Entwicklung und Implementierung eines Typinferenzsystems mit Dimensionstypen für funktionale Sprachen

Markus Schmidt

Diplomarbeit

19. Dezember 1995

eingereicht bei

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz / Softwaretechnologie

Johann Wolfgang Goethe-Universität

Frankfurt am Main

Fachbereich Informatik (20)

Hiermit bestätige ich, daß ich die vorliegende Arbeit selbständig verfaßt habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 19. Dezember 1995

Markus Schmidt

Danksagung

Ich möchte mich hiermit bei allen Personen bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben. Mein besonderer Dank gilt Sven-Erik Panitz und Prof. Dr. Manfred Schmidt-Schauß für ihre gute Betreuung und ihre wertvollen Anregungen.

Markus Schmidt

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die meisten modernen Programmiersprachen unterstützen das Konzept der *Typen*. Obgleich die formalen Darstellung von Typen recht unterschiedlich ist, liegt doch allen die gleiche Idee zugrunden, daß Anwendungen von Argumenten auf Funktionen wohl-definiert sein müssen.

Wie mathematische Funktionen haben Funktionen eines Programms einen Definitions- und Wertebereich. Daher ist eine einfache Form der automatischen Verifikation von Programmen möglich, indem überprüft wird, ob die Argumente einer Funktion aus ihrem Definitionsbereich sind. Diese Überprüfung (*type-checking*) kann vor (*statisch getypt*) oder während (*dynamisch getypt*) der Ausführung von Programmen erfolgen. Ein Beispiel für eine statisch getypte Programmiersprache ist C, wohingegen LISP dynamisch getypt ist.

In dieser Arbeit werden wir uns mit den Typen einer speziellen Klasse von Programmiersprachen beschäftigen und zwar den funktionale Programmiersprachen. Der Grund dafür ist, daß moderne funktionale Programmiersprachen die am weitesten entwickelten Typsystem besitzen.

1.1 Typen in funktionalen Sprachen

Als de-facto Standard für moderne funktionale Programmiersprachen werden wir in dieser Arbeit die Sprache Haskell[HPW92] verwenden. Haskell ist eine stark getypte Sprache, d.h. daß die Typüberprüfung vor der Ausführung des Programms erfolgt und im Falle einer fehlerfreien Überprüfung gesichert ist, daß zur Laufzeit keine Typfehler entstehen. Als Besonderheit bietet Haskell ein neues Konzept zur systematischen Überladung von Funktionen. Diese sogenannten *Typ-Klassen* sind aber zu komplex um sie hier zu betrachten.

Allen modernen funktionalen Programmiersprachen ist aber gemein, daß sie neben der einfachen Überprüfung von Typen auch die Herleitung von Typen beherschen. Diese sogenannte *Typinferenz* ist in der Lage den Typ von Funktion zu ermitteln, die ohne eine explizite Angabe des Typs definiert sind. Dazu folgendes Beispiel

```
test c zahl = if c == 's' then sin(zahl)
              else cos(zahl)
```

Die Typinferenz ermittelt für `test` den korrekten Typ `Char -> Float -> Float` und akzeptiert deshalb auch den folgenden Ausdruck.

```
test '?' 3.14
```

Dagegen wird eine Vertauschung der Argumente als Fehler erkannt, da bei der Typinferenz festgestellt wird, daß die Typen der Argumente nicht mit dem ermittelte Typ der Funktion `test` übereinstimmen. Damit scheint die Typinferenz einen guten Schutz vor Programmierfehlern zu bieten.

Fehler können aber nur erkannt werden, wenn Objekte die unterschieden werden müssen, auch unterschiedliche Typen haben.

Gehen wir einmal davon aus, daß wir den Preis und die Geschwindigkeit eines Autos mittels der Variablen `preis` und `geschwindigkeit` repräsentieren. Die beiden Variablen repräsentieren zwar unterschiedliche Objekte, haben aber den gleichen Typ `Float`. Dadurch können `preis` und `geschwindigkeit` bei der Typinferenz nicht unterschieden werden und die folgenden Ausdrücke werden nicht als Fehler erkannt.

```
preis + geschwindigkeit
preis * preis
sin(geschwindigkeit)
```

Damit bei der Typinferenz die obigen Ausdrücke als unzulässig erkannt werden, müssen die Typen von `preis` und `geschwindigkeit` unterschiedlich sein. Dies können wir erreichen, indem wir die neuen Datentypen `Geld` und `Speed` definieren[1].

```
data Geld  = DM  Float
data Speed = KMH Float

preis = DM 50000
geschwindigkeit = KMH 220
```

[1] Die Defintion von Datentypen geschieht in Haskell über das Schlüsselwort `data`. Ohne Typklassen sieht dies wie folgt aus: `data` $T\ a_1 \ldots a_n \;=\; K_1\ t_{1,1} \ldots t_{1,k_1} \mid \ldots \mid K_m\ t_{1,1} \ldots t_{1,k_m}$
Dabei ist T der neuen Datentyp (Typkonstruktor), a_i seine Typvariablen und K_i die Datenkonstruktoren mit ihren Typen t_{i,k_i}

Dadurch haben wir zwar unterschiedliche Typen, aber gleichzeitig auch ein neues Problem, denn auf die neuen Datentypen sind keine arithmetischen Operationen mehr anwendbar! So wird der Ausdruck `preis + preis` bei der Typinferenz als unzulässig erkannt, da `+` nicht auf dem Datentyp `Geld` definiert ist. Daher müssen alle arithmetischen Operationen, für beide neuen Datentypen, neu definieren. Für die Addition sieht dies wie folgt aus.

```
addGeld :: Geld -> Geld -> Geld
addGeld (DM g1) (DM g2) = DM (g1+g2)
addSpeed :: Speed -> Speed -> Speed
addSpeed (KMH s1) (KMH s2) = KMH (s1+s2)
```

Die korrekte Addition von Geldbeträgen erfolgt dann wie folgt.

```
 preis = addGeld preis1 preis2
```

Dies wirkt sehr umständlich und verschlechtert die Lesbarkeit von Programmen. Außerdem ergibt sich durch die zusätzliche Definiton von grundlegenden Funktionen für die neuen Datentypen ein erheblicher Aufwand. Es stellt sich daher folgende Frage:

Kann man diesen Aufwand reduzieren und wenn ja, wie ist dies zu realisieren ?

1.2 Ziel und Aufbau dieser Arbeit

Wie wir gesehen haben bieten die bestehenden Typsysteme nur eine sehr einfache Form der automatischen Verifikation von Programmen. Unser Ziel ist es deshalb ein Typsystem zu entwickeln, daß eine bessere automatische Programmverifikation ermöglicht und möglichst wenig zusätzlichen Aufwand beim Programmieren erfordert.

Im ersten Teil der Arbeit werden wir uns mit den nötigen theoretischen Grundlagen vertraut machen. Dabei werden wir uns in Kapitel 2 mit den Grundlagen der funktionalen Programmiersprachen befassen – dem λ-Kalkül . Beginnend mit dem ungetypten λ-Kalkül wird zuerst die explizite und danach die implizite Typisierung dargestellt. Danach werden wir die Sprache Exp einführen, die ein polymorphes Typsystem besitzt. Dieses Typsystem wird uns als Standardtypsystem für spätere Erweiterungen dienen. Am Ende dieses Kapitels werden wir aufzeigen, wie die einfache Sprache Exp um moderne Programmkonstrukte erweitert werden kann.

In Kapitel 3 wird das Problem aus dem vorherigen Abschnitt aufgegriffen und nach einer Lösung gesucht. Die Betrachtung von physikalischen Größen wird dabei die entscheidenden Erkenntnisse liefern. Auf der Grundlage dieser Erkenntnisse werden wir eine neue Art von Typen entwicklen – die *Dimensionstypen*. Um diese Dimensionstypen bei der Typinferenz zu verwenden, wird ein *Unifikationsalgorithmus* auf Dimensionen entwickelt.

Die Ergebnisse aus Kapitel 2 und 3 werden in Kapitel 4 benutzt um eine einfache funktionale Sprache, ExtCore, zu entwickeln. Das Typsystem dieser Sprache wird in der Lage sein, auf polymorphen Dimensionstypen zu arbeiten. Ein exakte Definition dieses Typsystems, ist der Hauptbestandteil dieses Kapitels sein.

Im Kapitel 5 wird die Implementierung eines kompletten Compiler Front-End für die Sprache ExtCore beschrieben. Diese Implementierung erfolgt in der funktionalen Sprache Haskell.

Kapitel 6 faßt diese Arbeit zusammen, diskutiert die erzielten Ergebnisse und zeigt auf, inwieweit diese Arbeit erweitert werden kann.

Kapitel 2

Die Grundlagen

Der λ-Kalkül ist ein formales System zur Behandlung von λ-Ausdrücken. Außer der Syntax, welche die Darstellung der λ-Ausdrücke festlegt, gibt es noch folgende wichtige Teilsysteme innerhalb des λ-Kalkül

1. Konversionssystem – dient zur Herleitung von Gleichheitsbeziehungen zwischen λ-Ausdrücken
2. Reduktionssystem – eine gerichtete Form des gleichungsbasierten Schließens
3. Modellsystem – eine Modelltheorie auf λ-Ausdrücken

In der Terminologie der Programmiersprachen würde man diese Systeme als axiomatische Semantik, operationale Semantik und denotationale Semantik bezeichnen. Da wir den λ-Kalkül und seine Derivate unter dem Aspekt der Programmiersprache betrachten, werden wir von den drei obigen Systemen nur das Reduktionssystem betrachten, weil es die Ausführung von Programmen am deutlichsten zum Ausdruck bringt. Für ausführlichere Informationen zum λ-Kalkül , insbesondere zum Konversions- und Modellsystem, muß auf [Bar84] verwiesen werden.

Der ungetypte λ-Kalkül wird in Abschnitt 2.1 eingeführt und dient uns als Basis für weitere Untersuchungen. Die dabei auftretenden Probleme werden wir in Abschnitt 2.2 versuchen zu lösen, wobei die Idee der *Typisierung* der entscheidende Lösungsansatz sein wird. Dabei wird uns die *implizite Typisierung* ein Verfahren names *Typinferenz* bereitstellen, daß in den meisten modernen Programmiersprachen nicht enthalten ist, obwohl es seit mehr als 50 Jahren bekannt ist. In Abschnitt 2.3 wir die Sprache Exp und deren *Typsystem* vorgestellt. Dabei werden vor allem die Eigenschaften des Typsystems, hinsichtlich einer automatischen Verifikation von Programmen, behandelt. Der Abschnitt 2.4 zeigt auf, wie die Sprache Exp erweitert werden kann, um einen gewissen Komfort beim Programmieren zu bieten.

2.1 Der ungetypte λ-Kalkül

2.1.1 Syntax

Definition 2.1 (λ-Ausdrücke) *Die Menge Λ der λ-Ausdrücke wird, ausgehend von einer abzählbaren Menge V von Variablen (typischerweise x,y,z) mittels der drei folgenden Regeln erzeugt.*

$$\begin{array}{lll} (1) & x \in V & \Rightarrow \; x \in \Lambda \\ (2) & e_1, e_2 \in \Lambda & \Rightarrow \; (e_1 e_2) \in \Lambda \\ (3) & e \in \Lambda \; , \; x \in V & \Rightarrow \; (\lambda x.e) \in \Lambda \end{array}$$

Ausdrücke der Form $(\lambda x.e)$ heißen Abstraktionen *und beschreiben die Definition einer Funktion. Funktionsanwendungen werden durch Ausdrücke der Form $(e_1 e_2)$, den* Applikationen*, beschrieben.*

Ungewohnt sind die durch Regel (3) entstehenden Ausdrücke. Um die neue Notation von Funktionsdefinitionen besser zu verstehen, vergleichen wir diese , mit der üblichen mathematischen Funktionsdefinition.

$$f(x) = e \;\; \widehat{=} \;\; f \equiv \lambda x.e$$

Das 'λ' bezeichnet eine folgende Funktionsdefinition mit einem formalen Parameter x und dem Funktionskörper e. Der '.' dient zur expliziten Trennung von Paramter und Funktionskörper. Die Anwendung der Funktion f auf einen Variable y würde wie folgt aussehen.

$$f(y) \;\; \widehat{=} \;\; (fy)$$

Um unnötige viele Klammerungen innerhalb von λ-Ausdrücken zu vermeiden legen wir fest, daß Applikationen linksassoziativ und Abstraktionen rechtsassoziativ sind. Applikationen und Abstraktionen lassen sich dann wie folgt schreiben.

$$ee_1 \ldots e_n \;\widehat{=}\; (\ldots(ee_1)\ldots e_n)$$

$$\lambda v_1 \ldots v_n.e \;\widehat{=}\; (\lambda v_1.(\ldots(\lambda v_n.e)\ldots))$$

Desweiteren lassen werden wir die äußeren Klammern weglassen.

Die λ-Ausdrücke $\lambda x.x$ und $\lambda y.y$ werden bisher noch unterschieden. Dies ist vom operationalen Standpunkt aus überflüssig, da beide die gleichen Funktionen repräsentieren. Wir wollen deshalb λ-Ausdrücke, die zwar eine unterschiedliche Syntax besitzen, aber die gleiche Funktion repräsentieren, nicht mehr unterscheiden. Bevor wir dies formal definieren können, müssen wir uns noch mit *freien* und *gebundenen* Variablen sowie der *Substitution* beschäftigen.

Definition 2.2 (freie und gebundene Variablen) *Die Menge $FV(e)$ der freien Variablen in e ist definiert durch*

$$\begin{array}{lrcl} (1) & FV(x) & = & \{x\} \\ (2) & FV(\lambda x.e) & = & FV(e) \setminus \{x\} \\ (3) & FV(e_1 e_2) & = & FV(e_1) \cup FV(e_2) \end{array}$$

Eine Variable x ist frei *in e, falls $x \in FV(e)$. Ansonsten ist sie in e* gebunden.

Variablen werden nur durch λ-Abstraktionen gebunden, da innerhalb eines Ausdrucks $\lambda x.e$ eine Beziehung zwischen dem x als formalen Parameter und weiteren Vorkommen von x im Funktionskörper e besteht. Freie Variablen besitzen solche Beziehungen nicht und können daher auch einfacher umbenannt werden.

Definition 2.3 (Substitution) *Seien e_1 und e_2 λ-Ausdrücke und x eine Variable. Dann bezeichnet $e_1[e_2/x]$ die Substitution jedes freien Vorkommens von x in e_1 durch e_2. Die Substitution ist definiert durch*

$$\begin{array}{rcl} x[e/x] & = & e \\ y[e/x] & = & y \\ (e_1 e_2)[e/x] & = & (e_1[e/x] e_2[e/x]) \\ (\lambda x.e_1)[e/x] & = & \lambda x.e_1 \\ (\lambda y.e_1)[e/x] & = & \begin{cases} \lambda y.e_1[e/x] & , x \notin FV(e_1) \\ \lambda z.(e_1[z/y])[e/x] & , x \in FV(e_1), z \notin (FV(e_1) \cup FV(e)) \end{cases} \end{array}$$

Besonders die letzte Regel ist zu beachten, weil hier ein Nameskonflikt auftreten kann, falls $x \in FV(e_1)$. Dieser wird durch eine Substitution von y in e_1 durch eine neue Variable z verhindert. Die folgende Substitution zeigt einen solchen Fall.

$$((\lambda y.x)(\lambda x.x)x)[y/x] \; = \; (\lambda z.y)(\lambda x.x)y$$

Nach diesen Vorarbeiten können wir nun definieren, wann unterschiedliche λ-Ausdrücke die gleiche Funktion repräsentieren.

Definition 2.4 (α-Kongruenz) *Zwei λ-Ausdrücke $\lambda x.e_1$ und $\lambda y.e_2$ sind α-kongruent, geschrieben $\lambda x.e_1 \equiv_\alpha \lambda y.e_2$, falls $e_2 \equiv e_1[y/x]$ und $y \notin FV(e_1)$.*

Der einzige Unterschied zwischen α-kongruenten Ausdrücken besteht also in der Bezeichnung der gebundenen Variablen. Man beachte, daß die Bedingung $y \notin FV(e_1)$ notwendig ist um eine Bindung von y durch die Substitution zu verhindern. Ohne diese Einschränkung wären $\lambda x.y$ und $\lambda y.y$ α-kongruent. Von nun an werden wir α-kongruente λ-Ausdrücke auch als identisch ansehen.

2.1.2 Operationelle Semantik

Wir werden λ-Ausdrücken Bedeutungen geben, indem wir angeben, wie wir sie benutzen.

Definition 2.5 (β-Reduktion) *Gegeben sei ein λ-Ausdruck der Form $(\lambda x.e_1)e_2$. Dann bezeichnen wir die Substitution von x durch e_2 in e_1 als* β-Reduktion, *geschrieben*

$$(\lambda x.e_1)e_2 \to e_1[e_2/x]$$

wobei der Ausdruck $(\lambda x.e_1)e_2$ Redex *heißt.*
Wir schreiben $e_1 \twoheadrightarrow e_2$, falls e_2 durch keine oder mehrere aufeinanderfolgende β-Reduktionen aus e_1 hervorgeht.

Da bei der Substitution eine Umbenennung von gebundenen Variablen notwendig sein kann, werden durch die β-Reduktion Ausdrücke erzeugt, die bis auf α-Kongruenz eindeutig bestimmt sind. Weil wir α-kongruente Ausdrücke als identisch ansehen, sind die durch die β-Reduktion erzeugten Ausdrücke, für uns eindeutig bestimmt. Betrachten wir die folgenden Reduktionen.

- $(\lambda x.xx)(\lambda y.y)z \to (\lambda y.y)(\lambda y.y)z \to (\lambda y.y)z \to z$
  oder auch kürzer
  $(\lambda x.xx)(\lambda y.y)z \twoheadrightarrow z$
- $(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$
- $(\lambda x.xxx)(\lambda x.xxx) \to (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)$

Das erste Beispiel zeigt die schrittweise β-Reduktion eines λ-Ausdrucks, hin zu einem λ-Ausdruck ohne Redex. Bei der vereinfachten Darstellung mittels '$\twoheadrightarrow$' werden die einzelnen Zwischenschritte weggelassen. An den Beispielen zwei und drei erkennt man, daß eine Reduktion nicht immer kürzere λ-Ausdrücke liefert. Im Gegensatz zum ersten λ-Ausdruck kann man an diesen beiden λ-Ausdrücken unendlich oft aufeinanderfolgende Reduktionen ausführen, da nie ein λ-Ausdruck ohne Redex entsteht.

Betrachtet man λ-Ausdrücke als Programme und die β-Reduktion als Berechnungsschritt, heißt dies, daß nur die Berechnung des ersten Programms terminiert.

Definition 2.6 (Normalform) *Ein λ-Ausdruck e heißt* Normalform (NF)*, falls es keinen λ-Ausdruck e_1 gibt, mit $e \to e_1$.*
Sind e_1, e_2 λ-Ausdrücke, dann hat e_1 die NF e_2, falls $e_1 \twoheadrightarrow e_2$ und e_2 eine NF ist.

Eine NF charakterisiert also das Ende einer Folge von β-Reduktionen und kann daher als Ergebnis einer Berechnung angesehen werden. Wie wir schon gesehen haben hat $(\lambda x.xx)(\lambda y.y)z$ die NF z. Es stellt sich sofort die Frage, ob solch eine NF eindeutig ist und wie sie erreicht werden kann. Die Eindeutigkeit der NF ist eine unmittelbare Folgerung des folgenden Satzes.

Satz 2.1 (Church-Rosser Theorem 1) *Sind e, e_1 und e_2 λ-Ausdrücke mit $e \twoheadrightarrow e_1$ und $e \twoheadrightarrow e_2$, dann existiert ein λ-Ausdruck e_3 mit $e_1 \twoheadrightarrow e_3$ und $e_2 \twoheadrightarrow e_3$.*

Eine weitere Folgerung des Satzes ist, daß eine NF durch unterschiedliche Reduktionsschritte erreicht werden kann. Die β-Reduktion ist ja definiert, durch die Reduzierung eines Redex. Da ein λ-Ausdruck mehr als einen Redex haben kann, ist nicht klar in welcher Weise die Wahl des Redex, das Erreichen der NF beeinflußt. Wir benötigen daher eine Strategie, die den nächsten Redex so wählt, daß eine mögliche NF auch erreicht wird. Die zwei wesentlichen Strategien werden im folgenden vorgestellt und hinsichtlich des Erreichens einer NF untersucht.

Definition 2.7 (Reduktionsstrategien) *Bei der Reduktion in* normaler Reihenfolge*, wird zuerst der am weitesten links stehende Teilausdruck reduziert. Bei der Reduktion in* applikativer Reihenfolge*, wird zuerst der innerste Teilausdruck reduziert.*

Da die Reduktion mit der Parameterübergabe in prozeduralen Programmiersprachen vergleichbar ist, kann man ein Reduktionsstrategie als eine spezielle Form der Parameterübergabe ansehen. So entspricht die Reduktion in normaler Reihenfolge, einer *call-by-name* Übergabe und die *call-by-value* Übergabe einer Reduktion in applikativer Reihenfolge.

Zwischen diesen beiden Strategien gibt es wesentliche Unterschiede. Die Reduktion in applikativer Reihenfolge kann effizienter sein, da der innerste Redex zuerst reduziert wird und so mehrfache Reduktionen dieses Redex zu einem späteren Zeitpunkt verhindert werden. Am folgenden Beispiel ist dies deutlich zu sehen.

$$
\begin{array}{ll}
\text{applikative Reihenfolge} & \text{normale Reihenfolge} \\
\ (\lambda x.xyxx)((\lambda x.x)z) & \ (\lambda x.xyxx)((\lambda x.x)z) \\
\rightarrow (\lambda x.xyxx)z & \rightarrow ((\lambda x.x)z)y((\lambda x.x)z)((\lambda x.x)z) \\
\rightarrow zyzz & \rightarrow zy((\lambda x.x)z)((\lambda x.x)z) \\
 & \rightarrow zyz((\lambda x.x)z) \\
 & \rightarrow zyzz
\end{array}
$$

Die Auswertung in applikativer Reihenfolge hat aber einen entscheidenden Nachteil. Es ist möglich, daß die Reduktion eines λ-Ausdrucks nicht terminiert, selbst wenn dieser eine NF hat. Das folgende Beispiel zeigt solch einen Fall.

applikative Reihenfolge	normale Reihenfolge
$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$	$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$
$\to (\lambda x.y)((\lambda x.xx)(\lambda x.xx))$	$\to y$
$\to \quad \vdots$	

Die Reduktion in normaler Reihenfolge terminiert, weil nur solche Redexe reduziert werden, deren Reduktion auch wirklich nötig ist. Im obigen Beispiel war die Reduktion des Teilausdrucks $((\lambda x.xx)(\lambda x.xx))$ nicht notwendig. Diese Eigenschaft, nur das benötigte zu reduzieren, hat eine wichtige Konsequenz, die der folgende Satz zeigt.

Satz 2.2 (Curch-Rosser Theorem 2) *Sind e_1 und e_2 λ-Ausdrücke mit $e_1 \twoheadrightarrow e_2$ und e_2 ist in NF, dann existiert eine Reduktion in normaler Reihenfolge von e_1 nach e_2.*

Die Reduktion in normaler Reihenfolge garantiert also das Erreichen der NF, falls diese existiert.

2.1.3 Berechnungsstärke

Funktionen werden ausschließlich durch λ-Abstraktionen definiert. Die λ-Abstraktionen sind *unbenannte* Funktionen und können sich daher nicht auf sich selbst beziehen. Rekursive Funktionen werden aber gerade mittels der Selbstreferenz definiert. Allgemein sehen rekursive Funktionen wie folgt aus

$$f \;\equiv\; \ldots f \ldots$$

Die Funktion f ist dabei eine *benannte* Funktion, wodurch eine Selbstreferenz möglich ist. Da es im ungetypten λ-Kalkül keine benannten Funktionen gibt, können keine rekursiven Funktionen definiert werden. Dadurch scheint auch die Berechnung von rekursiven Funktionen unmöglich.

Man könnte aber versuchen eine rekursive Funktion auf eine andere Art auszudrücken. Nämlich als eine Funktion, die sich selbst als Argument übergibt. Obige rekursive Funktion kann dann mittels λ-Abstraktion wie folgt beschrieben werden.

$$f \;\equiv\; (\lambda g.\ \ldots g \ldots)f$$

Dabei wird das Argument f an die Variable g gebunden und so kann im Funktionskörper von f auch auf f zugegriffen werden. Man erkennt, daß f ein Fixpunkt von $(\lambda g.\ \ldots g \ldots)$ sein muß. Könnten wir diesen Fixpunkt im ungetypten λ-Kalkül ausdrücken, dann könnten wir f auch berechnen. Das dies tatsächlich möglich ist zeigt der folgende Satz.

Satz 2.3 (Fixpunkt Theorem) *Jedes $e \in \Lambda$ hat einen Fixpunkt $(Ye) \in \Lambda$, d.h. $Ye \twoheadrightarrow e(Ye)$. Dabei ist der* Fixpunktkombinator Y *wie folgt definiert.*

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Damit können wir jede rekursiv definierte Funktion f, nicht rekursiv definieren durch

$$f \equiv Y(\lambda f.\ldots f\ldots)$$

Diese Fähigkeit, die Rekursion zu simulieren, ist der Schlüssel zur Berechnungsstärke des ungetypten λ-Kalkül . Dies wurde zuerst von Church erkannt.

Satz 2.4 (Berechenbarkeit) *Effektiv berechenbare Funktionen von natürlichen Zahlen zu natürlichen Zahlen, sind gerade die Funktionen, die im ungetypten λ-Kalkül definierbar sind.*

Die große Berechnungsstärke hat aber den Nachteil, daß nicht alle Reduktionen von λ-Ausdrücken terminieren. So hatten wir schon gesehen, daß $(\lambda x.xx)(\lambda x.xx)$ keine NF hat.

2.2 Getypte λ-Kalküle

In diesem Abschnitt werden wir die bisherigen Kombinationsmöglichkeiten von λ-Ausdrükken, auf sinnvolle Kombinationen einschränken. Sinnvolle Kombinationen werden wir dadurch erkennen, daß wir die Bedeutung von λ-Ausdrücken berücksichtigen.

2.2.1 Explizite Typisierung

Der in Abschnitt 2.1 eingeführte ungetypte λ-Kalkül erlaubte es beliebige λ-Ausdrücke zu kombinieren, solange die Syntax Regeln eingehalten wurden. Im allgemeinen haben die von uns konstruierten Ausdrücke eine intendierte Bedeutung, die aber bisher nicht berücksichtigt worden sind.

Definition 2.8 (Typ) *Die Annotation eines λ-Ausdruck e mit der zugehörigen Bedeutung τ heißt* Typisierung *und wird durch $e : \tau$ notiert.*
Diese Bedeutung τ werden wir im weiteren als Typ *bezeichnen.*

Die λ-Ausdrücke aus Definiton 2.1 werden nun um eine explizite Typisierung erweitert.

Definition 2.9 (typisierte Ausdrücke) *Die Menge Λ_τ der λ_τ-Ausdrücke wird, ausgehend von einer abzählbaren Menge V von Variablen (typischerweise x,y,z) und einer abzählbaren Menge T von Typausdrücken, mittels der drei folgenden Regeln erzeugt.*

$$\begin{array}{lll} (1) & x \in V & \Rightarrow \; x \in \Lambda_\tau \\ (2) & e_1, e_2 \in \Lambda & \Rightarrow \; (e_1 \; e_2) \in \Lambda_\tau \\ (3) & e \in \Lambda \;,\; x \in V \;,\; \tau \in T & \Rightarrow \; (\lambda x : \tau . e) \in \Lambda_\tau \end{array}$$

Die Regel (3) erzeugt nun Abstraktionen, bei denen der Typ der gebundene Variable explizit angegeben wird. Dadurch ist es möglich bei einer Applikation zu prüfen, ob das Argument den benötigten Typ hat.

Zusätzlich zu den neuen λ_τ-Ausdrücken benötigen wir nun auch *Typausdrücke*.

Definition 2.10 (Typausdrücke) *Die Menge T, der* Typausdrücke *ist definiert durch*

$$\begin{array}{lll} (1) & \alpha \in V & \Rightarrow \; \alpha \in T \\ (2) & b \in B & \Rightarrow \; b \in T \\ (3) & \tau_1, \tau_2 \in T & \Rightarrow \; \tau_1 \to \tau_2 \in T \end{array}$$

wobei V eine abzählbare Menge von Typvariablen *ist und B ein abzählbare Menge von* Basistypen *(z.B.* `Int`*) ist.*

Die Regel (2) erzeugt Typausdrücke, die den Typ ein Abstraktion $(\lambda x.e)$ bescheiben. Dabei bezeichnet τ_1 den Typ der gebundenen Variable x und τ_2 den Typ des Funktionskörpers e.

Typausdrücke repräsentieren Typen. So ist der Typ der Zahl 2, die Menge aller ganzen Zahlen[1] . Geschrieben wird dies durch 2:`Int`, d.h. `Int` repräsentiert die Menge aller ganzen Zahlen. Die Unterscheidung zwischen Typen und Typausdrücken ist zwar wichtig, aber in den meisten Fällen überflüssig, da aus dem Kontext hervorgeht, ob der Typ oder der Typausdruck gemeint ist. Daher werden wir im weiteren Verlauf auf eine explizite Unterscheidung verzichten, d.h. wir sagen: „der Typ von 2 ist `Int` “. Falls die Unterscheidung nicht aus dem Kontext hervorgeht, wird diese explizit angegeben.

Da der Typ eines Ausdrucks, auch von den Typen der freien Variablen abhängt, müssen diese auf eine geeignete Weise verwaltet werden. Dafür führen wir die *Typumgebung* ein.

Definition 2.11 (Typumgebung) *Die* Typumgebung Γ *eines Ausdrucks e, ist eine endliche Menge von Typisierungen $\{(x_1 : \tau_1), ..., (x_n : \tau_n)\}$, wodurch jedem $x_i \in FV(e)$ genau ein $\tau \in T$ zugewiesen wird.*
Die Umgebung ohne Typisierungen bezeichnen wir mit $\emptyset$. $\Gamma(x : \tau)$ bezeichnet die Umgebung, die entsteht, wenn aus Γ eine, eventuell vorhandene, Typisierung von x entfernt wird und $(x : \tau)$ hinzugefügt wird.

[1] Dies entspricht der intuitiven Vorstellung , daß Typen eine Menge von Werten sind. Dies ist aber nicht das einzig möglich Modell von Typen. So stellt Cartwright [Car85] Typen als Intervalle dar und MacQueen, Plotkin und Sethi [MPS86] betrachten Typen als mathematische Ideale. Vor allem das letzte Modell ist für die Behandlung von rekursiven Typen wichtig.

Damit sind wir in der Lage *Typregeln* zu erstellen, mit deren Hilfe wir die korrekte Typisierung eines Ausdrucks überprüfen können. Die Typregeln werden wir wie folgt darstellen.

$$\textbf{Name}\quad \frac{\begin{matrix}\text{Prämisse } 1\\ \vdots \\ \text{Prämisse } n \quad n \geq 0\end{matrix}}{\text{Konklusion}}$$

Diese Regeln sind wie folgt zu lesen : „Sind alle Prämissen gültig, dann ist auch die Konklusion gültig“. Die Prämissen und Konklusionen bestehen aus den *Typentscheidungen*. Diese Typentscheidungen haben die Form $\Gamma \vdash e : \tau$ und sind wie folgt zu lesen : „Aus der Typumgebung Γ folgt, daß der Ausdruck e den Typ τ hat“. Die Menge der Typregeln und der zugehörigen Typen, bezeichnen wir als *Typsystem*.

Das Typsystem über Λ_τ-Ausdrücke definieren wir, indem wir für alle 3 Formen der Ausdrücke angeben, wie die Typen der Teilausdrücke aussehen müssen, damit der Typ des Gesamtausdruck korrekt ist. Für Variablen ist die Typregel sehr einfach, da nur in der Umgebung nachgesehen werden muß, welche Typisierung die Variable hat. Die *lambda*-Abstraktion $(\lambda x : \tau_1.e)$ hat einen Typ der Form $\tau_1 \to \tau_2$. Daher muß e, in einer um $(x : \tau_1)$ erweiterten Umgebung, den Typ τ_2 haben. Damit die Applikation $(e_1 e_2)$ den Typ τ_2 hat, muß e_1 eine Funktion vom Typ $\tau_1 \to \tau_2$ sein und e_2 den entsprechenden Argumententyp τ_1 haben. Formal sieht diesen Typsystem daher wie folgt aus.

$$\textbf{VAR}\quad \Gamma(x : \tau) \vdash x : \tau$$

$$\textbf{ABS}\quad \frac{\Gamma(x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.e) : \tau_1 \to \tau_2}$$

$$\textbf{APP}\quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 e_2) : \tau_2}$$

Abbildung 2.1 : Typregeln für Λ_E

Definition 2.12 *Ein λ_τ-Ausdruck e heißt* wohlgetypt*, falls ein Typumgebung Γ und ein Typ τ existieren, für die gilt : $\Gamma \vdash e : \tau$.*

Definition 2.13 Λ_E *ist die Menge aller wohlgetypen λ_τ-Ausdrücke, d.h. sei $e \in \Lambda_\tau$ und es gibt Γ,τ mit $\Gamma \vdash e : \tau$, dann gilt $e \in \Lambda_E$.*

Man beachte, daß unsere neue Sprache Λ_E nicht mehr kontext-frei ist, da die Einschränkung auf wohlgetypte Ausdrücke kontext-sensitiv ist. Alle statisch-getypten Sprachen sind notwendigerweise kontex-sensitiv. Diese Sprachen sind nach der gleichen Methode definiert wie unsere Sprache Λ_E. Erst wird durch Angabe einer kontext-freien Sprache (bei uns Λ_τ) eine echte Obermenge aller Ausdrücke definiert. Die eigentliche Sprache wird dann, durch Einschränkung dieser Obermenge auf wohlgetypte Ausdrücke, definiert.

Neben der Überprüfung liefert das obige Typsystem auch den Typ des zu überprüfenden Ausdrucks zurück. Wir können dieses System also auch benutzen um den Typ eines Ausdruck zu ermitteln. Dies wollen wir am Beispiel des Ausdruck $\lambda x:\tau_1.(\lambda y:\tau_1{\rightarrow}\tau_2.yx)$ einmal demonstrieren.

$$\cfrac{\cfrac{\cfrac{\{x:\tau_1, y:\tau_1{\rightarrow}\tau_2\}\vdash y:\tau_1{\rightarrow}\tau_2\ \ \mathbf{VAR}\quad \{x:\tau_1, y:\tau_1{\rightarrow}\tau_2\}\vdash x:\tau_1\ \ \mathbf{VAR}}{\{x:\tau_1, y:\tau_1{\rightarrow}\tau_2\}\vdash yx:\tau_2}\ \mathbf{APP}}{\{x:\tau_1\}\vdash \lambda y:\tau_1{\rightarrow}\tau_2.yx:(\tau_1{\rightarrow}\tau_2){\rightarrow}\tau_2}\ \mathbf{ABS}}{\emptyset\vdash \lambda x:\tau_1.(\lambda y:\tau_1{\rightarrow}\tau_2.yx):\tau_1{\rightarrow}(\tau_1{\rightarrow}\tau_2){\rightarrow}\tau_2}\ \mathbf{ABS}$$

Ausgehend von den Variablen x und y, erhält man durch Anwendung der entsprechenden Typregel für die einzelnen Teilausdrücke, schließlich den Typ des Gesamtausdrucks.

Wir wollen nun untersuchen, welche Auswirkungen die Einschränkung auf wohlgetypte λ_τ-Ausdrücke, auf die Existenz einer NF von Ausdrücken aus Λ_E hat. Dazu überprüfen wir zuerst, ob es eine Typumgebung Γ gibt, in welcher der Ausdruck $(\lambda x:\tau.xx)$, wohlgetypt ist.

Falls es ein Γ und ein τ_1 gibt mit $\Gamma\vdash(\lambda x:\tau.xx):\tau_1$ dann muß die zuletzt angewandte Regel **ABS** gewesen sein. Es hätte daher folgende Situation bestanden :

$$\tau_1=\tau\rightarrow\tau_2 \text{ und } \{x:\tau\}\vdash xx:\tau_2$$

Davor kann nur die **APP** Regel angewandt worden sein. Vor der Anwendung dieser Regel muß die Situation wie folgt gewesen sein

$$\tau=\tau_3\rightarrow\tau_2\ \mathit{und}\ \{x:\tau_3\rightarrow\tau_2\}\vdash x:\tau_3\rightarrow\tau_2\quad \{x:\tau_3\rightarrow\tau_2\}\vdash x:\tau_3$$

Daraus folgt, daß **VAR** als erste Regel angewandt wurde und $\tau_3=\tau_3\rightarrow\tau_2$ gewesen sein muß. Dies ist aber unmöglich und deshalb gilt $(\lambda x:\tau.xx)\notin\Lambda_E$. Daraus folgt, daß auch $(\lambda x:\tau.xx)(\lambda x:\tau.xx)\notin\Lambda_E$.

Dieser Ausdruck ist die explizit getypte Form eines λ-Ausdrucks, der keine NF hat. Somit haben wir zumindest einen Ausdruck ohne NF aus der Sprache Λ_E verbannt. Könnte es vieleicht sein, daß durch die Einschränkung auf wohlgetypte λ_τ-Ausdrücke, sogar alle Ausdrücke ohne NF ausgesondert wurden ? Der folgende Satz gibt darauf eine positive Bestätigung.

Satz 2.5 *Jeder $e \in \Lambda_E$ hat eine NF. Somit ist jeder $e \in \Lambda_E$ stark normalisiert, d.h. jede Reduktion von e terminiert.*

Durch die Typisierung haben wir also eine garantierte Terminierung der Reduktion eines beliebigen $e \in \Lambda_E$ gewonnen. Anders ausgedrückt, jedes Programm aus Λ_E terminiert. Aber was haben wir durch die Typisierung verloren? Es stellt sich heraus, daß der Gewinn einer gesicherten Terminierung mit einem Verlust an Berechnungsstärke bezahlt worden ist. Wir haben gesehen, daß $(\lambda x : \tau.xx)(\lambda x : \tau.xx) \notin \Lambda_E$. Da dieser Ausdruck wesentlicher Bestandteil des Fixpuktkombinators Y ist, ist auch die explizit getypte Form von Y nicht in Λ_E.

Satz 2.6 *Der Fixpunktkombinator Y, ist nicht in Λ_E enthalten. Dies bedeutet, daß keine rekursiven Funktionen in Λ_E definierbar sind.*

Kommen wir aber nun noch einmal auf das vorherige Beispiel zurück. Betrachtet man es genauer, dann stellt man sich die Frage, ob die explizite Typisierung der Argumente von λ-Abstraktionen überhaupt nötig ist. Es scheint, als ob sich der Typ dieser Argumente während dem Durchlaufen der Typregeln ermitteln ließe.

Diese Idee, den Typ von Ausdrücken ohne explizite Typisierung nur mit Hilfe der Typregeln zu ermitteln, wird als *implizite Typisierung* bezeichnet. Wie der nächste Abschnitt zeigen wird, ist es tatsächlich möglich ohne explizite Typisierung den Typ eines Ausdruck zu ermitteln.

2.2.2 Implizite Typisierung

Zunächst definieren wir eine Sprache, die keine explizite Typisierung enthält, aber trotzdem nur aus wohlgetypten Ausdrücken besteht.

Definition 2.14 *Λ_I ist die Menge aller wohlgetypten λ-Ausdrücke, d.h. für alle $e \in \Lambda_I$ gilt : $e \in \Lambda$ und es gibt Γ τ mit $\Gamma \vdash e : \tau$.*

Die Typregeln für λ_I-Ausdrücke sind gegenüber den Regeln aus Abbildung 2.1 nur leicht verändert. Nur die **ABS** Regel mußte wegen der neuen Syntax verändert werden.

$$
\begin{array}{ll}
\textbf{VAR} & \Gamma(x:\tau) \vdash x:\tau \\[1em]
\textbf{ABS} & \dfrac{\Gamma(x:\tau_1) \vdash e:\tau_2}{\Gamma \vdash (\lambda x.e):\tau_1 \to \tau_2} \\[1.5em]
\textbf{APP} & \dfrac{\Gamma \vdash e_1:\tau_1 \to \tau_2 \quad \Gamma \vdash e_2:\tau_1}{\Gamma \vdash (e_1 e_2):\tau_2}
\end{array}
$$

Abbildung 2.2 : Typregeln für Λ_I

Wir wollen nun versuchen die Typen von ungetypten Ausdrücken herzuleiten. Dazu überlegen wir uns noch einmal, wie wir beim Beispiel aus dem vorherigen Abschnitt vorgegangen sind. Wir haben sukzessive die Typregeln von dem innersten Teilausdruck hin zum Gesamtausdruck angewandt. Dies konnten wir machen, weil wir die Typen der innernsten Teilausdrücken (x,y) schon kannten. Diese Information fehlt uns aber jetzt und um die Typregeln anzuwenden, benötigen wir diese Informationen. Daher werden wir uns mit einem Trick behelfen und so tun, als ob wir die benötigten Informationen hätten. Wir gehen dabei wie folgt vor.

> Wir *raten* anfangs die Typen von ungetypten Ausdrücken und können dann mit Hilfe der Typregeln überprüfen, ob unsere Annahme korrekt war.
> Dabei machen wir am Anfang die allgemeinste Annahme (*raten*) und spezifizieren den Typ genauer, während wir die Typregeln durchlaufen.

Wir müssen daher für jeden Teilausdruck e eine Variable t_e bereitstellen, die den Typ des Teilausdrucks repräsentiert. Bei jeder Regelanwendung werden diese Variablen genauer spezifiziert. So erhält man bei Anwendung der **ABS** Regel, die Zusatzinformation $t_e = t_x \to t_{e_1}$. Dies bedeutet, daß der Typ von e eine Funktion ist, dessen Argument vom Typ t_x ist und einen Wert vom Typ t_{e_1} liefert. Sind alle Teilausdrücke durchlaufen, erhält man eine Menge von Gleichungen. Diese Gleichungen spezifizieren die Typen der einzelnen Teilausdrücke. Hat dieses Gleichungssystem eine Lösung, dann ist der Gesamtausdruck wohlgetypt.

Im Unterschied zur reinen Typüberprüfung werden hier 2 Phasen benötigt. In der ersten Phase werden mit Hilfe der Typregeln, Informationen über die Typen der Teilausdrücke gesammelt. Erst in der zweiten Phase kann die Überprüfung der Typen beginnen.

Diese Vorgehensweise bezeichnen wir als *Typinferenz*, da die Typen nicht nur überprüft werden, sondern auch selbst hergeleitet werden.

Der zugehörige Algorithmus sieht wir folgt aus.

1. Die Typregeln liefern für jeden Teilausdruck folgender Gleichung

$$
\begin{array}{lll}
(\textbf{VAR}) & e \equiv x & t_x = \alpha_x \\
(\textbf{ABS}) & e \equiv (\lambda x.e_1) & t_e = t_x \to t_{e_1} \\
(\textbf{APP}) & e \equiv (e_1 e_2) & t_{e_1} = t_{e_2} \to t_e
\end{array}
$$

2. Bestimme die Lösung des resultierenden Gleichungssystems S.

Man erkennt, daß die Anwendbarkeit des Algorithmus nur vom 2. Schritt abhängt. Um festzustellen ob der Algorithmus tatsächlich anwendbar ist, müssen wir noch folgende Fragen beantworten.

- Was ist eine Lösung von S ?
- Wie berechnet man diese Lösung ?
- Ist diese Lösung eindeutig ?

Zur Beantwortung dieser Fragen betrachten wir das resultierend Gleichungssystem S genauer. S ist eine Folge $s_1 = t_1, ..., s_n = t_n$ von Gleichungen zwischen Typen τ_e von Ausdrücken und Variablen α_x. Allgemeiner gesagt ist S ein Gleichungssystem auf Ausdrücken, wobei die Ausdrücke entweder aus T (die Menge der Typausdrücke) oder Variablen sind. Eine Lösung von S ist daher eine Zuweisung von Typausdrücken an Variablen, wodurch alle Paare (s_i, t_i) syntaktisch identisch werden. Jetzt erkennt man den Zusammenhang zwischen einer Lösung von S und der *Unifikation*. Bei der Unifikation wird versucht durch eine bestimmte Substitution (*Unifikator* genannt) eine endliche Menge von Ausdrücken syntaktisch identisch zu machen. Existiert dieser Unifikator, dann ist diese Menge *unifizierbar*, d.h. durch Anwendung des Unifikators auf diese Ausdrücke werden diese syntaktisch identisch.

Die Konzepte der Unifikation gehen zurück auf Herbrand. Doch erst die Arbeiten von Robinson [Rob65] machte die Unifikation bekannt. Die wichtigsten Ergebnisse seiner Arbeit sind

- falls eine endliche Menge von Ausdrücken einen Unifikator besitzt, dann gibt es einen *allgemeinsten* Unifikator, der in allen anderen Unifikatoren enthalten ist.
- der allgemeinste Unifikator ist bis auf Variablenumbenennung *eindeutig* bestimmt.
- es gibt einen effektiven Algorithmus der den allgemeinsten Unifikator berechnet, falls dieser existiert. Existiert dieser nicht, dann erkennt dies der Algorithmus. Der Algorithmus terminiert also garantiert !

Übertragen wir diese Ergebnisse auf unser Problem, dann ergeben sich die folgenden Fakten

- wir haben einen Algorithmus, der uns die Lösung von S berechnet, falls diese existiert. Existiert keine Lösung, erhalten wir eine entsprechende Meldung.
- die Lösung ist eindeutig und so allgemein wie möglich.

Die Betrachtung der Unifikation werden wir auf den nächste Abschnitt verschieben, wo ein effizienterer Algorithmus behandelt wird. Dieser Algorithmus wurde angegeben, weil er die Typinferenz sehr übersichtlich darstellt. Deshalb betrachten wir am Beispiel des Ausdruck $\lambda xyz.xz(yz) \equiv \lambda x.(\lambda y.(\lambda z.(xz)(yz))))$, wie der Algorithmus arbeitet.

1.Phase: Aufstellen des Gleichungssystems

Typumgebung	Ausdruck	neue Gleichung
$\lambda xyz.xz(yz) : \tau_0$	$\lambda x.(\lambda y.(\lambda z.(xz)(yz))))$	$\tau_0 = \tau_1 \to \tau_2$
$\lambda xyz.xz(yz) : \tau_0, x : \tau_1$ $\lambda yz.xz(yz) : \tau_2$	$\lambda y.(\lambda z.(xz)(yz)))$	$\tau_2 = \tau_3 \to \tau_4$
$\lambda xyz.xz(yz) : \tau_0, x : \tau_1$ $\lambda yz.xz(yz) : \tau_2, y : \tau_3$	$\lambda z.(xz)(yz))$	$\tau_4 = \tau_5 \to \tau_6$
$\lambda xyz.xz(yz) : \tau_0, x : \tau_1$ $\lambda yz.xz(yz) : \tau_2, y : \tau_3$ $\lambda z.xz(yz) : \tau_4, z : \tau_5, (xz)(yz) : \tau_6$	$(xz)(yz)$	$\tau_7 = \tau_8 \to \tau_6$
$\lambda xyz.xz(yz) : \tau_0, x : \tau_1$ $\lambda yz.xz(yz) : \tau_2, y : \tau_3$ $\lambda z.xz(yz) : \tau_4, z : \tau_5, (xz)(yz) : \tau_6$ $(xz)(yz), (xz) : \tau_7, (yz) : \tau_8$	xz	$\tau_1 = \tau_5 \to \tau_7$
$\lambda xyz.xz(yz) : \tau_0, x : \tau_1$ $\lambda yz.xz(yz) : \tau_2, y : \tau_3$ $\lambda z.xz(yz) : \tau_4, z : \tau_5, (xz)(yz) : \tau_6$ $(xz)(yz), (xz) : \tau_7, (yz) : \tau_8$	yz	$\tau_3 = \tau_5 \to \tau_8$

2.Phase: Lösung des Gleichungssystems (nur τ_0)

$$\begin{aligned}
\tau_0 =& \tau_1 \to \tau_2 \\
\tau_0 =& (\tau_5 \to \tau_7) \to \tau_2 \\
\tau_0 =& (\tau_5 \to \tau_7) \to \tau_3 \to \tau_4 \\
\tau_0 =& (\tau_5 \to \tau_7) \to (\tau_5 \to \tau_8) \to \tau_4 \\
\tau_0 =& (\tau_5 \to \tau_7) \to (\tau_5 \to \tau_8) \to \tau_5 \to \tau_6 \\
\tau_0 =& (\tau_5 \to \tau_8 \to \tau_6) \to (\tau_5 \to \tau_8) \to \tau_5 \to \tau_6
\end{aligned}$$

Der Vorteil dieses Verfahrens ist sicherlich die Übersichtlichkeit. Durch die Trennung in 2 Phasen, erkennt man sehr deutlich den Unterschied zur reinen Typüberprüfung. Leider muß zuerst ein Gleichungsystem für das komplette Programm erstellt werden. Dadurch wird man bei entsprechend großen Programmen, mit dem Lösen eines riesigen Gleichungssystem konfrontiert. Wesentlich effizienter wäre es, wenn man ein Programm in seine unabhängigen Bereiche aufteilen würde und diese Bereich einzeln überprüft.

Dies wurde auch von Dale Milner erkannt und er entwickelte einen Algorithmus zur Typinferenz, der schon während der Erstellung des Gleichungssystems versucht, dieses zu lösen. Milners Arbeiten [Mil78] [DM82] sind die Standardlitertatur auf dem Gebiet der Typsysteme und auch wir werden sein Typsystem als Basis für unsere Erweiterung benutzen.

2.3 Die Sprache Exp

Wir betrachten nun eine Sprache, die als Standardsprache auf dem Gebiet der Typinferenz gilt. Diese Sprache Exp, wird in [DM82] wie folgt angegeben.

e	$\rightarrow$	x	Variable
	$\mid$	$e_1 e_2$	Applikation
	$\mid$	$\lambda x.e$	Abstraktion
	$\mid$	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	lokale Definition

Abbildung 2.3 : Syntax von Exp

Das neue **let**-Konstrukt dient zur Defintion von lokalen Funktionen. Dabei ist x der Funktionsname, e_1 der Funktionsrumpf und e_2 die Umgebung, in welcher die neue Funktion benutzt werden kann. Vom operationalen Standpunkt aus, ist das **let**-Konstrukt keine Erweiterung, da es funktionel identisch ist mit dem Ausdruck $(\lambda x.e_2)e_1$.

Die eigentliche Bedeutung des **let**-Konstrukt liegt in der besonderen Weise, wie es getypt wird. Das **let**-Konstrukt gestattet es, *polymorphe* Funktionen zu definieren, d.h. Funktionen die mehr als einen Typ haben. Für diese neue Art von Typen brauchen wir natürlich auch eine neue Darstellung. Diese sieht wir folgt aus.

Mono Typ	τ	$\rightarrow$	α	Typvariable
		$\mid$	$\tau_1 \rightarrow \tau_2$	Funktionstyp
Poly Typ	σ	$\rightarrow$	τ	monomorher Typ
		$\mid$	$\forall\ \alpha_1\ \ldots\ \alpha_n.\ \tau$	Typschema

Abbildung 2.4 : Syntax der Typen von Exp

Deutlich zu Erkennen ist die Unterteilung der Typen in 2 Bereiche. Die uns bekannten Typen τ, werden wir ab jetzt *monomorphe* Typen nennen, da sie genau einen Typ repräsentieren. Im Gegensatz dazu können die polymorphen Typen mehr als einen Typ repräsentieren. Dies ist möglich, weil die Typvariablen α_i innerhalb eines polymorphen Typs gebunden sind. Dadurch können die Typvariablen bei jeder Verwendung des polymorphen Typs neu *instanziiert* werden, d.h. durch einen neuen Typ ersetzt werden. Bei einem monomorphen Typ τ ist dies nicht möglich, da die Typvariablen in τ frei vorkommen. Eine polymorphe Typen bezeichnet man auch als Typschema.

Bevor wir die Typregeln von Exp angeben können, müssen wir noch einige Begriffe definieren.

Definition 2.15 (freie Typvariablen) *Die Menge $fv(\tau)$, der* freien Typvariablen *von τ, ist definiert durch*

$$\begin{array}{lrcl}
(1) & fv(\alpha) & = & \{\alpha\} \\
(2) & fv(\tau_1 \to \tau_2) & = & fv(\tau_1) \cup fv(\tau_2) \\
(3) & fv(\forall \alpha_1 \ldots \alpha_n.\tau) & = & fv(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}
\end{array}$$

Definition 2.16 (freie Typvariablen einer Typumgebung) *Die Menge $fv(\Gamma)$, der* freien Typvariablen einer Typumgebung Γ, *ist definiert durch*

$$\begin{array}{lrcl}
(1) & fv(\emptyset) & = & \emptyset \\
(2) & fv(x_1 : \tau_1, \ldots, x_n : \tau_n) & = & \bigcup_{i=1}^{n} fv(\tau_i)
\end{array}$$

Dadurch ist fv zwar überladen, aber durch die Verwendung ($fv(\sigma)$ oder $fv(\Gamma)$) geht eindeutig hervor, welche Menge gemeint ist.

Definition 2.17 (Substitution auf Typen) *Eine* Substitution auf Typen, *ist eine Abbildung von Typvariablen nach Typen und wird mit θ notiert. Die explizite Angabe einer Substitution erfolgt durch $\theta = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ und bezeichnet die Zuordnung von α_i nach τ_i für $1 \leq i \leq n$. Die Anwendung dieser Substitution auf einen Typ τ, geschrieben $\theta\tau$, erfolgt durch das Ersetzen jeden freien Vorkommens der α_i durch τ_i. Die identische Substitution bezeichen wir mit* id.

Definition 2.18 (Instanz eines Typs) *Ein Typ $\sigma = \forall \alpha_1 \ldots \alpha_n.\tau$ hat eine* Instanz $\sigma' = \forall \beta_1 \ldots \beta_m.\tau'$, *falls $\tau' = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau$ und $\beta_i \notin fv(\tau)$ $1 \leq i \leq m$. Dies notieren wir durch $\sigma \geq \sigma'$.*

Damit sind wir in der Lage, daß Typsystem von Exp anzugeben.

$$\begin{array}{ll}
\textbf{VAR} & \Gamma(x : \sigma) \vdash x : \sigma \\[1ex]
\textbf{INST} & \dfrac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma'} \quad (\sigma \geq \sigma') \\[2ex]
\textbf{GEN} & \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall \alpha_1 \ldots \alpha_n.\tau} \quad \alpha_i \notin FV(\Gamma) \;\; 1 \leq i \leq n \\[2ex]
\textbf{ABS} & \dfrac{\Gamma(x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2} \\[2ex]
\textbf{APP} & \dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 e_2) : \tau_2} \\[2ex]
\textbf{LET} & \dfrac{\Gamma \vdash e_1 : \sigma \quad \Gamma(x : \sigma) \vdash e_2 : \tau}{\Gamma \vdash (\textbf{let}\ x = e_1\ \textbf{in}\ e_2) : \tau}
\end{array}$$

Abbildung 2.5 : Typsystem von Exp

Die **LET**-Regel überprüft die Einführung polymorpher Funktionen. Dazu wird zuerst der Typ des Funktionsrumpf e_1 ermittelt und danach die Funktionsumgebung e_2, in einer erweiterten Typumgebung, überprüft. Diese erweiterte Typumgebung entsteht durch das Hinzufügen einer Typisierung von x. Der Typ des Funktionsnamens x entspricht dabei dem Typ des Funktionsrumpfs e_1. Die Typregel **INST** dient zur Herleitung von Instanzen von Typen. Die umgekehrte Richtung, die Verallgemeinerung von Typen, wird in der Typregel **GEN** beschrieben. Dabei ist es wichtig, daß nur solche Typvariablen gebunden werden dürfen, die in der Typumgebung nicht frei vorkommen.

Nachdem wir nun schon einige Typregeln angegeben haben, ist es an der Zeit zu überprüfen, inwieweit diese auch *korrekt* sind. Korrekt heißt dabei, daß die auf der syntaktischen Ebene ($\Gamma \vdash e : \tau$) hergeleitete Typisierung, auch auf der semantischen Ebene gültig ist. Milner bewies in [Mil78],daß das Typsystem aus Abbildung 2.5 in diesem Sinne korrekt ist. Wir werden hier sein Vorgehen nur kurz wiedergeben, da insbesondere die Definiton der Semantik und der zugehörigen Funktionen einiges an Aufwand erfordert.

Die Semantik von Exp definiert Milner als denotationale Semantik [SS71]. Die semantische Domain ist die Lösung der folgenden reflexiven Domaingleichung.

$$\begin{aligned}\mathcal{V} &= \mathcal{B}_0 \oplus \cdots \oplus \mathcal{B}_\mathrm{n} \oplus \mathcal{F} \oplus \{\mathsf{wrong}\}\\ \mathcal{F} &= \mathcal{V} \to \mathcal{V}\end{aligned}$$

Hierbei sind $\mathcal{B}_0, \ldots, \mathcal{B}_\mathrm{n}$ die Basisbereiche (mit $\perp$) wie z.B. Wahrheitswerte und $\mathcal{F}$ der Bereich der Funktionen. Die Funktion $\mathcal{E}$:Exp$\to$Env$\to\mathcal{V}$ weist Ausdrücken semantische Werte zu und ist so definiert, daß Ausdrücke mit Typfehlern den Wert wrong erhalten Die Umgebung Env wird über die Funktion $\rho : Var \to \mathcal{V}$ realisiert, welche Variablen semantische Werte zuordnet.

Die semantische Relation $\models$ identifiziert jeden Typen τ mit einem Ideal $\mathcal{V}^\tau$ aus dem Bereich $\mathcal{V}$. Ein denotationaler Wert $v \in \mathcal{V}$ *besitzt* den Typ τ, geschrieben $\models v : \tau$, falls $v \in \mathcal{V}^\tau$. Eine Umgebung ρ *beachtet* eine Typumgebung Γ, geschrieben $\models \rho : \Gamma$, falls der Wertebereich von ρ mindestens so groß ist wir die von Γ und für jede Variable aus dem Wertebereich von Γ gilt: $\models \rho(x) : \Gamma(x)$.

Nach diesen technischen Vorarbeiten, bewies Milner den folgenden Satz.

Satz 2.7 (Korrektheit des Typsystems) *Falls* $\Gamma \vdash e : \tau$ *und* $\models \rho : \Gamma$*, dann gilt:* $\models \mathcal{E}[\![e]\!]\rho : \tau$.

Der Beweis dieses Satzes und die genauen Definitonen der Semantik, sind in [Mil78] zu finden.

Damit haben wir zwar ein korrektes Typsystem, aber um es anwenden zu können müssen wir es implementieren. Dies realisieren wir, indem wir einen entsprechenden Algorithmus angeben. Um den Algorithmus zur Typinferenz einfacher angeben zu können, werden wir das Typsystem aus Abbildung 2.5 ein wenig verändern. Dieses neue Typsystem ist deterministisch, da es für jede Ausrücke aus Exp, genau eine Regel gibt. In [CDDK86] wurde gezeigt, das dieses Typsystem äqivalent ist zum nichtdeterministischen Typsytem aus Abbildung 2.5.

VAR	$\Gamma(x:\sigma) \vdash x : inst(\sigma)$
ABS	$\dfrac{\Gamma(x:\tau_1) \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2}$
APP	$\dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1 e_2) : \tau_2}$
LET	$\dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma(x : gen(\Gamma, \tau_1)) \vdash e_2 : \tau_2}{\Gamma \vdash (\textbf{let } x = e_1 \textbf{ in } e_2) : \tau_2}$

Abbildung 2.6 : deterministisches Typsystem von Exp

Die Typregeln **INST** und **GEN** sind weggefallen und dafür die beiden Funktionen *inst* und *gen* hinzugekommen. Diese beiden Funktionen übernehmen die Funktion der Typregeln **INST** und **GEN** in jeweils einer Typregel. Die Funktion *inst* instanziiert ein Typ, indem alle gebundenen Typvariablen, durch neuen Typvariablen ersetzt werden. Die umgekehrte Richtung übernimmt die Funktion *gen*. Diese Funktion bindet solche Typvariablen innerhalb des angegebene Typs τ, die in τ frei vorkommen und in der Typumgebung Γ nicht frei vorkommen. Es werden also nur solche Typvariablen gebunden, die ausschließlich von τ verwendet werden. Die genaue Definition der beiden Funktion sieht wie folgt aus.

$$inst(\forall \alpha_1 \ldots \alpha_n.\tau) = \tau[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \quad , \qquad \text{wobei } \beta_i \text{ neuen Typvariablen sind}$$
$$gen(\Gamma, \tau) = \forall (fv(\tau) \setminus fv(\Gamma)).\tau$$

Die Unifikation von Typausdrücken, ist die Basis aller Algorithmen zur Typinferenz. Wir werden nun einen Unifikationsalgorithmus angeben, der speziell auf die Typausdrücke von Exp zugeschnitten ist.

$$
\begin{aligned}
&\text{unify}(\alpha, \beta) = [\beta/\alpha] \\
&\text{unify}(\alpha, \tau) = \text{if occur}(\alpha, \tau) \text{ then } \textsf{error} \\
&\qquad\qquad\qquad\ \text{else } [\tau/\alpha] \\
&\text{unify}(\tau, \alpha) = \text{if occur}(\alpha, \tau) \text{ then } \textsf{error} \\
&\qquad\qquad\qquad\ \text{else } [\tau/\alpha] \\
&\text{unify}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') = \text{unify}(\tau_2, \tau_2') \circ \text{unify}(\tau_1, \tau_1')
\end{aligned}
$$

Abbildung 2.7 : Unifikation auf Typausdücken

Der Aufruf unify(τ_1, τ_2) führt eine Unifikation auf τ_1 und τ_2 durch, und liefert eine Substitution θ durch die gilt: $\theta\tau_1 = \theta\tau_2$. Existiert solch eine Substitution nicht, wird der Fehlerwert error zurückgegeben. Die verwendete Hilfsunktion $occur(\alpha, \tau)$ testet, ob α in τ vorkommt.

Die Eigenschaften des allgemeinen Unifikationsalgorithmus von Robinson [Rob65] gelten natürlich auch für den Algorithmus aus Abbildung 2.7.

Satz 2.8 *Der Unifikationsalgorithmus aus Abbildung 2.7 hat folgende Eigenschaften*

(1) unify(τ_1, τ_2) *terminiert immer.*
(2) unify$(\tau_1, \tau_2) = \theta \;\Leftrightarrow\; \theta\tau_1 = \theta\tau_2$
(3) $\theta_1\tau_1 = \theta\tau_2 \;\Rightarrow$ unify $(\tau_1, \tau_2) = \theta$
und es existiert eine Substitution $\theta_2 mit : \theta_1 = \theta_2\theta$

Der Algorithmus zur Typinferenz ist in Abbildung 2.8 angegeben.

$\mathrm{W}(\Gamma, x) = (\mathsf{id}, [\beta_1/\alpha_1] \dots [\beta_n/\alpha_n])$ wobei

$x : \forall\alpha_1 \dots \alpha_n.\tau \;\in\; \Gamma$
$\beta_1, \dots, \beta_n$ sind neue Typvariablen

$\mathrm{W}(\Gamma, e_1 e_2) = (\theta_3\theta_2\theta_1, \theta_3\beta)$ wobei

$\mathrm{W}(\Gamma, e_1) = (\theta_1, \tau_1)$
$\mathrm{W}(\theta_1\Gamma, e_2) = (\theta_2, \tau_2)$
β ist eine neue Typvariable
unify$(\theta_2\tau_1, \tau_2 \rightarrow \beta) = \theta_3$

$\mathrm{W}(\Gamma, \lambda x.e) = (\theta, \theta\beta \rightarrow \tau)$ wobei

β ist eine neue Typvariable
$\mathrm{W}(\Gamma(x : \beta), e) = (\theta, \tau)$

$\mathrm{W}(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) = (\theta_2\theta_1, \tau_2)$ wobei

$\mathrm{W}(\Gamma, e_1) = (\theta_1, \tau_1)$
$fv(\tau_1) \setminus fv(\Gamma) \;=\; \{\alpha_1, \dots, \alpha_n\}$
$\mathrm{W}(\theta_1\Gamma(x : \forall\alpha_1 \dots \alpha_n.\tau_1), e_2) = (\theta_2, \tau_2)$

Falls eine der obigen Bedingungen nicht zutrifft, wird error zurückgegeben.

Abbildung 2.8 : Algorithmus W

Der Aufruf $\mathrm{W}(\Gamma, e)$ führt eine Typinferenz von e in der Typumgebung Γ durch. Falls e in Γ wohlgetypt ist, wird ein Tupel zurück gegeben, welches die komplette Substitution und den Typ von e enthält. Andernfalls wird der Fehlerwert `error` zurückgegeben.

Die Korrektheit und Vollständigkeit von W bezüglich des Typsystems aus Abbildung 2.6, wird in [DM82] gezeigt. Wir werden hier nur die Resultate wiedergeben.

Satz 2.9 (Korrektheit von W) *Falls* $\mathrm{W}(\Gamma, e) = (\theta, \tau)$ *, dann gilt :* $\theta\Gamma \vdash e : \tau$*. Hierbei bezeichnet* $\theta\Gamma$ *die Umgebung, in der auf alle Typen aus* Γ*, die Substitution* θ *angewendet wird.*

Der Algorithmus W hat außerdem die Eigenschaft, den *allgemeinsten* Typ eines Ausdrucks zu liefern. Diese Fähigkeit wird von der Unifikation geerbt, die immer den allgemeinsten Unifikator liefert.

Definition 2.19 (allgemeinster Typ) *Sei* $\Gamma \vdash e : \tau$*. Dann ist* τ allgemeinster Typ *von* e *unter* Γ*, falls gilt:*

$$\textit{für alle } \tau' \textit{ mit: } \Gamma \vdash e : \tau' \textit{ existiert ein } \theta \textit{ mit: } \theta\tau = \theta\tau'$$

Bei der Anwendung der Funktion $\lambda x.x$ auf ganzen Zahlen, hätte diese den Typ $\texttt{Int}\rightarrow\texttt{Int}$. Der allgemeinste Typ von $\lambda x.x$ ist aber $\alpha\rightarrow\alpha$.

Satz 2.10 (Vollständigkeit von W) *Falls* $\Gamma \vdash e : \tau$ *dann gilt:*

1. $\mathrm{W}(\Gamma, e) = (\theta, \tau')$
2. $\tau' \geq \tau$ *ist der allgemeinste Typ von* e *unter* Γ

2.4 Erweiterungen von Exp

Die bisher vorgestellte Sprache Exp stellt nur das Grundgerüst einer Programmiersprache dar. Dies ist für theoretische Betrachtungen sehr vorteilhaft. Zur Kodierung von Programmen ist dies aber ein großer Nachteil. Im Vergleich mit realen Programmiersprachen, macht sich besonders das Fehlen folgender Beschreibungsmöglichkeiten negativ bemerkbar.

- vordefinierte Typen (z.B `Int`) und benutzerdefinierte Typen (z.B. `Tree`)
- vordefinierte Konstanten (z.B. Zahlen) und Funktionen (z.B. `+`)
- Definition rekursiver Funktionen

Wir werden in diesem Abschnitt zeigen, daß einige dieser Beschreibungsmöglichkeiten schon in Exp enthalten sind. Die noch fehlenden Beschreibungsmöglichkeiten werden wir in die Sprache Exp einbauen.

2.4.1 Rekursive Funktionen

In Abschnitt 2.2 (Satz 2.6) haben wir gesehen, daß kein Fixpunktkombinator in Λ_E enthalten ist und somit auch nicht in Exp verfügbar ist. Da wir rekursive Funktionen mittels des Fixpunktkombinator Y simuliert haben, scheint es, als ob die Definition von rekursiven Funktionen in Exp nicht möglich ist. Wenn aber $Y_\alpha : (\forall \alpha . \alpha \rightarrow \alpha) \rightarrow \alpha$ als Konstante definieren wird, umgehen wir die Überprüfung während der Typinferenz. Durch die Hinzunahme von Y_α sind wir zwar in der Lage rekursive Funktionen zu definieren, aber gleichzeitig verlieren wir die Eigenschaft, daß alle wohlgetypten Ausdrücke terminieren. Zur Definition von rekursiven Funktionen führen wir folgendes Konstrukt ein

$$\mathbf{letrec}\ x = e_1\ \mathbf{in}\ e_2$$

Im Unterschied zum **let**-Konstrukt ist x im Gültigkeitsbereich von e_2 *und* e_1. Die Semantik ist definiert durch

$$(\mathbf{letrec}\ x = e_1\ \mathbf{in}\ e_2)\ \equiv\ (\mathbf{let}\ x = Y(\lambda x.e_1)\ \mathbf{in}\ e_2)$$

In Anlehnung an die Typregel für **let**, könnte man folgende Typregel für **letrec** angeben.

$$\boxed{\mathbf{LETREC-P}\quad \frac{\Gamma(x:\sigma) \vdash e_1 : \sigma \quad \Gamma(x:\sigma) \vdash e_2 : \tau}{\Gamma \vdash (\mathbf{letrec}\ x = e_1\ \mathbf{in}\ e_2) : \tau}}$$

Abbildung 2.9 : Typregel für polymorphe rekursive Funktionen

Diese Typregel wurde von Mycroft in [Myc84] eingeführt. In dieser Arbeit erweitert er die Sprache Exp um die Definition von polymorphen, rekursiven Funktionen und zeigt, daß das resultierende Typsystem semantisch korrekt (in Bezug auf Milners Semantik) ist. Desweiteren gibt er einen Algorithmus zur Typinferenz an, der korrekt und vollständigt ist, falls der Algorithmus selbst terminiert. Die Terminierung wird allerdings nicht bewiesen.

Im Jahr 1988 wurde von Henglein [Hen88], und unabhängig davon, auch von Kfoury, Tiuryn und Urzyczyn [KTU88], gezeigt, daß das Problem der Typinferenz in polynomialer Zeit auf die Semi-Unifikation[2] reduzierbar ist. Ein Jahr später wurde die umgekehrte Richtung gezeigt [KTU89]. Ein weiters Jahr später wurde bewiesen, daß die Semi-Unifikation unentscheidbar[3] ist [KTU90]

Daraus ergibt sich, daß durch Einführung einer polymorphen **letrec**-Typregel, die Typinferenz unentscheidbar wird. Dies heißt aber nicht, daß auf die Rekursion verzichtet

[2]Im Unterschied zur Unifikation, muß bei der Semi-Unifikation ein System von Ungleichungen gelöst werden

[3]Es gibt Spezialfälle bei denen die Semi-Unifikation entscheidbar ist [KMNS88]. Wir benötigen aber die allgemeine Semi-Unifikation und diese ist unentscheidbar.

werden muß. Mit einer eingeschränkten **letrec**-Typregel, erhält man die Entscheidbarkeit der Typinferenz zurück.

$$\textbf{LETREC} \quad \frac{\Gamma(x:\tau_1) \vdash e_1 : \tau_1 \quad \Gamma(x:\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash (\textbf{letrec}\ x = e_1\ \textbf{in}\ e_2) : \tau_2}$$

Abbildung 2.10 : monomorphe Typegel für rekursive Funktionen

Die Einschränkung besteht darin, daß die rekursive Funktion nur noch monomorph sein darf, d.h. nur noch einen Typ hat. Dadurch erhalten wir eine entscheidbare Typinferenz, weil die einfache Unifikation ausreicht, um die bei der monomorphen Rekursion entstehenden Typen, zu unifizieren.

Wir werden hier nicht weiter auf die polymorphe Rekursion eingehen. Weitere Information darüber, findet man in [Hen93] [KTU93].

2.4.2 Vordefinierte Konstanten und Funktionen

Alle realen Programmiersprachen bieten vordefinierte Typen, Konstanten und Funktionen. Wir werden nun zeigen, wie man Wahrheitswerte und zugehörige Funktionen, in Exp realisieren kann.

Wahrheitswerte sind vom Typ Bool und werden über die Konstanten true und false erzeugt. Diese beiden Konstanten realisieren wir über folgende Ausdrücke.

$$\begin{array}{lll} \lambda xy.x & =: \mathsf{t} & \text{true} \\ \lambda xy.y & =: \mathsf{f} & \text{false} \end{array}$$

Dadurch lassen sich die üblichen Funktionen auf Wahrheitswerten, wie folgt darstellen.

$$\begin{array}{ll} \lambda xy.xy\mathsf{f} & \text{and} \\ \lambda xy.x\mathsf{t}y & \text{or} \\ \lambda x.x\mathsf{ft} & \text{not} \end{array}$$

Das dies auch wirklich funktioniert, zeigt die folgende Reduktionsfolge.

$$\begin{array}{ll} \text{true and (not false)} & \mathrel{\hat{=}} (\lambda xy.xy\mathsf{f})\mathsf{t}((\lambda x.x\mathsf{ft})\mathsf{f}) \\ & \rightarrow \mathsf{t}((\lambda x.x\mathsf{ft})\mathsf{f})\mathsf{f} \ = \ (\lambda xy.x)((\lambda x.x\mathsf{f})\mathsf{f})\mathsf{f} \\ & \rightarrow (\lambda x.x\mathsf{ft})\mathsf{f} \\ & \rightarrow \mathsf{fft} = (\lambda xy.y)\mathsf{ft} \\ & \rightarrow \mathsf{t} \end{array}$$

Außerdem läßt sich die **if-then-else** Anweisung in Exp realisieren.

$$\textbf{cond} := \lambda zxy.zxy$$

Der Ausdruck **cond** benötigt als 1. Argument die zu überprüfende Bedingung. Die beiden anderen Argumente enthalten den **then** und **else** Teil. Hier ein Beispielreduktion.

$$\begin{array}{rcl}\textbf{cond}\ \text{true}\ e_1\ e_2 & \widehat{=} & (\lambda zxy.zxy)(\lambda xy.x)e_1e_2 \\ & \to & (\lambda xy.(\lambda xy.x)xy)e_1e_2 \\ & \to & (\lambda y.(\lambda xy.x)e_1y)e_2 \\ & \to & (\lambda xy.x)e_1e_2 \\ & \to & (\lambda y.e_1)e_2 \\ & \to & e_1\end{array}$$

Man sieht also, daß die Wahrheitswerte und die zugehörigen Funktionen, schon in Exp enthalten sind. Auf ähnliche Weise lassen sich auch Zahlen oder Tupel darstellen. Wir werden dies hier nicht näher ausführen. Für weitere Informationen, wird auf das Buch von Barendregt [Bar84] verwiesen.

Wollen wir die obigen Ausdrücke bei der Typinferenz verwenden, bekommen wir aber Probleme. Betrachten wir noch einmal die Definiton von true und false, dann sehen wir, daß beide verschiedene Typen haben. Deshalb müssen beide als Konstanten gleichen Typs vordefiniert sein. Dazu muß aber zuerst die Syntax von Exp entsprechend geändert werden.

$$\begin{array}{rcl}e & \to & x \\ & | & c \quad , \ c \in \{\text{true,false,and,or,not}\} \\ & \vdots & \end{array}$$

Außerdem muß die Typumgebung so erweitert werden, daß sie auch Typen von Konstanten enthält. Um die bisherige Schreibweise beizubehalten, gehen wir davon aus, daß es zwischen Variablen und Konstanten keine Nameskonflikte gibt. Eine Basistypumgebung für Wahrheitswerte können wir dann wie folgt notieren.

$$\begin{array}{l}\Gamma_{\mathsf{Bool}} = \{(true : \mathsf{Bool}), (false : \mathsf{Bool}), (and : \mathsf{Bool}\to\mathsf{Bool}\to\mathsf{Bool}), \\ \qquad\quad (or : \mathsf{Bool}\to\mathsf{Bool}\to\mathsf{Bool}), (not : \mathsf{Bool}\to\mathsf{Bool})\}\end{array}$$

Desweiteren müssen noch Reduktionen (δ-Reduktionen) für die Konstanten gegeben sein. Die δ-Reduktionen für not sehen wie folgt aus.

$$\text{not true} \to \text{false} \qquad \text{not false} \to \text{true}$$

Eine Typregel für Konstanten ersparen wir uns, da sie identisch ist, mit der Typregel für Variablen.

Insgesamt ergeben sich bezüglich des Typsystems keine großen Veränderungen. Daher werden wir auch bei weiteren Betrachtungen von Typsystemen, die Konstanten nicht berücksichtigen.

2.4.3 Typsignaturen

Mit Typsignaturen werden Ausdrücke explizit getypt, d.h. der Typ eines Ausdrucks wird vorgegeben. Dies scheint hier ein wenig deplaziert zu sein, da es unser Ziel ist, den Typ eines Ausdrucks ohne explizite Typisierung herzuleiten. Es gilt aber zu beachten, daß die explizite Typisierung bei der Programmierung immer noch verwendet wird, weil dadurch die Lesbarkeit von Programmen verbessert wird. Außerdem kann mittels der Typsignatur erzwungen werden, daß ein Ausdruck einen spezielleren Typ erhält, als er eigentlich hat.

Eine Typsignatur stellen wir durch folgenden Ausdruck dar.

$$\textbf{sig}\ \tau\ e$$

Hierbei ist τ die explizite Typangabe für den Ausdruck e. Die Typregel für die Typsignatur sieht wie folgt aus.

$$\textbf{sig}\quad \frac{\Gamma \vdash e : \tau' \qquad \tau' \geq \tau}{\Gamma \vdash \textbf{sig}\ \tau\ e\ : \tau}$$

Abbildung 2.11 : Typregel einer Typsignatur

Wie schon oben erwähnt, wird nicht gefordert, daß der angegebene Typ τ dem tatsächlichen Typ τ' entspricht. Vielmehr muß der angegebene Typ nur eine Instanz ($\tau' \geq \tau$) des tatsächlichen Typs von e sein. Dadurch ist man in der Lage Ausdrücke auf spezielle Typen einzuschränken.

$$\textbf{sig}\ (\mathsf{Bool}{\rightarrow}\mathsf{Bool})\ \lambda x.x$$

Im obigen Beispiel wurde die Identitätsfunktion ($\lambda x.x$) so eingeschränkt, daß sie nur auf Ausdrücken vom Typ Bool arbeitet. Eine Vorgabe vom Typ $\mathsf{Bool}{\rightarrow}\alpha$ wird zurückgewiesen, weil $\lambda x.x$ den Typ $\alpha{\rightarrow}\alpha$ hat und $\mathsf{Bool}{\rightarrow}\alpha$ keine Instanz von $\alpha{\rightarrow}\alpha$ ist.

2.4.4 Strukturierte Datentypen

Die Erzeugung von neuen Datentypen ist von fundamentaler Bedeutung für Programmiersprachen. Dies geschieht bei funktionalen Programmiersprachen durch die Definiton von *strukturierten* (algebraischen) *Datentypen*. Die allgemeine Form zur Definition von strukturierten Datentypen ist

$$\begin{array}{rcl} Tycon & = & C_1\ \tau_{1,1}\ \ldots\ \tau_{1,t_1} \\ & | & C_2\ \tau_{1,2}\ \ldots\ \tau_{1,t_2} \\ & & \vdots \\ & | & C_n\ \tau_{1,n}\ \ldots\ \tau_{1,t_n} \end{array}$$

Der *Typkonstruktor* $Tycon$ definiert einen neuen Datentyp[4] gleichen Names. Dieser enthält die *Datenkonstruktoren* C_1 bis C_n, wobei ein Datenkonstruktor C_i die Stelligkeit t_i hat. Ein Datenkonstruktor C_i erzeugt mit t_i Argumenten der Typen $\tau_{i,1}$ bis τ_{i,t_i} einen neues Datum vom Typ $Tycon$. Die Definition eines Datentyps für Listen, sieht in Haskell wie folgt aus

```
data List a = Nil
            | Cons a (List a)
```

Der Typkonstruktor `List` hat ein Argument `a`, welches den Typ der Listenelemente angibt, d.h. `List` ist ein polymorpher Datentyp. Das Ende einer Liste wird durch den Datenkonstruktor `Nil` signalisiert. Das Anfügen eines Elements an eine Liste, wird durch `Cons` realisiert. Wie man sieht können die Datenkonstruktoren eine unterschiedliche Stelligkeit besitzen. Außerdem kann eine neuer Typkonstruktor in seiner eigenen Definition benutzt werden. Dadurch können rekursive Datentypen erzeugt werden.

Die Möglichkeit strukturierte Datentypen zu definieren, werden wir nun in unsere bisherige Sprache einbauen. Dabei werden wir nicht den syntaktischen Komfort wie in Haskell haben, sondern uns auf das wesentliche beschränken. Betrachten wir zuerst, wie die neuen Typen aussehen.

Mono Typ	τ	$\rightarrow$	α	Typvariable
		$\mid$	$\tau_1 \times \tau_2$	Tupeltyp
		$\mid$	$\tau_1 \rightarrow \tau_2$	Funktionstyp
		$\mid$	Bool , unit	Basistypen
		$\mid$	χ	Rekursive Typen
Rec Typ	χ	$\rightarrow$	$\mu\beta.K_1\tau_1 + \ldots + K_m\tau_m$	$K_i \neq K_j$ für $i \neq j$
Poly Typ	σ	$\rightarrow$	τ	monomorher Typ
		$\mid$	$\forall\ \alpha_1\ \ldots\ \alpha_n.\ \sigma$	Typschema

Abbildung 2.12 : Syntax der strukturierten Datentypen

Als vordefinierte Basistypen sind unit und Bool hinzugekommen. Der Einheitstyp unit wird bei der Deklaration von strukturierten Datentypen benötigt und hat als einzigen Wert (). Aus Gründen der Übersichtlichkeit legen wir fest, daß alle Datenkonstruktoren eines Typkonstruktors die gleiche Stelligkeit haben. Daher dient unit zum Auffüllen von Datenkonstruktoren, die eine zu geringe Stelligkeit haben. Der neue Tupeltyp wird benötigt, falls ein Datenkonstruktor mehr als 1 Argument hat. Der Datentyp Bool muß vordefiniert sein, weil wir unserer neune Sprache ein Konstrukt enthält, daß einen Wert vom Typ Bool liefert.

[4]Im Kontext von funktionalen Programmiersprachen, werden diese als Typkonstruktoren bezeichnet. Wir werden beide Bezeichnungen verwenden.

Die strukturierten Datentypen werden als *Summentypen* dargestellt, wobei die K_i die einzelnen Datenkonstruktoren sind. Intuitiv betrachtet, sind Summentypen die Vereinigung aller Teiltypen. Rekursive Typen werden mittels μ eingeführt. Die Semantik dieser Typen kann nun nicht mehr über einfache Mengen definiert werden, da ein rekursiver Typ die Form $\tau = \tau \rightarrow \tau'$ hat. Diese Mengen müssen *schwache Ideale* sein. Durch eine metrische Struktur auf diesen schwachen Idealen, kann mittels des Fixpunktsatzes von Banach, die eindeutige Existenz der obigen Gleichung gesichert werden. Diese umfangreiche Vorgehensweise wird hier nicht näher ausgeführt. Sie ist in [MPS86] ausführlich beschrieben.

Die Typumgebung Γ muß nun so erweitert werden, daß sie auch die Typen der einzelnen Datenkonstruktoren enthält. Um die übliche Darstellung beizubehalten, nehmen wir an, daß Variablen und Datenkonstruktoren aus verschiedenen Namensbereichen kommen und so keine Überschneidungen möglich sind. Der Typ eines Datenkonstruktors, läßt sich dann durch $\Gamma(K)$ ermitteln. Außerdem benötigen wir eine Funktion, die den Typ des Arguments eines Datenkonstruktors K, innerhalb des zugehörigen Typkonstruktors verändert. $\Sigma[K \tau]$ liefert den Summentyp $K_1\tau_1 + \ldots + K_n\tau_n$, wobei $K = K_i$ und $\tau_i = \tau$.

Die Syntax der neuen Sprache ExpStruct, sieht wie folgt aus

e	$\rightarrow$	x	Variable
	$\mid$	$e_1 e_2$	Applikation
	$\mid$	$\lambda x.e$	Abstraktion
	$\mid$	**let** $x = e_1$ **in** e_2	lokale Funktionen
	$\mid$	**letrec** $x = e_1$ **in** e_2	rekursive Funktionen
	$\mid$	**data** $\forall \alpha_1 \ldots \alpha_n.\chi$ **in** e	Deklaration strukturierter Datentypen
	$\mid$	K	Datenkonstruktor
	$\mid$	**is** K	Test auf Datenkonstruktor
	$\mid$	**let** $K\ x = e_1$ **in** e_2	pattern matching

Abbildung 2.13 : Syntax von ExpStruct

Eine **data** Deklaration definiert einen neue Datentyp. Im Gegensatz zur Syntax von Haskell, kann kein Name für den neuen Datentyp angegeben werden und der Typ muß explizit als Summentyp angegeben werden. Werte dieses neuen Typs, werden durch die Anwendung einer seiner Datenkonstruktoren K, auf die entsprechenden Argumente erzeugt. Über den **is**-Ausdruck kann zwischen den Datenkonstuktoren eines Typkonstruktors unterschieden werden. Mittels des neuen *pattern matching* **let**, kann auf die einzelnen Argumente eines Datenkonstruktors zugegriffen werden.

Wir geben nun einen Ausdruck in ExpStruct an, der die Länge einer Liste von Zahlen berechnet. Um diesen Ausdruck möglichst übersichtlich zu halten, nehmen wir an, daß Zahlen schon definiert sind und die Funktionen *if*, *plus* und *snd* zu Verfügung stehen.

$$
\begin{array}{l}
\textbf{data } \forall\alpha.(\mu\beta.\mathrm{Nil}\ \mathsf{unit} + \mathrm{Cons}\ \ \alpha\times\beta)\ \textbf{in} \\
\quad \textbf{letrec } \mathit{length} = \lambda xs.\ \mathit{if}\ (\textbf{is}\ \mathrm{Nil}\ \ xs) \\
\qquad\qquad\qquad\qquad\qquad 0 \\
\qquad\qquad\qquad\qquad\qquad (\textbf{let}\ \mathrm{Cons}\ ab = xs\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{plus}\ (\mathit{length}(\mathit{snd}\ ab))1) \\
\quad \textbf{in } \mathit{length}(\mathrm{Cons}(3,\mathrm{Cons}(7,\mathrm{Nil}\ ())))
\end{array}
$$

Die erste Zeile definiert eine polymorphe Liste, wobei α den Typ der Elemente repräsentiert, β den rekursiven Typ angibt und unit dafür sorgt, daß Nil und Cons die gleiche Stelligkeit haben. Das Ende einer Liste, wird in Zeile 2, durch den Test auf den Datenkonstruktor Nil, abgefragt. In Zeile 4 erfolgt das pattern matching, wodurch in Zeile 5 der rekursive Aufruf von *lenght* erfolgen kann.

Wir werden nun die Typregeln für ExpStruct angeben. Dabei beschränken wir uns auf die Typregeln der neuen Konstrukte.

$$
\textbf{DATA}\quad \frac{\sigma = \forall\alpha_1\ldots\alpha_n.\mu\beta.K_1\tau_1 + \ldots + K_m\tau_m \qquad fv(\sigma)=\emptyset \quad \Gamma(K_1:\sigma,\ldots,K_m:\sigma)\vdash e:\tau}{\Gamma\vdash \textbf{data}\ \sigma\ \textbf{in}\ e:\tau}
$$

$$
\textbf{CONS}\quad \frac{\Gamma(K)\geq \mu\beta.\Sigma[K\tau]}{\Gamma\vdash K:\tau\rightarrow\mu\beta.\Sigma[K\tau]}
$$

$$
\textbf{TEST}\quad \frac{\Gamma(K)\geq \mu\beta.\Sigma[K\tau]}{\Gamma\vdash \textbf{is}\ K:(\mu\beta.\Sigma[K\tau])\rightarrow\mathsf{Bool}}
$$

$$
\textbf{PAT}\quad \frac{\Gamma\vdash e_1:\mu\beta.\Sigma[K\tau] \qquad \Gamma(x:gen(\Gamma,[\mu\beta.\Sigma[K\tau]/\beta]\tau))\vdash e_2:\tau_2}{\Gamma\vdash \textbf{let}\ Kx=e_1\ \textbf{in}\ e_2:\tau_2}
$$

Abbildung 2.14 : Typregeln für strukturierte Datentypen

Die Typregel **DATA** überprüft die Deklaration eines rekursiven Datentyps, indem getestet wird ob der angegebene Typ keine freien Variablen enthält. Dann wird die Typumgebung um die Typinformationen zu den Datenkonstruktoren K_i, erweitert. In dieser Typumgebung wird der Typ von e hergeleitet.

Der Ausdruck K wird mit der Typregel **CONS** überprüft. Dabei wird der Typ des Datenkonstruktors K aus der Typumgebung geholt und ein entsprechender Funktionstyp zurückgegeben. Diese Regel ist sehr ähnlich zur Regel für Variablen, da auch hier nur Informationen zurückgegeben werden. Der einzige Unterschied besteht darin, daß Datenkonstruktoren Funktionen sind und daher einen Funktionstyp liefern.

Die Typregel **TEST** stellt sicher, daß **is** K nur auf solche Ausdrücke angewendet wird, dessen Typ dem Typ des Datenkonstruktors K entspricht. Da **is** eine Überprüfung vornimmt, ist der zurückgegebene Wert vom Typ Bool.

Das pattern matching wird über die Typregel **PAT** überprüft und ist sehr ähnlich zur Typregel von **let**. Zuerst wird getestet ob e_1 dem Ergebnistyp von K entspricht. Danach kann e_2 in einer erweiterten Typumgebung überprüft werden. Diese Erweiterung besteht darin, x den verallgemeinerten Typ von e_1 zuzuweisen.

Weitere Information über die Typisierung von strukturierten Datentypen, insbesondere auch abstrakte Datentypen, findet man in [Läu93].

2.4.5 Einfache Definition von Funktionen

Moderne funktionale Programmiersprachen bieten die Möglichkeit Funktionen an beliebigen Stellen im Programmtext zu definieren. So kann ein Haskell Programm wie folgt aussehen.

```
main = id True
succ y = id y + 1
id x = x
```

Dabei ist die Funktion `main`, ähnlich wie in C, die Startfunktion und wird somit ausgewertet. Ein erster Ansatz zur Umsetzung in Exp, wäre die Benutzung der **let** Umgebung. Um diese verwenden zu können, muß sie erst um die Definition von n lokalen Funktionen erweitert werden.

$$\begin{array}{lrcl} \textbf{let} & x_1 & = & e_1 \\ & & \vdots & \\ & x_n & = & e_n \\ \textbf{in} & e & & \end{array}$$

Die zugehörige Typregel wird in ähnlicher Weise erweitert. Dabei ist allerdings zu beachten, daß die Typen der schon definierten Funktionen, den nachfolgenden Funktionsdefinitionen zu Verfügung gestellt werden.

$$\textbf{LET}\!-\!\mathrm{n} \quad \frac{\begin{array}{c}\Gamma(x_1 : \tau_1, \ldots, x_{i-1} : \tau_{i-1}) \vdash e_i : \tau_i \quad 1 \le i \le n \\ \Gamma(x_1 : gen(\Gamma, \tau_1), \ldots, x_n : gen(\Gamma, \tau_n)) \vdash e : \tau\end{array}}{\begin{array}{lllcl} \Gamma \vdash & \textbf{let} & x_1 & = & e_1 \\ & & & \vdots & \\ & & x_n & = & e_n \\ & \textbf{in} & e & & \end{array}}$$

Abbildung 2.15 : Typregel für n-stelliges **let**

Damit können wir die Umsetzung beginnen. Hierbei nehmen wir an, daß Zahlen und Wahrheitswerte schon definiert sind und die Funktion *add*, die Addtion realisiert.

let $id \;= \lambda x.x$
    $succ \;= \lambda y.plus(id\ y)1$
in id true

Diese naive Umsetzung hat aber eine entscheidenen Nachteil. Das entstandene Exp Programm enthält eine Typfehler. Dies liegt daran, daß die definierten Funktionen innerhalb des Definitionsblocks nur monomorph benutzt werden dürfen. In unserem Beispiel wird die Funktion id, in der Definition von $succ$, auf den Typ Num→Num eingeschränkt[5]. Die Anwendung auf den Wahrheitswert true liefert deswegen eine Typfehler.

Eine Lösung für diese Problem ist die Verwendung von geschachtelten Funktionsdefinitionen. Dabei werden die einzelnen Funktionsdefinition so geschachtelt, daß vor der Verwendung einer Funktion ihre Definiton steht. Für unser Beispiel sieht dies wie folgt aus.

let $id \;= \lambda x.x$
in **let** $succ \;= \lambda y.plus(id\ y)1$
    in id true

Dieses Vorgehen kann analog auf die Definition von rekursiven Funktionen angewendet werden.

Die Reihenfolge der Schachtelungen läßt sich mit einer *Abhängigkeitsanalyse* ermitteln. Die Abhängigkeitsanalyse erzeugt aus einer Menge F von Funktionen, im folgenden *Bindungen* genannt, eine Liste von *Bindungsgruppen* $[bg_1, \ldots, bg_n]$. Für diese Liste gilt:

- $bg_1, \ldots, bg_n$ enthält genau die selben Bindungen wie F.
- Die Bindungen in jeder Bindungsgruppe bg_i sind *verschränkt rekursive*, d.h. sie rufen sich gegenseitig auf.
- Jede Funktion, die in bg_i angewendet wird, ist in bg_j definiert und $j \leq i$.

Die Implementierung einer Abhängigkeitsanalyse wird in Abschnitt 5.5 beschrieben.

---

[5] Num repräsentiert dabei den Typ von Zahlen

Kapitel 3

Dimensionstypen

In diesem Kapitel werden wir untersuchen, wie die Typen erweitert werden müssen, um die Probleme aus Abschnitt 1.1 zu lösen. Ausgangspunkt wird dabei das Rechnen mit physikalischen Größen sein.

3.1 Physikalische Größen

In der Physik wird nicht mit reinen Zahlenwerten, sondern mit *physikalischen Größen* gerechnet. Jede physikalische Größe G, ist das Produkt aus einem *Zahlenwert* $\{\mathrm{G}\}$ und einer *physikalischen Einheit* [G].

$$\mathrm{G} = \{\mathrm{G}\}\ [\mathrm{G}]$$

Beispiele von physikalischen Größen sind

220 m	Länge von 220 Meter
220 N	Kraft von 220 Newton
220 V	elektrische Spannung von 220 Volt

Bei diesen Beispielen sind zwar alle Zahlenwerte gleich, aber durch die zusätzliche physikalische Einheit, werden sie unterscheidbar. Jede physikalische Einheit läßt sich über die 7 Basiseinheiten des Internationalen Einheitensystems bilden.

Einheitenname	Einheitenzeichen	Bedeutung
Meter	m	Länge
Sekunde	s	Zeit
Kilogramm	kg	Masse
Ampere	A	Stromstärke
Kelvin	K	Temperatur
Mol	mol	Stoffmenge
Candela	cd	Lichtstärke

Eine physikalische Einheit hat daher die allgemeine Form

$$\mathrm{m}^i\mathrm{s}^j\mathrm{kg}^k\mathrm{A}^l\mathrm{K}^m\mathrm{mol}^n\mathrm{cd}^o \quad i,j,k,l,m,n,o \in \mathbb{Z}$$

Aus der Kombination von Basiseinheiten lassen sich *abgeleitete Einheit* bilden. Die Einheit für die Geschwindigkeit ist m s^{-1} und für die Leistung kg $\mathrm{s}^{-3}\mathrm{m}^2$. Jede physikalische Einheit läßt sich eindeutig[1] als 7-Tupel darstellen, wobei die Elemente, den Exponenten der Basiseinheiten entsprechen.

$$(i,j,k,l,m,n,o) \quad i,j,k,l,m,n,o \in \mathbb{Z}$$

Daher repräsentiert das 7-Tupel $(2,-3,1,0,0,0,0)$, die Einheit der Leistung.

Physikalische Einheiten geben den Zahlenwerten aber nicht nur Bedeutungen, sondern verändern diese Zahlenwerte auch, indem vor die Einheitennamen Vorsätze und vor die Einheitenzeichen Vorsatzzeichen plaziert werden. Hier einige bekannte Vorsätze.

Vorsatz	Vorsatzzeichen	Bedeutung
Giga	G	10^9
Mega	M	10^6
Kilo	k	10^3
Milli	m	10^{-3}
Mikro	μ	10^{-6}
Nano	n	10^{-9}

Dadurch lassen sich physikalische Größen auf verschiedene Arten darstellen.

$$100000\mathrm{mm} = 100\mathrm{m} = 0,1\mathrm{km}$$

Vorsätze dienen lediglich zur kompakteren Darstellung von physikalischen Größen, deren Zahlenwerte sehr groß oder sehr klein sind. Um physikalischen Größen eindeutig darstellen zu können, benutzt man die *Dimensionen*. Eine Dimension gibt den Zusammenhang einer physikalischen Größe mit den Basiseinheiten an.

Basisgröße	Dimensionszeichen
Länge	L
Zeit	Z
Masse	M
Stromstärke	I
Temperatur	T
Stoffmenge	N
Lichtstärke	J

[1] Eindeutigkeit bezüglich eines normalisierten 7-Tupel , d.h. ggT(i,j,k,l,m,n,o)=1.

Der Begriff der Dimension ist von dem der Einheit zu unterscheiden. So hat die Geschwindigkeit die Dimension $\frac{\text{Länge}}{\text{Zeit}}$ und die Einheit $\frac{\text{Meter}}{\text{Sekunde}}$.

Beispiele für Dimensionen sind:

Größenart	Dimension
Volumen	L^3
Beschleunnigung	$\mathsf{L}\ \mathsf{Z}^{-2}$
Elektrische Spannung	$\mathsf{L}^2\mathsf{Z}^{-3}\mathrm{M}\ \mathrm{I}^{-1}$
Entropie	$\mathsf{L}^2\mathsf{Z}^{-2}\mathsf{M}^1\mathsf{T}^{-1}$

Dimensionen lasen sich, ähnlich wie die Einheiten, durch ein 7-Tupel eindeutig darstellen.

$$(i, j, k, l, m, n, o) \quad i, j, k, l, m, n, o \in \mathbb{Z}$$

Die Kombination mit einem Zahlenwert ist dann ebenfalls eindeutig, da keine Vorsätze erlaubt sind.

Betrachten wir nun, wie die Typen von Addition und Multiplikation auf physikalischen Größen aussehen.

$$\begin{aligned} \mathrm{G}_1 +_p \mathrm{G}_2 &:= (\{\mathrm{G}_1\} + \{\mathrm{G}_2\})\ [\mathrm{G}_1] \quad \text{,falls } [\mathrm{G}_1] = [\mathrm{G}_2] \\ \mathrm{G}_1 *_p \mathrm{G}_2 &:= (\{\mathrm{G}_1\} * \{\mathrm{G}_2\})\ ([\mathrm{G}_1] + [\mathrm{G}_2]) \end{aligned}$$

Man erkennt, daß Zahlenwerte und Dimensionen unterschiedlich behandelt werden. Die Addition von physikalischen Größen ist nur dann zulässig, wenn beide die gleiche Dimension aufweisen. Bei der Multiplikation hingegen, werden die beiden Dimensionen kombiniert. Das + steht dabei für eine Vektoraddition, da wir Dimensionen als 7-Tupel betrachten.

3.2 Typen mit Dimensionen

Die Darstellung von Dimensionen physikalischer Größen als Tupel, ist sicherlich sehr übersichtlich, hat aber einige entscheidende Nachteile. Zum einen werden in der Regel nicht sehr viele Basisdimensionen miteinander verknüpft. So gibt es in der Physik nur eine Größe, die universelle Gaßkonstante, dessen Dimension aus 5 Basisdimensionen gebildet wird. Die Dimensionen aller anderen Größen, sind aus weit weniger Basisdimensionen zusammengestzt. Für die Darstellung als Tupel bedeutet dies, daß die meisten Komponenten 0 sind.

Ein weiterer Punkt ist die Benutzung von Dimensionsvariablen. Ähnlich wie bei den gewöhnlichen Typen, können auch Dimensionen variabel sein (z.B. bei der Addition). Dadurch müßte jede Komponente des Tupels mit einer Variable belegt sein.

Je mehr Basisdimensionen vorhanden sind, desto ineffizienter ist die Darstellung von Dimensionen als Tupel.

Daher werden wir die Dimensionen direkter darstellen, wie die folgende Abbildung zeigt.

$$
\begin{array}{llcll}
\text{Dimension} & \delta & \rightarrow & d & \text{Variable} \\
 & & \mid & D & \text{Basisdimension} \\
 & & \mid & \delta_1\delta_2 & \text{Kombination} \\
 & & \mid & \delta^{-1} & \text{reziproke Dimension} \\
 & & \mid & \mathbf{0} & \text{keine Dimension}
\end{array}
$$

Abbildung 3.1 : Syntax von Dimensionen

Die Darstellung erfolgt jetzt analog zur Darstellung von physikalischen Einheiten. Wichtig ist dabei, daß mittels **0** auch ausgedrückt werden kann, daß keine Dimension vorhanden ist.

Jetzt stellt sich die Frage, wie diese Dimensionen mit den bisherigen Typen kombiniert werden, damit Dimensionstypen entstehen. Ein erster Ansatz wäre, Dimensionstypen als Paare zu betrachten, die aus Typen und Dimensionen bestehen.

$$\tau_{dim} = (\tau, \delta)$$

Diese sehr naheliegende Darstellung hat aber den Nachteil, daß Dimensionstypen entstehen können, die keine konkrete Bedeutung haben.

$$\tau_{dim} = ((\alpha_1, \delta_1) \rightarrow (\alpha_2, \delta_2), \delta)$$

Hierbei haben zwar beide Teile des Funktionstyps eine Dimension, aber die Funktion auch. Die Tatsache, daß auch Funktionen Dimensionen haben, ist doch recht merkwürdig und entspricht nicht der eigentlichen Aufgabe einer Dimension. Betrachten wir noch einmal am Beispiel der Physik, wir dort das Dimensionen in Funktionen verwendet werden. Den Typ einer Multiplikation würde man wie folgt darstellen[2].

$$(\mathsf{Num}, \delta_1) \rightarrow (\mathsf{Num}, \delta_2) \rightarrow (\mathsf{Num}, \delta_1\delta_2)$$

Auch hier sind die einzelnen Teiltypen als Paare dargestellt. Das entscheidende ist aber, daß die Funktion keine Dimension hat. Es wird nur der Typ eines konkreten Werts (der

[2] Num repräsentiert dabei den Typ von Zahlen

Zahl) um eine Dimension erweitert. Deshalb werden wir nur spezielle Typen um Dimensionen erweitern. Diese werden die 0-stelligen Typkonstruktoren sein, da sie die Typen von atomaren Werten repräsentieren. Die Dimensionstypen sehen daher wie folgt aus.

Dimensiontypen	τ	$\rightarrow$	α	Variable
		$\mid$	$\tau_1 \rightarrow \tau_2$	Funktionstyp
		$\mid$	$\chi_0 \delta$	Typkonstruktor mit Dimension
		$\mid$	$\chi\ \tau_1 \ldots \tau_n$	Typkonstruktor , $n \geq 1$
	σ	$\rightarrow$	τ	monomorpher Typ
		$\mid$	$\forall \alpha_1 \ldots \alpha_n d_1 \ldots d_m . \tau$	polymorpher Typ , $n, m \geq 0$

Abbildung 3.2 : Syntax von Dimensionentypen

Die Typen von arithmetische Funktionen, lassen sich dann wie folgt darstellen.

$$\begin{array}{rl}
+ & : \mathsf{Num}\ d \ \rightarrow \mathsf{Num}\ d \ \rightarrow \mathsf{Num}\ d \\
* & : \mathsf{Num}\ d_1 \rightarrow \mathsf{Num}\ d_2 \rightarrow \mathsf{Num}\ d_1 d_2 \\
/ & : \mathsf{Num}\ d_1 \rightarrow \mathsf{Num}\ d_2 \rightarrow \mathsf{Num}\ d_1 d_2^{-1} \\
\mathrm{sqrt} & : \mathsf{Num}\ d^2 \rightarrow \mathsf{Num}\ d \\
\sin & : \mathsf{Num}\ \mathbf{0} \ \rightarrow \mathsf{Num}\ \mathbf{0}
\end{array}$$

Dadurch, daß wir nur die Darstellung der Typen geändert haben, sind keine großen Veränderungen an den Typregeln erforderlich. Lediglich die Typregel zur Deklaration von rekursiven Datenstrukturen müsste um Dimensionvariablen erweitert werden. Eine große Veränderung muß aber am Algorithmus zur Typinferenz vorgenommen werden. Dies betrifft ausschießlich den Bereich der Unifikation, wo der folgende Fall eingebaut werden muß.

$$\mathrm{unify}(\chi_0 \delta_1, \chi_0 \delta_2) = \mathrm{dimunify}\ (\delta_1, \delta_2)$$

3.3 Unifikation von Dimensionen

Die Unifikation von Dimensionen ist nicht so einfach wie die Unifikation von Typen. Bei den Typen mußte nur die syntaktische Struktur beachtet werden, daher reichte eine einfache *syntaktische Unifikation* aus. Dagegen müßen bei den Dimensionen, die folgenden Gleichungen berücksichtigt werden.

$$\begin{array}{rcll}
\delta_1 \delta_2 & =_D & \delta_2 \delta_1 & \text{Kommutativität} \\
(\delta_1 \delta_2) \delta_3 & =_D & \delta_1 (\delta_2 \delta_3) & \text{Assoziativität} \\
\mathbf{0} \delta & =_D & \delta & \text{Identität} \\
\delta \delta^{-1} & =_D & \mathbf{0} & \text{Inverses}
\end{array}$$

Betrachtet man diese Gleichungen genauer, dann erkennt man, daß die Dimensionen eine abelsche Gruppe bilden. Glücklicherweise ist die Unifikation von Termen aus einer abelschen Gruppe entscheidbar und unitär[3][BD92]. Ein entsprechender Algorithmus wurde 1984 von Lankford, Butler und Brady [LBB84] angegeben. Dieser Unifikationsalgorithmus beruht auf der Lösung von linearen Gleichungen über ganzen Zahlen. Zur deren Lösung verwenden wir den Algorithmus von Knuth [Knu69], wobei wir nur eine einzige Gleichung betrachten werden. Der Algorithmus für dimunify(δ_1, δ_2) sieht wie folgt aus.

1. Transformiere die Gleichung $\delta_1 \stackrel{?}{=}_D \delta_2$ in die folgende normalisierte Form

$$d_1^{x_1} \dots d_n^{x_n} \ D_1^{y_1} \dots D_m^{y_m} \stackrel{?}{=}_D \mathbf{0}$$

   dabei gilt: $d_i \neq d_j$, $D_i \neq D_j$ für $i \neq j$.

2. Initialisiere die Substitution, d.h. $\theta = \mathsf{id}$. Falls $n = 0$ und $m = 0$ sind wir fertig. Falls $n = 0$ und $m \neq 0$, dann existiert kein Unifikator.

3. Suche die Variable, dessen Exponent x_k den kleinsten absoluten Wert hat. Ist x_k negativ, dann negiere alle anderen Exponenten. Ohne Beschränkung der Allgemeinheit nehmen wir an, daß $k = 1$.

4. Falls $\forall i, j \ : \ x_i \bmod x_1 = 0$, $y_j \bmod x_1 = 0$, dann sind wir fertig und der Unifikator ist

$$[d_2^{-x_2/x_1} \dots d_n^{-x_n/x_1} \ D_1^{-y_1/x_1} \dots D_m^{-y_m/x_1} \ / \ d_1]\theta$$

5. Ansonsten erzeuge eine neue Variable d und setze die Substitution auf

$$[d \ d_2^{\lfloor -x_2/x_1 \rfloor} \dots d_n^{\lfloor -x_n/x_1 \rfloor} \ D_1^{\lfloor -y_1/x_1 \rfloor} \dots D_m^{\lfloor -y_m/x_1 \rfloor} \ / \ d_1]\theta$$

   und ändere die Gleichung auf

$$d^{x_1} d_2^{x_2 \bmod x_1} \dots d_n^{x_n \bmod x_1} \ D_1^{y_1 \bmod x_1} \dots D_m^{y_m \bmod x_1} \stackrel{?}{=}_D \mathbf{0}$$

6. Steht auf der linken Seite nur noch die Variablen d, d.h. $x_i \bmod x_1 = 0$, $2 \leq i \leq n$, dann gibt es keine Lösung. Anderenfalls wird bei 3. eine neue Iteration begonnen.

Dieses Verfahren terminiert, weil bei jeder Iteration eine Variable verschwindet.

[3] es gibt eine eindeutigt bestimmten allgemeinsten Unifikator

3.4 Einführung von Dimensionsgrenzen

Wir wollen nun untersuchen, inwieweit die Dimensionstypen, die Probleme, die in Abschnitt 1.1 angesprochen wurden, lösen können.

Die Typen der beiden Variablen `preis` und `geschwindigkeit`, sehen wie folgt aus, wobei wir voraussetzen, daß die Basisdimensionen DM und KMH schon vorhanden sind.

`preis` : Num DM
`geschwindigkeit` : Num KHM

In Kombination mit den Typen von `+` und `sin` (s. Seite 39), können die folgenden Ausdrücke als fehlerhaft erkannt werden.

```
preis + geschwindigkeit
sin(geschwindigkeit)
```

Der dritte Ausdruck der in Abschnitt 1.3 1 angegeben wurde, `preis * preis`, wird aber nicht als fehlerhaft erkannt. Er erhält gemäß des Typs von `*`, den Typ Num DM^2.

Um dies zu vermeiden, benötigen wir eine Möglichkeit, der Typinferenz zu sagen, daß DM^2 unzulässig ist. Etwas allgemeiner ausgedrückt bedeutet dies, daß wir für jede Basisdimension vorgeben, in welchem Bereich sich der zugehörige Exponent bewegen darf. Für die Basisdimension DM wäre dieser Bereich [-1,1], d.h. nur DM^{-1} und DM sind zugelassen.

Für die Typinferenz müßte eine Umgebung bereitgestellt werden, die jeder Basisdimension, ihre jeweiligen Dimensiongrenzen zuordnet. Während der Typinferenz kann dann überprüft werden, ob diese Grenzen überschritten werden.

Die genaue Realisierung dieser Dimensionsgrenzen, sowie die Deklaration von Basisdimensionen, werden im nächsten Kapitel angegeben.

Kapitel 4

Die Sprache ExtCore

In diesem Kapitel wird eine einfache funktionale Sprache entwickelt, deren Typsystem auf den Dimensionstypen aus Kapitel 3 arbeitet. Um den Aufwand in vertretbaren Grenzen zu halten werden wir einigen syntaktischen Zucker den moderne funktionale Sprachen bieten weglassen.

Die Basis für ExtCore ist die Sprache Core, die in [PJL92] eingeführte wurde. Diese einfache funktionale Sprache werden wir, außer der Einführung eines Typsystems, um folgende Eigenschaften erweitern.

- Definition strukturierter Typen
- Definition von Basisdimensionen
- Angabe von Typsignaturen für Top-level Funktionen
- Direkte Verwendung von Datenkonstruktoren

Außerdem wird die Syntax soweit als möglich an Haskell angepaßt.

Zunächst werden wir die konkrete Syntax von ExtCore also die Syntax aus der Sicht des Programmierers definieren. Um die Typregeln übersichtlich darstellen zu können, werden wir diese konkrete Syntax so weit als möglich vereinfachen. Die daraus entstehende abstrakte Syntax bildet dann die Grundlage, auf der wir die Typregeln definieren.

4.1 Die konkrete Syntax

Zuerst legen wir die lexikalische Syntax, d.h. die Menge der gültigen Zeichen fest. Diese Definition werden wir im nächsten Kapitel bei der Implementierung der lexikalischen Analyse benötigen.

```
program       → {lexeme | whitespace}
lexeme        → varid | conid | literal | resresid | resop | symbol
literal       → integer | float
whitespace    → whitestuff { whitestuff }
whitestuff    → whitechar | whitecommand
whitechar     → newline | formfeed | space
newline       → ascii code 10
formfeed      → ascii code 13
space         → ascii code 32
whitecommand  → -- { lexeme }
conid         → ucase { ucase | lcase | digit }
varid         → lcase { ucase | lcase | digit }
resid         → case | data | dims | in
              | let | letrec | of
              | not | sin | sqrt
              | Num | Bool | True | False
float         → integer . number [E [+|-]number]
integer       → [+|-] number
number        → digit { digit }
ucase         → A | ... | Z
lcase         → a | ... | z
digit         → 0 | ... | 9
resop         → * | + | - | / | < | = | > | &
symbol        → ( | ) | , | . | : | ; | \ | ^ | { | | | }
```

Abbildung 4.1 : Lexikalische Syntax von ExtCore (1)

Wie man schon an der lexikalischen Syntax sieht, ist die Sprache sehr einfach gehalten. Außer Zahlen und Wahrheitswerten sind keine weiteren Datentypen vordefiniert. Es sind gibt insgesamt 7 Schlüsselwörter und 3 vordefinierte Funktionen. Der Typ von Zahlen wird mit `Num` notiert und `Bool` repräsentiert, mit den Datenkonstruktoren `True` und `False`, die Wahrheitswerte. Desweiteren ist die Möglichkeit gegeben, Kommentare in den Programmtext zu integrieren. Dazu muß die entsprechende Zeile mit `--` beginnen.

Der erste Teil der konkreten Syntax von ExtCore beschreibt die Deklarationen. Dies ist die wesentliche Erweiterung zu Core und ist an der Sprache Haskell angelehnt. Im Gegensatz zu Haskell fehlen die Modul Deklarationen, die Dekalartion von Typ Klassen und die Deklaration von Typsynonymen. Stattdessen ist die Deklaration von Basisdimensionen möglich.

Ein Programm ist eine Folge von Deklarationen, die durch Semikolons getrennt werden. Dabei spielt die Reihenfolge keine Rolle, d.h. eine Funktion kann in Programmtext vor ihrer Definition verwendet werden.

program	$\rightarrow$	*declaration*$_1$; … ; *declaration*$_n$	$n \geq 1$
declaration	$\rightarrow$	*topdecl* | *decl*	
topdecl	$\rightarrow$	`data` *typelhs* `=` *condecls*	
	|	`dims =` *dimdecl*$_1$ `,` … `,`*dimdecl*$_n$	$n \geq 1$
decl	$\rightarrow$	*varid* :: *type*	
	|	*varid varid*$_1$ … *varid*$_n$ `=` *expr*	$n \geq 0$
dimdecl	$\rightarrow$	*conid* (*number*$_1$, *number*$_2$)	
	|	*conid*	
typelhs	$\rightarrow$	*conid varid*$_1$ … *varid*$_n$	$n \geq 0$
	|	*comid* `{` *varid*$_1$ … *varid*$_n$ `}`	$n \geq 1$
condecls	$\rightarrow$	*condecl* `|` *condecls*	
	|	*condecl*	
condecl	$\rightarrow$	*conid atype*$_1$ … *atype*$_n$	$n \geq 1$
	|	*tycon*	
type	$\rightarrow$	*btype*	
	|	*btype* `->` *type*	
btype	$\rightarrow$	*conid atype*$_1$ … *atype*$_n$	$n \geq 0$
	|	*atype*	
atype	$\rightarrow$	*varid*	
	|	*tycon*	
	|	(*type*)	
	|	(*type*$_1$, … , *type*$_n$)	$n \geq 2$
tycon	$\rightarrow$	*conid*	
	|	*conid* `{` *dim*$_1$ … *dim*$_n$ `}`	$n \geq 1$
dim	$\rightarrow$	*varid*	
	|	*conid*	
	|	*varid* `^` *number*	
	|	*conid* `^` *number*	

Abbildung 4.2 : Konkrete Syntax von ExtCore (1)

Die Deklaration von neuen Datentypen entspricht, bis auf das Fehlen von Typklassen, der Deklaration in Haskell. Neu hinzugekommen ist die Angabe von Dimensionen, die innerhalb von geschweiften Klammern erfolgen muß. Neben der üblichen Definiton von Datenkonstrkturen, könne auch Datenkonstruktoren definiert werden, die Dimensionen haben. Dies ist aber nur für 0-stellige Datenkonstruktoren, wie z.B. 1, erlaubt. Bei der Definition von Typkonstruktoren ist die Angabe von Dimensionsvariablen nur erlaubt, wenn keine Typvariablen verwendet werden, also der Typkonstruktor die Stelligkeit 0 hat.

Die Definition der natürlichen Zahlen sieht daher, natürlich nur rein theoretisch, wie folgt aus: `data Num {a} = 0 {a} | 1 {a} | 2 {a}` ....

Basisdimensionen werden mittels `dims` angegeben. Hierbei ist es auch möglich die Dimensionsgrenzen festzulegen. Wie in Abschnitt 3.4 erläutert, kann man dadurch die Verwendung der Basisdimension sehr genau kontrollieren.

expr	$\rightarrow$	`let` *defs* `in` *expr*	loakle Definition
	$\vert$	`letrec` *defs* `in` *expr*	lokale rekursive Definition
	$\vert$	`case` *expr* `of {` *alts* `}`	Selektion
	$\vert$	`\` $varid_1$ ... $varid_n$ `->` *expr*	λ-Abstraktion, $n \geq 1$
	$\vert$	*expr aexpr*	Applikation
	$\vert$	*aexpr*	atomarer Ausdruck
aexpr	$\rightarrow$	*varid*	Variable
	$\vert$	*number exprdim*	Zahl
	$\vert$	*conid exprdim*	Datenkonstruktor
	$\vert$	(*expr*)	geklammerter Ausdruck
	$\vert$	($expr_1, \ldots, expr_n$)	Tupel , $n \geq 2$
	$\vert$	*op*	Operator
defs	$\rightarrow$	def_1 ; ... ; def_n	$n \geq 1$
def	$\rightarrow$	*varid* `=` *expr*	$n \geq 0$
alts	$\rightarrow$	alt_1 ; ... ; alt_n	Alternativen , $n \geq 1$
alt	$\rightarrow$	*pat* `->` *expr*	einzelen Alternative
pat	$\rightarrow$	*number*	Zahl
	$\vert$	$(varid_1, \ldots, varid_n)$	Tupel , $n \geq 2$
	$\vert$	*conid* $varid_1 \ldots varid_n$	$n \geq 0$
exprdim	$\rightarrow$	`{` $edim_1 \ldots edim_n$ `}`	$n \geq 0$
	$\vert$		
edim	$\rightarrow$	*conid*	Basisdimension
	$\vert$	*conid* `^` *number*	
op	$\rightarrow$	*arithop* $\vert$ *relop* $\vert$ *boolop*	vordefinierte Operatoren
arithop	$\rightarrow$	`+` $\vert$ `-` $\vert$ `*` $\vert$ `/`	arithmetische Operatoren
relop	$\rightarrow$	`<` $\vert$ `<=` $\vert$ `==` $\vert$ `/=` $\vert$ `>=` $\vert$ `>`	relationale Operatoren
boolop	$\rightarrow$	`&&` $\vert$ `||`	bool'sche Operatoren

Abbildung 4.3 : Konkrete Syntax von ExtCore (2)

Der zweite Teil der konkreten Syntax, beschreibt alle Ausdrücke von ExtCore und orientiert sich sehr stark an der Sprache Core. Im Gegensatz zu Core können hier die Konstruktoren direkt verwendet werden ($pat \rightarrow conid\ var_1 \ldots var_n$). Außerdem wurde bei der λ-Abstraktion, der `.` durch, das in Haskell verwendete, `->` ersetzt. Im Unterschied zu Haskell, wurde hier die Trennung zwischen `let` und `letrec` beibehalten. Datenkonstruktoren der Stelligkeit 0, kann eine Dimension, wieder innerhalb geschweifter Klammern, vorgegeben werden. Die Angabe von Dimensionsvariablen ist dabei nicht zugelassen. Werden bei Zahlen oder 0-stelligen Datenkonstruktoren die Dimensionsangaben weggelassen, dann werden diese behandelt, als hätten sie die Dimension d. Dadurch kann man in ExtCore programmieren, ohne die Dimensionen verwenden zu müssen. Durch `{}` wird ausgedrückt, das keine Dimension vorliegt.

4.2 Die abstrakte Syntax

Die abstrakte Syntax beschreibt ExtCore in sehr kompakter Weise. Alle überflüssigen Informationen der konkreten Syntax wurden weggelassen. Außerdem wird angenommen, daß der Programmtext schon auf eine spezielle Weise analysiert wurde. Diese Analyse hat das Programm in die 3 Bereiche *dimdecls*, *typedecls* und *bgroups* aufgeteilt.

Alle Dimensionsdeklarationen befinden sich in *dimdecls* und alle Typdeklarationen in *typedecls*. Der Bereich *bgroups* enthält die einzelnen *Bindungsgruppen*, wobei diese aus den *Bindungen* besteht und den zu den Bindungen gehörigen Typsignaturen. Als Bindung bezeichnen wir im weitern eine Funktionsdefiniton. Enthält eine Bindungsgruppe nur eine Bindung, dann ist die zugehörige Funktion rekursiv (**recbind**) oder nicht-rekursiv (**nonrecbind**). Enthält eine Bindungsgruppe dagegen mehrere Bindungen, dann sind die zugehörigen Funktionen *verschränkt rekursiv*, d.h. die Funktionen rufen sich gegenseitig auf.

$$
\begin{array}{lcll}
\mathit{program} & \rightarrow & \mathit{dimdecls}\ \mathit{typedecls}\ \mathit{bgroups} & \\
\mathit{dimdecls} & \rightarrow & \mathit{dimdecl}_1\ \ldots\ \mathit{dimdecl}_n & n \geq 0 \\
\mathit{dimdecl} & \rightarrow & \textbf{dimfixed}\ \mathit{conid}\ \mathit{number}_1\ \mathit{number}_2 & \\
 & | & \textbf{dimany}\ \mathit{conid} & \\
\mathit{typedecls} & \rightarrow & \mathit{typedecl}_1\ \ldots\ \mathit{typedecl}_n & n \geq 0 \\
\mathit{typedecl} & \rightarrow & \textbf{data}\ \mathit{varid}_1\ \mathit{varid}_n & n \geq 0 \\
 & | & \textbf{datad}\ \mathit{varid}_1\ \mathit{varid}_n & n \geq 1 \\
\mathit{bgroups} & \rightarrow & \mathit{bgroup}_1\ \ldots\ \mathit{bgroup}_n & n \geq 1 \\
\mathit{bgroup} & \rightarrow & \textbf{nonrecbind}\ \mathit{sig}\ \mathit{bind} & \\
 & | & \textbf{recbind}\ \mathit{sig}_1\ \mathit{bind}_1\ \ldots\ \mathit{sig}_n\ \mathit{bind}_n & n \geq 1 \\
\mathit{sig} & \rightarrow & \textbf{nosig} & \\
 & | & \textbf{sig}\ \mathit{type} & \\
\mathit{bind} & \rightarrow & \mathit{varid}\ \mathit{varid}_1\ \ldots\ \mathit{varid}_n\ \mathit{expr} & n \geq 0 \\
\mathit{condecls} & \rightarrow & \mathit{condecl}_1\ \ldots\ \mathit{condecl}_n & n \geq 1 \\
\mathit{condecl} & \rightarrow & \mathit{conid}\ \mathit{atype}_1\ \ldots\ \mathit{atype}_n & n \geq 0 \\
\mathit{type} & \rightarrow & \mathit{btype} & \\
 & | & \textbf{tyfun}\ \mathit{btype}\ \mathit{type} & \\
\mathit{btype} & \rightarrow & \textbf{tycon}\ \mathit{conid}\ \mathit{atype}_1\ \ldots\ \mathit{atype}_n & n \geq 0 \\
 & | & \mathit{atype} & \\
\mathit{atype} & \rightarrow & \textbf{tyvar}\ \mathit{varid} & \\
 & | & \textbf{tyco\_}\ \mathit{conid}\ \mathit{dims} & \\
 & | & \textbf{tytup}\ \mathit{type}_1 \ldots \mathit{type}_n & n \geq 2 \\
 & | & \mathit{type} & \\
\mathit{dims} & \rightarrow & \mathit{dim}_1\ \ldots\ \mathit{dim}_n & n \geq 0 \\
\mathit{dim} & \rightarrow & \textbf{dimvar}\ \mathit{varid}\ \mathit{number} & \\
 & | & \textbf{dimbase}\ \mathit{conid}\ \mathit{number} &
\end{array}
$$

Abbildung 4.4 : Abstrakte Syntax von ExtCore (1)

Der Bereich *bgroups* ist nun so organisiert, daß eine Bindungsgruppe, in der eine Funktion f definiert wird, vor der Bindungsgruppe steht, in der die Funktion f angewendet wird. Desweiteren steht bereits fest, ob eine Bindung rekursiv ist und ob ihr eine Typsignatur gegeben wurde. Eine Unterscheidung zwischen Zahlen mit Dimensionen (**numdim**) und 0-stelligen Datenkonstruktoren mit Dimensionen (**condim**) ist notwendig, weil die Zahlen als vordefinierter Datentyp, intern anders behandelt werden als beliebige 0-stellige Datenkonstruktoren.

Die abstrakte Syntax der Ausdrücke von ExtCore enthält keinerlei Informationen über die Operatoren, die in der konkreten Syntax angegeben wurden. Dies liegt daran, daß diese Operatoren intern wie normale Funktionen behandelt werden. Daher ist es möglich, die binäre Addition auch unär anzuwenden. Der Ausdruck `+ 1` entspricht daher einer Funktion, die eine Zahl um eins erhöht.

expr	$\rightarrow$	**var** *varid*	Variable
	$\mid$	**num** *number*	Zahl
	$\mid$	**numdim** *number* $edim_1 \ldots edim_n$	Zahl mit Dimension , $n \geq 0$
	$\mid$	**tupel** $expr_1 \ldots expr_n$	Tupel , $n \geq 2$
	$\mid$	**con** *conid*	Datenkonstruktor
	$\mid$	**condim** *conid* $edim_1 \ldots edim_n$	Datenk. mit Dimension , $n \geq 0$
	$\mid$	**lambda** $varid_1 \ \ldots \ varid_n \ expr$	λ-Abstraktion, $n \geq 1$
	$\mid$	**app** *expr*1 *expr*2	Applikation
	$\mid$	**let** *defs expr*	loakle Definition
	$\mid$	**letrec** *defs expr*	lokale rekursive Definition
	$\mid$	**case** *expr alts*	Fallunterscheidung
defs	$\rightarrow$	**defs** $varid_1 \ldots varid_n \ expr_1 \ldots expr_n$	$n \geq 1$
alts	$\rightarrow$	$alt_1 \ \ldots \ alt_n$	$n \geq 1$
alt	$\rightarrow$	*pat expr*	
pat	$\rightarrow$	**patnum** *number*	Zahl
	$\mid$	**pattup** $varid_1 \ldots varid_n$	Tupel , $n \geq 2$
	$\mid$	**patcon** $varid_1 \ldots varid_n$	Datenkonstruktor , $n \geq 0$
edim	$\rightarrow$	**dimbase** *conid number*	Dimension in Ausdrücken

Abbildung 4.5 : Abstrakte Syntax von ExtCore (2)

4.3 Die Typinferenz

Nachdem sowohl konkrete als auch abstrakt Syntax definiert sind, kann das Erstellen der Typregeln beginnen. Vor der Definition dieser Regeln, müssen wir aber noch einiger

Vorbereitungen treffen. Dazu werden wir zunächst die Darstellung der Typen festlegen und uns überlegen, welche Typumgebungen wir mit welchen Informationen benötigen.

4.3.1 Darstellung der Dimensionstypen

Die Syntax der Dimensionstypen ist aus Abbildung 3.2 abgeleitet. Die Darstellung der Dimensionen wurde dahingehend geändert, daß sowohl die Dimensionsvariable , als auch eine Basisdimension, eine Exponenten besitzen. Außerdem ist der Tupeltyp in ExtCore schon vordefiniert. Dadurch ist man in der Lage, Tupel beliebiger Größe in der gewohnte Weise zu benutzen. Eine nachträgliche Defintion von Tupeltypen ist zwar auch möglich, hat aber den Nachteil, daß Tupel verschiedener Größe auch verschieden dargestellt werden müßten. Als Basistypen sind Num und Bool definiert.

Dimensionen	δ	$\rightarrow$	$\mathbf{0}$	keine Dimension
		$\mid$	d^n	Dimensionsvariable , $n \in \mathbb{Z} \setminus \{0\}$
		$\mid$	D^n	Basisdimension , $n \in \mathbb{Z} \setminus \{0\}$
		$\mid$	$\delta_1\ \delta_2$	Komposition
Mono Typ	τ	$\rightarrow$	α	Typvariable
		$\mid$	$\chi_0\ \delta$	Typkonstruktor mit Dimension
		$\mid$	$\tau_1 \rightarrow \tau_2$	Funktionstyp
		$\mid$	$(\tau_1, \ldots, \tau_n)$	Tupel Typ , $n \geq 2$
		$\mid$	$\chi(\tau_1, \ldots, \tau_n)$	strukturierte Typ , $n \geq 1$
Poly Typ	σ	$\rightarrow$	τ	monomorher Typ
		$\mid$	$\forall\ \alpha_1 \ldots \alpha_n, d_1 \ldots d_m.\ \tau$	$n \geq 1$, $m \geq 0$

Abbildung 4.6 : Syntax der Dimensionstypen von ExtCore

Auf diesen Typen benötigen wir noch folgende Funktionen.

- freie Typvariablen : fvT

$$\begin{aligned}
fvT(\alpha) &= \{\alpha\} \\
fvT(\chi_0\delta) &= \emptyset \\
fvT(\tau_1 \rightarrow \tau_2) &= fvT(\tau_1) \cup fvT(\tau_2) \\
fvT((\tau_1, \ldots, \tau_n)) &= \bigcup_{i=1}^{n} fvT(\tau_i) \\
fvT(\chi\tau_1 \ldots \tau_n) &= \bigcup_{i=1}^{n} fvT(\tau_i) \\
fvT(\forall\alpha_1 \ldots \alpha_n, d_1 \ldots d_m.\tau) &= fvT(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}
\end{aligned}$$

- freie Dimensionsvariablen : fvD

$$\begin{array}{ll}
fvD(\alpha) & = \emptyset \\
fvD(\chi_0\delta) & = freeD(\delta) \\
fvD(\tau_1 \rightarrow \tau_2) & = fvD(\tau_1) \cup fvD(\tau_2) \\
fvD((\tau_1,\ldots,\tau_n)) & = \bigcup_{i=1}^{n} fvD(\tau_i) \\
fvD(\chi\tau_1\ldots\tau_n) & = \bigcup_{i=1}^{n} fvD(\tau_i) \\
fvD(\forall\alpha_1\ldots\alpha_n, d_1\ldots d_m.\tau) & = fvD(\tau) \setminus \{d_1,\ldots,d_m\}
\end{array}$$

wobei

$$\begin{array}{ll}
freeD(\mathbf{0}) & = \emptyset \\
freeD(d^n) & = \{d\} \\
freeD(B^n) & = \emptyset \\
freeD(\delta_1\delta_2) & = freeD(\delta_1) \cup freeD(\delta_2)
\end{array}$$

Die Funktionen fvT und fvD werden auf die übliche Weise, auf Typumgebungen erweitert.

- Instanziierung eines Dimensionstyps : inst

  $$inst(\forall\alpha_1\ldots\alpha_n, d_1\ldots d_m.\tau) = [\beta_1/\alpha_1,\ldots,\beta_n/\alpha_n, c_1/d_1,\ldots,c_m/d_m]\tau$$
  wobei β_i neuen Typvariablen sind

- Verallgemeinerung eines Dimensionstyps : gen

  $$gen(\Gamma,\tau) = \forall((fvT(\tau) \setminus fvT(\Gamma)), (fvD(\tau) \setminus fvD(\Gamma)).\tau$$

Die Relation $\tau \geq \tau'$ gibt an, ob τ' eine Instanz von τ ist.

$$\tau \geq \tau' \;\Leftrightarrow \exists\theta : \theta\tau = \tau'$$

Die entsprechende Relation auf Dimensionen ist sehr ähnlich.

$$\delta \geq \delta' \;\Leftrightarrow \exists\theta_D : \theta_D\delta =_D \delta'$$

Hierbei ist die Gleichheit, die Gleichtheit auf Dimensionen (s. Seite 39) und θ_D die Substitution auf Dimensionsvariablen. Diese Substitution ist analog zur Substitution auf Typvariablen und wird daher nicht extra angegeben.

4.3.2 Die Umgebungen

Das in Kapitel 2 entwickelte Typsystem bediente sich einer Typumgebung Γ, um Informationen zu speichern und abzufragen. Aufgrund der einfachen Sprache und der Tatsache, daß die benötigten Informationen nur aus einfachen Typausdrücken besteht, war diese eine Umgebung vollkommen ausreichend.

Im Vergleich dazu ist die Sprache ExtCore um einiges komplexer. Vor allem die strukturierten Datentypen und die Dimensionstypen verlangen eine besondere Behandlung. Daher müssen wir in der Lage sein, mit den dabei entstehenden neuen Informationsarten umzugehen. Sicherlich wäre es möglich eine Umgebung zu konstruieren, die alle unterschiedlichen Informationsarten enthält. Aber aus Gründen der Übersichtlichkeit, werden wir verschiedene Umgebungen konstruieren, die unterschiedliche Arten von Information enthalten. Bei der Darstellung der Umgebungen folgen wir [PJW92].

Die Aufgabe von Umgebungen ist es, Namen mit speziellen Informationen zu assoziieren. Daher definieren wir Umgebungen als endliche Abbildungen von Namen nach Informationen.

$$Env : \text{Namen} \rightarrow \text{Informationen}$$

Umgebungen können wir explizit als endliche Mengen von Assoziationen, die wir mit $(name : info)$ notieren, angeben. So bezeichnet $Env = \{var_1 : \tau_1, var_2 : \tau_2\}$ die Umgebung Env, in der var_1 mit τ_1 und var_2 mir τ_2 assoziiert ist. Die leere Umgebung werden wir als $\{\}$ notieren.

Die Funktion *dom* liefert den Wertebereich einer Umgebung, also alle Namen die in dieser Umgebung mit einer Information assoziiert sind. Diese Funktion ist wie folgt definiert.

$$\begin{array}{l} dom : Env \rightarrow \text{Namen} \\ dom(\{\}) \ = \ \emptyset \\ dom(\{name_1 : info_1, \ldots, name_n : info_n\}) \ = \ \{name_1, \ldots, name_n\} \end{array}$$

Die Anwendung eines Namens auf eine Umgebung liefert die Information, die mit diesem Namen in der gegebenen Umgebung assoziiert ist. Ist keine Assoziation vorhanden wird $\perp$ zurückgeliefert. Die Definition sieht wie folgt aus.

$$Env(name) \ = \ \begin{cases} info & \text{falls } (name : info) \in Env \\ \perp & \text{sonst} \end{cases}$$

Die Funktion $\oplus$ kombiniert zwei Umgebungen gleiche Typs. Dabei wird vorausgesetzt, daß es keine Nameskonflikte geben kann.

$$Env_1 \ \oplus \ Env_2 \ = \ Env_1 \cup Env_2 \quad , \text{wobei } dom(Env_1) \cap dom(Env_2) = \emptyset$$

Alle Umgebungen sind in der folgenden Abbildung aufgelistet.

Umgebung	Bezeichnung	Zuordnung
Dimensionen	DI	$conid \mapsto (\mathrm{D}, number, number)$
Typkonstruktoren	TC	$conid \mapsto (\tau, number)$
Datenkonstruktoren	DC	$conid \mapsto \sigma$
Globale Variablen	GV	$varid \mapsto \tau$
Lokale Variablen	LV	$varid \mapsto \sigma$
Dimensionsvariablen	DV	$varid \mapsto d$
Typvariablen	TV	$varid \mapsto \alpha$

Abbildung 4.7 : Umgebungen der Typregeln

Die ersten 3 Umgebungen (DI, TC, DC), enthalten die Infromation der Top-level Deklarationen. Die Umgebung DI speichert die Informationen zu den definierten Basisdimensionen. Dabei ist D die interne Repräsentation der Basisdimension *conid* und die beiden Zahlen entsprechen der unteren bzw. der oberen Grenze des Exponenten. TC enthält Informationen zu jedem Typkonstruktoren in Form eines Typs τ und der Stelligkeit des Typkonstruktors. Der Typ τ entspricht dabei nicht dem Typ des Typkonstruktors, da dieser gar keine Typ hat, sondern dem Ergebnistyp jedes zugehörigen Datenkonstruktors. Dazu folgendes Beispiel.

```
data List a = Nil
            | Cons a (List a) ;
```

Dies ergibt folgende Umgebung: $TC = \{\texttt{List} : (\texttt{List}\ \alpha, 1)\}$

Hierbei ist die Typvariable α, die interne Repräsentation von `a`. Der Typ der Datenkonstruktoren besteht dann aus einem Funktionstyp, der zuerst alle Typen der Argumente enthält und am Ende den Typ, der in TC gespeichert ist. Also für `Cons` $\forall\alpha.\alpha\rightarrow\texttt{List}\ \alpha\rightarrow\texttt{List}\ \alpha$. Die Anzahl der gebundenen Typvariablen ist bekannt, da sie der Stelligkeit von `List` entspricht.

Die globale Umgebung GV enthält die Typen aller definierten Funktionen. Da diese Funktionen auf der äußersten Ebene definiert sind, sind alle Variablen dieser Typen, automatisch in diesem Typ gebunden. Deshalb können diese Typen auch als monomorphe Typen gespeichert werden. Bei Variablen, deren Typ in der lokalen Umgebung LV gespeichert ist, kann dies nicht angewendet werden. Dort können Variablen sowohl frei, als auch gebunden auftreten. Daher müssen die gebundenen Varibalen gekennzeichnet werden und somit der Typ als polymorpher Typ gespeichert werden. Die beiden letzten Umgebungen (DV, TV), werden bei der Überprüfung von Typdeklarationen und Typsignaturen verwendet. Sie weisen einer Variable des Programmtextes, die zugehörige interne Dimensions- oder Typvariable zu.

4.3.3 Entscheidungsformen

Bei der Erstellung der Inferenzregeln werden wir wie in Kapitel 2 vorgehen und die Regeln über den Aufbau der Sprache entwickeln. Da die Syntax von ExtCore umfangreicher ist als die des λ-Kalküls, wird für jedes Nichtterminal der abstrakten Syntax, eine spezielle *Entscheidungsform* bereitstellen. Eine Entscheidungsform eines Nichterminals besteht aus mindestens einer Infernzregel für jede Produktion dieses Nichtterminals.

Für jedes Nichtterminal gibt es eine spezielle Entscheidungsform, die die besondere Bedeutung des jeweiligen Nichtterminals berücksichtigt. Alle Formen basieren aber auf der folgende allgemeinen Form :

$$Umgebungen \overset{nichtterminal}{\vdash} Konstrukt \ : \ Informationen$$

Dies kann man wie folgt lesen : „Innerhalb der gegebenen *Umgebungen*, können wir über das *Konstrukt*, diese *Infomationen* herleiten. Dabei ist *Konstrukt* die rechte Seite einer Produktion von *nichtterminal* “.

Die wichtigsten Entscheidungsformen sehen wie folgt aus.

- Programm
  $DI, TC, DC, GV \overset{program}{\vdash} : dimdecls\ typedecls\ bgroups : (DI', TC', DC', GV')$
- Deklaration von Dimensionen
  $DI \overset{dimdecls}{\vdash} dimdecls : DI'$
- Deklaration von Typkonstruktoren
  $DI, TC, DC \overset{typedecls}{\vdash} typedecls : (TC', DC')$
- Bindungsgruppen
  $DI, TC, DC, GV \overset{bgroups}{\vdash} : bgroup_1 \ldots bgroup_n : GV'$
- Typsignaturen
  $DI, TC, GV \overset{sigs}{\vdash} sigs : GV'$
- Ausdrücke
  $DI, DC, GV, LV \overset{expr}{\vdash} expr : \tau$

4.3.4 Die Typregeln

Wir werden nun alle Typregeln für ExtCore angeben. Die Darstellung dieser Regeln erfolgt in gewohnter Weise. Die einzelnen Typregeln werden näher erläutert, indem wir angeben, wie wir die jeweilige Anwendung durchführen. Zur besseren Erläuterung der Typregeln führen wir die beiden Bezeichnungen *Argumenttypen* und *Ergebnistyp* ein. Bei einem Typ $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$, heißen die τ_i Argumenttypen und τ ist der Ergebnistyp.

4.3.4.1 Top-level Deklarationen

$$
\textbf{prg}\quad \frac{\begin{array}{rcl} DI & \overset{dimdecls}{\vdash} & dimdecls : DI' \\ DI', TC, DC & \overset{typedecls}{\vdash} & typedecls : (TC', DC') \\ DI', TC', DC', GV & \overset{bgroups}{\vdash} & bgroups : GV' \end{array}}{\begin{array}{rcl} DI, TC, DC, GV & \overset{program}{\vdash} & dimdecls\ typedecls\ bgroups \\ & & : (DI', TC', DC', GV') \end{array}}
$$

Abbildung 4.8 : Typregel für ein Programm

Das Überprüfen eines Programms besteht darin, die einzelnen Bereiche zu überprüfen und die jeweiligen Information weiterzuleiten. Dabei ist es wichtig, daß in der Reihenfolge *dimdecls*→*typedecls*→*bgroups* überprüft wird.

$$
\textbf{dims}\quad \frac{DI \oplus DI_1 \oplus \cdots \oplus DI_{i-1} \overset{dimdecl}{\vdash} dimdecl_i : DI_i \quad 1 \le i \le n}{DI \overset{dimdecls}{\vdash} dimdecl_1 \ \ldots \ dimdecl_n : DI_1 \oplus \cdots \oplus DI_n}
$$

$$
\textbf{types}\quad \frac{\begin{array}{c} TC \oplus TC_1 \oplus \cdots \oplus TC_{i-1} \overset{typecon}{\vdash} typedecl_i : TC_i \quad 1 \le i \le n \\ TC' = TC_1 \oplus \cdots \oplus TC_n \\ DI, TC', DC \oplus DC_1 \oplus \cdots \oplus DC_{i-1} \overset{datacon}{\vdash} typedecl_i : DC_i \quad 1 \le i \le n \\ DC' = DC \oplus DC_1 \oplus \cdots \oplus DC_n \end{array}}{DI, TC, DC \overset{typedecls}{\vdash} typedecl_1 \ldots typedecl_n : (TC', DC') \quad n \ge 0}
$$

$$
\textbf{bgroups}\quad \frac{DI, TC, DC, GV \oplus GV_1 \oplus \cdots \oplus GV_{i-1} \overset{bgroup}{\vdash} bgroup_i : GV_i \quad 1 \le i \le n}{DI, TC, DC, GV \overset{bgroups}{\vdash} bgroup_1 \ldots bgroup_n : GV_1 \oplus \cdots \oplus GV_n \quad n \ge 1}
$$

Abbildung 4.9 : Typregeln der Basis Strukturen

Die Überprüfung der 3 Bereiche geschieht, indem die einzelne Teile eines Bereiches nacheinander überprüft werden und die hergeleiteten Informationen weitergeleitet werden. Wie schon erwähnt, wird bei den *bgroups* verlangt, daß die einzelnen Bindungsgruppen bezüglich ihrer Abhängigkeit sortiert sind. Die Überprüfung der Typdeklarationen ist ein wenig aufwendiger, weil durch die Möglichkeit verschränkt rekursive Typen zu definieren, erst alle Typkonstruktoren mittels $\overset{typecon}{\vdash}$ überprüft werden müssen, bevor die Datenkonstruktoren überprüft werden können ($\overset{datacon}{\vdash}$).

$$\textbf{dimfixed}\quad \frac{DI(conid) = \bot \quad num_1 \leq num_2}{DI \overset{dimdecl}{\vdash} \textbf{dimfixed}\ conid\ num_1\ num_2 : \{conid : (\mathrm{D}, num_1, num_2)\}}$$

$$\textbf{dimany}\quad \frac{DI(conid) = \bot}{DI \overset{dimdecl}{\vdash} \textbf{dimany}\ conid : \{conid : (\mathrm{D}, -\infty, +\infty)\}}$$

Abbildung 4.10 : Typregeln für die Deklaration von Dimensionen

Die Typregel von Dimensionsdeklarationen ist ziemlich einfach, da die Informationen des Programmtextes einfach in die Umgebung DI kopiert werden. Falls die Dimensionsgrenzen nicht angegeben sind, werden $-\infty$ und $+\infty$ gespeichert.

$$\textbf{tcon1}\quad \frac{TC(conid) = \bot}{TC \overset{typecon}{\vdash} \textbf{data}\ conid\ varid_1 \ldots varid_n\ condecls : \{conid : (\chi, n)\}} \quad n \geq 0$$

$$\textbf{tcon2}\quad \frac{TC(conid) = \bot}{TC \overset{typecon}{\vdash} \textbf{datad}\ conid\ varid_1 \ldots varid_n\ condecls : \{conid : (\chi_0, n)\} \quad n \geq 1}$$

Abbildung 4.11 : Typregeln für die Deklaration eines Typkonstruktors

Im erste Teil der Überprüfung von Typdeklarationen, werden die obigen Regeln verwendet. Diese Regeln dienen nur dazu, alle Typkonstruktoren, vor der Überprüfung der Datenkonstruktor, in der Umgebunbg TC zu haben. Da wir zwischen Typdeklarationen mit Dimensionen und ohne Dimensionen unterscheiden, benötigen wir zwei Regeln. Ein Fehler tritt nur dann auf, wenn der Typkonstruktor schon definiert ist ($TC(conid) \neq \bot$).

Nachdem sich alle Typkonstruktoren in der Umgebung TC befinden, kann die Überprüfung der Datenkonstruktoren starten. Dafür gibt es wieder zwei Regeln, je nachdem ob Dimensionen im Spiel sind oder nicht. Diese beiden Regeln sind in Abbildung 4.12 zu finden.

Zuerst werden die beiden Umgebungen DV und TV gebildet, wobei eine Umgebung immer leer ist. Zusammen mit den vorgegebenen Umgebungen, bilden diese, die Umgebung, in der die einzelnen Datenkonstruktoren überprüft werden. Jede Überprüfung eines Datenkonstruktors liefert ein Umgebung DC_i, in der sich Informationen über diesen Datenkonstruktor befinden. Die Kombination aller DC_i, wird dann als Ergebnis zurückgegeben.

$$\textbf{dcon1}\quad \frac{\begin{array}{c} TC(conid) = (\chi, n) \\ TV = \{varid_1 : \alpha_1,\ \ldots\ , varid_n : \alpha_n\} \quad DV = \{\} \\ DI, TC, DC, DV, TV, \chi\ \alpha_1 \ldots \alpha_n \overset{condecl}{\vdash} condecl_i : DC_i \\ DC' = DC_1 \oplus\ \cdots\ \oplus DC_m \end{array}}{DI, TC, DC \overset{datacon}{\vdash} \textbf{data}\ conid\ varid_1 \ldots varid_n\ condecl_1 \ldots condecl_m : DC'}$$

$$\textbf{dcon2}\quad \frac{\begin{array}{c} TC(conid) = (\chi_0, n) \\ DV = \{varid_1 : d_1,\ \ldots\ , varid_n : d_n\} \quad TV = \{\} \\ DI, TC, DC, DV, TV, \chi_0\ d_1 \ldots d_n \overset{condecl}{\vdash} condecl_i : DC_i \\ DC' = DC_1 \oplus\ \cdots\ \oplus DC_m \end{array}}{DI, TC, DC \overset{datacon}{\vdash} \textbf{datad}\ conid\ varid_1 \ldots varid_n\ condecl_1 \ldots condecl_m : DC'}$$

Abbildung 4.12 : Typregeln für die Deklaration von Datenkonstruktoren

Die Überprüfung eines einzelnen Datenkonstruktors geschieht in der Typregel **condecl** aus Abbildung 4.13. Dabei wird zuerst getestet, ob der Datenkonstruktor schon definiert ist. Falls dies nicht der Fall ist, werden die einzelnen Typen des Datenkonstruktors überprüft und eine entsprechende Zuordnung von Datenkonstruktor zum resultierenden Funktionstyp zurückgegeben.

$$\textbf{condecl}\quad \frac{\begin{array}{c} DC(conid) = \bot \\ DI, TC, DV, TV \overset{monotype}{\vdash} atype_i : \tau_i \quad 1 \leq i \leq n \\ \{\alpha_1, \ldots, \alpha_n\} = fv(\tau) \end{array}}{\begin{array}{l} DI, TC, DC, DV, TV, \tau \overset{condecl}{\vdash} conid\ atype_1 \ldots\ atype_n \\ \qquad\qquad : \{conid : (\forall \alpha_1 \ldots \alpha_n .\ \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau, n)\} \end{array}}$$

Abbildung 4.13 : Typregeln für die Deklaration eines Datenkonstruktoren

Dabei ist zu beachten, daß alle vom Typkonstruktor bereitgestellten Variablen auch in jedem Typ des Datenkonstruktors gebunden sind, selbst wenn dieser nicht alle Variablen benötigt. Betrachten wir dazu folgendes Programm in ExtCore.

```
data List a = Nil | Cons a (List a) ;
len list = case list of
           { Nil       -> 0 ;
             Cons l rest -> 1 + len rest } ;
main = len (Cons 1 Nil) + len (Cons True Nil)
```

Die Datenkonstruktoren haben folgende Typen

Datenkonstruktor	alle Variablen	benötigte Variablen
`Nil`	$\forall a.List\ a$	$List\ a$
`Cons`	$\forall a.a \rightarrow List\ a \rightarrow List\ a$	$\forall a.a \rightarrow List\ a \rightarrow List\ a$

Das Programm ist also nur dann wohlgetypt, wenn auch `Nil` einen polymorphen Typ hat, d.h. `Nil`: $\forall a.List\ a$. Deswegen müssen alle Variablen des Typkonstruktors, in jedem Typ seiner Datenkonstruktoren gebunden werden.

Das Überprüfen von expliziten Typangaben, die bei der Deklaration von Typkonstruktoren und bei Typsignaturen vorkommen, wird durch die Typregeln aus Abbildung 4.14 beschrieben. Für jeder der 5 monomorphen Typarten, denn nur solche kann der Programmierer angeben, ist eine Regel vorhanden.

Bei einer Typvariable wird die entsprechende Information aus TV zurückgegeben. Bei einem Tupeltyp oder einem Funktionstyp, wird die jeweilige Kombination der Teiltypen zurückgegeben. Bei Typkonstruktoren, werden zusätzlich noch die Informationen des Typkonstruktors benötigt.

Bei einem Typkonstruktor mit Dimensionen muß zusätzlich noch überprüft werden, ob die angegebene Dimension δ', eine Instanz der der zulässigen Dimension δ ist. Die Überprüfung der Dimensionsgrenzen wird dabei von der Entscheidungsform $\overset{dim}{\vdash}$ vorgenommen.

$$\textbf{tyvar}\quad \frac{TV(varid) = \alpha}{DI, TC, DV, TV \overset{monotype}{\vdash} \textbf{tyvar}\ varid : \alpha}$$

$$\textbf{tytup}\quad \frac{DI, TC, DV, TV \overset{monotype}{\vdash} type_i : \tau_i \quad (1 \leq i \leq n)}{DI, TC, DV, TV \overset{monotype}{\vdash} \textbf{tytup}\ n\ type_1 \ldots type_n : (\tau_1, \ldots, \tau_n)}$$

$$\textbf{tyfun}\quad \frac{DI, TC, DV, TV \overset{monotype}{\vdash} type_1 : \tau_1 \qquad DI, TC, DV, TV \overset{monotype}{\vdash} type_2 : \tau_2}{DI, TC, DV, TV \overset{monotype}{\vdash} \textbf{tyfun}\ type_1\ type_2 : \tau_1 \rightarrow \tau_2}$$

$$\textbf{tyco\_}\quad \frac{\begin{array}{c} TC(conid) = (\chi_0\ \delta, 0) \\ DI, DV \overset{dims}{\vdash} dims : \delta' \quad \delta \geq \delta' \end{array}}{DI, TC, DV, TV \overset{monotype}{\vdash} \textbf{tyco\_}\ conid\ dims : \chi_0\ \delta'}$$

$$\textbf{tycon}\quad \frac{\begin{array}{c} TC(conid) = (\tau, n) \\ DI, TC, DV, TV \overset{monotype}{\vdash} type_i : \tau_i \quad (1 \leq i \leq n) \end{array}}{DI, TC, DV, TV \overset{monotype}{\vdash} \textbf{tycon}\ conid\ type_1 \ldots type_n : [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau}$$

Abbildung 4.14 : Typregeln für Typangaben

Das Überprüfen von Dimensionsdeklarationen ist in Abbildung 4.15 angegeben. Die Typregel **dims** überprüft eine Liste von Dimensionsdeklarationen, indem sie nacheinander die einzelnen Dimension durch $\overset{dim}{\vdash}$ überprüft. Die Entscheidungsfrom $\overset{dim}{\vdash}$ besteht aus 3 Regeln, jeweils eine für jede Dimensionsart.

dims $$\frac{DI, DV \overset{dim}{\vdash} dim_i : \delta_i \quad (1 \leq i \leq n)}{DI, DV \overset{dims}{\vdash} dim_1 \ldots dim_n : \delta_1 \ldots \delta_n}$$

nodim $$DI, DV \overset{dim}{\vdash} \; : \mathbf{0}$$

dimvar $$\frac{DV(varid) = d}{DI, DV \overset{dim}{\vdash} \textbf{dimvar}\ varid\ number : d^{number}}$$

dimbase $$\frac{\begin{array}{c} DI(conid) = (\mathrm{D}, lower, upper) \\ lower \leq number \leq upper \end{array}}{DI, DV \overset{dim}{\vdash} \textbf{dimbase}\ conid\ number : \mathrm{D}^{number}}$$

Abbildung 4.15 : Typregeln für die Deklaration von Dimensionen

Die einzelnen Regeln liefern die entsprechende Information aus der Umgebung zurück. Nur bei der Regel **dimbase** wird noch zusätzlich überprüft, ob die vorgegebenen Dimensionsgrenzen eingehalten wurden.

4.3.4.2 Bindungsgruppen

Das Überprüfen einer Bindungsgruppe erfolgt über die Entscheidungform $\overset{bgroup}{\vdash}$. Dabei gibt es eine Regel für nicht-rekursive (**bgroup1**) und eine für rekursive Bindungsgruppen (**bgroup1**). Die nicht-rekursiven Bindungsgruppen, bestehen immer aus nur einer Bindung. Rekursive Bindungsgruppe können dagegen mehr als nur eine Bindung. In diesem Fall sind die Bindungen verschränkt rekursiv.

$$\textbf{bgroup1}\quad \frac{DI, TC, DC, GV \overset{nonrecbind}{\vdash} sig\ bind : GV}{DI, TC, DC, GV \overset{bgroup}{\vdash} \textbf{nonrecbind}\ sig\ bind : GV}$$

$$\textbf{bgroup2}\quad \frac{\begin{array}{c} DI, TC, GV \overset{chksig}{\vdash} sig_i\ bind_i : LV_i \quad 1 \leq i \leq n \\ LV = LV_1 \oplus \cdots \oplus LV_n \\ DI, DC, GV, LV \overset{recbind}{\vdash} bind_i : \tau_i \quad 1 \leq i \leq n \\ LV_i, \tau_i \overset{cmpsigtype}{\vdash} sig_i : GV_i \\ GV = GV_1 \oplus \cdots \oplus GV_n \end{array}}{DI, TC, DC, GV \overset{bgroup}{\vdash} \textbf{recbind}\ sig_1\ bind_1 \ldots sig_n\ bind_n : GV}$$

Abbildung 4.16 : Typregeln für eine Bindungsgruppe

Ist die Bindungsgruppe nicht-rekursiv, dann wird die Entscheidungform $\overset{nonrecbind}{\vdash}$ angewandt. Die zurückgegebene Umgebung GV enthält die Zuordnung von Funktionsname zum inferierten Typ.

Bei einer rekursiven Bindungsgruppe ist wesentlich mehr Arbeit nötig. Zuerst werden die Signaturen der einzelnen Funktionen, soweit vorhanden, überprüft. Die Entscheidungsform $\overset{chksig}{\vdash}$ liefert für jede Bindung eine lokale Umgebung LV_i, in welcher dem Funktionsnamen, der entsprechende Typ aus der Signatur zugewiesen wird. Falls keine Signatur vorhanden ist, ist der Typ ein Funktionstyp aus lauter Typvariablen, wobei die Länge des Funktionstyps, der Anzahl der Variablen der Funktion entspricht.

Dieses Verfahren ist nötig, da es sich hierbei um verschränkt rekursive Bindungen handelt. Deshalb muß schon vor der ersten Überpüfung, allen Funktionsnamen ein Typ zugeordnet sein. Die Überprüfung der einzelnen Bindungen erfolgt über die Entscheidungsform $\overset{recbind}{\vdash}$. Im Gegensatz zu $\overset{nonrecbind}{\vdash}$ wird hier die Umgebung TC nicht benötigt, da die Überprüfung der Typsignaturen schon abgeschlossen ist. Dafür ist aber die Umgebung LV nötig, die die Typen der verschränkt rekursiven Bindung enthält.

Nachdem alle Bindungen überprüft worden sind, wird über $\overset{cmpsigtype}{\vdash}$ getestet, ob der Typ der Siganturen sig_i eine Instanz der Typen τ_i ist. Falls dies der Fall ist, liefert sie die entsprechende Umgebung GV zurück.

Für das Überprüfen von nicht-rekursiven Bindungen, sind die beiden Typregeln aus Abbildung 4.17 zuständig.

$$\textbf{nonrecbind1}\quad \frac{\begin{array}{c} LV = \{var_1 : \tau_1, \ldots, var_n : \tau_n\} \\ DI, DC, GV, LV \overset{expr}{\vdash} expr : \tau \\ GV = gen(\{var : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau\}) \end{array}}{DI, TC, DC, GV, \overset{nonrecbind}{\vdash} \textbf{nosig}\ var\ var_1 \ldots var_n\ expr : GV}$$

$$\textbf{nonrecbind2}\quad \frac{\begin{array}{c} DI, TC, GV \overset{sig}{\vdash} type : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \\ LV = \{var_1 : \tau_1, \ldots, var_n : \tau_n\} \\ DI, DC, GV, LV \overset{expr}{\vdash} expr : \tau' \\ \tau' \geq \tau \\ GV = gen(\{var : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau\}) \end{array}}{DI, TC, DC, GV \overset{nonrecbind}{\vdash} \textbf{sig}\ type\ var\ var_1 \ldots var_n\ expr : GV}$$

Abbildung 4.17 : Typregeln für eine nicht-rekursive Bindung

Die Typregel **nonrecbind1** wird angewandt, falls keine Signatur vorgegeben ist. In diesem Fall wird eine lokale Umgebung LV erstellt, die die Typen der Funktionsargumente enthält. Der Funktionsrumpf $expr$ wird über die Entscheidungsform $\overset{expr}{\vdash}$ geprüft. Dabei wird die Umgebung LV übergeben, wodurch sichergestellt ist, das allen Funktionsargumenten schon ein Typ zugeordnet worden ist. Danach wird der Typ der Funktion zusammengesetzt und eine entsprechende Umgebung GV zurückgegeben.

Ist die Bindung mit Signatur angegeben, dann erfolgt die Überprüfung ein wenig anders. Zum ersten wird die Signatur dazu verwendet, den einzelnen Argumenten der Funktion, den dort angegebene Typ zuzuordnen. Außerdem wird nach der Überprüfung von $expr$ getestet, ob die angegebene Signatur eine Instanz des angegebenen Typs ist. Hierzu müssen lediglich die Ergebnistypen τ' und τ getestet werden, da die Typen der Argumente bei der Herleitung von τ' verwendet wurden.

$$\textbf{recbind}\quad \frac{\begin{array}{c} LV(var) = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \\ LV' = \{var_1 : \tau_1, \ldots\ var_n : \tau_n\} \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr : \tau \end{array}}{DI, DC, GV, LV \overset{recbind}{\vdash} var\ var_1 \ldots var_n\ expr : \tau}$$

Abbildung 4.18 : Typregel für eine rekursive Bindung

Wenn die Typregel **recbind** angewendet wird, ist sichergestellt, daß für var ein entsprechender Eintrag in LV vorhanden ist. Der Funktionsrumpf $expr$, kann dadurch in einer Umgebung überprüft werden, in der sowohl der Typ von var, als auch der Typ aller Argumente bekannt ist.

4.3.4.3 Signaturen

Für Typsignaturen gibt es drei verschiedenen Entscheidungsformen. Eine für Signaturen von nicht-rekursiven Bindungen und zwei für Signaturen von rekursiven Bindungen.

Die Entscheidungsform $\overset{sig}{\vdash}$ hat nur die ein Regel **sig** und ist für nicht-rekursive Bindungen zuständig.

$$\textbf{sig}\quad \frac{\begin{array}{c} GV(var) = \bot \\ typevar(type) = \{varid_1, \ldots, varid_n\} \quad n \geq 0 \\ dimvar(type) = \{varid_1, \ldots, varid_m\} \quad m \geq 0 \\ TV = \{varid_1 : \alpha_1, \ldots, varid_n : \alpha_n\} \\ DV = \{varid_1 : d_1, \ldots, varid_n : d_m\} \\ DI, TC, DV, TV \overset{monotype}{\vdash} type : \tau \end{array}}{DI, TC, GV \overset{sig}{\vdash} type : \tau}$$

Abbildung 4.19 : Typregel für die Signatur einer nicht-rekursiven Bindung

Als erstes muß sichergestellt sein, daß dem Funktionsnamen var noch kein Typ zugeordnet wurde. Dann kann der Typ der Signatur, in seinen speziellen Umgebungen DV und TV hergeleitet werden. Die beiden Funktionen $typevar$ und $dimvar$ liefern, analog zu fvT und fvD, die Typ- bzw. Dimensionsvariablen eines Typs, allerdings auf der abstrakten Syntax. Trotzdem ist die Struktur identisch zu fvT und fvD, weswegen hier auf eine explizite Definition verzichtet wird.

Die nächsten beiden Entscheidungsformen, sind für rekursive Bindungsgruppen zuständig.

Die Entscheidungsform $\overset{checksig}{\vdash}$ wird dabei vor der Überprüfung der einzelnen rekursiven Funktionen angewendet. Ihre Aufgabe ist es, eine lokale Umgebung zu liefern, in der dem Funktionsnamen var, der Typ der Signatur zugeordnet wird.

$$\textbf{chknosig}\quad \frac{LV = \{var : \tau_1 \to \ldots \to \tau_n \to \tau\}}{DI, TC, GV \overset{checksig}{\vdash} \textbf{nosig}\ var\ var_1 \ldots var_n\ expr : LV}$$

$$\textbf{chksig}\quad \frac{\begin{array}{c} DI, TC, GV \overset{sig}{\vdash} type : \tau_1 \to \ldots \to \tau_n \to \tau \\ LV = \{var : \tau_1 \to \ldots \to \tau_n \to \tau\} \end{array}}{DI, TC, GV \overset{checksig}{\vdash} \textbf{sig}\ type\ var\ var_1 \ldots var_n\ expr : LV}$$

Abbildung 4.20 : Typregeln für die Signatur einer rekursiven Bindung

Die Entscheidungsform $\overset{cmpsigtype}{\vdash}$ wird angewendet, nachdem alle verschränkt rekursiven Bindungen einer rekursiven Bindungsgruppe, überprüft wurden. Falls diese Bindung keine Signatur besitzt, braucht nichts überprüft zu werden und die lokale Umgebung LV wird in eine globale Umgebung umgewandelt und zurückgegeben. Ist eine Signatur vorhanden, muß überprüft werden, ob der inferierte Ergebnistyp τ', eine Instanz des Ergebnistyps der Signatur ist.

$$\textbf{cmpsig1} \quad \frac{LV' = \{var : \tau_1 \to \ldots \to \tau_n \to \tau'\}}{LV, \tau' \overset{cmpsigtype}{\vdash} \textbf{nosig} : gen(LV)}$$

$$\textbf{cmpsig2} \quad \frac{\begin{array}{c} LV = \{var : \tau_1 \to \ldots \to \tau_n \to \tau\} \\ \tau' \geq \tau \end{array}}{LV, \tau' \overset{cmpsigtype}{\vdash} \textbf{sig}\ type : gen(LV)}$$

Abbildung 4.21 : Typregel für die Signatur einer rekursiven Bindung

4.3.4.4 Ausdrücke

Die Abbildung 4.22 enthält die einfachsten Typregeln der Entscheidungsform $\overset{expr}{\vdash}$. Bei allen Regeln wird die Information aus den Umgebungen, in entsprechender Form zurückgegeben. Für Variablen gibt es zwei Typregeln, da diese sowohl in der lokalen Umgebung LV, als auch in der gloablen Umgebung GV definiert sein können.

$$\textbf{lvar} \quad \frac{var \in dom(LV) \quad LV(var) = \sigma}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{var}\ var : inst(\sigma)}$$

$$\textbf{gvar} \quad \frac{var \in dom(GV) \quad GV(var) = \sigma}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{var}\ var : inst(\sigma)}$$

$$\textbf{num} \quad DI, DC, GV, LV \overset{expr}{\vdash} \textbf{num}\ num : \mathsf{Num}$$

$$\textbf{confun} \quad \frac{DC(conid) = \sigma}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{con}\ conid : inst(\sigma)}$$

$$\textbf{tupel} \quad \frac{DI, DC, GV, LV \overset{expr}{\vdash} expr_i : \tau_i \quad (1 \leq i \leq n)}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{tupel}\ expr_1 \ldots expr_n : (\tau_1, \ldots, \tau_n)}$$

Abbildung 4.22 : Typregeln für einfache Ausdrücke

Es gibt zwei Ausdrucksformen die eine explizite Dimension haben können. Dies sind die 0-stelligen Datenkonstruktoren und als Spezialfall davon, die vordefinierten Zahlen. Bei beiden wird der zugehörige Typkonstruktor mit der hergeleiteten Dimension zurückgegeben. Bei Zahlen muß nicht überprüft werden, ob die angegebene Dimension eine Instanz der zulässigen Dimension ist, weil die zulässige Dimension d ist. Bei den benutzerdefinierten Datenkonstruktoren muß diese Überprüfung erfolgen.

$$\textbf{condim}\quad \frac{DI(conid) = \chi_0\ \delta \qquad DI \overset{edims}{\vdash} edim_1 \ldots edim_n : \delta' \qquad \delta \geq \delta'}{DI, DC, GV, LV, DV \overset{expr}{\vdash} \textbf{condim}\ conid\ edim_1 \ldots edim_n : \chi_0\ \delta'} \ , n \geq 0$$

$$\textbf{numdim}\quad \frac{DI \overset{edims}{\vdash} edim_1 \ldots edim_n : \delta}{DI, DC, GV, LV, DV \overset{expr}{\vdash} \textbf{numdim}\ number\ edim_1 \ldots edim_n : \mathsf{Num}\ \delta} \ , n \geq 0$$

Abbildung 4.23 : Typregeln für einen Datenkonstruktor mit Dimension

Die Überprüfung von Dimensionen in Ausdrücken, verläuft analog zur Entscheidungsform $\overset{dims}{\vdash}$. Der einzige Unterschied besteht darin, daß keine Dimensionsvariablen zugelassen sind und deshalb auch die Umgebung DV nicht benötigt wird.

$$\textbf{edims1}\quad DI \overset{edims}{\vdash} \ : \mathbf{0}$$

$$\textbf{edims2}\quad \frac{DI \overset{edim}{\vdash} edim_i : \delta_i \ ,\ 1 \leq i \leq n}{DI \overset{edims}{\vdash} edim_1 \ldots edim_n : \delta_1 \ldots \delta_n \ ,\ n \geq 1}$$

$$\textbf{edim}\quad \frac{DI(conid) = (\mathrm{D}, lower, upper) \qquad lower \leq number \leq upper}{DI \overset{dim}{\vdash} \textbf{dimbase}\ conid\ number : \mathrm{D}^{number}}$$

Abbildung 4.24 : Typregeln für Dimensionen in Ausdrücken

λ-Ausdrücke werden getypt, indem der Körper der unbenannten Funktion ($expr$), in einer erweiterten lokalen Umgebung getypt wird. Diese Erweiterung besteht darin, jedem Argument var_i eine Typ vorzugeben. Der Typ eines λ-Ausdrucks ist dann eine Funktionstyp der alle Typen der Argumente enthält und als Ergebnistyp, den Typ von $expr$ hat.

$$\textbf{lambda} \quad \frac{\begin{array}{c} LV' = \{var_1 : \tau_1, \ \ldots \ , var_n : \tau_n\} \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr : \tau \end{array}}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{lambda}\ var_1 \ldots var_n\ expr : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau}$$

Abbildung 4.25 : Typregel für eine unbenannte Funktion

Der Typ einer Applikation $expr_1$ $expr_2$ ergibt sich, wenn der Typ von $expr_2$ der Argumenttyp von $expr_1$ ist. In diesem Fall entspricht der Typ der Applikation, dem Ergebinstyp von $expr_1$.

$$\textbf{app} \quad \frac{DI, DC, GV, LV \overset{expr}{\vdash} expr_1 : \tau \rightarrow \tau' \quad DI, DC, GV, LV \overset{expr}{\vdash} expr_2 : \tau}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{app}\ expr_1\ expr_2 : \tau'}$$

Abbildung 4.26 : Typregel für eine Applikationen

Die Typregeln von **let** und **letrec** sind sehr ähnlich. Bei beiden Programmkonstrukten wird *expr* in einer lokalen Umgebung überprüft, die um Informationen aus den einzelnen Definition erweitert ist. Der hergeleitete Typ für *expr*, entspricht dann dem Typ von **let** bzw. **letrec**. Der Unterschied zwischen **let** und **letrec** liegt in der Art und Weise, wie diese lokale Umgebung erstellt wurde. Bei **let** wird die Entscheidungsform $\overset{defs}{\vdash}$ verwendet, weil die lokalen Funktionen nicht-rekursiv sind. Für **letrec** muß die rekursive Variante $\overset{defs-rec}{\vdash}$ benutzt werden.

$$\textbf{let} \quad \frac{\begin{array}{c} DI, DC, GV, LV \overset{defs}{\vdash} defs : LV' \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr : \tau \end{array}}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{let}\ defs\ expr : \tau}$$

$$\textbf{letrec} \quad \frac{\begin{array}{c} DI, DC, GV, LV \overset{defs-rec}{\vdash} defs : LV' \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr : \tau \end{array}}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{letrec}\ defs\ expr : \tau}$$

Abbildung 4.27 : Typregeln für lokale Definitionen

Fallunterscheidungen kann man als Funktionen auffassen, wobei der zu überprüfende Ausdruck *expr* das Argument ist und die einzelnen Alternativen Funktionen sind, die dieses Argument übernehmen. Die Typregel besagt, daß der Typ eines **case**-Ausdrucks, dem Ergebnistyp der Alternativen entspricht, falls der Typ der zu überprüfende Bedingung, dem Argumententyp der Alternativen entspricht.

$$\textbf{case} \quad \frac{DI, DC, GV, LV \overset{expr}{\vdash} expr : \tau \quad DI, DC, GV, LV \overset{alts}{\vdash} alts : \tau \rightarrow \tau'}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{case}\ expr\ alts : \tau'}$$

Abbildung 4.28 : Typregel für Fallunterscheidung

Lokale Definitionen werden getypt, indem die Typen der einzelnen Funktionskörper ($expr_i$) hergeleitet werden und dann den zugehörigen Funktionsnamen (var_i) zugeordnet werden. Da diese Funktionen nach ihrer Defintion auf verschiedene Weise benutzt werden können, müssen die Typen verallgemeinert werden ($gen(LV')$). Sind die lokalen Funktionen rekursiv, muß schon vor der ersten Herleitung, jedem Funktionsname ein Typ zugeordnet sein.

$$\textbf{defs} \quad \frac{\begin{array}{c} DI, DC, GV, LV \overset{def}{\vdash} expr_i : \tau_i \quad (1 \leq i \leq n) \\ LV' = \{var_1 : \tau_1, \ \dots \ , var_n : \tau_n\} \end{array}}{DI, DC, GV, LV \overset{defs}{\vdash} \ var_1 \dots var_n \ expr_1 \dots expr_n : gen(LV') \quad n \geq 1}$$

$$\textbf{defsrec} \quad \frac{\begin{array}{c} LV' = \{var_1 : \tau_1, \ \dots \ , var_n : \tau_n\} \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr_i : \tau_i \quad (1 \leq i \leq n) \end{array}}{DI, DC, GV, LV \overset{defsrec}{\vdash} \ var_1 \dots var_n \ expr_1 \dots expr_n : gen(LV') \quad n \geq 1}$$

Abbildung 4.29 : Typregeln für Definitionen

Alternativen werden über die Entscheidungform $\overset{alts}{\vdash}$ getypt. Dabei wird gefordert, daß jede Alternative den gleichen Funktionstyp besitzt.

$$\textbf{alts} \quad \frac{DI, DC, GV, LV \overset{alt}{\vdash} alt_i : \tau \rightarrow \tau' \quad (1 \leq i \leq n)}{DI, DC, GV, LV \overset{alts}{\vdash} alt_1 \dots alt_n : \tau \rightarrow \tau'}$$

Abbildung 4.30 : Typregeln für Alternativen

Die Überrüfung einer einzigen Alternative erfolgt mit $\overset{alt}{\vdash}$. Alle Alternativen liefern eine Funktionstyp zurück, dessen Ergebnistyp dem Typ von $expr$ entspricht. Der Argumenttyp ist abhängig von dem speziellen pattern. Bei einer Zahl ist der Argumenttyp `Num` und bei einem Tupel der entsprechende Tupeltyp. Ist das pattern ein Datenkonstruktor, dann muß der Typ dieses Datenkonstruktors geholt werden und die einzelnen Variablen var_i, mit den entsprechenden Argumenttypen ausgestattet werden. Bei den pattern, die Variablen besitzen (**pattup**,**patcon**), muß *expr* in einer Umgebung hergeleitet werden, in der diesen Variablen ein entsprechender Typ zugeordnet ist.

$$\textbf{alt1}\quad \frac{DI, DC, GV, LV \overset{expr}{\vdash} expr : \tau}{DI, DC, GV, LV \overset{alt}{\vdash} \textbf{patnum}\ num\ expr : \mathsf{Num} \rightarrow \tau}$$

$$\textbf{alt2}\quad \frac{\begin{array}{c} LV' = \{var_1 : \tau_1,\ \ldots\ , var_n : \tau_n\} \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr : \tau \end{array}}{DI, DC, GV, LV \overset{alt}{\vdash} \textbf{pattup}\ var_1 \ldots var_n\ expr : (\tau_1, \ldots, \tau_n) \rightarrow \tau \quad (n \geq 2)}$$

$$\textbf{alt3}\quad \frac{\begin{array}{c} DC(conid) = \sigma \qquad inst(\sigma) = \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \\ LV' = \{var_1 : \tau_1,\ \ldots\ , var_n : \tau_n\} \\ DI, DC, GV, LV \oplus LV' \overset{expr}{\vdash} expr : \tau' \end{array}}{DI, DC, GV, LV \overset{alt}{\vdash} \textbf{patcon}\ conid\ var_1 \ldots var_n\ expr : \tau \rightarrow \tau' \quad (n \geq 0)}$$

Abbildung 4.31 : Typregeln für Alternativen

4.4 Beispiele

Die Programmierung in ExtCore kann auf die gewohnte Weise erfolgen.

```
test x = x + 2 ;
```

Hier hat die Funktion `test` den Typ `Num{a}->Num{a}`, d.h. die Dimensionen werden automatisch hinzugefügt.

```
f x y z = (x*x) + (y*y*y) + (z*z*z*z) ;
```

Der Typ von `f` zeigt, daß man durch die Hinzunahme von Dimensionen, einige überraschende Typen erhält. `f :: Num{a^6}->Num{a^4}->Num{a^3}->Num{a^12}`.

Durch die explizite Verwendung von Dimensionen, läßt sich das Problem aus Abschnitt 1.1 lösen.

```
dims = DM (-1,1) , KMH ;
preis = 28400.50 {DM} ;
geschwindigkeit = 220 {KMH} ;
```

Der Ausdruck `preis + geschwindigkeit` wird durch die obige Definition als fehlerhaft erkannt. Die explizite Angabe der Dimensionsgrenzen von `DM` ermöglicht es, daß `preis * preis` zurückgewiesen wird.

Da die Ermittlung der Dimensionen statisch, d.h. zur Übersetzungszeit, erfolgt, haben einige Funktionen nicht den erhofften Typ.

```
power n x = case (n==0) of
            { True  -> x ;
              False -> x * (power (n - 1) x) } ;
```

Hier wird nur der Typ `Num{a}->Num->Num` hergeleitet, da die Dimension des Ergebnisses, nur zur Laufzeit exakt festgestellt werden kann.

Um auf einfache Weise mit Dimensionen umgehen zu können, müssen einige Funktionen vordefiniert sein, die Dimensionen enthalten.

```
dims=M (-3,3),S (-2,2) ;
geschwindigkeit :: Num{M} -> Num{S} -> Num{M S^-1} ;
geschwindigkeit weg zeit = weg / zeit ;
```

Hier wurden Basisdimensionen für die Länge (`M`) und die Zeit (`S`) definiert. Außerdem ist eine Funktion vorgegeben, die eine Geschwindigkeit berechnet. Wird diese Funktion `geschwindigkeit` verwendet, dann vererbt sie ihre Dimensionen.

```
beschleunigung weg zeit = (geschwindigkeit weg zeit) / zeit ;
```

Durch die vorherige Definition von `geschwindigkeit`, wird ohne eine explizite Angabe, der korrekte Typ `Num{M}->Num{S}->Num{M S^-2}` hergeleitet.

Kapitel 5

Die Implementierung

Zur Implementierung von ExtCore wird die funktionale Sprache Haskell [HPW92] verwendet. Dabei wird nicht der volle Sprachumfang verwendet[1], sondern nur der Bereich genutzt, der auch von Gofer[2] [Jon94] untersützt wird. Dadurch kann ExtCore auf fast allen Systemen verwendet werden.

5.1 Überlegungen

Um ExtCore zu implementieren, genügt es sicherlich nicht, nur die Typregeln aus Kapitel 4 zu implementieren. Damit die Vorteile von Dimensionstypen in geeigneter Weise aufgezeigt werden können, ist es vielmehr notwendig ein komplettes Compiler Front-End zu entwickeln. Dieses Front-End erzeugt aus einem Programm in ExtCore, einen speziellen Zwischencode. Der Zwischencode ist die schon erwähnte Sprache Core [PJL92]. Dies hat zum einen den Vorteil, daß kein umfangreicher Generator zum Erzeugen des Zwischencodes notwendig ist, weil ExtCore auf Core basiert. Außerdem ist es dadurch relativ einfach möglich, die Ergebnisse von zwei anderen Diplomarbeiten [Sch94][Res96] einzubauen.

Das Front-End besteht aus den folgenden Bereichen.

1. Syntaktische Analyse
2. Statische Analyse
3. Typinferenz

[1] Damit sind hauptsächlich die Typklassen gemeint.

[2] Der Name Gofer beschreibt sowohl die Programmiersprache Gofer, als auch den zughörigen Interpreter. Beides wurde von Mark P. Jones entwickelt. Da er den C-Quelltext frei verfügbar gemacht hat, sind Interpreter für Unix Systemen, Apple und PC, sowie Atari und Amiga, erhältlich. Nur für den C64 gibt es bisher keine Implementierung ;-)

4. Ausgabefunktionen

Die syntaktische Analyse erzeugt aus dem Programmtext, eine abstrakte Repräsentation, auf der die nachfolgenden Phasen arbeiten. Die statische Analyse untersucht die Abhängigkeiten der definierten Funktionen und erzeugt die entsprechenden Bindungsgruppen. Außerdem werden soch hier die Top-level Deklarationen überprüft.

Die gesammelten Informationen der statischen Analyse, werden dann zur Typinferenz weitergeleitet. Diese überprüft die, von der Abhängigkeitsanalyse erstellten, Bindungsgruppen und liefert die entsprechenden Umgebungen zurück. Danach können alle gesammelten Informationen, auf verschiedene Arten ausgegeben werden.

Bevor wir mit der Beschreibung der Implementierung beginnen, stellen wir eine neue funktionale Programmiertechnik vor. Diese erlaubt es, imperative Konstrukte, wie z.B. globale Zähler, auf sehr übersichtliche Weise in funktionalen Programmen zu simulieren.

5.2 Monaden

Die Konzepte von *Monaden* kommen aus dem Bereich der Kategorie Theorie. Die Anwendbarkeit von Monaden beim funktionalen Programmieren, wurden zuerst von Wadler [Wad92] erkannt.

Eine Monade besteht aus einem Typkonstruktor `M` und den beiden *monadischen Funktionen* `returnM` und `thenM`.

```
returnM :: a -> M a
thenM :: M a -> (a -> M b) -> M b
```

Diese beiden Funktionen müssen die folgende Bedingungen erfüllen

linke Einheit	`(returnM a) ‘thenM‘ k = k a`
rechte Einheit	`m ‘thenM‘ returnM = m`
Assoziativität	`m ‘thenM‘ (\a -> (k a) ‘thenM‘ (\b -> h b)) =`
	`(m ‘thenM‘ (\a -> (k a)) ‘thenM‘ (\b -> h b)`

Dadurch ist eine korrekte Kombination gewährleistet.

Die Idee bei Monaden ist, einfache Werte vom Typ `a`, in monadische Werte vom Typ `M a` zu überführen und auf diesen zu rechnen.

```
f1 a ‘thenM‘ \b ->
f2 b ‘thenM‘ ...
```

Diesen Programmausschnitt könnten man auch einfacher als `f2(f1(a)) ...` angeben. Der große Vorteil von Monaden ist aber, daß Berechnungen verborgen werden können. Nach folgender Definition

```
type M a = (a,Int)
thenM (a,counter) f = (f a,counter + 1)
```

wird bei dem monadischen Programmausschnitt, die Anzahl der Funktionsaufrufe mitgezählt. Dies geschieht ohne eine Änderung am Programmausschnitt. Das Hinzufügen einer Fehlerbehandlung, würde nur eine Änderung von `M` und den beide Funktionen `returnM` und `thenM` erfordern. Hätte man diese Erweiterung an `f2(f1(a)) ...` vornehmen müssen, dann wären wesentlich umfangreichere Veränderungen nötig gewesen.

Mit Monaden können Programme so erstellt werden, daß Details, die zwar notwendig sind, aber ein Programm unübersichtlich machen, so versteckt werden, daß nur die eigentliche Programmstruktur sichtbar ist. Daher eignen sich Monaden besonders bei Programmen die auf Fehler reagieren müssen oder globale Zähler benutzen.

Wir werden Monaden sehr intensiv nutzen und diese an den entsprechenden Stellen, genauer betrachten.

5.3 Die Basiselemente

Da in vielen Funktionen des zu implementierenden Compiler Front-End Fehler auftreten können, wird ein Datentyp inklusive der zugehörigen Funktionen benötigt, der eine Fehlerbehandlung bieten. Da wir keinen Wert auf eine komplexe Fehlerbehandlung legen, werden wir lediglich zwischen erfolgreicher und fehlerhafter Ausführung einer Funktion unterscheiden. Deshalb definieren wir den folgenden Datentyp.

```
data Result a = Ok a
              | Fail FailFun
```

Hierbei repräsentiert `Ok` eine erfolgreiche Ausführung, wohingegen `Fail` einen Fehler signalisiert und eine entsprechende Fehlermeldung vom Typ `FailFun` zurückgegeben wird. Die wichtigsten Funktionen auf diesem Datentyp sind:

- `returnResult :: a -> Result a`
  erzeugt aus einem Datum, daß Ergebnis einer erfolgreichen Berechnung
- `failResult :: FailFun -> Result a`
  erzeugt das Ergebnis einer fehlerhaften Berechnung, wobei das Argument die entsprechende Fehlermeldung enthält

- `thenResult :: Result a -> (a -> Result b) -> Result b`
  führt die Funktion (2. Argument) nur dann aus, wenn das 1. Argument, daß Ergebnis einer erfolgreichen Berechnung ist.

- `mapResult :: (a -> Result b) -> [a] -> Result [b]`
  bietet `map` Funktionalität, auf `Result`

Wie das folgende Beispiel zeigt, gestatten uns die Monade `Result` und die zugehörigen Funktionen, sehr kompakte und übersichtliche Programme zu schreiben.

```
test a b = f1 a ‘thenResult‘ \c ->
           f2 b ‘thenResult’ \d ->
           returnResult (c,d)
```

Die Funktion `test` übergibt ihre Argumente an die Funktionen `f1` und `f2`. Nur wenn beide Funktionen fehlerfrei ausgeführt werden, gibt `test` das Ergebnispaar zurück. Ansonsten wird, je nachdem wo der Fehler auftrat, die Fehlermeldung von `f1` oder `f2` zurückgegeben.

Der Datentyp `FM`[3] realisiert die Umgebungen, die bei der Typinferenz benötigt werden. Die Umgebungen sind als Listen von Paaren implementiert, wobei jedes Paar, einer Zuordnung $name \mapsto information$ entspricht.

```
type FM a = [FMentry a]
type FMentry a = (FastName,a)
data FMresult a = Return  a
                | Bottom
```

Der Datentyp `FMresult` realisiert das Ergebnis einer Suche, in einer Umgebung. Hierbei signalisiert `Bottom` ($\perp$), daß ein angegebenen Name, in einer angegebenen Umgebung, nicht definiert ist. Die wichtigsten Funktionen auf diesem Datentyp sind folgende.

- `mkFM :: [FastName] -> [a] -> FM a`
  erzeugt aus einer Liste von Name und der zugehörigen Liste von Informationen die entsprechende Umgebung.

- `extFM :: FMentry a -> FM a -> FM a`
  fügt eine Eintrag in die angegebene Umgebung ein.

- `combineFM :: FM a -> FM a -> FM a`
  erzeugt aus den angegebenen Umgebungen eine neue, die alle Informationen aus den angegebenen Umgebungen enthält.

[3] „finite map“

- `lookupFM :: FM a -> FastName -> FMresult a`
  liefert aus der angegebenen Umgebung die Information zum angegebenen Name.

Außerdem definieren wir noch folgende Typsynonyme.

```
type Number = #Int
type FastName = Number
```

Hierbei entspricht `FastName` dem Typ der intern verwendeten Namen. Diese werden nach der syntaktischen Analyse, nur noch durch ihren Index in der Bezeichnertabelle angesprochen (s. nächster Abschnitt). Falls das Compiler-Front mit dem GHC[4] übersetzt wird, werden zur Repräsentation von Zahlen, die effizienten *unboxed* Integers verwendet. Bei Verwendung eines anderen Compilers oder des Interpreters Gofer, hat `Number` den gewöhnlichen Typ `Int`.

5.4 Die syntaktische Analyse

Die syntaktische Analyse erfolgt in zwei Schritten. Im erstem Schritt erzeugt der Lexer, aus der Folge von Eingabezeichen, eine Folge von Token. Im zweiten Schritt wird von dem Parser, aus dieser Folge von Token, eine abstrakte Programmdarstellung erzeugt. Diese Darstellung unterscheidet sich von dem üblichen Syntaxbaum, der bei der Überstzung von prozedurealen Sprachen, wie z.B. C, entsteht.

Dies resultiert daraus, daß bei einer funktionalen Sprache die Reihenfolge der Definitionen keine Rolle spielt und daher in Struktur des Programms erst nach dem parsen bestimmt werden kann.

Üblicherweise werden Lexer und Parser nicht von Hand erstellt, sondern mittels Generatoren erzeugt. Wir werden hier den Lexer selbst erstellen, da dies in einer funktionalen Sprache schnell, kompakt und übersichtlich möglich ist. Aus Zeitgründen und zur Wahrung der Übersichtlichkeit, werden wir den Parser von einem Generator erzeugen lassen.

5.4.1 Der Lexer

Der Lexer kennt zwei verschiedene Arten von Token – Token mit und Token ohne Werte. Token ohne Werte existieren für jedes Schlüsselwort (*resid*), jedes Symbol (*symbol*) und für *newline*. Das Token für eine neue Zeile existiert, damit beim Parser, im Falle eines Fehlers, auch die zugehörige Zeile angegeben werden kann. Für *conid*, *varid* und *float*, existieren drei Token, die den zugehörigen Wert erhalten. Bei *float* ist der Wert die

[4]Glasgow Haskell Compiler

angegebene Zahl und dei den beiden Bezeichnern, entspricht dieser Wert einem Index, aus einer Bezeichnertabelle. Diese Bezeichnertabelle wird während der Analyse erzeugt und dient dazu, in späteren Phasen des Front-End, die Bezeichner nur noch über ihren zugehörigen Index anzusprechen. Es existieren zwei solcher Tabelle, jeweils eine für *conid* und eine für *varid*.

Die reservierten Operatoren (*resop*), werden besonders behandelt. Um sie nicht nur binär, sondern auch unär anwenden zu können, werden sie als normales Wort (*varid*) behandelt. Dadurch ist für den Parser jeder reservierte Operator eine ganz normale Funktion und kann deswegen auch unär angewendet werden (z.B. `+ 1`).

Intern arbeitet der Lexer auf der Monade `L`. Diese Monade erledigt die Verwaltung der beiden Bezeichnertabelle.

```
type L a = IDtable -> (a,IDtable)
```

Hierbei ist `IDtable` der Typ der beiden Bezeichnertabellen.

Betrachten wir dazu, wie der Lexer *varid*s behandelt.

```
lexlname :: String -> L [Token]
lexlname cs = case span isValidChar cs of
  ("in",rest)     -> returnL TKin     `noadd` lexer rest
  ("of",rest)     -> returnL TKof     `noadd` lexer rest
  ("let",rest)    -> returnL TKlet    `noadd` lexer rest
  ("dims",rest)   -> returnL TKdims   `noadd` lexer rest
  ("data",rest)   -> returnL TKdata   `noadd` lexer rest
  ("case",rest)   -> returnL TKcase   `noadd` lexer rest
  ("letrec",rest) -> returnL TKletrec `noadd` lexer rest
  (name,rest)     -> returnL name     `addLL` lexer rest
```

Falls ein reserviertes Schlüsselwort vorliegt, wird das entsprechende Token (vom Typ `Token`), über die monadische Funktion `returnL` zurückgegeben. Die monadische Funktion `noadd` sorgt dafür, daß die Bezeichnertabellen für die weitere Analyse (`lexer rest`) zu Verfügung steht. Ist kein Schlüsselwort erkannt worden, wird mittels `addLL`, der Name in die Bezeichnerliste gesucht. Ist er vorhanden, wird der entsprechende Index zurückgegeben. Anderenfalls wird der Name eingetragen und der neue Index zurückgegeben.

5.4.2 Der Parser

Der Parser wird von dem Parser-Generator Happy[GM95] erzeugen. Dieser erzeugt aus einem `yacc`[5]-ähnlichen Eingabefile einen LALR(1) Parser in Haskell. Aus der Tokenlist

[5] Parser-Generator unter UNIX

des Lexer, wird eine abstrakte Programmrepräsentation gemäß Abbildung 4.4 und Abbildung 4.5 erzeugt. Die interne Darstellung folgt der Struktur der abstrakten Syntax. So ist die Deklaration einer Basisdimension über die folgende Produktion festgelegt.

$$
\begin{array}{rcl}
\mathit{dimdecl} & \rightarrow & \textbf{dimfixed}\ \mathit{conid}\ \mathit{number}_1\ \mathit{number}_2 \\
 & | & \textbf{dimany}\ \mathit{conid}
\end{array}
$$

Die interne Darstellung dafür, sieht wie folgt aus.

```
data DimdeclAS = ASdimfixed FastName     -- Name der Basisdimension
                            Number       -- untere Grenze
                            Number       -- obere Grenze
               | ASdimany   FastName     -- Name der Basisdimension
```

In der Syntax von Happy, läßt sich die Produktion *dimdecl* wie folgt schreiben.

```
Dimdecl  : uname '(' num ',' num ')' { ASdimfixed $1 $3 $5 }
         | uname                     { ASdimany $1 }
```

Dabei steht das Nichtterminal `num` für eine Zahle und `uname` für eine Wort, welches mit einem Großbuchstaben beginnt. Die Meta-Variable `$i` repräsentiert das i-te Symbol einer Produktionszeile. Auf das Format des Eingabefiles werden wir nicht näher eingehen, es ist zusammen mit einigen Beispielen im Happy Paket[GM95] beschrieben.

Wie schon angedeutet, erzeugt der Parser keinen von prozeduralen Sprachen bekannten Syntaxbaum. Dies liegt an der Tatsache, daß es auf die Reihenfolge der Definitionen in funktionalen Programmen nicht ankommt. Deshalb müssen wir eine Struktur erzeugen, in der die Informationen eines Programms nach bestimmten Arten geordnet ist. Diese Struktur ist als Datentyp `ExtCorePRG` definiert.

```
data ExtCorePRG = ASExtCore [DimdeclAS] [TypedeclAS] [SigAS] [BindAS]
```

Die Informationen eines Programms werden demnach auf die folgenden 4 Listen verteilt

Liste	gespeicherte Information
`[DimdeclAS]`	Dimensionsdekalrationen
`[TypeDeclAS]`	Typdekalrationen
`[SigAS]`	Typsignaturen von Funktionen
`[BindAS]`	Bindungen

Im Gegensatz zu der Dreiteilung, die bei der Typinferenz angegeben ist (*dimdecls*, *typedecls*, *bgroups*), sind hier die Typsignaturen noch von den zugehörigen Bindungen getrennt. Dies liegt daran, daß diese Zuordnung erst nach dem Durchlauf des Parsers erfolgen kann.

Der Parser verwendt die Monade `Result`, um mit Fehlern umgehen zu können. Der Typ von `parse` sieht daher wie folgt aus.

```
parse :: [Token] -> Result ExtCorePRG
```

5.5 Die statische Analyse

Die statische Analyse ermittelt Informationen, die bei der Typinferenz benötigt werden. Dabei wird zuerst eine Abhängigkeitsanalyse durchgeführt und danach die Top-level Deklarationen analysiert.

5.5.1 Abhängigkeitsanalyse

Das Ziel der Abhängigkeitsanalyse ist es, die Abhängigkeiten der im Programm definierten Funktionen zu ermitteln. Als Beispiel betrachten wir dazu folgendes Fragment eines Programms, bei dem nur die freien Variablen innerhalb einer Defintion angegeben sind.

```
f = ...g...a...
a = ...
g = ...f...
b = ...a...
c = ...c...b...f..
```

Diese Programm enthält die folgenden Bindungsgruppen, die im Kontext der Abhängigkeitsanalyse auch als *starke Zusammenhangskomponenten* bezeichnet werden.

{f,g}, {a}, {b}, {c}

Hierbei ist {f,g} eine verschränkt rekursive Bindungsgruppe und {c} ein Bindungsgruppe, dessen einzige Bindung rekursiv ist. Alle anderen Bindungsgruppen bestehen aus jeweils einer nicht-rekursiven Bindung. In der obigen Darstellung wird nicht zwischen Bindungsgruppen unterschieden, die aus nur einer rekursiven oder nicht-rekursiven Bindung bestehen. Bei der Typinferenz muß aber zwischen rekursiv und nicht-rekursiv unterschieden werden. Daher wird dies bei der Implementierung gesondert betrachtet.

Sortiert man diese bezüglich ihrer Abhängigkeit, den erhält man folgende Reihenfolge[6]

{a}→{f,g}→{b}→{c}

Wir das Programm bei der Typinferenz in dieser Reihenfolge durchlaufen, dann ist gesichert, daß vor der Verwendung einer Funktion, schon dessen Definition überprüft wurde.

Die Abhängigkeitsanalyse realisieren wir durch die Implementierung des Algorithmus von Sharir [Sha81].

[6] In diesem Fall ist die Reihenfolge nicht eindeutig, da es keine Abhängigkeit zwischen {b} und {f,g} gibt. Die folgende Reihenfolge ist daher auch korrekt: {a}→{b}→{f,g}→{c}

```
sccs g = dfs (transposeG g) (reverse (postOrd g))
```

Die Berechnung der starken Zusammenhangskomponenten (`sccs`) des Graphen `g`, erfolgt mittels eine Tiefensuche (`dfs`) auf dem transponierten Graphen, wobei die Startordnung der Kanten durch `reverse (postOrdG g)` festgelegt ist. Dabei liefert `postOrd g`, die Kanten von `g` in Post-Ordnung. Die Korrektheit dieses Algorithmus wird, neben anderen Graphalgorithmen, auch in [KL94] gezeigt.

Da bei der Aufstellung des Graphen `g`, alle Bindungen nach Namen durchsucht werden, wird dies ausgenutzt um gleichzeitig zu überprüfen, ob es Ausdrücke gibt die undefinierte Variablen erhalten. Außerdem erfolgt hier die Zuordnung von Typsignaturen zu den Bindungen.

5.5.2 Top-level Analyse

Die Top-level Analyse dient zur Überprüfung der Top-level Deklarationen *dimdecls* und *typedecls*. Daher besteht die Implementierung der Top-level Analyse, aus der Implementierung der Entscheidungsformen $\overset{dimdecls}{\vdash}$ und $\overset{typedecls}{\vdash}$, sowie von ihnen benutzte Entscheidungsformen (z.B. $\overset{monotype}{\vdash}$). Die verwendeten Umgebungen sind.

Umgebung	Information
DI	Basisdimensionen
TC	Typkonstruktoren
CF	Datenkonstruktoren
CI	Position der Datenkonstruktoren in Typkonstruktoren

Die Umgebung CI wird in den Typregeln nicht erwähnt und ist bei der Typinferenz eigentlich überflüssig. Die Informationen die CI speichert, werden erst bei der Erzeugung des Zwischencode Core gebraucht (Darstellung von Datenkonstruktoren mittels `Pack`). Die Realisierung der einzelnen Umgebung verschieben wir auf den nächsten Abschnitt. Dort wird die Implementierung der Typinferenz mit allen benötigten Umgebungen besprochen.

Betrachten wir nun einmal wie die Entscheidungsform $\overset{typedecls}{\vdash}$ implementiert ist. Die Definiton von $\overset{typedecls}{\vdash}$ sah wie folgt aus.

$$
\textbf{types}\quad \frac{\begin{array}{c} TC \oplus TC_1 \oplus \cdots \oplus TC_{i-1} \overset{typecon}{\vdash} typedecl_i : TC_i \quad 1 \leq i \leq n \\ TC' = TC_1 \oplus \cdots \oplus TC_n \\ (DI, TC', DC \oplus DC_1 \oplus \cdots \oplus DC_{i-1}) \overset{datacon}{\vdash} typedecl_i : DC_i \quad 1 \leq i \leq n \\ DC' = DC \oplus DC_1 \oplus \cdots \oplus DC_n \end{array}}{(DI, TC, DC) \overset{typedecls}{\vdash} typedecl_1 \ldots typedecl_n : (TC', DC') \quad n \geq 0}
$$

Abbildung 5.1 : Typregel aus Kapitel 4

Die Funktion `chkTypedecls` ist die Implementierung von $\overset{typedecls}{\vdash}$.

```
chkTypedecls :: [TypedeclAS] -> DI -> TC -> DC -> Result (TC,DC,CI)
chkTypedecls types di tc cf
 = foldResult chkTypecon tc types                     `thenResult` \tc' ->
   foldResult (chkDatacon di tc') (cf,baseCI) types `thenResult` \(cf',ci) ->
   returnResult (tc',cf',ci)
```

Man erkennt, daß die Funktion `chkTypedecls` die Monade `Result` verwendet wird. Die gesamten Analyse von Top-level Deklarationen ist über diese Monade realisiert. Die monadische Funktion `foldResult`, ist die Implementierung der gewöhnlichen `fold` Funktion auf der Monade `Result`. Die Funktionen `chkTypecon` und `chkDatacon` sind Implementierungen der Entscheidungsformen $\overset{typecon}{\vdash}$ und $\overset{datacon}{\vdash}$.

Wie schon erwähnt, wird die Umgebung CI nur für eine spätere Übersetzung benötigt und ist deswegen in der Typregel **types** nicht aufgeführt.

Die Gegenüberstellung von Definition ($\overset{typedecls}{\vdash}$) und Implementierung (`chkTypedecls`) zeigt deutlich, welche Vorteile sich durch die Verwendung von Monaden ergeben. Aus der Definiton von `chkTypedecls` läßt sich nur die Definition von $\overset{typedecls}{\vdash}$ ablesen. Die notwendige Fehlerbehandlung, wird von der Monade `Result` versteckt.

5.6 Die Typinferenz

Den größten Umfang, nimmt die Implementierung der Typinferenz ein. Zur besseren Übersicht ist diese in die folgenden Bereiche aufgeteilt.

1. Typen und zugehörige Funktionen

2. Umgebugen und zugehörige Funktionen

3. Unifikation und Substitutionen
4. die Typinferenz Monade
5. die Inferenzfunktionen

5.6.1 Typen

Zunächst benötigen wir eine interne Repräsentation der Typen. Wir werden dabei die Trennung zwischen monomorphen und polymorphen Typen beibehalten und zwei Datentypen definieren, die monomorphe bzw. polymorphe Typen repräsentieren.

```
data MonoType = Tyvar FastName
              | Tytup [MonoType]
              | Tyfun MonoType MonoType
              | Tycon FastName [MonoType]
              | Tyco_ FastName [Dim]

data PolyType = Poly [FastName] [FastName] MonoType
```

Die obige Definition entspricht der Definition aus Abbildung 4.6. Ein monomorpher Typ ist entweder eine Typvariable (`Tyvar`), ein Tupel (`Tytup`), ein Funktionstyp (`Tyfun`) oder ein strukturierter Typ (`Tycon`). Der Datenkonstruktor `Tyco_` repräsentiert einen Typkonstruktor mit Dimensionen. Im Gegensatz zu den monomorphen Typen, wo alle Variablen (Typ und Dimensionsvariablen) frei sind, können in polymorphen Typen sowohl freie, als auch gebundenen Variablen auftreten. Daher werden alle gebundenen Variablen eines polymorphen Typ explizit aufgelistet (jeweils eine Liste für Typvariablen und Dimensionsvariablen). Die Implementierung einer Dimensionen sieht wie folgt aus.

```
data Dim = DimVar  FastName Number
         | DimBase FastName Number
```

Ein Datenkonstruktur zur Repräsentation von **0** (keine Dimension), braucht nicht zu Verfügung gestellt werden, da Dimensionen vom Typ `[Dim]` sind. Deshalb wird **0**, durch die leere Liste (`[]`) repräsentiert.

Aufgrund dieser Definitonen, sind die beiden eingebauten Datentypen Num und Bool, wie folgt definieren.

```
tyconNUM  = Tyco_ 1 [DimVar 1]
tyconBOOL = Tyco_ 2 [DimVar 1]
```

Dabei wird vorausgesetzt, daß die Bezeichnertabelle für *conids* als erstes `Num` und als zweites `Bool` enthält.

Betrachten wir einige Beispiele[7]zur internen Darstellung von Typen.

Typausdruck	interne Repräsentation
Num$d \rightarrow \alpha$	`Tyfun (Tyco_ 1 [DimVar 1]) (Tyvar 1)`

$\forall \beta . \alpha \rightarrow \beta \rightarrow (\alpha, \beta)$	`Poly [2] (Tyfun (Tyvar 1)`
	`(Tyfun (Tyvar 2)`
	`(Tytup (Tyvar 1) (Tyvar2)))`

Betrachtet man die Typen von Top-level Funktionen, dann erkennt man, daß diese keine freien Typvariablen enthalten können. Eine Repräsentation mittels `PolyType` wäre für diese Typen nicht sehr geschickt, da die explizite Auflistung der gebundenen Variablen ohne Informationsverlust entfallen kann. Dies hätte aber den Nachteil, daß bei einer Instanziierung eines solchen Typs, jedes mal der gesamte Typ nach den darin vorhandenen Variablen durchsucht werden müßte. Dieses Problem läßt sich lösen, indem nur die Anzahl der Variablen explizit vermerkt wird und alle Variablen aufsteigend durchnummeriert werden. Die Definition des neuen Datentyp sieht daher wie folgt aus.

```
data GVType   = GVType Number Number MonoType
```

Der Datenkonstruktor `GVType` enthält die Anzahl der Typ- und Dimensionsvariablen, sowie den eigentlichen Typ. Die Instanziierung eines solchen Typs erfolgt dadurch sehr schnell. Erst wird die benötigte Anzahl von neuen Dimensions- und Typvariablen erzeugt und danach die alten Variablen, durch die neuen ersetzt. Bei der Benutzung von `PolyType`, muß für diese Ersetzung eine Tabelle erzeugt werden, die explizit angibt, durch welche neue Variable, eine alten Variable ersetzt wird. Diese Tabelle ist bei `GVType` nicht nötig, da hier eine direkte Zuordnung gegeben ist. Jede alte Variable i, wird durch eine neue Variable mit gleicher Nummer ersetzt.

5.6.2 Die Umgebungen

Die Umgebungen werden mittels des Datentyp `FM` (s. Abschnitt 5.2) implementiert. Um die Lesbarkeit zu erhöhen definieren wir folgendes Typ-Synonym.

```
type Env a = FM a
```

Entsprechend den Definitionen aus Abbildung 4.7, ergeben sich dadurch folgende Implementierungen.

[7]Dabei gehen wir davon aus, daß α und d den Index 1 haben und β den Index 2 hat.

```
type DI = Env BaseDimInfo
type TC = Env (Number,MonoType)
type DC = Env GVType
type GV = Env GVType
type LV = Env PolyType
type CI = Env (Number,Number)
type TV = Env FastName
type DV = Env FastName
```

Der Datentyp `BaseDimInfo` enthält die Informationen, der einzelnen Basisdimensionen.

```
data BaseDimInfo = Dimany
                 | Dimfixed Number Number
```

Die Umgebung CI ist für die Typinferenz nicht von Bedeutung, sondern dient lediglich der Aufnahme von Informationen, die bei der Erzeugung des Zwischencode Core, notwendig sind.

$$CI \ : \ conid \mapsto (Index, Stelligkeit)$$

Für jeden Datenkonstruktor *conid* wird vermerkt, an welcher Stelle ($Index$) er innerhalb der Definiton des zugehörigen Typkonstruktors steht und wieviel Argumente ($Stelligkeit$) er benötigt. Die folgende Definition

```
data List a = Nil | Cons a (List a) ;
```

ergibt diese Umgebung $CI \ = \ \{\texttt{Nil} : (1,0) \ , \ \texttt{Cons} : (2,2)\}$.

Um mit den Umgebungen arbeiten zu können bedienen wir uns den schon definierten Funktion auf `FM`. So ist sieht beispielsweise die Implementierung von $\oplus$ wie folgt aus.

```
plusEnv :: Env a -> Env a -> Env a
plusEnv = combineFM
```

Das Suchen innerhalb einer Umgebung, wird durch die Funktion `lookupFM` realisiert.

5.6.3 Die Unifikation

Die Unifikation von zwei Typen, ist durch die Funktion `unify` realisiert.

```
unify :: MonoType -> MonoType -> Unify -> Result Unify
```

Da die Unifikation auch fehlschlagen kann, ist wieder die Monade `Result` im Einsatz. Der Datentyp `Unify` enthält zwei Substitutionen. Jeweils eine für Typvariablen und eine für Dimensionsvariablen.

```
type Unify =(SubsT,SubsD)
```

Die Substitutionen sind über den Datentyp `FM` definiert.

```
type Substitution a  = FM a
type SubsT  = Substitution MonoType
type SubsD  = Substitution [Dim]
```

Die Unifikation ist durch die übliche Fallunterscheidung, gemäß der Defintion in Abbildung 2.7, implementiert.

Die einzige Neuerung besteht darin, daß Unifizieren von 0-stelligen Typkonstruktoren besonders zu behandeln.

```
unify (Tyco_ n dim1) (Tyco_ m dim2) subsU
 = case (n == m) of
   False -> failResult dummyFailFun
   True  -> unifyDim dim1 dim2 subsU
```

Falls es sich um die gleichen Typkonstruktoren handelt, wird `unifyDim` aufgerufen um die beiden Dimensionen zu unifizieren. Die Funktion `unifyDim` führt den Algorithmus aus, der in Abschnitt 3.3 angegeben ist.

5.6.4 Die Typinferenz Monade

Die Monade für die Typinferenz ist recht komplex, da neben der Fehlerbehandlung, auch noch globale Werte benötigt werden. Da wären zunächst einmal die Substitutionen. Diese werden während der Typinferenz erzeugt und müssen für Typvariablen und Dimensionsvariablen verwaltet werden. Außerdem werden noch zwei Zähler benötigt, mit deren Hilfe neue Variablen erzeugt werden können. Diese globalen Werte sind in dem Typ `TIMinternal` zusammengefaßt.

```
type TIMinternal = (FastName,FastName,Subs)
```

Der Typ `Subs` beinhaltet schon beide Substitutionen (s. Unifikation). Um mit dieser internen Struktur arbeiten zu können, muß eine entsprechender Funktionstyp vorhanden sein.

```
type InternalTIM out = TIMinternal -> out
```

Damit können die eigentlichen Monaden-Typen definiert werden. Eine Monade mit und eine Monade ohne Fehlerbehandlung.

```
type TIM a   = InternalTIM (Result (a,TIMinternal))
type TIMNF a = InternalTIM (a,TIMinternal)
```

`TIMNF`[8] wird aus Gründen der Effizienz eingeführt, da es bei der Typinferenz auch Situationen gibt, in denen keine Fehler auftreten können.

Auf diesen Typen sind die normalen Basisfunktionen von Monaden definiert. Betrachten wir einmal den Unterschied zwischen `TIM` und `TIMNF`, am Beispiel der monadischen Funktion `then`.

```
thenTIM :: TIM a -> (a -> TIM b) -> TIM b
f1 `thenTIM` f2 = \timInternal ->
                  f1 timInternal `thenResult` \(a,timInternal') ->
                  f2 a timInternal'

thenTIMNF :: TIMNF a -> (a -> InternalTIM b) -> InternalTIM b
f1 `thenTIMNF` f2 = \timInternal1 ->
                    let (a,timInternal2) = f1 timInternal1
                     in  f2 a timInternal2
```

`thenTIM` arbeitet nur auf `TIM` Werten und verwendet intern die monadische Funktion `thenResult` um mit Fehler umzugehen. Bei `thenTIMNF` ist es so, daß nur das erste Argument vom Typ `TIMNF` sein muß. Das zweite Argument kann vom Typ `TIM` oder vom Typ `TIMNF` sein. Deshalb ist das zweite Argument eine Funktion, dessen Ergebnis vom Typ `InternalTIM b` ist. Das Gesamtergebnis ist natürlich auch vom Typ `InternalTIM b`. Da das erste Argument vom Typ `TIMNF` ist, kann im Unterschied zu `thenTIM`, das Zwischenergebnis direkt übergeben werden.

Die eigentliche Aufgabe der Monade ist aber, den Zugriff auf die internen Werte zu ermöglichen. Dazu sind einige monadische Funktionen definiert, die wir hier nicht alle angegeben werden. Als Beispiel für diese Funktionen, betrachten wir, wie eine neue Typvariable erzeugt wird.

```
newTyvar :: TIMNF MonoType
newTyvar timInternal
 = (mkTyvar tCounter' ,(tCounter',dCounter,subs))
  where (tCounter,dCounter,subs) = timInternal
        tCounter' = tCounter + 1
```

[8] „TIM no fail“

Zuerst wird die internen Struktur aufgetrennt und der Zähler für die Typvariablen entnommen. Dieser Zähler wird erhöht und in die neue interne Struktur eingebaut. Die eigentliche Typvariable wird von der Funktion `mkTyvar` erzeugt. Alle anderen monadischen Funktionen sind nach dem selben Prinzip aufgebaut. Wie diese Funktionen verwendet werden, zeigt eine weitere Ausschnitt aus dem Quelltext.

```
instGVTypeTIMNF :: GVType -> TIMNF MonoType
instGVTypeTIMNF (GVType nrTyvars nrDimvars monoType)
 = newTyvars  nrTyvars  `thenTIMNF` \alphas ->
   newDimvars nrDimvars `thenTIMNF` \ds ->
   returnTIMNF (instGVType alphas ds monoType)
```

Die Funktion `instGVTypeTIMNF` instanziiert einen `GVType` innerhalb der Monadenumgebung. Dazu erzeugt sie über `newTyvars` und `newDimvars` die neuen Variablen und übergibt diese an die Funktion `instGVType`, die die eigentliche Instanziierung vornimmt. Die Funktionen `newTyvars` und `newDimvars` sind nach dem gleichen Schema aufgebaut wie `newTyvar`, mit dem einen Unterschied, daß sie gleich mehrere neue Variablen erzeugen.

5.6.5 Die Inferenzfunktion

Nach diesen umfangreichen Vorarbeiten, gestaltet sich die Implementierung der einzelnen Entscheidungsformen relativ einfach. Durch die intensive Verwendung von monadischen Funktionen, lassen sich die einzelnen Typregeln fast direkt übertragen. Um dies zu demonstrieren, werden wir die Typregel **app** der Entscheidungform $\overset{expr}{\vdash}$ implementieren. Diese Typregel ist in Kapitel 4 wie folgt definiert.

$$\textbf{app} \quad \frac{DI, DC, GV, LV \overset{expr}{\vdash} expr_1 : \tau{\rightarrow}\tau' \quad DI, DC, GV, LV \overset{expr}{\vdash} expr_2 : \tau}{DI, DC, GV, LV \overset{expr}{\vdash} \textbf{app}\ expr_1\ expr_2 : \tau'}$$

Abbildung 5.2 : Typregel aus Kapitel 4

Die Umsetzung sieht wie folgt aus.

```
tiExpr di dc gv lv (ASapp expr1 expr2)
 = tiExpr di dc gv lv expr1                `thenTIM`   \tau2tau' ->
   tiExpr di dc gv lv expr2                `thenTIM`   \tau ->
   newTyvar                                `thenTIMNF` \tau' ->
   (unifyTIM tau2tau' (mkTyfun tau tau'),
    printErrApp expr1 expr2)               `thenTIMF`  \() ->
   returnTIM tau'
```

Die Variablen `di`,`dc`,`gv`,`lv` repräsentieren die jeweiligen Umgebungen und `ASapp expr1 expr2` ist die abstrakte Dartstellung einer Applikation. Die Beziehung zwischen Definition und Implementierung sieht wie folgt aus

Definition	Implementierung
$expr_1$, $expr_2$	`expr1 , expr2`
$\tau \rightarrow \tau'$	`tau2tau'`
τ	`tau`
τ'	`tau'`

Die Struktur der Definition ist in der Implementierung deutlich zu erkennen. Im Unterschied zur Definition, muß bei der Implementierung, die Überprüfung der Typen explizit angegeben werden. Dazu wird eine neue Typvariable erzeugt und der Typ von `expr1` mit dem Funktionstyp `mkTyfun tau tau'` unifiziert. Ist die Unifikation erfolgreich, wird `tau'` als Typ der Applikation zurückgegeben. Da die Unifikation die einzige Stelle ist, an der die Typisierung einer Applikation scheitern kann, wird `unifyTIM` über die monadische Funktion `thenTIMNF` aufgerufen. Dieser Funktion kann eine Fehlermeldung , in diesem Fall `printErrApp expr1 expr2`, übergeben werden, die im Falle eines Fehlers, an die rufende Funktion zurückgegeben wird. Die Funktion `unifyTIM` ist eine Variante der Funktion `unify`, die ein `TIM` Interface besitzt.

Alle anderen Typregeln sind ähnlich aufgebaut. Ihre Implementierung ist im Anhang B angegeben.

5.7 Ausgabefunktionen

Die Informationen die während der Typinferenz gesammelt wurden, können auf 4 verschiedene Arten ausgegeben werden. Zum ersten können die Typen der definierten Funktion, ausgegeben werden. Die Darstellung erfolgt dabei in der gewohnten Weise, wobei die Dimensionsangabe innerhalb von geschweiften Klammern erfolgen.

```
+ :: Num{a}->Num{a}->Num{a}
```

Die Implementierung erfolgt in der übliche Weise und wird deshalb nicht weiter behandelt.

Die zweite Darstellungsart, die Ausgabe der Bindungsgruppen, liefert mehr Informationen über die internen Bearbeitung. Die einzelnen Bindungsgruppen werden in der Reihenfolge ihrer Abhängigkeiten ausgegeben und zu jeder Bindungsgruppe wird angegeben, wieviele Variablen und Substitutionen benötigt wurden. Diese Daten befinden sich in der internen Struktur der Monade `TIM` und werden über die folgende Funktion ausgegeben.

```
printTimInternal :: TIMinternal -> String
printTimInternal (tyvar,dimvar,(subsT,subsD))
 = " <("++show tyvar ++","++show (length (getSubs subsT))++
   ") ("++show dimvar++","++show (length (getSubs subsD))++")>"
```

Die einzelnen Umgebungen die bei der Typinferenz entstanden sind, lassen sich natürlich auch ausgeben. Da alle Umgebungen vom Typ `FM a` sind, kann die Funktion `printFM` verwendet werden. Diese Funktion wendet auf jeden Eintrag in einer Umgebung, eine gegeben Ausgabefunktion an.

```
printEnvs :: IDtable -> DI -> TC -> DC -> CI -> String
printEnvs idTab di tc cf ci
 = "\nDI:\n" ++ (printFM di (\e -> "  "++(printDIEntry idTab e))) ++
   "\nTC:\n" ++ (printFM tc (\e -> "  "++(printTCEntry idTab e))) ++
   "\nDC:\n" ++ (printFM cf (\e -> "  "++(printDCEntry idTab e))) ++
   "\nCI:\n" ++ (printFM ci (\e -> "  "++(printCIEntry idTab e)))
```

Die letzte Art der Ausgabe, gibt das eingegebene ExtCore Programm in dem Zwischencode Core aus. Dieser Code enthält weder Deklarationen noch Typangaben. Da die Form der Ausdrücke von Core, den Ausdrücken von ExtCore sehr ähnlich ist, ist die Erzeugung der Ausgabe relativ einfach. Nur die Datenkonstruktoren müssen anders dargestellt werden. Datenkonstruktorn werden in Core wie folgt dargestellt.

`Pack`$\{tag, arity\}$

Hierbei bezeichnet *arity* die Stelligkeit und *tag* die Position, an welcher der Datenkonstruktor innerhalb des zugehörigen Typkonstruktors definiert wurde. Durch diese Definiton in ExtCore

```
data List a = Nil | Cons a (List a) ;
```

werden die beiden Datenkonstruktoren in Core wie folgt dargestellt

`Nil` = `Pack`{1,0}
`Cons` = `Pack`{2,2}

Um diese Darstellung erzeugen zu können, ist die Umgebung *CI* entwickelt worden. Die Erzeugung dieser Darstellung, zeigt der folgende Programmauschnitt.

```
printExpr :: CI -> IDtable -> String -> ExprAS -> String
printExpr ci idTab spaces expr = case expr of
 (ASnum num)          -> "Pack{"++(show num)++",0}"
 (AStupel exprs)      -> "Pack{1,"++(show (length exprs))++"} "++
                          mapPrint (printExpr ci idTab spaces) " " exprs
 (AScon name)         -> "Pack{"++(printCIinfo ci name)++"}"
```

Hierbei ist `mapPrint` eine Funktion, die eine Ausgabefunktion (hier `printExpr`), auf eine Liste anwendet.

Wie man die Art der Ausgabe steuern kann, wird im nächsten Abschnitt behandelt.

5.8 Das Hauptprogramm

Das Hauptprogramm liest ein Programm in ExtCore ein, analysiert dieses und gibt die angeforderten Informationen zurück. Die Art der Ausgabe kann durch die folgenden Optionen gesteuert werden

`-t`	Ausgabe der Typen aller definierten Funktionen
`-b`	Ausgabe der Bindungsgruppen und ihrer `TIMinternal`
`-c`	Ausgabe des ExtCore Programms als Core Programm
`-e`	Ausgabe der erzeugten Umgebungen
`-n`	Ausgabe der vordefinierten Name

Ferner wird durch die Option `-h`, ein kurzer Benutzungshinweis ausgegeben.

Um das I/O nicht über den übliche, aber sehr unübersichtlichen, continuation-based I/O abwickeln zu müssen, wird eine entsprechende Monade definiert.

```
type NewIO a = (a -> Dialogue) -> Dialogue
```

Über die monadische Funktion `thenNewIO`, läßt sich auf einfache die Reihenfolge von Ausgaben steuern.

```
thenNewIO :: NewIO a -> (a -> NewIO b) -> NewIO b
thenNewIO a b cont = a (\x -> b x cont)
```

Außer `thenNewIO` existieren noch weitere monadische Funktionen, wie z.B. `readFileNewIO` oder `writeFileIO`.

Das Startfunktion `main` zeigt, wie komfortabel sich Ausgaben, mit Hilfe von `NewIO`, in funktionalen Sprachen realisieren lassen.

```
main :: Dialogue
main = exitNewIO
  ( printNewIO mWelcome              `thenNewIO` \() ->
    evalArgs                         `thenNewIO` \setting ->
    getInputFile setting             `thenNewIO` \input ->
    doSyntaxAnalysis input baseTable `thenNewIO` \(extCore,idTab) ->
    mainAnalysis extCore idTab       `thenNewIO` \(gv,di,tc,ci,cf,binds,tim,depi) ->
    printNewIO mOk                   `thenNewIO` \() ->
    doSetting (elems (snd setting))
```

```
               [printTypes idTab gv,
                printBindgroups idTab depi tim gv,
                printBinds ci idTab binds,
                printBuilt_inNames,
                printEnvs idTab di tc cf ci]
  )
```

Die Startfunktion `main` besteht aus einer Folge von Funktionen, die über `thenNewIO` Eingabe oder Ausgaben in der festgelegten Reihenfolge vornehmen. Dies wird innerhalb der Funktion `exitNewIO` vorgenommen, die dafür sorgt, daß das Ergebnis vom Typ `Dialogue` ist.

```
exitNewIO :: NewIO () -> Dialogue
exitNewIO io = io (\_ _ -> [])
```

Die wichtigste Funktion ist `mainAnalysis`. Sie liefert auf Eingabe der abstrakten Syntax und der Bezeichnertabellen (diese werden nur für Fehlermeldungen benötigt), die vollständige Information der Analyse, als 8-Tupel zurück.

```
mainAnalysis :: ExtCorePRG -> IDtable ->
                NewIO (GV,DI,TC,CI,CF,[BindAS],[TIMinternal],DEPI)
mainAnalysis (ASExtCore dims types sigs binds) idTab
 = doDepencyAnalysis binds sigs idTab             `thenNewIO` \(bgroups,depi) ->
   doTopDeclAnalysis dims types idTab             `thenNewIO` \(di,tc,cf,ci) ->
   doTypeInferenz binds sigs di tc cf bgroups idTab `thenNewIO` \(gv,tim) ->
   returnNewIO (gv,di,tc,ci,cf,binds,tim,depi)
```

Die einzelnen Teilanalysen, werden in der schon besprochenen Reihenfolge durchgeführt. Auftretende Fehler werden, auch wenn dies hier nicht zu sehen ist, direkt ausgegeben und die Analyse abgebrochen.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Diese Diplomarbeit untersuchte, inwieweit Typsysteme bei der automatischen Verifikation von Programmen behilflich sein können. Aus der Erkenntnis, daß die bestehenden Typsysteme nur eine sehr begrenzte Form der automatischen Verifikation liefern, entwickelten wir ein erweitertes Typsystem.

Dieses Typsystem basiert auf Dimensionstypen, die als Erweiterung zu den normalen Typen, auch mit den aus der Physik bekannten Dimensionen umgehen können. Um die Vorteile von Dimensionstypen darstellen zu können, wurde die funktionale Sprache ExtCore entwickelt und mit diesem neuen Typsystem ausgestattet. Die Implementierung eines Compiler Front-End für ExtCore erfolgte in Haskell.

6.2 Ausblick

Eine Erweiterung dieser Arbeit, ist in zwei verschiedenen Bereiche möglich.

- Erweiterung der Sprache ExtCore
- Erweiterung des Compiler Front-End für ExtCore

6.2.1 Erweiterung der Sprache

6.2.1.1 Neue Sprachkonstrukte

Der bisherige Sprachumfang von ExtCore entspricht dem, was im Rahmen einer Diplomarbeit möglich ist. Um komfortabel programmieren zu können, ist eine Anpassung an die Sprache Haskell erforderlich. Dies betrifft vor allem das pattern matching und List Komprehensionen. Um dies zu erreichen, sind zwei verschieden Vorgehensweisen denkbar.

1. Das gesamte Compiler Front-End wird auf die erweiterte Sprache angepaßt. Dies bedeutet insbesondere, daß die gesamte Analyse erweitert werden muß.

2. Zwischen dem Parsen und der Analyse wird eine Übersetzung vorgenommen, die die neuen Elemente der Sprache so umwandelt, daß sie in der bisherigen Sprache ausgedrückt werden können.

   Betrachten wir dies am Beispiel der logischen Funktion `and`.

erweiterte ExtCore	ExtCore

`and False x = False ;`
`and True  x = x ;`
	`and x y = case x of`
	`        { False -> False ;`
	`          True  -> x } ;`

Hierbei erscheint ein Vorgehen nach der zweiten Methode, die bessere Lösung zu sein, da keinerlei Veränderungen an der Analyse, dem Kern des Front-End, vorgenommen werden müssen. Außerdem sind die benötigten Übersetzungen schon so oft untersucht worden, daß deren Implementierung, relative einfach, im Vergleich zu einer Erweiterung der gesamten Analyse, zu realisieren ist.

6.2.1.2 Erweiterung des Typsystems

Zunächst einmal erscheint eine Erweiterung der Dimension um Vorsätze (s. Abschnitt 3.1) angebracht. Ein Ausdruck der Form `2000 {M}` wäre dann äquivalent zum Ausdruck `2 {kM}`. Dies kann relativ einfach realisiert werden, indem eine Tabelle angelegt wird, die angibt, welcher Vorsatz welchen Faktor besitzt. Im obigen Beispiel also `k`=1000. Problematisch wird dies aber, wenn diese Vorsätze auch in den Typen auftauchen. So könnten Typen wie `Num {mM}` oder `Num {kM}` entstehen. Eine sinnvolle Lösung scheint daher, die Vorsätze nur bei Dimensionen innerhalb von Ausdrücken zuzulassen. Innerhalb der Ausdrücke können diese Vorsätze, durch Anwendung der entsprechenden Faktoren, eleminiert werden, wodurch der zugehörige Typ keine Vorsätze enthält.

Eine ähnliche Idee ist, eine Basisdimension über eine Menge von Einheiten zu definieren. Dies läßt sich gut am Beispiel von Geldbeträgen demonstrieren. Die Basisdimension `Geld` besteht aus Einheiten, den Währungen, und den zugehörigen Faktoren, den Umrechnungskursen. Dies könnte wie folgt aussehen.

```
dims = Geld { Dollar=1 , DM=1.47 , Pfund=2.4 }
```

Dadurch würde eine Umrechnung während der Analyse durchgeführt werden. Das große Problem an diesem Beispiel ist, daß die Umrechnungskurse und somit die einzelnen Faktoren, nicht konstant sind. Die Umrechnung kann daher nicht zur Übersetzungszeit erfolgen und es müste an den entsprechenden Stellen ein Code eingefügt werden, der zur Laufzeit diese Umrechnung vornimmt. Diese Umrechnung erfolgt, indem aus einer ständig aktualisierten Datenbank, die jeweiligen Umrechnungsfaktoren entnommen werden.

Neben der Erweiterung von Dimensionstypen, ist vor allem die Einführung von *Typklassen* interessant. Durch die Typklassen hat man die Möglichkeit, Funktionen systematisch überladen. Die Einführung von Typklassen, stellt aber eine beachtliche Aufwand dar, da der größte Teil der Typinferenz angepaßt werden müßte. Interessant ist in diesem Zusammenhang, wie Typklassen und Dimensionstypen kombiniert werden können und inwieweit Typklassen durch Dimensionstypen simuliert werden können.

6.2.2 Erweiterung des Compiler Front-End

Zur Zeit erzeugt das Compiler Front-End nur eine Zwischencode in der Sprache Core. Die Kombination mit einem entsprechenden Back-End, wird aber in nächster Zukunft (Mitte 1996) realisiert werden können. Ein Übersetzer von Core in STG[1], wird zur Zeit, im Rahmen einer Diplomarbeit [Res96], entwickelt. In Kombination mit dem schon zu Verfügung stehenden Übersetzer [Kie95] von STG nach C, ist dann eine Übersetzung von ExtCore nach C möglich.

Eine weitere Erweiterung, hin zu einem kompletten Compiler, ist die Integration des Strickheits Analysers aus [Sch94]. Die Strickheits Analyse ist bei nicht strikten Sprachen wie ExtCore so wichtig, weil sie (fast immer) erkennt, welche Funktionen strikt ausgewertet werden können, ohne die Sematik zu verändern. Diese Information kann dann verwendet werden, um effizienteren Code zu erzeugen. Die Integration des Strickheits Analysers, in das Compiler Front-End, dürfte mit vertretbarem Aufwand machbar sein, da er der Sprache Core arbeitet.

[1] eine einfache funktionale Sprache

Anhang A

Beispiel-Analysen

Wir werden in diesem Anhang einige Programme in ExtCore angegben und zeigen, welche Informationen die Typinferenz ermittelt hat.

A.1 Prelude

Das folgende Programm ist ein kleiner Ausschnitt aus der Standard-Prelude von Gofer. Um die einzelnen Funktionen in ExtCore verwenden zu können, wurde das pattern matching in `case`-Ausdrücke übersetzt. Die ermittelten Information werden nach dem Programmtext angegeben.

```
-- Standard Kombinatoren

const k x = k ;
id x = x ;
curry f a b = f (a,b) ;
uncurry f tupel = case tupel of { (a,b) -> f a b } ;
fst tupel = case tupel of { (x,y) -> x } ;
snd tupel = case tupel of { (x,y) -> y } ;
fst3 triple = case triple of { (x,y,z) -> x } ;
snd3 triple = case triple of { (x,y,z) -> y } ;
thd3 triple = case triple of { (x,y,z) -> z } ;
flip f x y = f y x ;

-- Bool'sche Funktionen

and = foldr (&&) True ;
or = foldr (||) False ;
any p list = or (map p list) ;
all p list = and (map p list) ;
otherwise = True ;
```

```
-- Numerische Funktionen

max x y = case (x >= y) of { True  -> x;
                             False -> y } ;
min x y = case (x >= y) of { True  -> y;
                             False -> x } ;
subtract = flip (-) ;
signum x = case (x == 0) of { True  -> 0 ;
                              False -> case (x > 0) of
                                       { True  -> 1 ;
                                         False -> -1 } } ;
sums = scanl (+) 0 ;
products = scanl (*) 1 ;

-- Funktionen auf Listen

data List a = Nil | Cons a (List a) ;

head list = case list of { Cons x xs -> x } ;
last list = case list of { Cons x xs ->
                           case xs of { Nil -> x ;
                                        Cons y ys -> last xs } } ;
tail list = case list of { Cons x xs -> xs } ;
init list = case list of { Cons x xs ->
                           case xs of { Nil -> Nil ;
                                        Cons y ys -> Cons x (init xs) } };
append l1 l2 = case l1 of { Nil -> l2 ;
                            Cons x xs -> Cons x (append xs l2) } ;
length = foldl (\n tmp -> (n+1)) 0 ;
select list pos = case list of
                  { Cons x xs -> case (pos == 0) of
                                 { True -> x ;
                                   False -> select xs (pos - 1) } } ;
iterate f x = Cons x (iterate f (f x))  ;
repeat x = Cons x (repeat x) ;
cycle list = letrec list2 = append list list2
                 in list2 ;
copy n x = letrec xs = Cons x xs
               in take n xs ;
reverse = foldl (flip Cons) Nil ;
concat = foldr (append) Nil ;
transpose = foldr (\xs xss -> zipWith (Cons) xs (append xss (repeat Nil))) Nil ;
null list = case list of { Nil -> True ;
                           Cons x xs -> False } ;
map f list = case list of { Nil -> Nil ;
                            Cons x xs -> Cons (f x) (map f xs) } ;
filter p list = case list of
               { Nil       -> Nil ;
                 Cons x xs -> let xxs = filter p xs
                                 in case (p x) of { True -> Cons x xxs ;
                                                    False -> xxs } } ;
```

```
foldl f z list = case list of { Nil      -> z ;
                                Cons x xs -> foldl f (f z x) xs } ;
foldl1 f list = case list of { Cons x xs -> foldl f x xs } ;
scanl f q list = Cons q (case list of { Nil      -> Nil ;
                                        Cons x xs -> scanl f (f q x) xs } ) ;
scanl1 f list = case list of { Cons x xs -> scanl f x xs } ;
foldr f z list = case list of { Nil      -> z ;
                                Cons x xs -> f x (foldr f z xs) } ;
foldr1 f list = case list of
                { Cons x xs -> case xs of
                               { Nil -> x ;
                                 Cons y ys -> f x (foldr1 f xs) } } ;
scanr f q list = case list of
                 { Nil -> Cons q Nil ;
                   Cons x xs -> let qs = scanr f q xs
                                in Cons (f x (head qs)) qs } ;
scanr1 f list = case list of
                { Cons x xs -> case xs of
                               { Nil -> Cons x Nil ;
                                 Cons y ys -> let qs = scanr1 f xs
                                              in Cons (f x (head qs)) qs } } ;

-- Aufteilung von Listen

take n list = case (n == 0) of
              { True  -> Nil ;
                False -> case list of { Nil -> Nil ;
                                        Cons x xs -> Cons x (take (n - 1) xs) } } ;
drop n list = case (n == 0) of
              { True -> list ;
                False -> case list of { Nil -> Nil ;
                                        Cons x xs -> drop (n - 1) xs } } ;
splitAt pos list = case (pos == 0) of
                   { True -> (Nil,list) ;
                    False -> case list of
                             { Nil -> (Nil,Nil) ;
                               Cons x xs -> let tuple = splitAt (pos - 1) xs
                                            in (Cons x (fst tuple),snd tuple) } } ;
takeWhile p list = case list of
                   { Nil -> Nil ;
                     Cons x xs -> case (p x) of
                                  { True  -> Cons x (takeWhile p xs) ;
                                    False -> Nil } } ;
takeUntil p list = case list of
                   { Nil -> Nil ;
                     Cons x xs -> case (p x) of
                                  { True  -> Cons x Nil ;
                                    False -> Cons x (takeUntil p xs) } } ;
dropWhile p list  = case list of
                   { Nil -> Nil ;
                     Cons x xs -> case (p x) of
                                  { True  -> dropWhile p xs ;
```

```
                                    False -> Cons x xs } } ;
span p list = case list of
              { Nil -> (Nil,Nil) ;
                Cons x xs -> case (p x) of
                             { True -> let y = span p xs
                                           in (Cons x (fst y),snd y) ;
                               False -> (Nil,xs) } } ;
break p list = span (\x -> not (p x)) list ;
zip   = zipWith  (\a b -> (a,b)) ;
zipWith z l1 l2 = case l1 of
                 { Nil -> Nil ;
                   Cons x xs -> case l2 of
                                { Cons y ys -> Cons (z x y) (zipWith z xs ys) } } ;

-- Sonstige Funktionen

until p f x = case (p x) of
              { True  -> x ;
                False -> until p f (f x) }
```

Hier nun die einzelnen Ausgaben.

- Die Typen

```
end :: a
until :: (a->Bool{a})->(a->a)->a->a
zip :: List a->List b->List (a,b)
break :: (a->Bool{a})->List a->(List a,List a)
span :: (a->Bool{a})->List a->(List a,List a)
dropWhile :: (a->Bool{a})->List a->List a
takeUntil :: (a->Bool{a})->List a->List a
takeWhile :: (a->Bool{a})->List a->List a
splitAt :: Num{a}->List a->(List a,List a)
drop :: Num{a}->List a->List a
scanr1 :: (a->a->a)->List a->List a
scanr :: (a->b->b)->b->List a->List b
foldr1 :: (a->a->a)->List a->a
scanl1 :: (a->a->a)->List a->List a
foldl1 :: (a->a->a)->List a->a
filter :: (a->Bool{a})->List a->List a
null :: List a->Bool{a}
transpose :: List List a->List List a
zipWith :: (a->b->c)->List a->List b->List c
concat :: List List a->List a
reverse :: List a->List a
```

```
copy :: Num{a}->a->List a
take :: Num{a}->List a->List a
cycle :: List a->List a
repeat :: a->List a
iterate :: (a->a)->a->List a
select :: List a->Num{a}->a
length :: List a->Num{a}
foldl :: (a->b->a)->a->List b->a
append :: List a->List a->List a
init :: List a->List a
tail :: List a->List a
last :: List a->a
head :: List a->a
products :: List Num->List Num{a}
sums :: List Num{a}->List Num{a}
scanl :: (a->b->a)->a->List b->List a
signum :: Num{a}->Num{b}
subtract :: Num{a}->Num{a}->Num{a}
min :: Num{a}->Num{a}->Num{a}
max :: Num{a}->Num{a}->Num{a}
otherwise :: Bool{a}
all :: (a->Bool{a})->List a->Bool{a}
any :: (a->Bool{a})->List a->Bool{a}
map :: (a->b)->List a->List b
or :: List Bool{a}->Bool{a}
and :: List Bool{a}->Bool{a}
foldr :: (a->b->b)->b->List a->b
flip :: (a->b->c)->b->a->c
thd3 :: (a,b,c)->c
snd3 :: (a,b,c)->b
fst3 :: (a,b,c)->a
snd :: (a,b)->b
fst :: (a,b)->a
uncurry :: (a->b->c)->(a,b)->c
curry :: ((a,b)->c)->a->b->c
id :: a->a
const :: a->b->a
```

- Die Bindungsgruppen

  Das Format der einzelnen Bindungsgruppen ist

  * *Funktionsnamen* <(# Typvariablen , # Substitutionen)
(# Dimensionsvaraiblen , # Substitutionen)>

```
Number of bindinggroups : 58
 * const <(2,0) (0,0)>
 * id <(1,0) (0,0)>
 * curry <(4,1) (0,0)>
 * uncurry <(7,4) (0,0)>
 * fst <(4,2) (0,0)>
 * snd <(4,2) (0,0)>
 * fst3 <(5,2) (0,0)>
 * snd3 <(5,2) (0,0)>
 * thd3 <(5,2) (0,0)>
 * flip <(5,2) (0,0)>
 * foldr <(12,10) (0,0)>
 * and <(4,4) (2,1)>
 * or <(4,4) (2,1)>
 * map <(13,11) (0,0)>
 * any <(7,6) (1,0)>
 * all <(7,6) (1,0)>
 * otherwise <(0,0) (1,0)>
 * max <(5,5) (3,2)>
 * min <(5,5) (3,2)>
 * subtract <(4,4) (1,0)>
 * signum <(7,7) (11,9)>
 * scanl <(16,14) (0,0)>
 * sums <(4,4) (2,1)>
 * products <(4,4) (3,2)>
 * head <(3,2) (0,0)>
 * last <(8,7) (0,0)>
 * tail <(3,2) (0,0)>
 * init <(12,11) (0,0)>
 * append <(11,10) (0,0)>
 * foldl <(12,10) (0,0)>
 * length <(8,7) (3,2)>
 * select <(12,11) (6,5)>
 * iterate <(9,8) (0,0)>
 * repeat <(6,5) (0,0)>
 * cycle <(5,4) (0,0)>
 * take <(18,17) (6,5)>
 * copy <(9,8) (1,0)>
 * reverse <(10,9) (0,0)>
 * concat <(6,5) (0,0)>
 * zipWith <(18,15) (0,0)>
 * transpose <(20,19) (0,0)>
 * null <(4,3) (2,1)>
```

```
* filter <(14,13) (2,1)>
* foldl1 <(9,8) (0,0)>
* scanl1 <(9,8) (0,0)>
* foldr1 <(12,11) (0,0)>
* scanr <(21,19) (0,0)>
* scanr1 <(21,20) (0,0)>
* drop <(14,13) (6,5)>
* splitAt <(25,24) (6,5)>
* takeWhile <(15,14) (2,1)>
* takeUntil <(18,17) (2,1)>
* dropWhile <(14,13) (2,1)>
* span <(22,21) (2,1)>
* break <(8,7) (2,1)>
* zip <(6,4) (0,0)>
* until <(10,9) (2,1)>
* end <(1,0) (0,0)>
```

- Die Umgebungen

```
DI:

TC:
  (1,List a)
  (0,Bool{a})
  (0,Num{a})
DC:
  Cons :: a->List a->List a
  Nil :: List a
  False :: Bool{a}
  True :: Bool{a}
CI:
  Cons: <2,2>
  Nil: <1,0>
  False: <2,0>
  True: <1,0>
```

A.2 Effiziente Verwendung von Variablen

Die Effizienz der Typinferenz hängt im wesentlichen davon ab, wie effizient die Verwaltung der Variablen (Typ- und Dimensionsvariablen) ist. Im allgemeinen kann die Größe eines Typ exponentiell in der Größe des zugehörigen Ausdrucks sein [KMM91].

$$
\begin{aligned}
W_n \;:=\; \lambda\ \mathrm{x}.\mathbf{let}\ & x_0 = (x, x)\ \mathbf{in} \\
& \mathbf{let}\ x_1 = (x_0, x_0)\ \mathbf{in} \\
& \quad \ddots \\
& \qquad \mathbf{let}\ x_n = (x_{n-1}, x_{n-1})\ \mathbf{in}\ x_n
\end{aligned}
$$

Der allgemeinste Typ von W_n enthält $2^{\Omega(n)}$ viele Typvariablen (s. [KMM91]).

Wir wollen an einem ähnlichen Ausdruck testen, wie effizient die Typinferenz von ExtCore mit Typvariablen umgeht. Wir werden im folgenden zwar nur noch von Typvariablen sprechen, damit sind aber auch die Dimensionsvariablen gemeint, da sie auf die selbe Weise behandelt werden wie die Typvariablen.

Der verwendtet Ausdruck ist:

$$
\begin{aligned}
V_n \;:=\; \mathbf{let}\ & x_0 = \lambda x.(x, x)\ \mathbf{in} \\
& \mathbf{let}\ x_1 = \lambda x.x_0(x_0 x)\ \mathbf{in} \\
& \quad \ddots \\
& \qquad \mathbf{let}\ x_n = \lambda x.x_{n-1}(x_{n-1} x)\ \mathbf{in}\ x_n(\lambda x.x)
\end{aligned}
$$

Eine entsprechende Funktion für V_3, sieht in ExtCore wie folgt aus.

```
v3 x = let x0 = \x -> (x,x) in
           let x1 = \x-> x0 (x0 x) in
               let x2 = \x -> x1 (x1 x) in
                   let x3 = \x -> x2 (x2 x) in x3 (\x -> x)
```

Um zu überprüfen, wie effizient die implementierte Typinferenz Variablen verwaltet, wurden die Funktionen V_n in ExtCore implementiert und analysiert. Die folgende Tabelle enthält die Ergebnisse, die durch Ausgabe der Bindungsgruppen erfolgten.

n	# Typvariablen	# Substitutionen
1	10	6
2	15	10
3	20	14
4	25	18
5	30	22

Anhang B

Der Quelltext

Das Compiler Front-End für ExtCore besteht aus den folgenden Modulen.

Modul	Inhalt	Programmzeilen
`Basics`	Basisdatenstruktruen und Funktionen	166
`Built_in`	Eingebaute Funktionen und Datentypen	78
`Core`	Ausgabe in der Sprache Core	84
`Depency`	Abhängigkeitsanalyse	108
`Dims`	Dimensionen und zugehörige Funktionen	228
`Extcore`	interne Repräsentation von ExtCore	191
`Ldfs`	Lazy Depth-First Search	72
`Lexer`	Lexikalische Analyse	207
`Main`	Das Hauptprogramm	193
`Parser`	Der von Happy erzeugte Parser	1611
`Print`	Die Ausgabefunktionen	69
`Subst`	Substitution und zugehörige Funktionen	112
`Tim`	Typinferenz Monade und zugehörige Funktionen	201
`Topdecls`	Überprüfung der Top-level Deklarationen	196
`Type_env`	Umgebungen und zugehörige Funktionen	162
`Type_int`	Typen und zugehörige Funktionen	237
`Type_inf`	Funktionen der Typinferenz	282
`Unifier`	Die Unifikation	102

Aufgrund des großen Umfangs des Quelltextes, sind anschließend nur die Module `Main`, `Tim` und `Type_inf` angegeben. Der komplette Quelltext wird über die folgende Web-Seite zur Verfügung gestellt:

`http://kiste-5.ki.informatik.uni-frankfurt.de/papers/masters.html`

B.1 Das Hauptprogramm

```
> module Main where
> import Basics (Result(..),thenResult,returnResult,foldResult,failResult,
>                caseResult,IDtable(..))
> import Built_in (baseTable,baseEnvs,baseGV,printBuilt_inNames)
> import Lexer (Token,lexer)
> import Extcore (ExtCorePRG(..),DimdeclAS,TypedeclAS,BindAS,SigAS)
> import Parser (parse)
> import Dims (Dim)
> import Ldfs_h (DEPI(..))
> import Depency (Bindgroup,depency)
> import Topdecls (chkTopDecls)
> import Type_env (DI(..),TC(..),DC(..),GV(..),CI(..),BaseDimInfo,domEnv)
> import Type_int (GVType,MonoType)
> import Tim (TIMinternal(..))
> import Type_inf (tiBgroups)
> import Core (printCore)
> import Print (printBindgroups,printTypes,printEnvs)
```

B.1.1 Die Monade NewIO

Die Monade `NewIO` realisiert eine einfache Benutzung des I/O.

```
> type NewIO a = (a -> Dialogue) -> Dialogue

> returnNewIO :: a -> NewIO a
> returnNewIO a cont = cont a

> exitNewIO :: NewIO () -> Dialogue
> exitNewIO io = io (\_ _ -> [])

> doneNewIO :: NewIO a
> doneNewIO cont = \_ -> []

> thenNewIO :: NewIO a -> (a -> NewIO b) -> NewIO b
> thenNewIO a b cont = a (\x -> b x cont)

> thenNewIOResult :: (Result a,IDtable) -> (a -> NewIO b) -> NewIO b
> thenNewIOResult (result,idTab) fkt
>  = result `caseResult` (\prFail -> printErrNewIO (prFail idTab)
>                                    `thenNewIO` \() -> doneNewIO ,
>                         \a -> fkt a)

> readFileNewIO :: String -> NewIO String
> readFileNewIO fname = readFile fname exit'
```

```
> writeFileNewIO :: String -> String -> NewIO ()
> writeFileNewIO fname contents cont = writeFile fname contents exit' (cont ())

> printNewIO :: String -> NewIO ()
> printNewIO message cont = appendChan stdout message exit' (cont ())

> printErrNewIO :: String -> NewIO ()
> printErrNewIO err cont = appendChan stderr ("\n"++err) exit' (cont ())

> getArgsNewIO :: NewIO [String]
> getArgsNewIO = getArgs exit'

> exit' :: FailCont
> exit' err = appendChan stderr ("\n" ++ msg ++ "\n") abort done
>       where msg = case err of
>                   ReadError   s -> s
>                   WriteError  s -> s
>                   SearchError s -> s
>                   FormatError s -> s
>                   OtherError  s -> s
```

B.1.2 Einstellungen

Die vom Benutzer angegebene Ausgabeform wird in `dSetting` gespeichert.

```
> type Setting  = (String,Switches)
> type Switches = Array Int Bool

> dSetting :: Setting
> dSetting = ("",                       -- no filename
>            array (0,4) [0 := False,  -- no types
>                         1 := False,  -- no bindinggroups
>                         2 := False,  -- no core output
>                         3 := False,  -- no built-in names
>                         4 := False]) -- no enviornments

> doSetting :: [Bool] -> [String] -> NewIO ()
> doSetting      []         []      = doneNewIO
> doSetting (switch:bs) (string:ss) = case switch of
>                                      False -> doSetting bs ss
>                                      True  -> printNewIO string `thenNewIO` \() ->
>                                               doSetting bs ss
```

B.1.3 Die Hauptroutine

Die Funktion `main` erledigt die komplette Steuerung des Compiler Front-End. Dabei ist `main` für das Einlesen der Daten und die Umwandlung in die interne Darstellung zuständig.

Die eigentliche Analyse erfolgt in `mainAnalysis`.

```
> main :: Dialogue
> main = exitNewIO
>  ( printNewIO mWelcome               `thenNewIO` \() ->
>     evalArgs                         `thenNewIO` \setting ->
>     getInputFile setting             `thenNewIO` \input ->
>     doSyntaxAnalysis input baseTable  `thenNewIO` \(extCore,idTab) ->
>     mainAnalysis extCore idTab        `thenNewIO` \(gv,di,tc,ci,cf,binds,tim,depi) ->
>     printNewIO mOk                    `thenNewIO` \() ->
>     doSetting (elems (snd setting))
>                       [printTypes idTab gv,
>                        printBindgroups idTab depi tim gv,
>                        printCore ci idTab binds,
>                        printBuilt_inNames,
>                        printEnvs idTab di tc cf ci]
>  )

> mainAnalysis :: ExtCorePRG -> IDtable ->
>                 NewIO (GV,DI,TC,CI,DC,[BindAS],[TIMinternal],DEPI)
> mainAnalysis (ASExtCore dims types sigs binds) idTab
>  = doDepencyAnalysis binds sigs idTab              `thenNewIO` \(bgroups,depi) ->
>    doTopDeclAnalysis dims types idTab              `thenNewIO` \(di,tc,cf,ci) ->
>    doTypeInferenz binds sigs di tc cf bgroups idTab `thenNewIO` \(gv,tim) ->
>    returnNewIO (gv,di,tc,ci,cf,binds,tim,depi)
```

Die einzelnen Teilanalysen geben ihre entsprechenden Meldungen aus und rufen die jeweilige Analysefunktion auf.

```
> doSyntaxAnalysis :: String -> IDtable -> NewIO (ExtCorePRG,IDtable)
> doSyntaxAnalysis input idTab
>  = printNewIO mSyntaxAnalysis `thenNewIO` \() ->
>    lexer input idTab `caseResult`
>    (\prFail         -> printErrNewIO (prFail idTab) `thenNewIO` \ () ->
>                        doneNewIO,
>     \(tkList,idTab) -> parse tkList `caseResult`
>                        (\prFail  -> printErrNewIO (prFail idTab)  `thenNewIO` \ () ->
>                        doneNewIO,
>                         \extCore -> returnNewIO (extCore,idTab)   ) )

> doTopDeclAnalysis :: [DimdeclAS] -> [TypedeclAS] -> IDtable -> NewIO (DI,TC,DC,CI)
> doTopDeclAnalysis dims types idTab
>  = printNewIO mTopDeclsAnalysis            `thenNewIO` \() ->
>    (chkTopDecls dims types baseEnvs,idTab)  `thenNewIOResult` \newEnvs ->
>    returnNewIO newEnvs

> doDepencyAnalysis :: [BindAS] -> [SigAS] -> IDtable -> NewIO ([Bindgroup],DEPI)
> doDepencyAnalysis binds sigs idTab
>  = printNewIO mDepenciesAnalysis              `thenNewIO`      \() ->
```

```
>    (depency binds sigs (domEnv baseGV),idTab)  `thenNewIOResult` \result ->
>    returnNewIO result

> doTypeInferenz :: [BindAS] -> [SigAS] -> DI -> TC -> DC -> [Bindgroup] ->
>                   IDtable -> NewIO (GV,[TIMinternal])
> doTypeInferenz binds sigs di tc cf bgroups idTab
>  = printNewIO mTypeInference                            `thenNewIO `      \() ->
>    (tiBgroups binds sigs di tc cf baseGV bgroups,idTab) `thenNewIOResult` \res ->
>    returnNewIO res
```

B.1.4 Auswertung der Argumente

Die übergebenen Argumente werden überprüft und `dSetting` entsprechend verändert.

```
> evalArgs :: NewIO Setting
> evalArgs = getArgs exit'  `thenNewIO` \args ->
>            case (evalArgs' dSetting args) of
>            Ok globals -> case (testInput globals) of
>                          True  -> returnNewIO globals
>                          False -> printErrNewIO "error: no input file\n" `thenNewIO` \() ->
>                                   doneNewIO
>            Fail prFail -> printErrNewIO (prFail baseTable) `thenNewIO` \() ->
>                           doneNewIO

> evalArgs' :: Setting -> [String] -> Result Setting
> evalArgs' globals  []  = failResult (\_ -> mHelp)
> evalArgs' globals args = foldResult evalArg globals args

> evalArg :: Setting -> String -> Result Setting
> evalArg globals ('-':switch) = evalSwitch globals switch
> evalArg globals    fname     = returnResult (setFileName globals fname)

> evalSwitch (fName,switches) switch
>  | switch == "t" = returnResult (fName,(switches // [0 := True]))
>  | switch == "b" = returnResult (fName,(switches // [1 := True]))
>  | switch == "c" = returnResult (fName,(switches // [2 := True]))
>  | switch == "n" = failResult (\_ -> printBuilt_inNames)
>  | switch == "e" = returnResult (fName,(switches // [4 := True]))
>  | switch == "h" = failResult (\_ -> mHelp)
>  | otherwise     = failResult (\_ -> ("error: unknown switch '"++ switch ++
>                                      "'\n"++ mSwitches) )

> getInputFile :: Setting -> NewIO String
> getInputFile setting
>  = printNewIO ("loading '"++fName++"' ...  ") `thenNewIO` \() ->
>    readFileNewIO fName
>  where fName = getFileName setting
```

```
> testInput :: Setting -> Bool
> testInput (input,bSwitch) = not (null input)

> setFileName (input,bSwitch) new = (new,bSwitch)
> getFileName (input,bSwitch) = input
```

B.1.5 Meldungen

Hier noch die einzelnen Meldungen.

```
> mWelcome           = "Mudit 1.0 by Markus Schmidt\n"
> mSyntaxAnalysis    = "ok\nanalysing syntax ...  "
> mTopDeclsAnalysis  = "ok\nchecking top-level declarations ..."
> mDepenciesAnalysis = "ok\nanalysing depencies ...  "
> mTypeInference     = "ok\ninfering types ...  "
> mOk                = "ok\n\n"

> mHelp     = mUsage ++ mSwitches
> mUsage    = "usage : Mudit <switch1 ... switchN> file\n"
> mSwitches = "switches are : -t  print types of all defined functions"++
>             "\n              -b  print bindinggroups info"++
>             "\n              -c  print core syntax"++
>             "\n              -n  print built-in names"++
>             "\n              -e  print enviornments"++
>             "\n              -h  show this message\n"
```

B.2 Die Typinferenz Monade

```
> module Tim (TIM(..),TIMNF(..),TIMinternal(..),baseTIM,
>             returnTIM,returnTIMNF,failTIM,thenTIM,thenTIMF,thenTIMNF,
>             mapTIM,newTyvar,newTyvars,lookupEnvTIM,lookupEnvTIMNF,
>             lookupEnvLTIMNF,unifyTIM,genMonoTypeTIMNF,genMonoTypesTIMNF,
>             instGVTypeTIMNF,instGVTypeTIM,instPolyTypeTIMNF,
>             setTIMCounter,printTimInternal,applySubsMono,applySubsMonos)
> where

> import Basics (FastName(..),Number(..),IDtable(..),Result(..),FMresult(..),
>                FailFun(..),thenResult,thenResultF,returnResult,failResult,
>                caseResult)
> import Dims (Dim,mkDimVar,mkDimVarList,dimVarList2Dims)
> import Type_int (MonoType,mkTyvar,mkTyvarList,GVType(..),PolyType(..),
>                  instGVType,instPolyType,genMonoType,printMonoType)
> import Type_env (Env(..),LV(..),lookupEnv,lookupEnvNF,fvLV)
> import Subst (Subs(..),SubsT(..),applySubs,initSubs,applySubsLV,
>               setFixVars,getSubs)
> import Unifier (unify,isInst)
```

B.2.1 Definiton der Monade

Die Monade `TIM` verwaltet intern jeweils einen Zähler für Typ- und Dimensionsvariablen, sowie die Substitution. Außerdem steht durch die Kombination mit `Result` eine Fehlerbehandlung zur Verfügung.

`TIMNF` ist eine Variante von `TIM`, bei der keine Fehlerbehandlung erfolgt.

```
> type TIMinternal = (FastName,FastName,Subs)

> type InternalTIM out = TIMinternal -> out
> type TIM a   = InternalTIM (Result (a,TIMinternal))
> type TIMNF a = InternalTIM (a,TIMinternal)

> baseTIM = (0,0,initSubs) :: TIMinternal
```

Hier folgen die gewöhnlichen monadischen Funktionen. Dabei kann bei `caseTIM` eine Fehlermeldung vorgegeben werden.

```
> returnTIM :: a -> TIM a
> returnTIM a = \timInternal -> returnResult (a,timInternal)

> returnTIMNF :: a -> TIMNF a
> returnTIMNF a = \timInternal -> (a,timInternal)

> failTIM :: FailFun -> TIM a
> failTIM pprFun = \timInternal -> failResult pprFun

> thenTIM :: TIM a -> (a -> TIM b) -> TIM b
> f1 `thenTIM` f2 = \timInternal ->
>                   f1 timInternal `thenResult` \(a,timInternal') ->
>                   f2 a timInternal'

> thenTIMF :: (TIM a,FailFun -> FailFun) -> (a -> TIM b) -> TIM b
> (f1,fail) `thenTIMF` f2 = \timInternal ->
>                           (f1 timInternal,fail) `thenResultF` \(a,timInternal') ->
>                           f2 a timInternal'

> thenTIMNF :: TIMNF a -> (a -> InternalTIM b) -> InternalTIM b
> f1 `thenTIMNF` f2 = \timInternal1 ->
>                     let (a,timInternal2) = f1 timInternal1
>                      in  f2 a timInternal2

> caseTIM :: TIM a -> (FailFun -> TIM b,a -> TIM b) -> TIM b
> func `caseTIM` (fail,cont)
>  = \timInternal1 ->
>    case (func timInternal1) of
>    (Ok (result,timInternal2)) -> cont result timInternal2
```

```
>    (Fail prFail) -> fail prFail timInternal1

> mapTIM :: (a -> TIM b) -> [a] -> TIM [b]
> mapTIM _ [] = returnTIM []
> mapTIM f (a:as) = f a         `thenTIM` \b ->
>                   mapTIM f as `thenTIM` \bs ->
>                   returnTIM (b:bs)
```

B.2.2 Monadische Funktionen

Die folgenden Funktionen gestatten es, auf die internen Zähler zuzugreifen.

```
> newTyvar :: TIMNF MonoType
> newTyvar timInternal
>  = (mkTyvar tCounter' ,(tCounter',dCounter,subs))
>   where (tCounter,dCounter,subs) = timInternal
>         tCounter' = tCounter + 1

> newDimvar :: TIMNF Dim
> newDimvar timInternal
>  = (mkDimVar dCounter' 1,(tCounter,dCounter',subs))
>   where (tCounter,dCounter,subs) = timInternal
>         dCounter' = dCounter + 1

> newTyvars :: Number -> TIMNF [MonoType]
> newTyvars nr timInternal
>  | nr == 0   = ([],timInternal)
>  | otherwise = (mkTyvarList (tCounter+1) tCounter',(tCounter',dCounter,subs))
>  where (tCounter,dCounter,subs) = timInternal
>        tCounter' = tCounter + nr

> newDimvars :: Number -> TIMNF [Dim]
> newDimvars nr timInternal
>  | nr == 0   = ([],timInternal)
>  | otherwise = (mkDimVarList (dCounter+1) dCounter',(tCounter,dCounter',subs))
>  where (tCounter,dCounter,subs) = timInternal
>        dCounter' = dCounter + nr

> setTIMCounter :: (FastName,FastName) -> TIMNF ()
> setTIMCounter (n,m) timInternal = ((),(n,m,subs'))
>   where (tCounter,dCounter,subs) = timInternal
>         subs' = setFixVars subs n m

> setTyvarCounter :: FastName -> TIMNF ()
> setTyvarCounter n timInternal = ((),(n,dCounter,subs))
>   where (tCounter,dCounter,subs) = timInternal

> setDimvarCounter :: FastName -> TIMNF ()
```

```
> setDimvarCounter n timInternal  = ((),(tCounter,n,subs))
>   where (tCounter,dCounter,subs) = timInternal
```

Der Zugriff auf die Substitution erfolgt über die nachfolgenden Funktionen.

```
> getSubsTIMNF :: TIMNF Subs
> getSubsTIMNF timInternal = (subs,timInternal)
>   where (tCounter,dCounter,subs) = timInternal

> putSubsTIM :: Subs -> TIM ()
> putSubsTIM subs' timInternal = returnResult ((),(tCounter,dCounter,subs'))
>   where (tCounter,dCounter,subs) = timInternal

> applySubsLVTIMNF :: LV -> TIMNF LV
> applySubsLVTIMNF lv timInternal = (applySubsLV subs lv,timInternal)
>   where (tCounter,dCounter,subs) = timInternal

> applySubsMono :: MonoType -> TIMNF MonoType
> applySubsMono tau timInternal = (applySubs subs tau,timInternal)
>   where (tCounter,dCounter,subs) = timInternal

> applySubsMonos :: [MonoType] -> TIMNF [MonoType]
> applySubsMonos taus timInternal = (map (applySubs subs) taus,timInternal)
>   where (tCounter,dCounter,subs) = timInternal
```

Für einige Funktionen wird ein `TIM`-Interface definiert. So erledigt `instGVTypeTIMNF` die vollständige Instanziierung, indem die benötigten Variablen erzeugt werden und die eigentliche Instanziierungfunktion `instGVType` aufgerufen wird.

```
> instGVTypeTIMNF :: GVType -> TIMNF MonoType
> instGVTypeTIMNF (GVType nrTyvars nrDimvars monoType)
>  = newTyvars  nrTyvars  `thenTIMNF` \alphas ->
>    newDimvars nrDimvars `thenTIMNF` \ds ->
>    returnTIMNF (instGVType alphas (dimVarList2Dims ds) monoType)

> instGVTypeTIM :: GVType -> TIM MonoType
> instGVTypeTIM gvType = instGVTypeTIMNF gvType `thenTIMNF` \mType ->
>                        returnTIM mType

> instPolyTypeTIMNF :: PolyType -> TIMNF MonoType
> instPolyTypeTIMNF (Poly tyVars dimVars monoType)
>  = case (nrPolyTyvars) of
>    0 -> returnTIMNF monoType
>    n -> newTyvars nrPolyTyvars `thenTIMNF` \alphas ->
>             returnTIMNF (instPolyType tyVars alphas monoType)
>  where nrPolyTyvars = length tyVars

> genMonoTypeTIMNF :: LV -> MonoType -> TIMNF PolyType
```

```
> genMonoTypeTIMNF lv tau timInternal = (genMonoType freeLV tau',timInternal)
>   where (tCounter,dCounter,subs) = timInternal
>         tau' = applySubs subs tau
>         freeLV = fvLV (applySubsLV subs lv)

> genMonoTypesTIMNF :: LV -> [MonoType] -> TIMNF [PolyType]
> genMonoTypesTIMNF lv taus timInternal = (map (genMonoType freeLV) taus',timInternal)
>   where (tCounter,dCounter,subs) = timInternal
>         taus' = map (applySubs subs) taus
>         freeLV = fvLV (applySubsLV subs lv)
```

Auch für die Verwendung der Umgebungen werden Funktionen mit einem `TIM`-Interface benötigt.

```
> lookupEnvTIM :: Env a -> FastName -> TIM a
> lookupEnvTIM env name = case (lookupEnv env name) of
>                         Bottom      -> failTIM (\x -> "lookupEnvTIM")
>                         Return info -> returnTIM info

> lookupEnvTIMNF :: Env a -> FastName -> TIMNF a
> lookupEnvTIMNF env name = returnTIMNF (lookupEnvNF env name)

> lookupEnvLTIMNF :: Env a -> [FastName] -> TIMNF [a]
> lookupEnvLTIMNF env names = returnTIMNF (map (lookupEnvNF env) names)
```

Die Funktion `unifyTIM` startet die Unifizierung und setzt bei erfolgreicher Bearbeitung die neue Substitution.

```
> unifyTIM :: MonoType -> MonoType -> TIM ()
> unifyTIM tau tau' timInternal
>  = unify tau tau' (subs,dCounter) `caseResult`
>     (\fail -> failResult (printUnifyError (applySubs subs tau)
>                                           (applySubs subs tau')),
>      \(newSubs,dCounter') -> returnResult ((),(tCounter,dCounter',newSubs)) )
>   where (tCounter,dCounter,subs) = timInternal
```

B.2.3 Ausgabefunktionen

Die erste Funktion gibt die internen Daten von `TIM` aus und die zweite ist die Fehlermeldung von `unifyTIM`.

```
> printTimInternal :: TIMinternal -> String
> printTimInternal (tyvar,dimvar,(subsT,subsD))
```

```
>  = " <("++show tyvar ++","++show (length (getSubs subsT))++
>    ") ("++show dimvar++","++show (length (getSubs subsD))++")>"

> printUnifyError :: MonoType -> MonoType -> IDtable -> String
> printUnifyError tau1 tau2 (uTab,lTab)
>  = "types: "++(printMonoType tau1 uTab)++"\n       "++(printMonoType tau2 uTab)
```

B.3 Die Typinferenz

```
> module Type_inf (tiBgroups)
> where

> import Basics (FastName(..),Number(..),Result(..),FMresult(..),
>                FailFun(..),IDtable(..),
>                thenResult,returnResult,failResult,caseResult,printName)
> import Built_in (tyconNUM)
> import Extcore
> import Topdecls (printErrVarSig)
> import Depency (Bindgroup(..),Signature(..))
> import Type_env (DI(..),TC(..),DC(..),GV(..),LV(..),TV(..),CI(..),
>                  BaseDimInfo,mkLVentry,emptyLV,addEntry,mkArgsLV2,
>                  mkLV,mkMonoLV,LVEntry(..),mkArgsLV,mkArgsLV',mkGV,
>                  mkVarEnv,plusEnv,lookupEnv,lookupEnvNF,chkDimBases)
> import Type_int (GVType(..),MonoType(..),PolyType,
>                  mkTytup,mkTyfun,mkTycon,mkFuntyL,mkFuntyList,instGVType,
>                  mkEmptyPolyType,typeDepth,getLastTypePart,mono2GV,poly2mono,
>                  printMonoType)
> import Topdecls (getMonotype)
> import Tim (TIMinternal(..),TIM(..),thenTIM,returnTIM,failTIM,mapTIM,baseTIM,
>             thenTIMNF,returnTIMNF,thenTIMF,
>             genMonoTypesTIMNF,genMonoTypeTIMNF,
>             instGVTypeTIMNF,instGVTypeTIM,instPolyTypeTIMNF,
>             unifyTIM,newTyvar,newTyvars,setTIMCounter,
>             applySubsMono,applySubsMonos)
> import Unifier (isInst)
```

B.3.1 Bindungsgruppen

Die nachfolgenden Funktionen organisieren die Überprüfung der Bindungsgruppen.

```
> type SigFlag = (FastName,Bool)

> tiBgroups :: [BindAS] -> [SigAS] -> DI -> TC -> DC -> GV -> [Bindgroup]
>         -> Result (GV,[TIMinternal])
```

```
> tiBgroups binds sigs di tc dc gv [bgroup]
>  = tiBgroup di tc dc gv bgroup binds sigs baseTIM `thenResult` \(gv',timInternal) ->
>    returnResult (gv',[timInternal])

> tiBgroups binds sigs di tc dc gv (bgroup:bgroups)
>  = tiBgroup di tc dc gv bgroup binds sigs baseTIM        `thenResult` \(gv1,tcI1) ->
>    tiBgroups binds sigs di tc dc (plusEnv gv gv1) bgroups `thenResult` \(gv2,tcI2) ->
>    returnResult (plusEnv gv1 gv2,tcI1:tcI2)

> tiBgroup :: DI -> TC -> DC -> GV -> Bindgroup -> [BindAS] -> [SigAS] -> TIM GV
> tiBgroup di tc dc gv (NonRecBind sigIdx bindIdx) binds sigs
>  = tiNonRecBind di tc dc gv sigIdx sigs (binds !! bindIdx)

> tiBgroup di tc dc gv (RecBinds sigIdx bindIdx) binds sigs
>  = tiCheckSigs di tc gv sigs sigIdx binds'  `thenTIM` \(lv,sigFlags) ->
>    mapTIM (tiRecBind di dc gv lv) binds'    `thenTIM` \taus ->
>    tiCmpSigTypes lv sigFlags taus
>  where binds' = map (binds !!) bindIdx
```

Die Funktion `tiNonRecBind` überprüft nicht-rekursive Bindungsgruppen. Dabei gibt es zwei verschiedene Vorgehensweisen, je nachdem ob eine Signatur vorgegeben wurde oder nicht.

```
> tiNonRecBind :: DI -> TC -> DC -> GV -> Signature -> [SigAS] -> BindAS -> TIM GV
> tiNonRecBind di tc dc gv NoSig sigs (ASbind var vars expr)
>  = case (null vars) of
>    True -> (tiExpr di dc gv emptyLV expr,printErrFunc var) `thenTIMF` \tau ->
>            createGV tau

>    False -> newTyvars (length vars)                          `thenTIMNF` \taus ->
>             (tiExpr di dc gv (mkArgsLV' vars taus)    expr,
>              printErrFunc var)                               `thenTIMF` \tau ->
>              createGV (mkFuntyL taus tau)
>  where createGV tau = applySubsMono tau  `thenTIMNF` \tau' ->
>                       returnTIM (mkGV var (mono2GV tau'))

> tiNonRecBind di tc dc gv (Sig i) sigs (ASbind var vars expr)
>  = tiSig di tc gv (sigs !! i)                `thenTIM`   \tauSig ->
>    returnTIMNF (mkArgsLV vars tauSig)        `thenTIMNF` \(lv,tau) ->
>    (tiExpr di dc gv lv expr,printErrFunc var) `thenTIMF` \tau' ->
>    testInst var tau tau' tauSig
```

Rekursive Bindungen werden durch die Funktion `tiRecBind` überprüft.

```
> tiRecBind :: DI -> DC -> GV -> LV -> BindAS -> TIM MonoType
> tiRecBind di dc gv lv (ASbind var vars expr)
>  = (tiExpr di dc gv (plusEnv lv lv')    expr,printErrFunc var) `thenTIMF` \tau ->
>    returnTIM tau
>  where lv' = mkArgsLV2 vars (lookupEnvNF lv var)
```

B.3.2 Signaturen

Bei der Überprüfung von verschränkt rekursiven Bindungsgruppen, werden zuerst mit `tiCheckSigs` die Signaturen getestet. Nach der Überprüfung der einzelnen Funktionen stellt `tiCmpSigTypes` fest, ob die Signatur eine Instanz des inferierte Typs ist.

```
> tiCheckSigs :: DI -> TC -> GV -> [SigAS] -> [Signature] -> [BindAS] -> TIM (LV,[SigFlag])
> tiCheckSigs di tc gv sigs signatures [] = returnTIM (emptyLV,[])
> tiCheckSigs di tc gv sigs (signature:signatures) (bind:binds)
>  = tiCheckSig  di tc gv sigs signature  bind  `thenTIM` \(lvEntry,sigFlag) ->
>    tiCheckSigs di tc gv sigs signatures binds `thenTIM` \(lv,sigFlags) ->
>    returnTIM (addEntry lvEntry lv,sigFlag:sigFlags)

> tiCheckSig :: DI -> TC -> GV -> [SigAS] -> Signature -> BindAS -> TIM (LVEntry,SigFlag)
> tiCheckSig di tc gv sigs NoSig (ASbind var vars expr)
>  = newTyvars ((length vars)+1)   `thenTIMNF` \alphas ->
>    returnTIM (mkLVentry var (mkEmptyPolyType (mkFuntyList alphas)),(var,False))
> tiCheckSig di tc gv sigs (Sig i) (ASbind var vars expr)
>  = tiSig di tc gv (sigs !! i)    `thenTIM`   \tauSig ->
>    returnTIM (mkLVentry var (mkEmptyPolyType tauSig),(var,True))

> tiCmpSigTypes :: LV -> [SigFlag] -> [MonoType] -> TIM GV
> tiCmpSigTypes lv [sigFlag] [tau] = tiCmpSigType lv sigFlag tau
> tiCmpSigTypes lv (sigFlag:sigFlags) (tau:taus)
>  = tiCmpSigType  lv sigFlag tau   `thenTIM` \gv ->
>    tiCmpSigTypes lv sigFlags taus `thenTIM` \gv' ->
>    returnTIM (plusEnv gv gv')

> tiCmpSigType :: LV -> SigFlag -> MonoType -> TIM GV
> tiCmpSigType lv (var,signature) tauExprInfered
>  = case signature of
>    True  -> testInst var tauExprSig tauExprInfered tauSig
>    False -> unifyTIM tauExprSig tauExprInfered  `thenTIM`   \() ->
>             applySubsMono tauSig                `thenTIMNF` \tau ->
>             returnTIM (mkGV var (mono2GV tau))
>  where tauSig = poly2mono (lookupEnvNF lv var)
>        tauExprSig = getLastTypePart tauSig
```

Typsignaturen von nicht-rekursiven Bindungsgruppen werden mit `tiSig` überprüft.

```
> tiSig :: DI -> TC -> GV -> SigAS -> TIM MonoType
> tiSig di tc gv sigAS
>  = tiSig' di tc gv sigAS `caseResult`
>    (\prFail      -> failTIM prFail,
>     \(tau,counter) -> setTIMCounter counter `thenTIMNF` \() ->
>                       returnTIM tau )

> tiSig' :: DI -> TC -> GV -> SigAS -> Result (MonoType,(FastName,FastName))
```

```
> tiSig' di tc gv (ASsig var atype)
>  = case (lookupEnv gv var) of
>    Bottom   -> getMonotype di tc dv tv atype `thenResult` \tau ->
>                returnResult (tau,(length tv,length dv))
>    Return n -> failResult (printErrVarSig var)
>  where tv = mkVarEnv (varsinTypeAS atype)
>        dv = mkVarEnv (dimVarsinTypeAS atype)
```

Die Funktion `testInst` wird von `tiCmpSigType` verwendet und testet, ob das zweite Argument eine Instanz des dritten Arguments ist.

```
> testInst :: FastName -> MonoType -> MonoType -> MonoType -> TIM GV
> testInst var tauExprSig tauExprInf tauSig
>  = applySubsMono tauExprInf `thenTIMNF` \tauExprInf' ->
>    case (isInst tauExprSig tauExprInf') of
>    True  -> returnTIM  (mkGV var (mono2GV tauSig))
>    False -> failTIM (printErrNotInst var tauSig tauExprInf')
```

B.3.3 Ausdrücke

Die Überprüfung der Ausdrücke erfolgt gemäß der Typregeln. Dabei ist zu beachten, daß die Überprüfung der Dimensionsgrenzen einer Basisdimension, nur bei einer Applikation erfolgt.

```
> tiExpr :: DI -> DC -> GV -> LV -> ExprAS -> TIM MonoType
> tiExpr di dc gv lv (ASvar name)
>  = case (lookupEnv lv name) of
>    Return sigma -> instPolyTypeTIMNF sigma `thenTIMNF` \tau -> returnTIM tau
>    Bottom       -> instGVTypeTIM (lookupEnvNF gv name)

> tiExpr di dc gv lv (ASnum number) = instGVTypeTIM (GVType 0 1 tyconNUM)

> tiExpr di dc gv lv (AScon conid)
>  = case (lookupEnv dc conid) of
>    Return sigma -> instGVTypeTIM sigma
>    Bottom       -> failTIM (printErrConstr conid)

> tiExpr di dc gv lv (ASnumdim number dims)
>  = returnTIM (instGVType [] [dims] tyconNUM)

> tiExpr di dc gv lv (AScondim conid dims)
>  = case (lookupEnv dc conid) of
>    Return (GVType n m tau) -> instGVTypeTIM (GVType n m tau)
>    Bottom       -> failTIM (printErrConstr conid)

> tiExpr di dc gv lv (ASlambda vars expr)
```

```
>  = newTyvars (length vars)                  `thenTIMNF` \taus ->
>    returnTIMNF (mkArgsLV' vars taus)        `thenTIMNF` \lv' ->
>    tiExpr di dc gv (plusEnv lv lv')  expr   `thenTIM`   \tau ->
>    returnTIM (mkFuntyL taus tau)

> tiExpr di dc gv lv (ASapp expr1 expr2)
>  = tiExpr di dc gv lv expr1                `thenTIM`   \tau2tau' ->
>    tiExpr di dc gv lv expr2                `thenTIM`   \tau ->
>    newTyvar                                `thenTIMNF` \tau' ->
>    (unifyTIM tau2tau' (mkTyfun tau tau'),
>     printErrApp expr1 expr2)               `thenTIMF`  \() ->
>    applySubsMono tau'                      `thenTIMNF` \tauD ->
>    case (chkDimBases tauD di) of
>    True  -> returnTIM tauD
>    False -> failTIM (printErrDimBaseExp (ASapp expr1 expr2) tauD)

> tiExpr di dc gv lv (ASlet defs expr)
>  = tiDefs di dc gv lv defs               `thenTIM` \lv' ->
>    tiExpr di dc gv (plusEnv lv lv') expr `thenTIM` \tau ->
>    returnTIM tau

> tiExpr di dc gv lv (ASletrec defs expr)
>  = tiDefsRec di dc gv lv defs            `thenTIM` \lv' ->
>    tiExpr di dc gv (plusEnv lv lv') expr `thenTIM` \tau ->
>    returnTIM tau

> tiExpr di dc gv lv (AScase expr alts)
>  = tiExpr di dc gv lv expr              `thenTIM`   \tau ->
>    tiAlts di dc gv lv alts              `thenTIM`   \tau2tau' ->
>    newTyvar                             `thenTIMNF` \tau' ->
>    (unifyTIM (mkTyfun tau tau') tau2tau',
>     printErrCase expr)                     `thenTIMF`  \() ->
>    returnTIM tau'

> tiExpr di dc gv lv (AStupel exprs)
>  = mapTIM (tiExpr di dc gv lv) exprs `thenTIM` \taus ->
>    returnTIM (mkTytup taus)



> tiDefs :: DI -> DC -> GV -> LV -> DefsAS -> TIM LV
> tiDefs di dc gv lv (ASdefs vars exprs)
>  = mapTIM (tiDef di dc gv lv) exprs `thenTIM` \sigmas ->
>    returnTIM (mkLV vars sigmas)

> tiDef :: DI -> DC -> GV -> LV -> ExprAS -> TIM PolyType
> tiDef di dc gv lv expr
>  = tiExpr di dc gv lv expr `thenTIM`   \tau ->
>    genMonoTypeTIMNF lv tau    `thenTIMNF` \sigma ->
>    returnTIM sigma

> tiDefsRec :: DI -> DC -> GV -> LV -> DefsAS -> TIM LV
```

```
> tiDefsRec di dc gv lv (ASdefs vars exprs)
>  = newTyvars (length vars)                        `thenTIMNF` \alphas ->
>    returnTIMNF (mkMonoLV vars alphas)             `thenTIMNF` \lv' ->
>    mapTIM (tiExpr di dc gv (plusEnv lv lv')) exprs `thenTIM`   \taus ->
>    genMonoTypesTIMNF lv taus                      `thenTIMNF` \sigmas ->
>    returnTIM (mkLV vars sigmas)
```

```
> tiAlts :: DI -> DC -> GV -> LV -> [AltAS] -> TIM MonoType
> tiAlts di dc gv lv [alt] = tiAlt di dc gv lv alt
> tiAlts di dc gv lv (alt:alts)
>  = tiAlt  di dc gv lv alt  `thenTIM` \tau1 ->
>    tiAlts di dc gv lv alts `thenTIM` \tau2 ->
>    unifyTIM tau1 tau2      `thenTIM` \() ->
>    returnTIM tau1

> tiAlt :: DI -> DC -> GV -> LV -> AltAS -> TIM MonoType
> tiAlt di dc gv lv (ASpatNum number expr)
>  = tiExpr di dc gv lv expr `thenTIM` \tau ->
>    returnTIM (mkTyfun tyconNUM tau)

> tiAlt di dc gv lv (ASpatTup vars expr)
>  = newTyvars (length vars)               `thenTIMNF` \taus ->
>    returnTIMNF (mkArgsLV' vars taus)     `thenTIMNF` \lv' ->
>    tiExpr di dc gv (plusEnv lv lv') expr `thenTIM`   \tau ->
>    returnTIM (mkTyfun (mkTytup taus) tau)

> tiAlt di dc gv lv (ASpatCon conID vars expr)
>  = case (lookupEnv dc conID) of
>    Bottom   -> failTIM (\x -> "unknown constructor in tiAlt")
>    Return sigma -> instGVTypeTIMNF sigma                  `thenTIMNF` \tau' ->
>                    returnTIMNF (mkArgsLV vars tau')        `thenTIMNF` \(lv',tau) ->
>                    tiExpr di dc gv (plusEnv lv lv') expr   `thenTIM`   \tauE ->
>                    returnTIM (mkTyfun tau tauE)
```

B.3.4 Fehlermeldungen

Hier noch die einzelnen Fehlermeldungen.

```
> printErrFunc :: FastName -> FailFun -> FailFun
> printErrFunc var prFail (uTab,lTab)
>  = "error in function '"++(printName var lTab)++"'\n"++(prFail (uTab,lTab))

> printErrNotInst :: FastName -> MonoType -> MonoType -> IDtable -> String
> printErrNotInst var tauSig tauInfered (uTab,lTab)
>  = "error: type signature given to '"++(printName var lTab)++
>    "' is not an instance of the infered type !\nsignature: "++
```

```
>    (printMonoType tauSig uTab)++"\ninfered: "++
>    case (n == m) of
>    True  -> printInfered
>    False -> "...->"++printInfered
>  where printInfered = printMonoType tauInfered uTab
>        n = typeDepth tauSig ; m = typeDepth tauInfered

> printErrConstr :: FastName -> IDtable -> String
> printErrConstr conID (uTab,lTab)
>  = "error: unknown constructor "++(printName conID uTab)

> printErrApp :: ExprAS -> ExprAS -> FailFun -> FailFun
> printErrApp expr1 expr2 prFail idTab
>  = "type error in application e1 e2\ne1: "++(printExprAS idTab expr1)++
>    "\ne2: "++(printExprAS idTab expr2)++"\n"++(prFail idTab)

> printErrCase :: ExprAS -> FailFun -> FailFun
> printErrCase expr prFail idTab
>  = "type error in case expression case e of alts\ne: "++(printExprAS idTab expr)++
>    "\nalts: { ... }\n"++(prFail idTab)

> printErrDimBaseExp :: ExprAS -> MonoType -> FailFun
> printErrDimBaseExp expr tau (uTab,lTab)
>  = "error: dimbase out of bounds !\n   expr: "++(printExprAS (uTab,lTab) expr)++
>    "\n   type: "++(printMonoType tau uTab)
```

Literaturverzeichnis

[Bar84] H.P. Barendregt . The lambda calculus, its syntax and semantics. *North Holland Publishing Company, Amsterdam 1984.*

[BD92] F. Baader, J.H. Siekmann. Unification Theory. In D.M. Gabbay, C.J. Hogger and J.A. Robinson (eds.) *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, 1992.

[Car85] R. Cartwright. Types as Intervals. In *Proc. 12th ACM Symposium on Principles of Programming languages*, 22-36, 1985.

[CDDK86] D. Clement, J. Despeyroux, T.Despeyroux, G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming* 13-27, 1986.

[Chu41] A. Church . A formalisation of the simple theory of types. In *Journal of Symbolic Logic* 5,56-68.

[Cur34] H.B. Curry . Functionality in combinatory logic. In *Proc. Nat. Acad. Science USA* 20,584-590.

[CW85] L. Cardelli, P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471-522, Dec. 1985.

[DM82] L. Damas, R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming languages*, 207-212, 1982.

[GM95] A. Gill, S. Marlow. Happy Manual second Edition, Happy version 0.8 `http://www.dcs.gla.ac.uk/fp/software/happy.html`

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. ACM Symposium Lisp and Functional Programming*, 1988.

[Hen93] F. Henglein. Type Inference with Polymorphic Recursion. *ACM TOPLAS 12,* 2(Apr. 93), 253-289.

[HPW92] P.Hudak, S.L. Peyton Jones, P. Wadler editors. Report on the Programming Languages Haskell, Version 1.2. *SIGPLAN Notices, May 1992.*

[Jon94] M. P. Jones. Gofer 2.30. `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`

[Kie95] F. Kiesel. Implementierung eines Übersetzers zur Erzeugung von effizientem C-Code aus funktionalem STG-Code. *Diplomarbeit*, Johann Wolfgang Goethe-Universität, Frankfurt am Main, April 1995.

[KL94] D.J. King, J. Launchbury. Lazy Depth-First Search and Linear Graph Algorithms in Haskell. In J. T. O'Donnell and Kevin Hammond, editors, *Glasgow functional programming workshop*, Springer Verlag, 1993.

[KMM91] P. Kanellakis, H. Mairson, J. Mitchell. Unification and ML type reconstruction. In *Computational Logic – Essays in Honor of Alan Robinson*, J.L. Lassez and G. Plotkin, Eds. MIT Press, 1991.

[KMNS88] D. Kapur, D. Musser, P. Narendran, J. Stillman. Semi-unification. In *Proc. 8th Conference on Foundation of Software Technology and Theoretical Computer Science*, Prune, India, 1988.

[Knu69] D. Knuth. *The Art of Computer Programming, Vol 2.* Addison-Wesley, 1969.

[KTU88] A.J. Kfoury, A. Tiuryn, P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Proc. 15th ACM Symposium Principles of Programming Languages*, 58-59, 1988.

[KTU89] A.J. Kfoury, A. Tiuryn, P. Urzyczyn. Computational consequences and partial solutions of a generalized unification problem. In *Proc. of IEEE 4th Symposium Logic in Computer Science*, 98-105, 1989.

[KTU90] A.J. Kfoury, A. Tiuryn, P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. ACM Symposium Theory of Computing*, 1990.

[KTU93] A.J. Kfoury, A. Tiuryn, P. Urzyczyn. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM TOPLAS 12,* 2(Apr. 93), 290-311.

[LBB84] D. Lankford, G. Butler, B. Brady. Abelain group unification algorithms for elementary terms. *Contemporary Mathematics,* 29:193-199, 1984.

[Läu93] K. Läufer. Polymorphic Type Inference and Abstract Data Types. *PhD thesis*, New York University, Department of Computer Science, July 1992.

[Mil78] R. Milner . A theory of type polymorphism in programming.*Journal of Computer System Sciences*, 17(3), 1978.

[MPS86] D. MacQueen, G. Plotkin, R. Sethi. An Ideal Model for Recursive Polymorphic Types. *Information and Control* **71**, 95-130, 1986.

[Myc84] A. Mycroft . Polymorphic type schemes and recursive definitions. In *Proc. of the International Symposium on Programming*, Toulouse 1984 , LNCS 167.

[PJL92] S.L. Peyton Jones, D. Lester. *Implementing Functional Languages*. Prentice Hall, 1992.

[PJW92] S.L. Peyton Jones, P. Wadler. A static semantic for Haskell. *Unpublished Technical Repot*, University of Glasgow, 1992.

[Res96] B. Rest. *Diplomarbeit in Vorbereitung*, Johann Wolfgang Goethe-Universität, Frankfurt am Main.

[Rob65] J.A. Robinson. A Machine-oriented Logic based on the Resolution Principle *J.ACM* **12** 1, 23-41.

[Sch94] M. Schütz. Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache. *Diplomarbeit*, Johann Wolfgang Goethe-Universität, Frankfurt am Main, August 1994.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and mathematics with applications* **7**(1), 67-72.

[SS71] D. Scott, C. Strachey . Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata* 19-46 , Brooklyn , New York 1971. MRI Symp. Proc's, Vol. XXI.

[Tiu92] J. Tiuryn. Type Inference Problems: A Survey. In *Proc. 15th Symp. on Mathematical Foundations of Computer Science*, LNCS 452, 1990.

[Wad92] P. Wadler . The essence of functional programming. In *Proc. 19th ACM Symposium on Principles of Programming languages*, 1992.

Jetzt ist aber wirklich Schluß !