3 Semantische Analyse

Ein Variablenname, etwa „`listenElement`“, repräsentiert in einer Programmiersprache ein Objekt, das mehrere Eigenschaften besitzt:

1) Eine **Speicheradresse**, ab der dieses Objekt im Speicher abgelegt ist. Diese Speicheradresse wird vom Compiler, manchmal in Verbindung mit dem Linker, festgelegt und kann sich während des Programmlaufs durchaus ändern (etwa bei lokalen Variablen, die in rekursiven Prozeduren auftreten).

2) Eine Codierung des Objektes selbst, d.h. die zugeordneten Speichereinheiten und das darin abgelegte Bitmuster, das den **Wert** des Objektes darstellt.

Um den Wert eines Objektes manipulieren zu können, muß man wissen, wie lang diese Codierung ist und wie man sie zu interpretieren hat. Diese Information zu dem Objekt wird durch eine **Typ-Information** festgelegt und dem Objekt als *tag* oder der Variablen per Deklaration zugeordnet. In den meisten Programmiersprachen wird dies durch eine Typ-Angabe bei der Variablendeklaration eingeführt. In objektorientierten Sprachen wird der Begriff „Typ“ duch „Klasse“ ersetzt.
Jeder Teilausdruck eines arithmetischen Ausdrucks hat natürlich ebenfalls einen Typ (oder sogar bei gewissen Programmiersprachen mehrere Typen, z.B. sind nicht negative Integer in Modula-2 vom Typ `INTEGER` und `CARDINAL`). Die in der Programmiersprache verfügbaren Operatoren verlangen üblicherweise Operanden eines speziellen Typs. Ein Compiler kann in gewissem Rahmen prüfen, ob die durch die Sprachdefinition der Programmiesprache gegebenen Beschränkungen bzgl. der Typen im vorgelegten Programm eingehalten werden (**Type-Checking**) und ob eventuell Typen automatisch umgewandelt werden müssen.
Bei einer Programmiersprache mit **starker Typ-Prüfung** (strongly typed language) kann vom Compiler im gewissen Umfang garantiert werden, dass kein Typ-Fehler zur Laufzeit auftritt, also die Operanden einer Operation den „richtigen“ Typ haben.
Diese Typ-Prüfung kann, speziell bei einfachen Typregeln, während des Parsings bzw. bei der Zwischencode-Erzeugung geschehen (etwa Pascal) oder aber auch bei entsprechend komplexen Regeln (etwa Ada, C^{++} oder Java) in einem zusätzlichen Pass des Compilers geschehen (**statische Typ-Prüfung**).
Andererseits kann diese Überprüfung auch ganz oder teilweise erst zur Laufzeit durchgeführt werden (**dynamische Typ-Prüfung**). Man denke etwa an vom Compiler generierten Code zum Testen auf Bereichsüberschreitungen bei Arrayzugriffen auf dynamische Arrays oder die dynamische Auswahl virtueller Funktionen in C^{++} (dynamisches Binden).
Auch gibt es Sprachen (etwa Smalltalk oder Scheme), in denen keine Typdeklaration vorgesehen ist. In diesen Fällen ist jedem Datenobjekt auch eine Kodierung des Typs (bzw. in Smalltalk der Klasse) zugeordnet und diese Kodierung muß ebenfalls gespeichert werden (**tag-System**).
Eine andere Klasse von Sprachen, wie etwa ML, Caml oder Haskell oder neuerdings Swift, erlaubt die Deklaration von Variablen ohne Angabe eines Typs, führt aber trotzdem eine starke Typ-Prüfung durch. Dies geschieht durch ein komplexes System, bei dem durch logisches Schließen durch den Gebrauch der Variablen auf deren Typ geschlossen wird und damit eine konsistente Verwendung der Variablen gewährleistet wird.
Den Prozeß des Ableitens von Typen für Konstrukte der Programmiersprache aus dem Gebrauch der Konstrukte nennt man **Typ-Inferenz**. Zum Beispiel kann aus der Anweisung `x := 1.0` geschlossen werden, dass `x` vom Typ `real` sein muss und dass daher der nachfolgende Teilausdruck `a[x] := 0` zu einer Fehlermeldung wegen Typ-Unverträglichkeit führt.

Wird der Typ eines Ausdrucks dagegen aus den bekannten Typen der Operanden bestimmt, spricht man von **Typ-Synthese**. Diese Vorgehensweise erfordert die Deklaration von Namen vor ihrer Verwendung.

3.1 Typ-Ausdrücke

Es soll im folgenden jedem Objekt des Programms ein **Typ-Ausdruck** zugeordnet werden. Typ-Ausdrücke sind Terme, die mittels Konstruktoren aus elementaren Typen gebildet werden. Dazu werden rekursiv ausgehend von sogenannten **Basistypen** immer komplexere Typ-Ausdrücke aufgebaut.
Was ein Basistyp ist, hängt von der speziellen Programmiersprache ab; so besitzt z.B. FORTRAN 90 den Basistyp *complex*, Pascal jedoch nicht. Das **Typ-System** einer Programmiersprache ist ein Satz von Regeln, mit denen allen Objekten eines Programms ein Typ-Ausdruck zugeordnet werden kann.

Definition einfacher Typ-Ausdrücke

1) Ein Basistyp ist ein Typ-Ausdruck. Zu den Basistypen gehören etwa: *boolean*, *char*, *integer*, *real*, *void*.
   Bemerkung: Auch Aufzählungen wie etwa `1..7` oder `(mo, di, mi, do, fr)` und Unterbereiche von einigen Basistypen sowie der spezielle (Fehler)-Typ *type-error* können als Basistypen angesehen werden. Aufzählungs- und Unterbereichstypen bringen jedoch viele Probleme in das Typ-System und werden daher in modernen Sprachen meist nicht mehr zu den Basistypen gezählt.

2) Sind Namen als Abkürzungen für Typen erlaubt, dann sind diese Namen ebenfalls Typ-Ausdrücke.

   Beispiel: Durch `type Zeichen = char` wird *Zeichen* ein Typ-Ausdruck.

3) Ist T ein Typ-Ausdruck, so ist *array(T)* ebenfalls ein Typ- Ausdruck, der einen Feld-Typ beschreibt.

   Bemerkung: *array* ist ein Beispiel eines **Typ-Konstruktors**, der einen neuen Typ erzeugt. In manchen Programmiersprachen betrachtet man den Indexbereich des Feldes als mit zum Typ gehörig. In diesem Fall hätte der Typ-Ausdruck die Form *array(I,T)*, wobei I ein Typ-Ausdruck ist, der einen Unterbereich von *integer* oder eine Aufzählung bezeichnet. Ist der untere Index des Feldes fest, so reicht es aus, die Länge des Feldes mit in den Typ-Ausdruck aufzunehmen.

   Beispiel: Durch die Deklaration `var A : array[1..10] of integer` ist *array(integer)* oder auch *array(10,integer)* Typ-Ausdruck für `A`.

4) Ist T ein Typ-Ausdruck, so ist *list(T)* ebenfalls ein Typ-Ausdruck, der einen Listen-Typ beschreibt.
   Dieser Typ-Konstruktor wird natürlich nur dann benötigt, wenn Listen elementare Datenstrukturen der Programmiersprache sind.

5) Sind $T_1, \ldots, T_r$ Typ-Ausdrücke und sind $\texttt{name}_1, \ldots, \texttt{name}_r$ unterschiedliche Namen, dann ist *record(*$name_1$*:*T_1 $\times \cdots \times$ $name_r$*:*T_r*)* ein Typ-Ausdruck, der eine Record-Konstruktion beschreibt.

Beispiel:
```
type t=record
    x:integer;
    y:array[1..15] of char
end
```

ordnet `t` den Typ-Ausdruck *record(x: integer* × *y: array(15,char))* zu.

Bemerkung Analog würde man in obejektorientierten Sprachen einen Typ-Konstruktor für neue Klassen definieren.

6) Sind $T_1, \ldots, T_r$ Typ-Ausdrücke mit $r \geq 1$, dann ist $T_1 \times \ldots \times T_r$ ein Typ-Ausdruck, der ein Tupel mit r Komponenten von Typ $T_1, \ldots, T_r$ beschreibt.

   Bemerkung: Mit diesem Konstrukt kann man etwa die Typen der Parameter einer Funktion zu einem Typ zusammenfassen oder aber auch bei Funktionen mit mehreren Rückgabewerten die Typen festlegen.

7) Ist T ein Typ-Ausdruck, dann ist *pointer(T)* ein Typ-Ausdruck, der einen Zeigertyp auf ein Objekt vom Typ T beschreibt.

   Beispiel: `var p:↑integer` ordnet der Variablen `p` den Typ-Ausdruck *pointer(integer)* zu. Dieser Typ-Konstruktor wird natürlich nur benötigt, wenn die Programmiersprache die Verwendung von Pointer-Variablen zulässt.

8) Sind T_1 und T_2 Typ-Ausdrücke, dann ist $T_1 \rightarrow T_2$ ein Typ- Ausdruck, der eine Funktion mit einem Parameter vom Typ T_1 beschreibt, die ein Objekt vom Typ T_2 zurückgibt.

   Bemerkung: Mit der Definition von Tupeln kann man natürlich auch Prozeduren mit mehreren Parametern definieren,

   Man beachte, dass der Operator × linksassoziativ ist und eine größere Priorität als der rechtsassoziative Operator → hat. Daher ist der Typ-Ausdruck
   char×*char* → *integer* → *pointer(integer* → *integer)* als
   (char×*char)* → *(integer* → *pointer(integer* → *integer))* zu interpretieren.

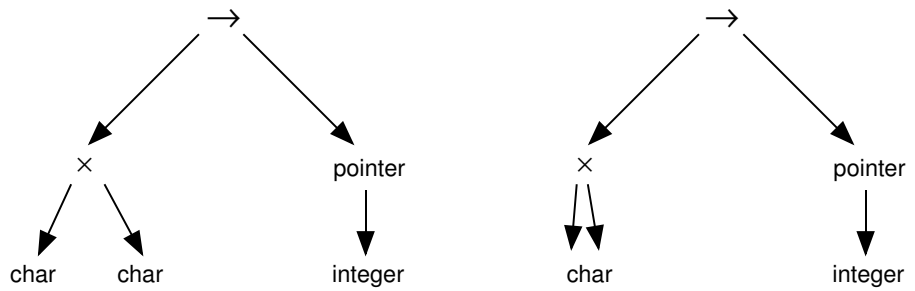
Beispiel 3.1:
Die Deklaration `function f(a,b:char): ↑integer` in Pascal oder die äquivalente Deklaration `int *f(char a, char b)` in C liefert als Typ-Ausdruck

char×*char* → *pointer(integer).*

während in C die Deklaration `int (*f)(char a, char b)` den Typausdruck

pointer (char×*char* → *integer)*

liefern würde.
Typ-Ausdrücke werden auch häufig als Bäume oder Graphen dargestellt. Man erhält Graphen, wenn man Teilausdrücke eines Typausdrucks nicht doppelt aufführt, sondern mehrere Kanten auf diesen Teilausdruck zulässt wie etwa in der rechten Abbildung:



Beispiel 3.2:
Als Beispiel soll hier die Kodierung von Typ-Ausdrücken im Java class File Format vorgestellt werden. Typ-Ausdrücke werden dort „Descriptoren“ genannt. Typ-Ausdrücke werden als Zeichenketten im UTF-8 Format kodiert. Es gibt eine Reihe von Basistypen und nur einen Typ-Konstruktor:

Typ	Kodierung	Bemerkung
byte	`B`	signed byte
char	`C`	Unicode Zeichen
double	`D`	Floating-Point-double
float	`F`	Floating-Point single
int	`I`	integer
long	`J`	long integer
reference	`L<classname>`	Objekt der Klasse <classname>
short	`S`	short
boolean	`Z`	boolean
reference	`[`	array

Der `<classname>` repräsentiert dabei einen vollen qualifizierten Klassen- oder Interfacenamen. Der kodierte Typ-Ausdruck für eine Instanzenvariable vom Typ Integer ist also einfach `I`. Der Typ-Ausdruck für eine Instanzenvariable vom Typ `Object` ist `Ljava/lang/Object;` .
Hat man ein 2-dimensionales Feld `d` durch `float d[][]` deklariert, ist der zugehörige Typ-Ausdruck `[[F`.
Die Kodierung von Typ-Ausdrücken von Methoden haben die Form:

(<Typ-Ausdrücke der Parameter >) <Typ-Ausdruck des Rückgabewerts>

Der Rückgabetyp `void` wird dabei mit `V` kodiert.
Der Deklaration `Object myMethod (int i, double d, Thread t)` würde also die Kodierung `(IDLjava/lang/Thread;)Ljava/lang/Object;` zugeordnet.

3.2 Einfache Typ-Synthese

Den Prozess des Ableitens von Typ-Ausdrücken für Konstrukte der Programmiersprache aus verfügbaren Typinformationen nennt man **Typ-Synthese**. Man diesen Vorgang bei einfachen Programmiersprachen während des Parsens durchführen.
Als Beispiel soll ein SDTS angegeben werden, das diese Aufgabe erfüllt. Als einziges Attribut tritt hier das synthetische Attribut *type* auf. Im Deklarationsteil wird das Attribut benutzt, um den Typ-Ausdruck der Deklaration zu konstruieren. Im beispielhaften Ausführungsteil enthält das Attribut *type* den Typ-Ausdruck des Teilausdrucks. Bei der Anwendung von Operationen wird geprüft, ob die Operanden den „richtigen“ Typ haben. Andernfalls wird eine Fehlermeldung ausgegeben.

Ein einfaches Übersetzungsschema zur Typ-Synthese und Typ-Prüfung

$$
\begin{array}{rl}
P \to D;E & \\
D \to D;D & \\
D \to \texttt{id} : T & \{\ \texttt{addtype}(\texttt{id}.entry, T.type)\ \} \\
T \to \texttt{char} & \{\ T.type := char\ \} \\
T \to \texttt{integer} & \{\ T.type := integer\ \} \\
T \to \uparrow T_1 & \{\ T.type := pointer(T_1.type)\ \} \\
T \to \texttt{array[num] of}\ T_1 & \{\ T.type := array(\texttt{num}.val, T_1.type)\ \} \\
E \to \texttt{literal} & \{\ E.type := char\ \} \\
E \to \texttt{num} & \{\ E.type := integer\ \} \\
E \to \texttt{id} & \{\ E.type := \texttt{lookup}(\texttt{id}.entry)\ \} \\
E \to E_1\ \texttt{op}\ E_2 & \{\ E.type := \texttt{if}\ E_1.type = integer\ \texttt{and}\ E_2.type = integer \\
 & \quad \texttt{then}\ integer\ \texttt{else}\ type\text{-}error\ \} \\
E \to E_1[E_2] & \{\ E.type := \texttt{if}\ E_2.type = integer\ \texttt{and}\ E_1.type = array(s,\ t) \\
 & \quad \texttt{then}\ t\ \texttt{else}\ type\text{-}error\ \} \\
E \to E_1 \uparrow & \{\ E.type := \texttt{if}\ E_1.type = pointer(t) \\
 & \quad \texttt{then}\ t\ \texttt{else}\ type\text{-}error\ \}
\end{array}
$$

Bemerkungen zum Übersetzungsschema:

- Hier wird angenommen, dass Felder als unteren Index immer die 1 haben und die Feldlänge Teil des Typs sind.
- `addtype(st, T)` speichert den Typ-Ausdruck T in dem Eintrag der Symboltabelle, auf den `st` zeigt. Bei Programmiersprachen, in denen eine Vielzahl von Typen definiert werden kann, wird der Compiler alle definierten Typen in einer **Typ-Tabelle** festhalten und in der Symboltabelle nur einen Verweis auf den entsprechenden Eintrag in der Typ-Tabelle machen.
- `lookup(st)` liefert den Typ-Ausdruck des Eintrags in der Symboltabelle, auf den `st` verweist.
- Als Folge der Seiteneffekte befinden sich für dieses Übersetungsschema die Informationen in der Symboltabelle bevor sie abgefragt werden.
- Durch Produktionen wie etwa $E \to E < E$ könnte man Ausdrücke vom Typ *boolean* erzeugen.

Erweiterung des vorangehenden Beispiels auf Wertzuweisungen (Wertzuweisungen haben hier angenommen den zugeordneten Basis-Typ *void*):

$$
\begin{array}{rl}
P \to D\,;\,S & \\
S \to \texttt{id} := E & \{\ S.type := \texttt{if lookup(id}.entry) = E.type \\
 & \quad \texttt{then}\ void\ \texttt{else}\ type\text{-}error\ \} \\
S \to \texttt{if}\ E\ \texttt{then}\ S_1 & \{\ S.type := \texttt{if}\ E.type = boolean \\
 & \quad \texttt{then}\ S_1.type\ \texttt{else}\ type\text{-}error\ \} \\
S \to \texttt{while}\ E\ \texttt{do}\ S_1 & \{\ S.type := \texttt{if}\ E.type = boolean \\
 & \quad \texttt{then}\ S_1.type\ \texttt{else}\ type\text{-}error\ \} \\
S \to \texttt{begin}\ L\ \texttt{end} & \{\ S.type := L.type\ \} \\
L \to S; L_1 & \{\ L.type := \texttt{if}\ S.type = void \\
 & \quad \texttt{then}\ L_1.type\ \texttt{else}\ type\text{-}error\ \} \\
L \to \varepsilon & \{\ L.type := void\ \}
\end{array}
$$

Hier folgt jetzt eine Erweiterung des vorigen Beispiels auf einfache Funktionen mit einem Parameter. Die erste Produktion soll eine stark vereinfachte Form der Deklaration einer Funktion repräsentieren.

$$
\begin{array}{rl}
T \to T_1\ '\to'\ T_2 & \{\ T.type := T_1.type \to T_2.type\ \} \\
E \to E_1(E_2) & \{\ E.type := \texttt{if}\ E_2.type = s\ \texttt{and}\ E_1.type = s \to t \\
 & \quad \texttt{then}\ t\ \texttt{else}\ type\text{-}error\ \}
\end{array}
$$

3.3 Äquivalenz von Typ-Ausdrücken

Bei der Typ-Prüfung muss immer wieder getestet werden, ob zum Beispiel der Typ-Ausdruck *typ1* eines Operanden mit dem Typ-Ausdruck *typ2* eines Operanden einer Operation zusammenpasst. Man sagt in diesem Fall, dass die Typen *typ1* und *typ2* **äquivalent** sind.
Solange keine Namen als Abkürzungen für Typ-Ausdrücke auftreten, ist die Sache relativ einfach. Die offensichtliche Definition der Typ-Äquivalenz wäre in diesem Fall:

Zwei Typ-Ausdrücke sind äquivalent, wenn sie identisch sind.

Der Typ-Prüfer muß also nur beide Typ-Ausdrücke bzgl. ihres Aufbaus miteinander vergleichen. Dies geschieht am besten rekursiv. (Das funktioniert, falls keine rekursiv definierten Typ-Ausdrücke möglich sind, d.h. der zugeordnete Graph azyklisch ist!)

Beispiel einer Prüfung auf Typ-Äquivalenz:

```
function typequiv(s, t) : boolean
begin
   if s und t sind vom gleichen Basistyp then
      return (true);
   if s = pointer(s1) and t = pointer(t1) then
      return (typequiv(s1, t1));
            .
            .
            .
   if s = array(s1, s2) and t = array(t1, t2) then
      return (s1 = t1) and typequiv(s2, t2));
   else
      return (false);
end
```

Es gibt leider viele Konstruktionen in höheren Programmiersprachen, die implementationsabhängig sind. In Pascal betrachte man etwa

```
var x: integer;
    y: 1..20;
    z: 10..50;
    a: array[1..10] of integer;
    b: array[0..9] of integer;
```

Sollte der Term `y + z` erlaubt sein? Und welchen Typ hat das Ergebnis? Gehört der Indexbereich oder die Länge eines Feldes zum Typ oder nicht? Ist im Beispiel eine Wertzuweisung `a := b` erlaubt? Auch ein Überschreiten einer Feldgrenze beim Zugriff auf ein Feld kann als Typ-Fehler interpretiert werden, speziell wenn der Indexbereich Teil des Typs des Feldes ist. So ein Fehler ist jedoch nicht in jedem Fall bereits zur Übersetzungszeit lokalisierbar und leider wird bei einigen Programmiersprachen eine Überprüfung zur Laufzeit aus Effizienzgründen nicht durchgeführt.

Läßt man zu, dass Namen für Typ-Ausdrücke benutzt werden können, dann gibt es zusätzliche Probleme.
Man betrachte das folgende Pascal-Programmfragment:

```
type link = ↑cell;
      slk = link;
var  next : link;
     last : link;
        s : slk;
     p    : ↑cell;
     q,r  : ↑cell;
```

Es stellt sich die Frage, welche Variablen vom selben Typ sind? Bemerkenswert ist, dass im ersten veröffentlichten Referenz-Handbuch zu Pascal (Jensen, Wirth (1976)) zwar häufig „Typgleichheit“ von Konstrukten der Programmiersprache gefordert wird, der Begriff aber nirgends definiert wird, mit der Folge, dass die obige Frage implementationsabhängig war!

Es gibt in diesem Fall zwei prinzipielle Möglichkeiten der Definition von Typ-Äquivalenz für Typ-Ausdrücke:

Namensäquivalenz: Jeder Typ-Name legt einen neuen Typ fest. Daraus folgt, dass zwei Typ-Ausdrücke äquivalent sind, wenn sie identisch sind. (Ada, abgeschwächt in Pascal und Modula 2) Zum Beispiel wären in Ada `type sec is range 0..60;` und `type min is range 0..60;` sinnvollerweise unterschiedlich. Objektorientierte Sprachen mit einem strengen Klassenkonzept (Smalltalk, Eiffel, Java) fallen ebenfalls unter diesen Punkt.

Strukturäquivalenz: Jeder Typ-Name ist nur eine Abkürzung für den definierten Typ-Ausdruck. Daraus folgert, dass bis auf rekursive Definitionen zwei Typ-Ausdrücke äquivalent sind, wenn man jeden auftretenden Namen durch den zugeordneten Typ-Ausdruck ersetzt und die resultierenden Typ-Ausdrücke identisch sind (Algol 60, Algol 68, FORTRAN, COBOL). Diese Form der Äquivalenz wird in modernen Sprachen nicht mehr verwendet.

Beispiel 3.3:
Im obigen Beispiel sind bzgl. der Namensäquivalenz `next` und `last` sowie `p`, `q` und `r` äquivalent, aber z.B. nicht `p` und `last`. Andrerseits sind natürlich alle Variablen strukturäquivalent.

In Pascal gab es eine noch verwirrendere Konvention, die besagte, dass man implizite Namen für jede direkte Typ-Deklaration verwendet. Dies führt dann dazu, dass im obigen Beispiel `q` und `r` typäquivalent sind, nicht jedoch `p` und `q`! Andererseits werden aber `last` und `s` als äquivalent betrachtet.
In Ada wären sogar `q` und `r` nicht vom gleichen Typ, da eine derartige Deklaration die gleiche Bedeutung wie zwei getrennte Deklarationen haben soll!

Es gibt aber noch weitere Problempunkte. Man betrachte das folgende Programmfragment:

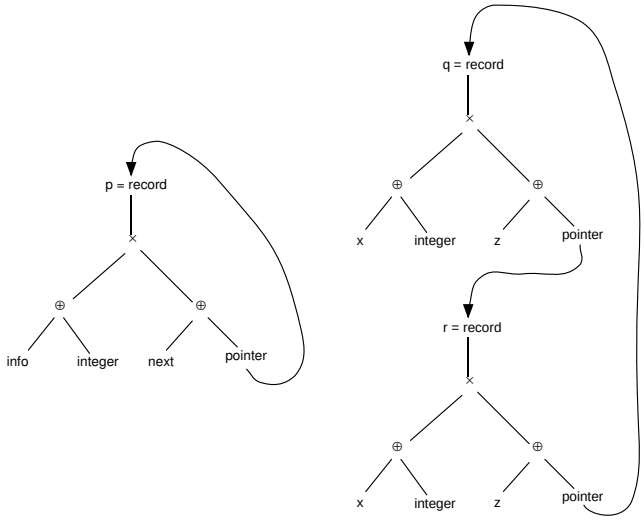
```
t1 = array[-1..9] of integer;
t2 = array[0..10] of integer;
rec1 = record
    x: boolean;
    y: integer;
    end;
rec2 = record
    a: boolean;
    b: integer;
    end;
```

Sind `t1` und `t1` strukturell äquivalent? Falls der Indexbereich zum Typ gehört sind sie nicht äquivalent! Und wie sieht es mit der strukturellen Äquivalenz von `rec1` und `rec2` aus? Gehören die Namen der Komponenten des Records zum Typ, dann sind sie nicht äquivalent.

In C wird wie in vielen anderen Programmiersprachen eine Mischform zwischen Namens- und Strukturäquivalenz benutzt, denn es gilt die strukturelle Äquivalenz bis zur Record-Ebene hinunter. Jeder Record- (Aufzählungs-, Union-) Name wird dagegen als eigenständiger Typ interpretiert.

Weiterhin gibt es Probleme bei der Strukturäquivalenz falls rekursive Definitionen erlaubt sind. Man betrachte den Fall:

```
type p=record
        info:integer;
        next:^p;
     end;
     q=record
        x:integer;
        z:^r;
     end;
     r=record
        x:integer;
        z:^q;
     end;
```



Man sieht an den Beispielen, dass die Namensäquivalenz viel leichter zu handhaben ist und im Gegensatz zur Strukturäquivalenz zu weniger unangenehmen „Überraschungen“ führt.
Bei objektorientierten Programmiersprachen sind kompliziertere Typkonstruktionen nur über die Klassenbildung möglich und die Klassenhierarchie erlaubt eine relativ einfache Überprüfung der Typverträglichkeiten.

3.4 Typ-Umwandlungen

Speziell für die Basistypen gibt es in vielen Programmiersprachen eine Reihe von Umwandlungsregeln, um eine automatische Anpassung der Typen von Operanden an zulässige Typen eines Operators zu ermöglichen.
Man betrachte etwa das folgende Beispiel:

`j := x + i;` mit `x` real und `i, j` integer

Da es i.A. keinen Operator für eine derartige Additionsoperation mit unterschiedlichen Operandentypen gibt, muß der Compiler oder aber auch der Programmierer etwas tun:

- Der Compiler führt eine automatische Typ-Anpassung durch Einfügen einer Typ-Umwandlung durch. Dies ist üblicherweise nur dann erlaubt, wenn damit keine Genauigkeitsverluste verbunden sind, z.B. in C bei der Umwandlung `char` → `integer` (Typ-Erweiterung).

- Der Compiler signalisiert eine Fehlersituation, etwa ”MIXED MODE” in Fortran. Dann

  – kann der Programmier eine explizite Typ-Umwandlung durch Aufruf einer Bibliotheksfunktion programmieren, etwa `x + float(i)` oder

  – der Programmierer kann eine explizite Typ-Umwandlung durch sogenanntes „casting“ vornehmen. Dies bewirkt eine Änderung des zugehörigen Typ-Ausdrucks, aber nicht notwendig eine Änderung der internen Darstellung des Wertes, etwa in `(char)20`.

Ein abschreckendes Beipiel für eine zu weitgehende automatische Typ-Umwandlung liefert PL/1. Numerische Werte sind entweder vom Typ `BINARY` oder `DECIMAL` und haben weitere Attribute wie etwa `FIXED`, `FLOAT` oder diverse andere Genauigkeitsangaben. Dazu gibt es eine Vielzahl von automatischen Typumwandlungen.

Beispiel 3.4:
Es seien die Standardwerte in PL/1 gesetzt.
Dann liefert `1/3 + 25` das Ergebnis `5.33333333333333`. Laut Regel liefert `1/3` das Ergebnis mit maximaler Genauigkeit, d.h. 15-stellig mit 14 Stellen nach dem Komma. Bei einer Addition muß die Genauigkeit erhalten bleiben. Dies führt zwar zu einer `OVERFLOW`-Exception, die kann aber eventuell ignoriert werden!

Auch in Java gibt es derartige automatische Typumwandlungen, die eventuell zu Überraschungen führen, etwa in

```
byte  b1, b2, b3;
b1 = 1;
b2 = 2;
b3 = b1 + b2;   /* führt zur Fehlermeldung */
```

Das Ergebnis der Addition wird automatisch vom Compiler auf den Typ `integer` gesetzt. Eine Wertzuweisung zu einer Byte-Variablen könnte also zu einem Genauigkeitsverlust führen und ist deshalb nicht erlaubt!

3.5 Operator-Identifikation (Überladene Funktionen)

Manchmal haben verschiedene Operatoren eine identische lexikale Darstellung, z.B. kann in Java `+` eine Integer-Addition, eine Float-Addition oder auch eine String-Konkatenation repräsentieren. Man bezeichnet `+` daher als **überladenen Operator**. Aufgabe des Compiler ist es, aus dem Kontext den „richtigen“ Operator zu identifizieren.
Das Problem verschärft sich bei Programmiersprachen, die dem Programmierer mehrere Definitionen einer Funktion mit unterschiedlichen Parameter- oder Ergebnistypen erlaubt.
Beispiel: In Ada sind z.B folgende Definitionen gleichzeitig erlaubt:

```
function '*' (i,j:integer) return complex;
function '*' (x,y:complex) return complex;
```

Damit hat eine Funktion (genauer ein Funktionsname) eine Vielzahl von Typ-Ausdrücken und auch arithmetische Ausdrücke haben nicht notwendigerweise nur einen Typ!

Beispiel 3.5:
Nehmen wir an, dass $*$ unter anderen die folgenden Typ-Ausdrücke hat:

- *integer* $\times$ *integer*$\rightarrow$ *integer*
- *integer* $\times$ *integer*$\rightarrow$ *complex*
- *complex* $\times$ *complex*$\rightarrow$ *complex*

Haben dann 2, 3 und 5 den einzig möglichen Typ *integer* und sei `z` eine Variable vom Typ *complex*, dann kann $3 * 5$ den Typ *integer* oder *complex* haben. Im Ausdruck $(3 * 5) * 2$ **muss** $3 * 5$ den Typ *integer* haben, im Ausdruck $(3 * 5) * z$ **muss** $3 * 5$ den Typ *complex* haben.
Eine Regel in Ada ist, dass der Gesamtausdruck nur genau einen Typ haben darf. Weitere Komplikationen ergeben sich, wenn, wie in C^{++}, selbstdefinierte und automatische Typ-Konvertierung erlaubt ist.

Beispielhaft soll die Identifikation eines Operators oder einer einer Funktion an folgendem Beispiel einer Übersetzung in Postfix-Notation erläutert werden.

Vorgehensweise bei der Operatoridentifikation:

1) Bottom-Up die möglichen Typ-Mengen eines Teilausdrucks berechnen (mit einem synthetischen Attribut *types*)

2) Top-Down die identifizierten Typen für jeden Teilausdruck festlegen (mit einem inheriten Attribut *utype*)

3) Bottom-Up den Postfix-Code erzeugen (mit dem synthetischen Attribut *code*)

$E' \to E$ { $E'.types := E.types$;
$E.utype :=$ `if` $E.types = \{t\}$ `then` t `else` *type-error*;
$E'.code := E.code$; }

$E \to$ `id` { $E.types :=$ `lookup( id.entry)`;
$E.code :=$ `concat(id.lexstring, ":",` $E.utype$ `)`; }

$E \to E_1$ `op` E_2 { $E.types := \{t \mid$ es gibt Typ $r \in E_1.types$ und Typ $s \in E_2.types$ und
Typ $r \times s \to t \in$ `op`$.types$;
Sei $\sigma = \{(r,s) \mid r \times s \to t \in$ `op`$.types$,
$r \in E_1.types, s \in E_2.types$ und $t = E.utype\}$;
$E_1.utype :=$ `if` $\sigma = \{(r,s)\}$ `then` r `else` *type-error*;
$E_2.utype :=$ `if` $\sigma = \{(r,s)\}$ `then` s `else` *type-error*;
$E.code :=$ `concat(`$E_1.code, E_2.code,$ `op` $: r \times s \to t,\ r,s,t$ wie oben ; }

Viel komplexer ist das Problem bei objekt-orientierten Sprachen mit Klassenhierarchie und statischer Typprüfung. Hier geht es um die Auswahl der korrekten Methode, die zum Teil auch erst zur Laufzeit getroffen werden kann. Folgendes Beispiel[2] zeigt die Problematik

```
class T {
     T n() {return new R(); }
}
class S extends T {
     T n() {return new S();}
}
 class R extends S {
     T n() {return new R();}
}
main () {
    T a;
    if (....)
        a = new T();
    else
        a = new S();
     a = a.n();
}
```

Welchen Typ kann der Compiler `a` zuordnen und welche Methode mit Namen `n` wird benutzt?

[2] nach dem Buch von Aho, Lem, Sethi und Ullman

Da Java das dynamische Laden von Klassen erlaubt und viele größere Programme auch noch mit *reflection* arbeiten, ist eine genaue Auswahl der „passenden“ Methode vom Compiler nicht immer zu erreichen und man muss die Auswahl auf die Laufzeit des Programms verschieben (dynamische Methodenauswahl).

3.6 Untertypen

Das Bilden von Untertypen erzeugt eine Relation auf den Typen, die es erlaubt, Werte eines Typs anstelle von Werten eines anderen Typs zu benutzen. Betrachtet man in einer üblichen Programmiersprache mit Typ-Prüfung die Anwendung einer Funktion `f` auf ein Argument `x`, so bestimmt der Typ-Prüfer den Typ $A \to B$ der Funktion `f` und den Typ C des Arguments `x` und prüft, ob $C = A$ gilt.
In Programmiersprachen, die die Bildung von Untertypen erlauben, ist die Situation komplexer. Ist X Untertyp von Y, geschrieben $X <: Y$, dann kann jeder Ausdruck vom Typ X ohne Hervorrufen eines Typ-Fehlers in jedem Kontext benutzt werden, in dem ein Ausdruck vom Typ Y benötigt wird.
Für das obige Beispiel bedeutet dieses, dass der Typ-Prüfer nach Bestimmung der Typen $A \to B$ für `f` und C für `x` prüfen muss, ob $C <: A$ gilt.
Das Konzept der Untertypen erlaubt eine konsistente Behandlung heterogen zusammengesetzter Daten, die alle Untertyp eines gemeinsamen Typs sind. So kann man etwa eine Liste von unterschiedlichen Arten von Bankkonten bilden, die alle Untertypen eines Typs `bank_konto` sind. Ohne die Möglichkeit, Untertypen bilden zu können, müssten alle Listenelemente vom gleichen Typ sein, mit Untertypen müssen die Listenelemente nur Untertypen eines gemeinsamen Typs sein.
Dieses Konzept ist auch wichtig bei der Erweiterung der Funktionalität eines Programms. Wenn Objekte eines bestimmten Typs X nicht die gewünschte Funktionalität haben und man möchte sie durch Objekte eines Typs Y ersetzen, die die gewünschte Funktionalität besitzen, so wird man in vielen Fällen ohne größere Änderungen des gesamten Programms die neuen Objekte mit Y als Untertyp von X einführen können. Dies ist besonders hilfreich, wenn man sich über eine Reihe von Prototypen an das gewünschte Programm annähert.
Einer der großen Vorteile objektorientierter Sprachen ist die Möglichkeit der Definition und der Verwendung von Untertypen. Häufig werden Untertypen implizit über Vererbung definiert.
Vererbung ist ein Konzept, das es erlaubt, neue Objekte durch Erweiterung bereits existierender Objekte zu definieren. Durch Vererbung werden Attribute und Methoden einer Klasse X an eine andere Klasse Y weitergegeben. Die Klasse Y ist dann eine Unterklasse von X bzw. die Klasse X ist Oberklasse der Klasse Y. In einer Unterklasse können Attribute und Methoden zu den geerbten der Oberklasse hinzugefügt werden. Es können aber auch vererbte Methoden neu definiert und/oder implementiert werden.
Bei einer einfachen Vererbung hat jede Klasse höchstens eine Oberklasse. Dies führt zu einer hierarchischen Anordnung der Klassen. Ist eine mehrfache Vererbung erlaubt, so hat eine Klasse mehrere Oberklassen. In diesem Fall muss man darauf achten, dass in der Vererbungsfolge keine Zyklen und keine Namenskonflikte auftreten.
Vom Prinzip her könnte man Vererbung durch Duplizieren von Code realisieren. Durch das Vererben spart man sich aber das Kopieren. Man muss natürlich dabei beachten, dass eine Änderung in der Oberklasse auch eine Änderung in den Objekten der Unterklasse bewirkt (sofern die geänderten Teile in der Unterklasse nicht überschrieben werden).
Wichtig ist in diesem Zusammenhang der Unterschied zwischen Untertyp-Bildung und Vererbung. Untertypen bilden eine Relation auf den Typen, Vererbung dagegen bildet eine Relation auf den Implementationen. Dass diese Begriffe häufig durcheinander gebracht werden liegt an

den einzelnen Programmiersprachen, die in ihren Klassenkonzepten beides kombinieren. In C^{++} wird A als Untertyp von B nur dann akzeptiert, wenn B public base class von A ist.

Folgendes Beispiel[3] zeigt die Problematik:

Beispiel 3.6:
Es soll ein Programm geschrieben werden, das mit den Datenstruktutren `stack`, `queue` und `dequeue` arbeitet.

`stack`: Eine Datenstruktur mit Einsetz- und Löschoperation, so dass das zuerst eingesetzte Objekt als letztes entfernt wird (first-in, last-out).

`queue`: Eine Datenstruktur mit Einsetz- und Löschoperation, so dass das zuerst eingesetzte Objekt als erstes entfernt wird (first-in, first-out).

`dequeue`: Eine Datenstruktur mit zwei Einsetz- und zwei Löschoperationen. Eine dequeue (doubly ended queue) kann man sich als eine Liste vorstellen, bei der sowohl am Anfang als auch am Ende der Liste Objekte eingesetzt oder entfernt werden können. Daher benötigt man für Anfang und Ende jeweils eine Einsetz- und eine Löschoperation.

Die Datenstruktur `dequeue` kann sowohl die Aufgaben der Datenstrukturen `stack` als auch `queue` übernehmen. Also wäre es eine Möglichkeit, zunächst die Klasse `dequeue` zu implementieren und dann die Klassen `stack` und `queue` als Unterklassen von `dequeue` zu definieren, indem man die geerbten Methoden zum Einsetzen und Löschen umbenennt bzw. überschreibt. Für die Klasse `queue` würde man z.B. die Löschoperation für den Anfang und die Einsetzoperation für das Ende der `dequeue` umbenennen und die beiden anderen Operation durch Methoden überschreiben, die etwa eine Fehlermeldung produzieren.
Obwohl `stack` und `queue` Unterklassen von `dequeue` sind, bilden sie *keinen* Untertyp von `dequeue`. Angenommen eine Funktion `f` hat ein Objekt `d` der Klasse `dequeue` als Argument und fügt an jedem Ende ein Objekt ein. Wäre jetzt `stack` oder `queue` ein Untertyp von `dequeue`, so müsste die Funktion `f` auch mit einem Objekt der Klasse `stack` oder `queue` arbeiten. Dies führt aber zu einem Fehler, also sind `stack` und `queue` keine Untertypen von `dequeue`.
Dagegen kann man in jedem Kontext, in dem man etwa ein Objekt der Klasse `stack` bzw. `queue` benutzt, ohne Schwierigkeiten auch ein Objekt der Klasse `dequeue` benutzen. Folglich ist `dequeue` Untertyp sowohl von `stack` als auch von `queue`.

3.7 Polymorphe Funktionen

Weitere Probleme treten auf, wenn in der Programmiersprache sogenannte **polymorphe Funktionen** oder **polymorphe Operatoren** zugelassen sind. Dies sind Funktionen, bei denen der Typ der Parameter nicht eindeutig festgelegt werden muß. Dieses Konzept ist wichtig und hilfreich, z.B. möchte man allgemeine Sortierprogramme schreiben können, ohne Rücksicht auf den speziellen Typ der zu sortierenden Elemente. Ein anderes Beispiel ist eine Funktion zum Bestimmen der Länge einer Liste, etwa in Scheme:

```
(define (length liste)
   (if (null? liste)
        0
        (+ 1 (length (cdr liste)))))
```

[3] lt. J.C. Mitchell ist es von Alan Snyder

Ähnliche Konstruktionen findet man in anderen funktionalen Sprachen wie etwa ML oder Haskell oder in rein objektorientierten Sprachen wie Smalltalk, aber z.B. nicht in Pascal. Welchen Typ-Ausdruck kann man der Funktion `length` zuordnen?
Man benötigt also wieder ein Konzept ähnlich dem der Untertypen im vorigen Abschnitt, um eine Typ-Prüfung vornehmen zu können. Allerdings kann die Definition von Untertypen in diesen Sprachen aber nicht an eine Klassenhierarchie gebunden sein.
Um auch Typ-Ausdrücke für polymorphe Funktionen angeben zu können, muß man das Konzept einer **Typ-Variablen** einführen. Eine Typ-Variable steht dabei für einen beliebigen Typ. Für das obige Beispiel könnte man der Funktion `length` den Typ-Ausdruck $\forall\alpha.\mathit{list}(\alpha) \rightarrow \mathit{integer}$ zuordnen, wobei α die Typ-Variable ist.
Wir wollen im folgenden den Quantor weglassen und schreiben einfach nur $\mathit{list}(\alpha) \rightarrow \mathit{integer}$
Die Typ-Prüfung von Programmen, in denen polymorhe Funktionen auftreten können, ist signifikant schwieriger als die Typ-Prüfung in den vorangegangenen Fällen.

- Verschiedene Auftreten einer polymorphen Funktion in einem Ausdruck können durchaus Argumente unterschiedlichen Typs haben.

- Da in Typ-Ausdrücken Variablen auftreten können, ist die Typ-Äquivalenz neu zu definieren. Um zwei Typ-Ausdrücke mit Variablen „anzupassen“, müssen die auftretenden Variablen durch Typ-Ausdrücke (eventuell wieder mit neuen Variablen) ersetzt werden. Diese Problematik ist unter dem Namen **Unifikation** bekannt und wird auch in der logischen Programmierung, etwa in PROLOG, benutzt.

  Beispiel: Die Typ-Ausdrücke $\mathit{pointer}(\alpha)$ und $\mathit{pointer}(\mathit{pointer}(\beta))$ könnte man z.B. durch $\alpha \mapsto \mathit{pointer}(\gamma)$ und $\beta \mapsto \gamma$ unifizieren.

Es gibt nun mehrere Möglichkeiten, wie man die Typ-Prüfung in Programmiersprachen, die polymorphe Funktionen und Operatoren erlauben, durchführen kann.

1) Man verzichtet völlig auf die Typisierung von Variablen usw. in der Programmiersprache und auf eine Überprüfung zur Übersetzungszeit und benutzt eine dynamische Typ-Überprüfung zur Laufzeit (etwa in LISP, Scheme oder Smalltalk)

2) Man verzichtet auf die Typisierung von Variablen usw. in der der Programmiersprache, prüft aber in der Übersetzungsphase jeden Gebrauch eines Namens oder eines anderen Sprachkonstrukts auf einen konsistenten Gebrauch (etwa in ML, Caml oder Haskell)

3) Man erweitert die Syntax der Programmiersprache um Typ-Variable oder Typ-Parameter, damit man generische Typen definieren kann. Oder man „durchlöchert“ das übliche Typ-System durch die Verwendung von Zeigern oder durch beliebiges „casting“.

4) Man benutzt eine Kombination der obigen Möglichkeiten.

Beispiel 3.7:
in einer Programmiersprache mit Typ-Variablen wären etwa die folgenden Deklarationen möglich:

```
deref : pointer(α) → α;
    q : pointer(pointer(integer));
```

Tritt dann im Programm der Ausdruck `deref(deref(q))` auf, so kann durch Typ-Inferenz geschlossen werden, dass dieser Ausdruck den Typ *integer* hat.

Wie kann man nun zwei Typ-Ausdrücke unifizieren? Das Problem besteht darin, für alle in beiden Ausdrücken auftretenden Typ-Variablen Ausdrücke zu finden, so dass nach einer Substitution der gefundenen Ausdrücke die Typ-Ausdrücke identisch sind. So kann z.B. der Typ-Ausdruck $\alpha \to \alpha$ zu *integer* $\to$ *integer* oder aber auch zu *pointer*(β) $\to$ *pointer*(β) werden. Man nennt das Ergebnis einer derartigen Substitution auch eine **Instanz** von $\alpha \to \alpha$.
Nun sucht man nicht irgendeine Substitution, die zwei Typ-Ausdrücke identisch macht, sondern nach einer möglichst allgemeinen Substitution mit dieser Eigenschaft, d.h. eine möglichst allgemeine Instanz der zwei Ausdrücke.
Man bezeichnet eine Substitution σ, die zwei Typ-Ausdrücke t_1 und t_2 identisch macht als **allgemeinste Substitution**, falls gilt:

1) $\sigma(t_1) = \sigma(t_2)$, d.h. die Substitution macht beide Ausdrücke identisch
2) für jede andere Substitution σ' mit $\sigma'(t_1) = \sigma'(t_2)$ ist $\sigma'(t_1)$ eine Instanz von $\sigma(t_1)$.

Bemerkung: Wenn man von einer Unifikation zweier Typ-Ausdrücke spricht, so wird darunter meist die Anwendung der allgemeinsten Substitution auf die Typ-Ausdrücke verstanden.

Beispiel: Gegeben seien die beiden Typ-Ausdrücke

$$\begin{aligned} t_1 &= ((\alpha_1 \to \alpha_2) \times \mathit{list}(\alpha_3)) \to \mathit{list}(\alpha_2) \quad \text{und} \\ t_2 &= ((\alpha_3 \to \alpha_4) \times \mathit{list}(\alpha_3)) \to \alpha_5 \end{aligned}$$

Die allgemeinste Substitution σ wäre dann

$$\begin{aligned} \alpha_1 &\mapsto \alpha_3 \\ \alpha_2 &\mapsto \alpha_2 \\ \alpha_3 &\mapsto \alpha_3 \qquad \sigma(t_1) = \sigma(t_2) = ((\alpha_3 \to \alpha_2) \times \mathit{list}(\alpha_3)) \to \mathit{list}(\alpha_2) \\ \alpha_4 &\mapsto \alpha_2 \\ \alpha_5 &\mapsto \mathit{list}(\alpha_2) \end{aligned}$$

Eine speziellere Substitution σ' wäre

$$\begin{aligned} \alpha_1 &\mapsto \alpha_1 \\ \alpha_2 &\mapsto \alpha_1 \\ \alpha_3 &\mapsto \alpha_1 \qquad \sigma'(t_1) = \sigma'(t_2) = ((\alpha_1 \to \alpha_1) \times \mathit{list}(\alpha_1) \to \mathit{list}(\alpha_1) \\ \alpha_4 &\mapsto \alpha_1 \\ \alpha_5 &\mapsto \mathit{list}(\alpha_1) \end{aligned}$$

und natürlich ist $\sigma'(t1)$ eine Instanz von $\sigma(t_1)$.

Eine interessante Anwendung erfährt dieses Konzept in der Programmiersprache ML, einer funktionalen Sprache, die polymorphe Funktionen erlaubt. Der Programmierer muß aber nicht selbst seine Funktionen deklarieren, sondern ein ausgefeiltes Typ-System in ML berechnet automatisch den Typ eines eingegebenen Ausdrucks.
Definiert man also z.B. eine Funktion, etwa

```
fun ac(oper, init, seq) =
       if null(seq) then init
       else oper(hd(seq), ac(oper, init, tl(seq)));
```

so erhält man als Antwort des Systems:

```
val ac = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
```

wobei der Typ-Ausdruck gemäß unserer Notation wie folgt wäre:

$$(\alpha \times \beta \to \beta) \times \beta \times \ \mathit{list}(\alpha) \to \beta.$$

Ein anderes Beispiel wäre etwa die ML-Version der `filter`-Prozedur

```
fun filter(pred, seq) =
        if null(seq) then nil
        else if pred(hd (seq)) then
                hd(seq) :: filter(pred, tl(seq))
                else filter(pred, tl(seq));
```

die zu der Ausgabe

```
val filter = fn : ('a -> bool) * 'a list -> 'a list
```

führt, die in unserer Notation

$$(\alpha \to \ \mathit{boolean}) \times \ \mathit{list}(\alpha) \to \ \mathit{list}(\alpha)$$

lauten würde.

Beispiel 3.8:
Typ-Inferenz für die ML-Funktion

```
fun length(lptr) =
   if null(lptr) then 0
   else length(tl(lptr)) + 1;
```

Bemerkung: Das Schlüsselwort `fun` leitet eine Funktionsdefinition ein. Definiert wird hier eine Funktion `length` mit einem Parameter `lptr`. Das Prädikat `null` testet, ob das Argument eine leere Liste ist. Die Funktion `tl` angewendet auf eine Liste liefert die Ausgangsliste ohne das erste Listenelement zurück. Die Funktion `length` berechnet also die Länge einer Liste. Ihr ist damit der Typ-Ausdruck $\mathit{list}(\alpha) \to \mathit{integer}$ zugeordnet.

Wie kann nun das Typ-System auf diesen Typ-Ausdruck schließen?
Betrachtet man zunächst den Ausdruck links vom „=“-Zeichen, dann kann man den beiden Symbolen `length` und `lptr` zum Beispiel die Typ-Ausdrücke $\gamma \to \delta$ und γ zuordnen, da es sich ja um eine Funktionsdefinition einer Funktion mit einem Parameter handelt. Der Ausdruck rechts vom „=“-Zeichen muss also einen Wert ergeben, dem der Typ-Ausdruck δ zugeordnet ist. Betrachtet man nun den `null` zugeordneten Typ-Ausdruck $\mathit{list}(\alpha_n) \to \mathit{boolean}$, wobei α_n eine neue Typ-Variable ist, so kann man schließen, dass $\gamma = \mathit{list}(\alpha_n)$ sein muss. Aus dem Typ-Ausdruck $\mathit{list}(\alpha_t) \to \mathit{list}(\alpha_t)$ für `tl` kann man wiederum schließen, dass $\alpha_t = \alpha_n$ sein muss. Die Anwendung der `length`-Funktion auf `tl(lptr)` liefert ein Objekt vom Typ δ.
Da die Konstante 1 vom Typ *integer* ist und der Addition ein Typ-Ausdruck $\mathit{integer} \times \mathit{integer} \to \mathit{integer}$ zugeordnet ist, muss $\delta = \mathit{integer}$ gelten. Damit ist aber auch der Typ-Ausdruck $\mathit{boolean} \times \alpha_i \times \alpha_i \to \alpha_i$ für die `if`-Anweisung mit $\alpha_i = \mathit{integer}$ erfüllt und man erhält so den Typ-Ausdruck $\mathit{list}(\alpha_n) \to \mathit{integer}$ für die Funktion `length`.

Um das ganze etwas formaler darzustellen, werden die einzelnen Schritte jetzt noch tabellarisch aufgeführt:

Die folgende Funktionen und ihre Typ-Ausdrücke sind vorgegeben:

$$\begin{aligned}
\texttt{if} &: \mathit{boolean} \times \alpha \times \alpha \to \alpha \\
\texttt{null} &: \mathit{list}(\alpha) \to \mathit{boolean} \\
\texttt{tl} &: \mathit{list}(\alpha) \to \mathit{list}(\alpha) \\
0 &: \mathit{integer} \\
1 &: \mathit{integer} \\
+ &: \mathit{integer} \times \mathit{integer} \to \mathit{integer}
\end{aligned}$$

Hier nun die Tabelle der einzelnen Schlussfolgerungen:

Zeile	Ausdruck :	Typ-Ausdruck	Substitution
(1)	`lptr` :	γ	
(2)	`length` :	β	
(3)	`length(lptr)` :	δ	$\beta = \gamma \to \delta$
(4)	`lptr` :	γ	
(5)	`null` :	$\mathit{list}(\alpha_n) \to \mathit{boolean}$	
(6)	`null(lptr)` :	$\mathit{boolean}$	$\gamma = \mathit{list}(\alpha_n)$
(7)	`0` :	$\mathit{integer}$	
(8)	`lptr` :	$\mathit{list}(\alpha_n)$	
(9)	`tl` :	$\mathit{list}(\alpha_t) \to \mathit{list}(\alpha_t)$	
(10)	`tl(lptr)` :	$\mathit{list}(\alpha_n)$	$\alpha_t = \alpha_n$
(11)	`length` :	$\mathit{list}(\alpha_n) \to \delta$	
(12)	`length(tl(lptr))` :	δ	
(13)	`1` :	$\mathit{integer}$	
(14)	`+` :	$\mathit{integer} \times \mathit{integer} \to \mathit{integer}$	
(15)	`length(tl(lptr)) + 1` :	$\mathit{integer}$	$\delta = \mathit{integer}$
(16)	`if` :	$\mathit{boolean} \times \alpha_i \times \alpha_i \to \alpha_i$	
(17)	`if( .... )` :	$\mathit{integer}$	$\alpha_i = \mathit{integer}$