4

Semantische Analyse

4.1 Aufgabe der semantischen Analyse

Einige Eigenschaften von Programmen sind nicht durch kontextfreie Grammatiken beschreibbar. Diese Eigenschaften werden durch *Kontextbedingungen* beschrieben. Die wichtigsten Beispiele für Kontextbedingungen sind Anforderungen an die *Deklariertheit* von Bezeichnern und die *Typkonsistenz*. Grundlegend für diese Anforderungen sind die Regeln der Programmiersprache für die *Deklariertheit*, die *Gültigkeit* und die *Sichtbarkeit* von Bezeichnern (engl. Identifier).

Die Regeln fur die *Deklariertheit* bestimmen, inwiefern zu einem Bezeichner eine explizite Deklaration gegeben werden muss, wo diese zu platzieren ist und und ob Mehrfachdeklarationen verboten sind. Die *Gültigkeitsregeln* legen für deklarierte Bezeichner fest, in welchem Teil des Programms ihre Deklaration einen Effekt haben kann. Die *Sichtbarkeitsregeln* wiederum bestimmen, wo in seinem Gültigkeitsbereich ein Bezeichner *sichtbar* bzw. *verdeckt* ist.

Ziel einiger Einschränkungen ist es, häufige Programmierfehler auszuschließen. Dazu gehören Regeln zur *Initialisierung* von Variablen oder Attributen oder zur *Typkonsistenz*. Initialisierungsregeln versuchen, Programmzugriffe auf *uninitialisierte* Variablen zu verhindern, deren Ergebnis undefiniert wäre. Die *Typkonsistenz* eines Programms garantiert, dass zur Ausführungszeit keine Operation auf Operanden angewendet wird, auf die sie von ihren Argumenttypen her nicht passt.

Einige Begriffe

Wir benutzen die folgenden Begriffe, um einige Aufgaben der semantischen Analyse zu beschreiben.

Ein *Bezeichner* (Identifier) ist ein Symbol (im Sinne der lexikalischen Analyse), welches in einem Programm zur Benennung eines Programmelements benutzt werden kann. Programmelemente imperativer Sprachen, die benannt werden, sind etwa Module, Funktionen bzw. Prozeduren, Sprungziele, Konstanten, Variablen, Parameter und ihre Typen. In objekt-orientierten Sprachen wie JAVA kommen Klassen und Interfaces mit ihren Attributen und Methoden sowie gegebenenfalls ihren Parametern hinzu. In funktionalen Sprachen wie OCAML können auch Module Parameter haben. Variablen und Funktionen haben zwar eine etwas andere Semantik wie in imperativen Sprachen, können aber ebenfalls mit einem Bezeichner versehen werden. Eine wichtige Klasse von Datenstrukturen wird mit Hilfe von *Konstruktoren* aufgebaut, deren Bezeichner zusammen mit dem entsprechenden Datentyp eingeführt werden. Das Konzept solcher Konstruktoren kann als Verallgemeinerung von Aufzähldatentypen in imperativen Sprachen aufgefasst werden, die eine feste Folge von Konstanten bereit stellen, deren Bezeichner jedoch ebenfalls zusammen mit der Typdeklaration eingeführt werden. In logischen Sprachen wie PROLOG gibt es Bezeichner für Prädikate, Konstanten, Datenkonstruktoren und Variablen.

Manche Bezeichner werden in einer expliziten *Deklaration* eingeführt. Das Vorkommen des Bezeichners in der Deklaration ist das *definierende Vorkommen* des Bezeichners, alle anderen Vorkommen sind *angewandte Vorkommen*. In imperativen Programmiersprachen wie C und objekt-orientierten wie JAVA müssen sämtliche Bezeichner explizit eingeführt werden. Das Gleiche gilt im Wesentlichen auch für funktionale Programmiersprachen wie OCAML. In PROLOG dagegen werden weder die verwendeten Konstruktoren und Atome noch die lokalen Variablen in Klauseln explizit eingeführt. Um sie

unterscheiden zu können, entstammen deren Bezeichner aus unterschiedlichen *Namensräumen*. So beginnen Variablen etwa mit einem Großbuchstaben oder einem Unterstrich, während Konstruktoren und Atome mit einem kleinen Buchstaben beginnen. Der Term $f(X, a)$ etwa entsteht durch Anwendung des zweistelligen Konstruktors $f/2$ auf die Variable X und das Atom a. Anstatt durch explizite Deklaration wird hier eine Bezeichnung implizit durch ihr syntaktisch erstes Vorkommen in einer Klausel eingeführt.

In jeder Programmiersprache gibt es Strukturierungsmöglichkeiten für Programme, die bei der Benutzbarkeit von Bezeichnern berücksichtigt werden müssen. In imperativen Sprachen dienen etwa Pakete, Module, Funktions- oder Prozedurdeklarationen zur Strukturierung sowie Blöcke, die mehrere Anweisungen (und gegebenenfalls Deklarationen) zusammen fassen. In objekt-orientierten Sprachen wie JAVA kommen als weitere Strukturierungsmöglichkeiten Klassen und Interfaces hinzu, die selbst wiederum in Hierarchien organisiert sind. Funktionale Sprachen wie OCAML bieten ebenfalls Module an, um eine Menge von Deklarationen zusammenzufassen. Deklarationen von Variablen und Funktionen erlauben es, die Verwendbarkeit auf einen Teilausdruck des Programms einzuschränken. Die Strukturierungsmöglichkeiten in PROLOG beschränken sich zuerst einmal nur auf Klauseln. Moderne Versionen bieten jedoch ebenfalls eine Strukurierung von Programmen durch Module an.

Vorkommen solcher Strukturierungsmöglichkeiten in Programmen nennen wir ganz allgemein *Blöcke*. Bei diesem Sprachgebrauch ist also eine Methodendeklaration in einem JAVA-Programm ein Block, ebenso wie ein Modul in OCAML oder eine Klausel in PROLOG.

Der *Typ* eines Programmelements schränkt ein, was während der Ausführung des Programms mit dem Element gemacht werden kann. Ist das Programmelement ein Modul, gibt er an, welche Operationen, Datenstrukturen oder sonstige Programmelemente exportiert werden. Ist es eine Funktion oder Methode, gibt er an, welche Typen die Argumente haben dürfen und welchen Typ das Ergebnis hat. Ist das Programmelement eine Programmvariable in einer imperativen oder objekt-orientierten Programmiersprache, schränkt der Typ ein, welche Werte in der Variablen abgespeichert werden dürfen. In rein funktionalen Sprachen können Werte nicht explizit einer Variablen zugewiesen d.h. in dem Speicherbereich abgelegt werden, die dieser Variablen zugeordnet ist. Eine Variable ist hier nicht der Bezeichner eines Speicherbereichs, sondern für einen Wert selbst. Der Typ einer Variablen muss deshalb ebenfalls zu den Typen der Werte passen, die die Variable möglicherweise haben kann. . Repräsentiert das Programmelement schließlich einen Wert, lässt sich aus dem Typ ableiten, wieviel Platz vom Laufzeitsystem für diesen Wert bereit gestellt werden muss, um seine interne Repräsentation abzuspeichern. Ein Wert vom Typ **int** in der Programmiersprache JAVA benötigt z.B. gegenwärtig 32 Bits oder vier Byte für seine Internrepräsentation, ein Wert vom Typ **double** dagegen 64 Bits oder acht Byte. Weiterhin schränkt der Typ ein, welche Operationen auf den Wert angewendet werden dürfen. Ein *int*-Wert kann z.B. durch arithmetische Operationen mit anderen Werten vom Typ **int** verknüpft werden, um neue Werte vom Typ **int** zu berechnen. Das ist mit Werten vom Typ **string** zum Beispiel erst einmal nicht möglich.

Konkrete und abstrakte Syntax

Eingabe für die semantische Analyse ist eine hierarchische Aufgliederung des Programms, die von der Syntaxanalyse bereit gestellt wird. Die Aufgliederung repräsentiert die statische Schachtelung der Programmkonstrukte. Wird der Syntaxbaum des Programms gemäß der kontextfreien Grammatik für die Programmiersprache verwendet, sprechen wir von der *konkreten Syntax* des Programms. Die kontextfreie Grammatik zu einer Programmiersprache enthält jedoch oft Informationen, die für die weitere Verarbeitung von Programmen nicht wichtig sind. Dazu gehören einige Terminalsymbole, die zwar für die syntaktische Analyse und für das Lesen von Programmen wichtig sind, aber keinen weiteren semantischen Wert beinhalten wie z.B. die Schlüsselwörter `if`, `else` oder `while`. Präzedenzen von Operatoren führen in vielen Grammatiken zu Schachtelungen von Nichtterminalen, typischerweise einem Nichtterminal pro Präzedenztiefe. Diese Nichtterminale und die zugehörigen Produktionen sind oft nicht mehr von Bedeutung, wenn die syntaktische Struktur erkannt ist.

Deshalb benutzen Übersetzer zur expliziten Darstellung der syntaktischen Struktur von Programmen oft vereinfachte Syntaxbäume. Wir sprechen hier deshalb von *abstrakter Syntax*. Sie identifiziert nur noch die im Programm auftretenden Konstrukte und ihre Schachtelungsbeziehung.

Beispiel 4.1.1 Den konkreten Syntaxbaum für das Programmstück

$$\begin{array}{l} \mathsf{if}\ (x+1>y) \\ \quad z \leftarrow 1; \\ \mathsf{else}\ z \leftarrow 2; \end{array}$$

zeigt Abbildung 4.1. Dabei gingen wir davon aus, dass die zugehörige kontextfreie Grammatik zwischen den Präzedenzstufen für Zuweisungen, Vergleiche, Additions- bzw. Multiplikationsoperatoren unterscheidet. Auffällig sind die langen Abfolgen von Kettenproduktionen, die für die Überbrückung der Präzedenzunterschiede eingeführt wurden. Eine abstraktere Darstellung erhalten wir, indem wir zuerst die Anwendungen der Kettenproduktionen aus dem Syntaxbaum entfernen und in einem zweiten Schritt gegebenenfalls zusätzlich überflüssige Terminalsymbole entfernen. Das Ergebnis dieser beiden Vereinfachungen zeigt Abbildung 4.2.

H: die Bilder muessen noch korrigiert werden!!! □

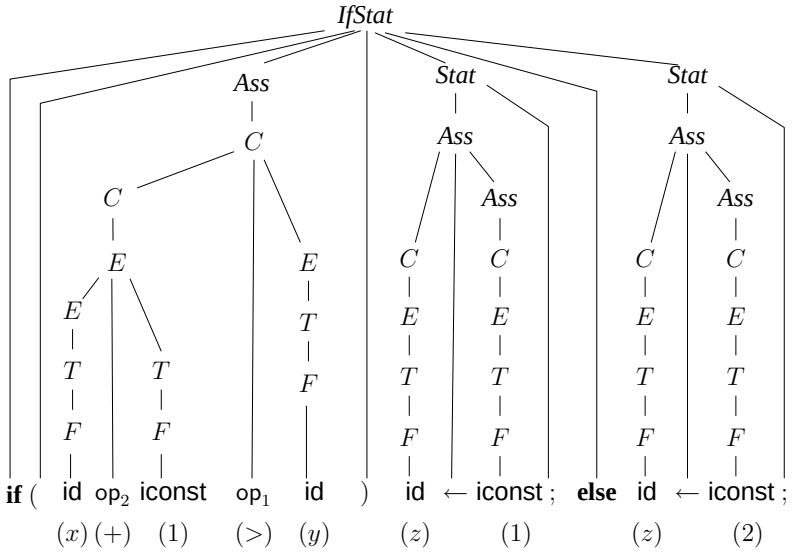


Abb. 4.1. Repräsentation der konkreten Syntax

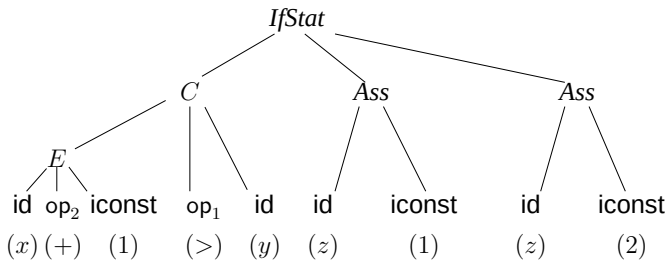


Abb. 4.2. Repräsentation der abstrakten Syntax

Der erste Schritt der Transformation in eine abstrakte Syntax, den wir in Beispiel 4.1.1 manuell durchgeführt haben, braucht nicht für jeden Syntaxbaum erneut durchgeführt zu werden. Stattdessen können wir auch die Grammatik G systematisch so umschreiben, dass keine Kettenproduktionen mehr vorkommen. Es gilt:

Satz 4.1.1 *Zu jeder kontextfreien Grammatik G kann eine kontextfreie Grammatik G' ohne Kettenproduktionen konstruiert werden mit den folgenden Eigenschaften:*

1. $L(G) = L(G')$;
2. *Ist G eine (starke) $LL(k)$-Grammatik, dann auch G';*
3. *Ist G eine $LR(k)$-Grammatik, dann auch G'.*

Beweis. Nehmen wir an, die kontextfreie Grammatik G sei gegeben durch $G = (V_N, V_T, P, S)$. Die kontextfreie Grammatik G' erhalten wir dann als das Tupel $G' = (V_N, V_T, P', S)$ mit den gleichen Mengen V_N und V_T an Nichtterminal- und Terminalsymbolen und dem gleichen Startsymbol S, wobei die neue Menge P' der Produktionsregeln von G' aus allen Produktionen $A \to \beta$ besteht mit

$$A_0 \underset{G}{\Longrightarrow} A_1 \underset{G}{\Longrightarrow} \ldots \underset{G}{\Longrightarrow} A_n \underset{G}{\Longrightarrow} \beta \quad \text{wobei} \quad A = A_0, \beta \notin V_N$$

für ein $n \geq 0$. Die Anzahl $\#P'$ der Produktionen in P' ist damit möglicherweise deutlich gößer als die Anzahl $\#P$ der Produktionen der ursprünglichen Grammatik, aber zumindest beschränkt durch $\#N \cdot \#P$.

Sei $R \subseteq V_N \times V_N$ mit $(A, B) \in R$ falls $a \to B \in P$. Dann reicht es, den *reflexiven und transitiven Abschluss* R^* dieser Relation berechnen, um die Menge aller Paare (A, B) zu ermitteln, bei denen B durch eine beliebige Folge von Anwendungen von Kettenproduktionen aus A ableitbar ist. Zur Berechnung des reflexiven und transitiven Abschlusses einer binären Relation lässt sich z.B. der Floyd-Warshall-Algorithmus einsetzen [**?**].

Den Beweis der Eigenschaften $(1), (2)$ und (3) überlassen wir der Leserin. □

Beispiel 4.1.2 Betrachten Sie unsere Beispiel-Grammatik G_1 mit den Produktionen:

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathsf{Id} \end{aligned}$$

Die Relationen R und R^* sind gegeben durch:

R	E	T	F
E	0	1	0
T	0	0	1
F	0	0	0

R^*	E	T	F
E	1	1	1
T	0	1	1
F	0	0	1

Durch Beseitigung der Kettenproduktionen erhalten wir eine Grammatik mit den Produktionen:

$$\begin{aligned} E &\to E + T \mid T * F \mid (E) \mid \mathsf{Id} \\ T &\to T * F \mid (E) \mid \mathsf{Id} \\ F &\to (E) \mid \mathsf{Id} \end{aligned}$$

□

Kettenproduktionen sind damit, zumindest was die Ausdruckskraft angeht, überflüssig. Die Duplizierung rechter Seiten jedoch, die zur Beseitigung der Kettenproduktionen erforderlich ist, wird bei der *Spezifikation* von Grammatiken gerne vermieden. Bei der *Benutzung* der Syntaxbäume der Grammatik möchten wir dagegen gerne auf die Kettenproduktionen wieder verzichten. Ein Kompromiss kann darin bestehen, die Beseitigung der Kettenproduktionen dem Parsergenerator zu überlassen.

Im Folgenden werden wir uns je nachdem, was vorteilhafter ist, auf die konkrete oder die abstrakte Syntax beziehen.

4.1.1 Gültigkeits- und Sichtbarkeitsregeln

Programmiersprachen lassen es im Allgemeinen zu, dass derselbe Bezeichner für unterschiedliche Programmelemente verwendet wird und damit gegebenenfalls mehrere Deklarationen besitzt. Deshalb muss geregelt werden, auf welches definierende Vorkommen sich ein angewandtes Vorkommen beziehen soll. Dies regeln die *Gültigkeitsbereichsregeln* und die *Sichtbarkeitsregeln* der Programmiersprache.

Der *Gültigkeitsbereich* (scope, range of validity) eines definierenden Vorkommens eines Bezeichners x ist der Teil des Programms (oder mehrerer Programme), in dem sich ein angewandtes Vorkommen von x auf dieses definierende Vorkommen beziehen kann.

Die *Sichtbarkeit* eines definierenden Vorkommens eines Bezeichners kann in seinem Gütligkeitsbereich durch Verdeckung eingeschränkt werden. Andererseits gibt es Möglichkeiten, Sichtbarkeit eines Bezeichners durch explizite Maßnahmen auch außerhalb seines Gültigkeitsbereichs herzustellen. Dazu gehören die *Qualifikation* von Komponenten von strukturierten Typen und der *Import* von Bezeichnern z.B. durch use-Klauseln.

Die Aufgabe, jedem angewandten Vorkommen eines Bezeichners das gemäß der Gültigkeits- und der Sichtbarkeitsregeln zugehörige definierende Vorkommen oder die zugehörigen definierenden Vorkommen zuzuordnen, nennt man die *Identifizierung von Bezeichnern* (identification of identifiers). Die Gültigkeits- und die Sichtbarkeitsregeln einer Programmiersprache hängen stark davon ab, welche Art von Schachtelung von Scope-Konstrukten die Sprache erlaubt.

COBOL erlaubt keine Schachtelung von Blöcken; alle Bezeichner sind überall gültig und sichtbar. FORTRAN77 erlaubt nur die Schachtelungstiefe 1, also nicht weiter geschachtelte Blöcke, d.h. Prozedur-/ Funktionsdeklarationen in einem Hauptprogramm. Bezeichner, die in einem Block definiert sind, sind nur innerhalb dieses Blocks sichtbar. Ein im Hauptprogramm deklarierter Bezeichner ist ab der Deklaration überall sichtbar, außer in Prozedurdeklarationen, die eine neue Deklaration des Bezeichners enthalten.

Modernere imperative oder objekt-orientierte Sprachen wie PASCAL, ADA, C, C++, C♯ oder JAVA und funktionale Programmiersprachen erlauben, Blöcke unbeschränkt tief zu schachteln. Der Gültigkeits- und der Sichtbarkeitsbereich definierender Vorkommen von Bezeichnern wird dann durch zusätzliche Festlegungen geregelt. Bei einem *let*-Konstrukt

$$\textbf{let } x = e_1 \textbf{ in } e_0$$

in OCAML ist der Bezeichner x nur in dem *Rumpf* e_0 des *let*-Konstrukts gültig. Angewandte Vorkommen von x in dem Ausdruck e_1 beziehen sich also auf definierende Vorkommen von x in umfassenden Blöcken. Der Gültigkeitsbereich der Bezeichner $x_1, \ldots, x_n$ eines *let-rec*-Konstrukts

$$\textbf{let rec } x_1 = e_1 \textbf{ and } \ldots \textbf{ and } x_n = e_n \textbf{ in } e_0$$

besteht dagegen aus den Ausdrücken $e_0, e_1, \ldots, e_n$. Diese Regelung macht die 1-Pass-Übersetzbarkeit einer Programmiersprache schwierig: der Übersetzer benötigt zur Übersetzung einer Deklaration gegebenenfalls Informationen über einen Bezeichner, dessen Deklaration noch nicht bearbeitet wurde. In PASCAL, ADA und C gibt es deshalb *forward*-Deklarationen, um dieses Problem zu vermeiden.

PROLOG hat mehrere Klassen von Bezeichnern, die durch ihre syntaktische Position charakterisiert sind. Da es keine Verdeckung gibt, stimmen Gültigkeit und Sichtbarkeit überein. Die Bezeichner aus den verschiedenen Klassen haben die folgenden Gültigkeitsbereiche:

- Prädikate, Atome und Konstruktoren haben globale Gültigkeit; sie sind im ganzen Prolog-Programm und in zugehörigen Anfragen gültig.
- Bezeichner von Klauselvariablen haben Gültigkeit nur in der Klausel, in der sie vorkommen.

Explizite Deklarationen gibt es nur für Prädikate: diese werden durch die Liste ihrer Alternativen spezifiziert.

Beispiel: JAVA

Als etwas komplexeres Beispiel betrachten wir die Programmiersprache JAVA. In JAVA gibt es Bezeichner für Pakete, für Typen bzw. Klassen oder Interfaces, für Attribute, Konstruktoren und Methoden, für lokale Variablen und lokale Sprungziele in Methoden und für Parameter von generischen Klassen und Methoden.

Bezeichner von Paketen, öffentlichen Klassen oder Interfaces, Attributen, Konstruktoren und Methoden sind grundsätzlich im gesamten Programm gültig. Fehlt jede Qualifizierung der Gültigkeit, sind Klassen, Interfaces, Attribute, Konstruktoren und Methoden nur innerhalb des aktuellen Pakets gültig. Formale Parameter einer generischen Klasse sind grundsätzlich nur innerhalb dieser Klasse gültig. Formale Parameter oder lokale Variablen oder Sprungziele einer Methode sind nur innerhalb der Methode gültig.
Ein Bezeichner, der an einer Stelle im Programm gültig ist, muss dort jedoch noch lange nicht *sichtbar* sein, d.h. verwendet werden dürfen. Betrachten wir dazu zuerst einmal den einfachen Fall eines Bezeichners x, der innerhalb eines Blocks b deklariert wurde. Dann ist er ab seiner Deklaration in dem gesamten Block gültig und sichtbar. Ein anderer Bezeichner mit dem gleichen Namen x darf innerhalb des Blocks b nicht deklariert werden. Der Block:

```
{
    int x ← 1;
    {
        double x ← 2.0;
    }
}
```

wird deshalb vom JAVA-Übersetzer zurück gewiesen. Die beiden aufeinander folgenden Blöcke:

```
{
    int x ← 1;
}
{
    double x ← 2.0;
}
```

sind vielleicht nicht besonders sinnvoll, werden aber vom Übersetzer akzeptiert. Im Hinblick auf mehrfache Deklaration des gleichen Bezeichners ist die Programmiersprache JAVA offenbar restriktiver als funktionale Programmiersprachen wie SML, HASKELL oder OCAML.

Attribute und Methoden einer Klasse C verhalten sich etwas anders als lokale Variablen. Ein Attribut oder eine Methode x ist zuerst einmal innerhalb der gesamten Klasse C sichtbar, kann aber durch die Deklaration eines anderen Bezeichners gleichen Namens *verdeckt* werden. Ist der verdeckende Bezeichner eine lokale Variable, ist der verdeckte Bezeichner nur bis zu der letzten Anweisung vor der verdeckenden Deklaration sichtbar. Innerhalb des restlichen Blocks, in dem der verdeckende Bezeichner gültig ist, ist der verdeckte Bezeichner nicht sichtbar.

Beispiel 4.1.3 Betrachten Sie die Klasse:

```
class Block {
    int a ← 1;
    int foo(int x) {
        int a ← x − 1;
        return a;
    }
}
```

In der *return*-Anweisung der Methode $foo()$ ist die lokale Variable a sichtbar, das Attribut a dagegen verdeckt. Deshalb liefert die Methode $foo(x)$ nicht den Wert des Attributs a zurück, sondern den Wert

der lokalen Variablen a. In der Anweisung selbst, die die lokale Variable a deklariert und initialisiert, ist das Attribut a selbst nicht mehr sichtbar. □

Ist der verdeckende Bezeichner keine lokale Variable, sondern ein Attribut oder eine Methode einer inneren Klasse, dann ist der verdeckte Bezeichner x innerhalb der gesamten inneren Klasse nicht sichtbar.

Beispiel 4.1.4 Betrachten Sie die Klasse:

```
class Outer {
    int a ← 1;
    class Inner {
        int a ← Outer.this.a;
    }
}
```

In der inneren Klasse *Inner* der äußeren Klasse *Outer* ist das Attribut a der äußeren Klasse nicht direkt sichtbar, d.h. verwendbar. Ein Vorkommen von a kann jedoch durch die explizite Qualifizierung mit dem Namen der äußeren Klasse und dem Schlüsselwort **this** sichtbar gemacht werden. □

Sehen wir uns die Regeln zur Sichtbarkeit von Bezeichnern für Attribute, Methoden und Klassen im Hinblick auf die Paketstruktur und Vererbungshierarchie eines Programms etwas näher an. Ist der Bezeichner x innerhalb einer Klasse C als **private** deklariert, ist sein Sichtbarkeitsbereich auf die aktuelle Klasse (und ihre inneren Klassen) beschränkt. Nur dort darf er verwendet werden.

Ist x dagegen als **public** deklariert, kann er nicht nur in allen Unterklassen verwendet werden, sondern auch im gesamten übrigen Programm. Ist keine Qualifizierung angegeben, kann x im gesamten aktuellen Paket verwendet werden. Ist x in der Klasse C als **protected** deklariert, ist sein Sichtbarkeitsbereich auf das aktuelle Paket und alle Unterklassen der Klasse C eingeschränkt. Die Einschränkung **protected** in JAVA ist damit weniger einschränkend als die Einschränkung *protected* in der Programmiersprache C++. In C++ ist ein Bezeichner, der als *protected* deklariert wurde, alleine in der aktuellen Klasse und ihren Unterklassen sichtbar.

Ein Attribut oder eine Methode x einer Oberklasse C, welches oder welche prinzipiell in der Unterklasse A sichtbar wäre, kann aber durch eine neue Deklaration von x verdeckt werden. Bei Methoden ist jedoch zu beachten, dass die Folge der Typen der Parameterliste eines Vorkommens von x verwendet wird, um zwischen mehreren möglichen Deklarationen von x zu unterscheiden. Gegebenenfalls kann die Instanz von x der Oberklasse von A auch durch explizite Qualifizierung mit dem Schlüsselwort $\mathbf{super}$ wieder sichtbar gemacht werden.

Innerhalb einer Klasse A, die verschieden von C, aber weder eine innere Klasse noch eine Unterklasse von C ist, mag der Bezeichner x der Klasse C vielleicht gültig sein. Er kann jedoch nur verwendet werden, wenn die Klasse bestimmt werden kann, aus der x stammt. Die Art, wie die Klasse des Bezeichners x ermittelt wird, hängt davon ab, ob x $\mathbf{static}$ ist, d.h. nur einmal für die gesamte Klasse C existiert, oder nicht, d.h. für jedes Objekt der Klasse C in einer eigenen Instanz vorliegt.

Betrachten wir zuerst einmal den statischen Fall. Gehört die Klasse C zu einem anderen Paket als die Klasse A, muss zur Identifizierung von x nicht nur die Klasse C, sondern zusätzlich dieses Paket angegeben werden. Ein Aufruf der statischen Methode *newInstance*() beispielsweise aus der Klasse *DocumentBuilderFactory* des Pakets *javax.xml.parsers* hat die Form:

javax.xml.parsers.DocumentBuilderFactory.newInstance()

Solche Bandwurmqualifizierungen sind bei wiederholtem Vorkommen etwas mühsam. Deshalb stellt JAVA eine *import*-Direktive zur Verfügung. Z.B. macht die Direktive:

import *javax.xml.parsers.*$*$

am Anfang einer Datei alle öffentlichen Klassen des Pakets *javax.xml.parsers* ohne Qualifizierung des Pakets in allen Klassen der gegenwärtigen Datei sichtbar. Der Aufruf *newInstance*() der Klasse *DocumentBuilderFactory* kann dann abgekürzt werden zu:

DocumentBuilderFactory.newInstance()

Die Direktive:

static import *javax.xml.parsers.DocumentBuilderFactory.*∗

am Anfang einer Datei macht dagegen in dieser Datei nicht nur die Klasse *DocumentBuilderFactory*, sondern auch alle statischen öffentlichen Attribute und Methoden der Klasse *DocumentBuilderFactory* sichtbar. Ähnliche Direktiven, die gültige, aber nicht direkt sichtbare Bezeichner im aktuellen Kontext sichtbar machen, gibt es in vielen Programmiersprachen. In OCAML kann man etwa in einem Modul B durch: **open** *A* alle Variablen und Typen des Moduls A in B sichtbar machen, die öffentlich verfügbar sind.

Anders verhält es sich, wenn der Bezeichner x nicht **static** ist. Dann gibt es eine gesonderte Instanz von x für jedes Objekt der Klasse C. Die Klasse, zu der ein Vorkommen des Bezeichners x gehört, ergibt sich aus der *statischen* Klasse des Ausdrucks, dessen Wert zur Laufzeit das Objekt liefert, für das x selektiert wird.

Beispiel 4.1.5 Betrachten Sie die Klassendeklarationen:

```
class A {                    class B extends A {
    int a ← 1;                   int b ← 2;
}                                int foo() {
                                     A o ← new B();
                                     return o.a;
                                 }
                             }
```

Der statische Typ des Attributs *o* ist A. Zur Laufzeit erhält das Attribut *o* als Wert jedoch ein Objekt der Unterklasse B von A. Sichtbar sind an den Objekten, zu denen sich *o* möglicherweise auswerten lässt, jedoch nur die sichtbaren Attribute, Methoden und inneren Klassen der Oberklasse A. □

An dem letzten Beispiel wird deutlich, dass die Menge der Bezeichner, die an einer bestimmten Programmstelle sichtbar sind und damit verwendet werden dürfen, gegebenenfalls von den Typen von Ausdrücken abhängt.

Zusammenfassung

Nicht an jeder Stelle des Gültigkeitsbereichs eines definierenden Vorkommens von x meint ein angewandtes Auftreten von x tatsächlich dieses definierende Vorkommen. Ist das definierende Vorkommen *global* zum aktuellen Block, d.h. nicht in dessen Deklarationsteil, so kann eine lokale Deklaration von x es *verdecken*. Es ist dann nicht *direkt sichtbar*. Es gibt aber mehrere Möglichkeiten, ein nicht direkt sichtbares definierendes Vorkommen eines Bezeichners x innerhalb seines Gültigkeitsbereichs sichtbar zu machen. Die Sichtbarkeitsregeln einer Programmiersprache legen fest, auf welche definierenden Vorkommen eines Bezeichners sich ein angewandtes Vorkommen beziehen kann. Die *Erweiterung eines Bezeichners* um den Bezeichner eines die Deklaration enthaltenden Konstrukts ermöglicht den Bezug auf ein verdecktes definierendes Vorkommen. Wie komplex solche Regeln sein können, haben wir am Beispiel der Programmiersprache JAVA gesehen.

- Einige Direktiven erlauben es, ein verdecktes definierendes Vorkommen eines Bezeichners in einem Teil des Gültigkeitsbereichs, einer *Region*, ohne Bezeichnererweiterung sichtbar zu machen. Bei JAVA hatten wir hier die *import*-Anweisungen kennen gelernt. Diese Direktiven sind Teil der statischen und nicht der dynamischen Semantik. Die Grenzen der Region sind entweder durch eine eigene Anfang- und Endklammerung bestimmt, wie bei der *with*-Direktive in PASCAL, oder sie sind gleich den Grenzen der sie direkt enthaltenden Programmeinheit. In JAVA ist dies die Datei. Die *use*-Direktive in ADA listet Bezeichner von umgebenden Programmeinheiten auf, deren Deklarationen dadurch sichtbar werden. Die Sichtbarkeit dieser Bezeichner erstreckt sich vom Ende der *use*-Direktive bis zum Ende der umfassenden Programmeinheit.

- Kennt eine Sprache (und nicht nur ihre Implementierung) das Konzept der getrennten Übersetzung von Programmeinheiten, gibt es Direktiven, die Definitionen aus getrennt übersetzten Einheiten sichtbar machen. Jede getrennt übersetzbare Programmeinheit kann einige ihrer Definitionen zur Benutzung anbieten. In JAVA kann dies mit Hilfe der Modifizierer **public**, **protected** und **private** gesteuert werden. In ADA bietet ein Paket (Modul) die Definitionen aus seinem öffentlichen Teil an, eine einzelne Prozedur ihre formalen Parameter. Diesen Bezeichnern ordnen wir einen Gültigkeitsbereich zu, der alle Programme umfasst, die nach getrennter Übersetzung zusammengebunden werden. Die *with*-Direktive in ADA macht dann solche von separat übersetzten Einheiten angebotenen und in der *with*-Liste erwähnten Bezeichner sichtbar.

Zusammenfassend halten wir fest: Der Gültigkeitsbereich eines definierenden Auftretens eines Bezeichners x ist der Teil eines Programms, in dem der Bezeichner benutzt werden kann, um das ihm in der Definition zugeordnete Objekt anzusprechen. In ihrem Gültigkeitsbereich ist die Definition entweder direkt sichtbar, oder sie kann sichtbar gemacht werden.

4.1.2 Überprüfung der Kontextbedingungen

Wir werden nun skizzieren, wie man in Übersetzern die Einhaltung der Kontextbedingungen überprüft. Dazu betrachten wir den einfachen Fall einer Programmiersprache mit geschachtelten Blöcken aber ohne Überladung, die jedoch Neudeklarationen von Bezeichnern in inneren Blöcken erlaubt.

Die Aufgabe wird in zwei Teilaufgaben zerlegt. Die erste Aufgabe besteht darin, Bezeichner zu identifizieren und ihre Deklariertheitseigenschaften zu überprüfen. Diese Aufgabe nennen wir *Deklarations-Analyse*. Hier gehen die Gültigkeits- und die Sichtbarkeitsregeln der Programmiersprache ein. Die zweite Teilaufgabe, die *Typüberprüfung*, untersucht, ob die angegebenen Typinformationen für Programmbestandteile zusammen passen und leitet gegebenenfalls für bestimmte Programmteile, für die keine Typen angegeben wurden, einen Typ her.

Identifizierung von Bezeichnern

Gemäß den Gültigkeits- und Sichtbarkeitsregeln gehört (in unserem einfachen Fall) zu jedem angewandten Vorkommen eines Bezeichners in einem korrekten Programm genau ein definierendes Vorkommen. Die Identifizierung von Bezeichnern besteht darin, diesen Bezug von angewandten Vorkommen auf definierende Vorkommen herzustellen, bzw. festzustellen, dass kein solcher Bezug oder kein eindeutiger besteht. Das Ergebnis dieser Identifizierung wird von der Typüberprüfung und der Codeerzeugung benutzt. Deshalb muss es diese Phase überleben. Für die Darstellung der Korrespondenz zwischen angewandten und definierenden Vorkommen gibt es eine Reihe von Möglichkeiten. Traditionell erstellt ein Übersetzer eine sogenannte *Symboltabelle*, in der für jedes definierende Vorkommen eines Bezeichners die zugehörige deklarative Information abgespeichert ist. Diese Symboltabelle ist meist analog zur Blockstruktur des Programms organisiert, so dass man von jedem angewandten Vorkommen (schnell) zu dem korrespondierenden definierenden Vorkommen gelangen kann. Eine solche Symboltabelle ist nicht das Ergebnis der Identifizierung sondern dient nur dazu, diese vorzunehmen. Das Ergebnis der Identifizierung besteht darin, dass bei jedem Knoten für ein angewandtes Vorkommen eines Bezeichners x ein Verweis auf den Knoten für die Deklaration dieses Vorkommens von x abgespeichert ist.

Welche Operationen muss die Symboltabelle anbieten? Wenn der Deklarations-Analysator eine Deklaration antrifft, muss er den deklarierten Bezeichner und einen Verweis auf den zugehörigen Deklarationsknoten im Syntaxbaum in die Symboltabelle eintragen. Solch eine Deklaration steht in einem Block. Eine weitere Operation muss das Öffnen von Blöcken vermerken, eine andere das Schließen von Blöcken. Letztere kann die Einträge zu Deklarationen des geschlossenen Blocks aus der Symboltabelle entfernen. Dadurch enthält die Symboltabelle zu jeder Zeit genau die Einträge zu Deklarationen aller zu dieser Zeit geöffneten, aber noch nicht geschlossenen Blöcke. Trifft der Deklarationsanalysator auf ein angewandtes Vorkommen eines Bezeichners, so sucht er die Symboltabelle gemäß den Gültigkeits- und Sichtbarkeitsregeln nach dem Eintrag des zugehörigen definierenden Vorkommens ab. Hat er es

gefunden, so kopiert er den dort eingetragenen Verweis auf die Deklarationsstelle zum Knoten für das angewandte Vorkommen.

Damit sind insgesamt die folgenden Operationen auf der Symboltabelle notwendig:

(a)	*create_symb_table*	legt eine leere Symboltabelle an.
(b)	*enter_block*	vermerkt das Öffnen eines neuen Blocks.
(c)	*exit_block*	setzt die Symboltabelle auf den Stand zurück, den sie vor dem letzten *enter_block* hatte.
(d)	*enter_id*(id, *decl_ptr*)	fügt einen Eintrag für Bezeichner id in die Symboltabelle ein. Dieser enthält den Verweis auf seine Deklarationsstelle, die in *decl_ptr* übergeben wird.
(e)	*search_id(*id*)*	sucht das definierende Vorkommen zu id und gibt den Verweis auf die Deklarationsstelle zurück, wenn er existiert.

Die beiden letzten Operationen bzw. Funktionen arbeiten relativ zu dem letzten geöffneten Block, dem aktuellen Block.

Bevor die Implementierung der Symboltabelle, d.h. der oben aufgelisteten Prozeduren und Funktionen angegeben wird, wird ihre Benutzung bei der Deklarationsanalyse vorgeführt. Dazu nehmen wir ADA-ähnliche Gültigkeitsregeln an; d.h. ein definierendes Vorkommen eines Bezeichners ist erst ab dem Ende seiner Deklaration gültig.

```
proc analyze_decl (k : node);
    proc analyze_subtrees (root: node);
    begin
      for i := 1 to #descs(root) do      (* #descs: Zahl der Kinder *)
          analyze_decl(root.i)           (* i-tes Kind von root *)
      od
    end;
begin
    case symb(k) of          (* Markierung von k *)
    block:      begin
                    enter_block;
                    analyze_subtrees(k);
                    exit_block
                end;
    decl:       begin
                    analyze_subtrees(k);
                    foreach hier dekl. Bezeichner id do
                        enter_id(id, ↑ k)
                    od
                end;
    appl_id:    (* angew. Vorkommen eines Bezeichners id *)
                speichere search_id(id) an k;
    otherwise: if k kein Blatt then analyze_subtrees(k) fi
    od
end
```

Die Ada-Gültigkeitsregeln drücken sich darin aus, dass in dem *decl*-Fall der *case*-Anweisung erst alle Deklarationen rekursiv abgearbeitet werden, bevor die lokalen Deklarationen eingetragen werden. Die Modifikation dieses Algorithmus für ALGOL-ähnliche und PASCAL-ähnliche Gültigkeitsregeln bleiben dem Leser überlassen (siehe Übung 1.4).

Überprüfung der Typkonsistenz

Die Überprüfung der Typkonsistenz kann in einem bottom up-Pass über Ausdrucksbäume erfolgen. Für terminale Operanden, die Konstanten sind, steht der Typ schon fest; für Bezeichner besorgt man sich den Typ von seiner Definitionsstelle. Für jeden Operator schlägt man in einer Tabelle nach (siehe Abb. 4.4), ob die Typen der Operanden zu ihm passen und welches der Ergebnistyp ist. Bei der Überladung eingebauter Operatoren wird dabei noch die richtige Operation ausgewählt.

Erlaubt die Programmiersprache Typanpassungen, etwa von integer $\rightarrow$ real, so wird für jeden Operator und jede Kombination aus Operandentypen, die nicht zu ihm passen, geprüft, ob die Operandentypen durch Typanpassung zu einer für den Operator gültigen Kombination von Operandentypen gemacht werden können.

Implementierung der Symboltabelle

Bei der Implementierung einer Symboltabelle muss man darauf achten, dass die *search_id*-Funktion zu jedem Zeitpunkt von eventuell mehreren möglichen Einträgen für einen Bezeichner den gemäß der Sichtbarkeitsregeln richtigen findet.

Als erste Lösung könnte einem eine lineare Liste aus *enter_block*- und *enter_id*-Einträgen einfallen. Neue Einträge werden hinten angehängt und *exit_block* löscht von hinten alle *enter_id*-Einträge bis einschließlich dem letzten *enter_block*-Eintrag. *search_id* durchsucht die Liste von hinten und wird dabei alle gemäß der Ada-Gültigkeitsregel aktuell gültigen Bezeichner finden. Da diese lineare Liste offensichtlich kellerartig verwaltet wird, kann man sie auch als Keller organisieren.

An dieser Lösung stört der Aufwand für *search_id*, der linear von der Zahl der deklarierten Bezeichner abhängt. Logarithmische Suchzeit in jedem Block wird erreicht, wenn man für jeden Block die Einträge in einem binären Suchbaum verwaltet. Die Suche beginnt dann bei dem Suchbaum des aktuellen Blocks, fährt beim Suchbaum des umfassenden Blocks fort, bis ein definierendes Vorkommen gefunden wird oder festgestellt wird, dass kein solches existiert.

Wenn man davon ausgeht, dass jeder definierte Bezeichner mehrfach angewandt vorkommt, so sollte vor allen Dingen *search_id* sehr effizient, am besten in konstanter Zeit ablaufen. Das lässt sich mit folgender Datenstruktur verwirklichen:

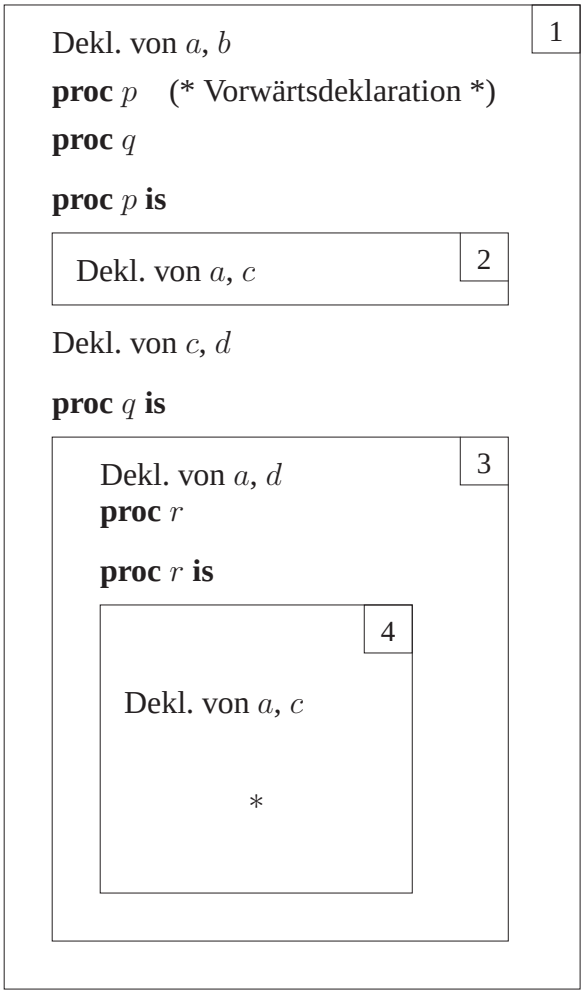
Die Einträge aller aktuell gültigen definierenden Vorkommen eines Bezeichners werden linear verkettet; ein neuer Eintrag wird hinten an diese Kette angefügt. Auf den jeweils letzten eingefügten Eintrag zeigt eine durch den Bezeichner indizierte Komponente eines Feldes. Außerdem sind alle zum gleichen Block gehörenden Einträge verkettet, um die Prozedur *exit_block* zu unterstützen. Auf diese Kette zeigt ein dem Block zugeordneter Listenkopf. Diese Listenköpfe können kellerartig verwaltet werden.

Beispiel 4.1.6
Für das Programm in Abbildung 4.3 und den mit $*$ markierten Punkt ergibt sich die in Abbildung 4.4 dargestellte Symboltabelle. □

Die Implementierung der Symboltabellenoperationen ist die folgende:

```
proc create_symb_table;
    begin
        kreiere leeren Keller von Blockeinträgen
    end;

proc enter_block;
    begin
        kellere neuen Eintrag für den neuen Block
    end;

proc exit_block;
    begin
        foreach Deklarationseintrag des aktuellen Blocks do
```

Abb. 4.3. Geschachtelte Gültigkeitsbereiche

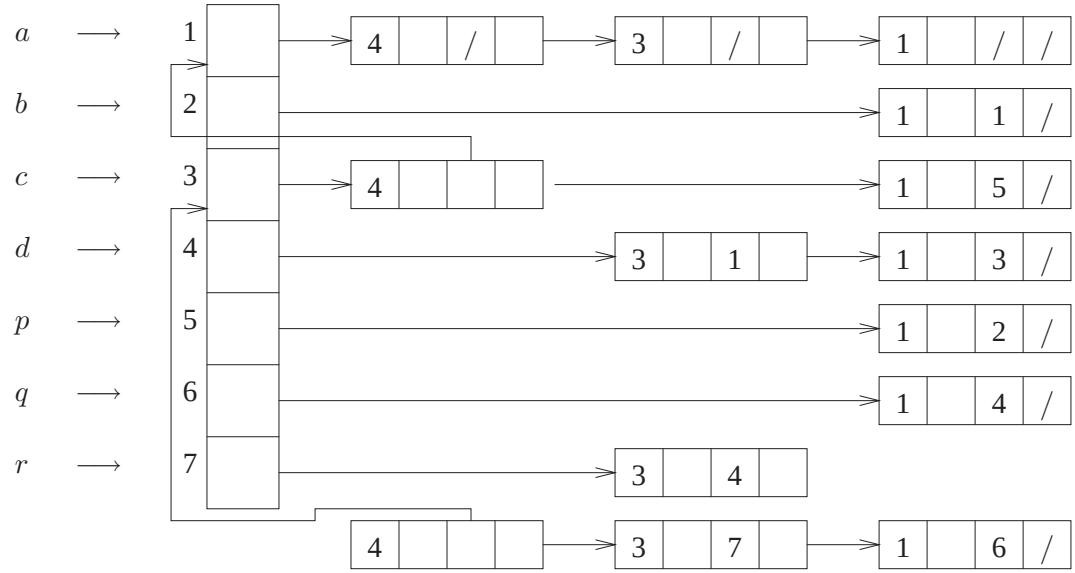


lösche Eintrag
od;
entferne Blockeintrag aus dem Keller
end;

proc *enter_id* (*id: Idno; decl:* ↑ *node*);
begin
if exist. bereits ein Eintrag für *id* in diesem Block
then *error*(″Doppeldeklaration″)
fi;
kreiere neuen Eintrag mit *decl* und Nr. des akt. Blocks;
füge diesen Eintrag hinten an die lineare Liste für *id* an;
füge diesen Eintrag hinten an die lineare Liste für diesen Block an
end;

function *search_id* (*id: idno*) ↑ *node;*
begin
if Liste für *id* ist leer

Abb. 4.4. Symboltabelle zum Programm aus Abbildung 4.3. Die Verkettung der Einträge zu einem Block ist nur für Block 4 angegeben, da sonst das Diagramm zu unübersichtlich würde. Sonst sind die "Adressen"der Kästchen, die Zahlen 1 - 7 angegeben. Das nicht ausgefüllte Kästchen in jedem Eintrag enthält jeweils den Verweis auf den Unterbaum für die Deklaration.



then *error*(″undeklarierter Bezeichner″)
else return (Wert des decl-Feldes aus erstem Eintrag in Liste *id*)
fi
end

4.1.3 Überladung von Bezeichnern

Ein Symbol heißt *überladen*, wenn es an einer Stelle im Programm mehrere Bedeutungen haben kann. Schon die Mathematik kennt überladene Symbole, etwa die arithmetischen Operatoren, die je nach Kontext Operationen auf den ganzen, den reellen oder den komplexen Zahlen oder sogar allgemein in Ringen oder Körpern bedeuten. Entsprechend der mathematischen Tradition haben schon die frühen Programmiersprachen Fortran und Algol60 die arithmetischen Operatoren überladen. Eine Typberechnung, wie sie im vorherigen Abschnitt vorgestellt wurde, dient im Übersetzer dazu, abhängig vom Typ der Operanden und oft auch noch vom verlangten Typ des Ergebnisses die richtige Operation zu einem überladenen Operator auszuwählen.

Programmiersprachen erlauben häufig die Überladung von benutzerdefinierten Symbolen, etwa von Prozedur- und Funktionsnamen. Dann können auch in korrekten Programmen bei einem angewandten Vorkommen eines Bezeichners x mehrere definierende Vorkommen von x sichtbar sein. Eine Redeklaration eines Bezeichners x verbirgt nur dann eine äußere Deklaration von x, wenn beide den gleichen Typ haben. Das Programm ist nur dann korrekt, wenn aufgrund der "Typumgebung"des angewandten Vorkommens genau eines der definierenden Vorkommen ausgewählt werden kann. Die Typumgebung bei Prozedur- oder Funktionsaufrufen besteht dabei in der Kombination der Typen der aktuellen Parameter.

Die Sichtbarkeitsregeln von Ada kombiniert mit den Möglichkeiten für die Überladung von Symbolen erfordern eine kaum überschaubare und verständliche Menge von Konfliktauflösungsregeln für die Fälle, wo auf verschiedene Weise sichtbare oder sichtbar gemachte aber nicht überladene Bezeichner in Konkurrenz stehen.

Beispiel 4.1.7 (Ada-Programm (Istvan Bach))

```
procedure BACH is
    procedure put (x: boolean) is begin null; end;
    procedure put (x: float)   is begin null; end;
    procedure put (x: integer) is begin null; end;
    package x is
       type boolean is (false, true);
       function f return boolean;                          -- (D1)
    end x;
    package body x is
       function f return boolean is begin null; end;
    end x;
    function f return float is begin null; end;            -- (D2)
    use x;
begin
    put (f);                                               -- (A1)
    A: declare
       f: integer;                                         -- (D3)
    begin
       put (f);                                            -- (A2)
       B: declare
          function f return integer is begin null; end;    -- (D4)
          begin
             put (f);                                      -- (A3)
           end B;
    end A;
end BACH;
```

Das Paket `x` deklariert in seinem öffentlichen Teil zwei neue Bezeichner, nämlich den Typ-Bezeichner `boolean` und den Funktions-Bezeichner `f`. Diese beiden Bezeichner werden durch die use-Anweisung `use x;` (siehe hinter (D2)) ab dem Semikolon (potentiell) sichtbar gemacht. Funktions-Bezeichner sind in Ada überladbar. Da die beiden Deklarationen von `f`, bei (D1) und (D2), verschiedene „Parameterprofile“ haben, d.h. in diesem Fall unterschiedliche Ergebnistypen, sind sie beide am Punkt (A1) (potentiell) sichtbar.

Die Deklaration `f: integer` in der Programmeinheit `A` (siehe (D3)) verdeckt die äußere Deklaration (D2) von `f`, da Variablen-Bezeichner in Ada nicht überladbar sind. Aus diesem Grunde ist auch die Deklaration (D1) nicht sichtbar. Die Deklaration (D4) von `f` in der Programmeinheit `B` verdeckt wiederum die Deklaration (D3), und da diese die Deklaration (D2) verdeckt, transitiv auch D2. Die durch die *use*-Anweisung (potentiell) sichtbar gemachte Deklaration (D1) wird allerdings nicht verdeckt, sondern ist nach wie vor potentiell sichtbar. Im Kontext `put(f)` (siehe (A3)) kann sich `f` nur auf die Deklaration (D4) beziehen, da die erste Deklaration von `put` einen anderen Typ, `boolean`, benutzt als der Ergebnistyp von `f` in (D1). □

Die Auswahl des richtigen definierenden Vorkommens eines überladenen Symbols nennt man die *Auflösung der Überladung* (overload resolution). Die Auflösung von Überladungen findet nach der Identifizierung von Bezeichnern innerhalb bestimmter Konstrukte der Sprache statt, d.h. beschränkt auf Ausdrücke, (zusammengesetzte) Bezeichner etc.

Der Auflösungsalgorithmus läuft auf der Darstellung des Ada-Programms als abstraktem Syntaxbaum ab. Konzeptionell benutzt er dabei vier Läufe über einen Ausdrucksbaum. Allerdings lässt sich der erste mit dem zweiten und der dritte mit dem vierten verschmelzen.

Um den Algorithmus zu formulieren, führen wir die folgende Notation ein: An jedem Knoten k des abstrakten Syntaxbaums erhalten wir über

$\#descs(k)$	die Zahl der Kindknoten von k,
$symb(k)$	das Symbol mit dem k markiert ist,
$vis(k)$	Menge der an k sichtbaren Definitionen von $symb(k)$
$ops(k)$	die Menge der aktuellen Kandidaten für das überladene Symbol $symb(k)$ und
$k.i$	wie üblich, das i-te Kind von k.

Für jedes definierende Vorkommen eines überladenen Symbols op mit Typ $t_1 \times \cdots \times t_m \rightarrow t$ sei

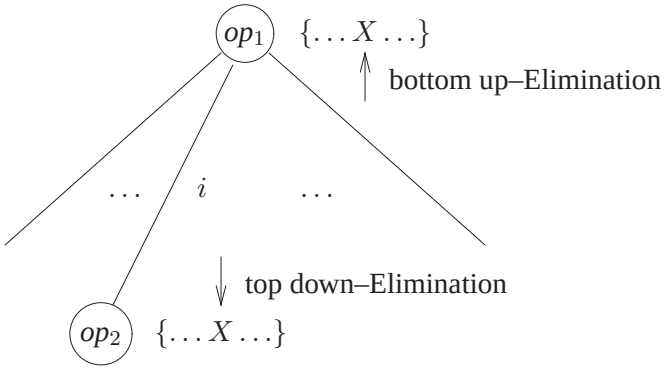
$$\begin{aligned} rank(op) &= m \\ res\_typ(op) &= t \\ par\_typ(op,i) &= t_i \quad (1 \leq i \leq m). \end{aligned}$$

Die beiden letzteren erweitern wir auf Mengen von Operatoren. Für jeden Ausdruck, in welchem die Überladung von Operatoren aufgelöst werden soll, wird aus seinem Kontext ein Typ, der sogenannte a-priori-Typ, berechnet.

```
proc resolve_overloading (root: node, a_priori_type: type);

func pot_res_types (k: node): set of type;
                (* potentielle Typen des Resultats *)
    return {res_typ(op) | op ∈ ops(k)}

func act_par_types (k: node, i: integer): set of type;
    return {par_typ(op, i) | op ∈ ops(k)}

proc init_ops
begin
    foreach k
        ops(k) := {op | op ∈ vis(k) und rank(op) = #descs(k)}
    od;
    ops(root) := {op ∈ ops(root) | res_typ(op) = a_priori_typ}
end;

proc bottom_up_elim (k: node);
begin
    for i := 1 to #descs(k) do
        bottom_up_elim (k.i);
        ops(k) := ops(k) − {op ∈ ops(k) | par_typ(op, i) ∉ pot_res_types(k.i)}
                (* entferne die Operatoren, deren i-ter Parametertyp zu keinem
                der möglichen Resultattypen des i-ten Operanden passt *)
    od;
end;

proc top_down_elim (k: node);
begin
    for i := 1 to #descs(k) do
        ops(k.i) := ops(k.i) − {op ∈ ops(k.i) | res_typ(op) ∉ act_par_types(k, i)};
                (* entferne die Operatoren, deren Resultattyp nicht zu
                irgendeinem Typ des zugehörigen Parameters passt *)
        top_down_elim(k.i)
    od;
end;

begin
    init_ops;
    bottom_up_elim(root);
    top_down_elim(root);
    prüfe, ob jetzt alle ops-Mengen einelementig sind; sonst Fehlermeldung
end
```

Es sieht so aus, als ob die bottom up-Elimination und die top down-Elimination das Gleiche täten. Das ist auch fast richtig. Abbildung 4.5 zeigt eine Kombination von op_1- und op_2- markierten Knoten. Mit jedem der Knoten ist eine Menge von möglichen Definitionen des Operators assoziiert. Die bottom up-Elimination löscht eventuell Kandidaten aus der Definitionsmenge von op_1, die top down-Elimination aus der von op_2.

Abb. 4.5.



4.2 Typinferenz

In imperativen Sprachen brauchen üblicherweise nur die Typen von Bezeichnern angegeben werden, aus denen die Typen von Ausdrücken abgeleitet werden. Im Gegensatz dazu werden in modernen *funktionalen* Programmiersprachen nicht nur die Typen von Ausdrücken, sondern auch die Typen der Bezeichner automatisch hergeleitet. Deshalb brauchen hier bei der Einführung neuer Bezeichner für Werte (i.A.) keine Typen angegeben werden.

Beispiel 4.2.1 Betrachten Sie die folgende OCAML-Funktion:

$$\begin{aligned}\textbf{let rec } \mathsf{fac} = \textbf{ fun } x \ \rightarrow \textbf{if } \ x \leq 0 \ \textbf{ then } \ 1 \\ \textbf{else } \ x \cdot \mathsf{fac}\ (x-1)\end{aligned}$$

Auf das Argument x der Funktion fac wird eine arithmetische Operation für ganze Zahlen angewendet. Deshalb muss der Typ des Arguments **int** sein. Weil der Rückgabewert entweder 1 ist oder mit Hilfe des Operators $\cdot$ für ganze Zahlen berechnet wird, muss der Typ des Rückgabewerts ebenfalls **int** sein. Daraus schließt der OCAML-Compiler, dass die Funktion fac den Typ: $\textbf{int} \rightarrow \textbf{int}$ hat, d.h. eine Funktion darstellt, die *int*-Werte als Argument erwartet und einen *int*-Wert zurück liefert. □

Die Idee automatischer Ableitung von Typen geht auf J.R. Hindley und R. Milner zurück. Ihnen folgend stellen wir Axiome und Regeln auf, die den Typ eines Ausdrucks in Beziehung setzen zu den Typen seiner Teilausdrücke, um die Menge möglicher Typen eines Ausdrucks zu charakterisieren. Der Einfachkeit halber betrachten wir nur eine funktionale *Kernsprache*, die an OCAML angelehnt ist. Eine ähnliche funktionale Kernsprache ist uns bereits im ersten Band: *Übersetzerbau: Virtuelle Maschinen* begegnet. Ein Programm in dieser Programmiersprache ist ein Ausdruck ohne freie Variablen, wobei Ausdrücke e gemäß der folgenden Grammatik gebildet werden:

$$
\begin{aligned}
e ::= {} & b \mid x \mid (\square_1\ e) \mid (e_1\ \square_2\ e_2) \\
& \mid (\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\
& \mid (e_1, \ldots, e_k) \mid [\,] \mid (e_1 :: e_2) \\
& \mid (\textbf{match}\ e_0\ \textbf{with}\ [\,] \rightarrow e_1 \mid h :: t \rightarrow e_2) \\
& \mid (\textbf{match}\ e_0\ \textbf{with}\ (x_1, \ldots, x_k) \rightarrow e_1) \\
& \mid (e_1\ e_2) \mid (\textbf{fun}\ (x_1, \ldots, x_m) \rightarrow e) \\
& \mid (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0) \\
& \mid (\textbf{let rec}\ x_1 = e_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = e_n\ \textbf{in}\ e_0)
\end{aligned}
$$

Dabei sind b Basiswerte, x Variablen und $\square_i$ $(i = 1, 2)$ i-stellige Operatoren auf Basiswerten. Der Einfachheit halber betrachten wir als zusammengesetzte Datenstrukturen nur Tupel und Listen. Pattern Matching kann dazu verwendet werden, um zusammengesetzte Werte in ihre Bestandteile zu zerlegen. Als Muster (engl. Patterns) für die Zerlegung verwenden wir dabei nur Muster, die genau einen Daten-Konstruktor enthalten.

Weiterhin benutzen wir die üblichen Präzedenz-Regeln und Assoziativitäten, um hässliche Klammern zu sparen.

Beispiel 4.2.2 Ein Programm in unserer funktionalen Kernsprache könnte etwa so aussehen:

$$
\begin{aligned}
&\textbf{let}\ \mathsf{rev} = \textbf{fun}\ x \rightarrow \\
&\qquad \textbf{let rec}\ \mathsf{r} = \textbf{fun}\ x \rightarrow \textbf{fun}\ y \rightarrow \textbf{match}\ x\ \textbf{with} \\
&\qquad\qquad\qquad\qquad [\,] \rightarrow y \\
&\qquad\qquad\quad \mid \qquad h :: t \rightarrow \mathsf{r}\ t\ (h :: y) \\
&\qquad \textbf{in} \qquad\quad \mathsf{r}\ x\ [\,]; \\
&\textbf{in} \qquad \mathsf{rev}\ [1; 2; 3]
\end{aligned}
$$

Dabei haben wir für Listen

$$a_1 :: (a_2 :: (\ldots (a_n :: [\,]) \ldots)$$

wie üblich die Kurzschreibweise

$$[a_1; \ldots; a_n]$$

verwendet. □

Auch für Typen verwenden wir eine Syntax, die an OCAML angelehnt ist – das heißt, dass der einstellige Typkonstruktor list für Listen rechts hinter sein Argument geschrieben wird und nicht davor. Typen t werden damit gemäß der folgenden Grammatik gebildet:

$$t ::= \textbf{int} \mid \textbf{bool} \mid (t_1 * \ldots * t_m) \mid t\ \mathsf{list} \mid (t_1 \rightarrow t_2)$$

Als *Basistypen* betrachten wir hier nur einen Typ **int** für ganze Zahlen und einen Typ **bool** für boolesche Werte. Ausdrücke können *freie Variablen* enthalten. Der Typ eines Ausdrucks hängt davon ab, welche Typen für die freien Variablen gewählt wurden. Unsere Annahmen über die Typen freier Variablen sammeln wir deshalb in einer *Typumgebung*. Eine Typumgebung Γ ist eine Abbildung von einer endlichen Menge von Variablen in die Menge der Typen. Eine Typumgebung Γ für einen Ausdruck e ordnet jeder freien Variable x von e einen Typ t zu, kann aber auch weiteren Variablen Typen zuordnen. Die Typaussage, dass der Ausdruck e unter der Annahme Γ den Typ t besitzt, schreiben wir kurz:

$$\Gamma \vdash e : t$$

Ein *Typsystem* gibt uns eine Menge von Axiomen und Regeln vor, mit deren Hilfe wir *gültige* Typaussagen herleiten können. Axiome sind dabei Aussagen, die ohne weitere Voraussetzung gültig sind, während es Regeln erlauben, aus gültigen Voraussetzungen neue gültige Aussagen abzuleiten. Im Folgenden listen wir die Axiome und Regeln für unsere funktionale Kernsprache auf. Als Axiome benötigen wir:

Const:	$\Gamma \vdash b : t_b$	(t_b Typ des Basiswerts b)
Nil:	$\Gamma \vdash [\,] : t\ \mathsf{list}$	(t beliebig)
Var:	$\Gamma \vdash x : \Gamma(x)$	(x Variable)

Jeder Familie von Axiomen haben wir dabei einen Namen gegeben, um später darauf Bezug nehmen zu können. Weiterhin haben wir angenommen, dass einem Basiswert b syntaktisch eindeutig ein Basistyp t_b zugeordnet werden kann.

Auch Regeln benennen wir mit Namen. Die Voraussetzungen oder Annahmen einer Regel schreiben wir oberhalb ihres Bruchstrichs auf, während die Schlussfolgerung darunter steht.

$$\textsc{Op}: \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \qquad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$

$$\textsc{Comp}: \quad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \mathbf{bool}}$$

$$\textsc{If}: \quad \frac{\Gamma \vdash e_0 : \mathbf{bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (\mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) : t}$$

$$\textsc{Tupel}: \quad \frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_m : t_m}{\Gamma \vdash (e_1, \ldots, e_m) : (t_1 * \ldots * t_m)}$$

$$\textsc{Cons}: \quad \frac{\Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t\ \mathsf{list}}{\Gamma \vdash (e_1 :: e_2) : t\ \mathsf{list}}$$

$$\textsc{Match}_1: \quad \frac{\Gamma \vdash e_0 : (t_1 * \ldots * t_m) \qquad \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} \vdash e_1 : t}{\Gamma \vdash (\mathbf{match}\ e_0\ \mathbf{with}\ (x_1, \ldots, x_m) \to e_1) : t}$$

$$\textsc{Match}_2: \quad \frac{\Gamma \vdash e_0 : t_1\ \mathsf{list} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \oplus \{x \mapsto t_1, y \mapsto t_1\ \mathsf{list}\} \vdash e_2 : t}{\Gamma \vdash (\mathbf{match}\ e_0\ \mathbf{with}\ [\,] \to e_1 \mid x :: y \to e_2) : t}$$

$$\textsc{App}: \quad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

$$\textsc{Fun}: \quad \frac{\Gamma \oplus \{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash \mathbf{fun}\ x \to e : t_1 \to t_2}$$

$$\textsc{Let}: \quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \oplus \{x_1 \mapsto t_1\} \vdash e_0 : t}{\Gamma \vdash (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0) : t}$$

$$\textsc{Letrec}: \quad \frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma' \vdash e_0 : t}{\Gamma \vdash (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_m = e_m\ \mathbf{in}\ e_0) : t} \quad \text{wobei} \quad \Gamma' = \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}$$

In der Regel OP haben wir als Beispiel den ganzzahligen Operator $+$ angenommenen. Analoge Regeln gibt es für die übrigen unären oder binären Operatoren. Im Falle boolescher Operatoren sind sowohl die Argumente wie die Ergebnisse vom Typ **bool**. Entsprechend wurde die Regel COMP für den Vergleichsoperator $=$ angegeben. Analoge Regeln gibt es in OCAML auch für die übrigen Vergleichsoperatoren. Beachten Sie, dass gemäß der Semantik von OCAML Vergleiche zwischen beliebigen Werten erlaubt sind, sofern sie nur desselben Typ haben.

Beispiel 4.2.3 Für den Rumpf der Funktion fac aus Beispiel 4.2.1 und die Typumgebung

$$\Gamma = \{\mathsf{fac} \mapsto \mathbf{int} \to \mathbf{int}, x \mapsto \mathbf{int}\}$$

ergibt sich die Ableitung:

$$\dfrac{\dfrac{\Gamma \vdash x : \mathbf{int} \quad \Gamma \vdash 0 : \mathbf{int}}{\Gamma \vdash x \leq 0 \ : \ \mathbf{bool}} \qquad \Gamma \vdash 1 \ : \ \mathbf{int} \qquad \dfrac{\Gamma \vdash x : \mathbf{int} \qquad \dfrac{\Gamma \vdash \mathsf{fac} : \mathbf{int} \to \mathbf{int} \qquad \dfrac{\Gamma \vdash x : \mathbf{int} \quad \Gamma \vdash 1 : \mathbf{int}}{\Gamma \vdash x - 1 : \mathbf{int}}}{\Gamma \vdash \mathsf{fac}\ (x-1) \ : \ \mathbf{int}}}{\Gamma \vdash x \cdot \mathsf{fac}\ (x-1) \ : \ \mathbf{int}}}{\Gamma \vdash \mathbf{if}\ x \leq 0\ \mathbf{then}\ 1\ \mathbf{else}\ x \cdot \mathsf{fac}\ (x-1) \ : \ \mathbf{int}}$$

Unter der Annahme, dass fac den Typ $\mathbf{int} \to \mathbf{int}$ und x den Typ $\mathbf{int}$ hat, lässt sich ableiten, dass der Rumpf der Funktion fac den Typ $\mathbf{int}$ hat. □

Die Regeln sind so gewählt, dass der Typ eines Ausdrucks bei der Auswertung erhalten bleibt. Diese Eigenschaft nennt man *Subject Reduction*. Könnten wir bei jeder Definition einer Variablen den zugehörigen Typ *raten*, ließe sich mithilfe der Regeln überprüfen, dass unsere Wahl *konsistent* war. Beachten Sie, dass ein Ausdruck mehrere Typen besitzen kann.

Beispiel 4.2.4 Der Ausdruck id, der gegeben ist durch

$$\mathbf{fun}\ x \to\ x$$

beschreibt die Identitätsfunktion. In jeder Typumgebung Γ und jeden Typ t lässt sich

$$\Gamma \vdash \mathsf{id} \ : \ t \to t$$

ableiten. □

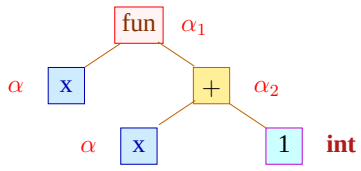
Um systematisch die Menge aller möglichen Typen eines Ausdrucks zu ermitteln, stellen wir ein *Gleichungssystem* auf, dessen Lösungen alle konsistenten Belegungen von Variablen und Ausdrücken mit Typen charakterisieren. Zum Aufstellen dieses Gleichungssystems gehen wir wie folgt vor.

Zuerst einmal machen wir die Namen der verschiedenen Variablen des Programms *eindeutig*. Dann erweitern wir Typterme, indem wir zusätzlich *Typvariablen* für unbekannte Typen von Variablen oder Teilausdrücken erlauben. Schließlich sammeln wir die Gleichungen zwischen den Typvariablen, die aufgrund der Axiome und Regeln des Typsystems notwendigerweise gelten müssen.

Beispiel 4.2.5 Betrachten Sie die Funktion:

$$\mathbf{fun}\ x \to x + 1$$

mit dem Syntaxbaum:



Als Typvariable für die Variable x wählen wir α, während α_1 und α_2 die Typen für die Ausdrücke $\mathbf{fun}\, x \to x+1$ und $x+1$ repräsentieren. Aus den Typregeln für Funktionen und Operatoranwendungen erhalten wir die Gleichungen:

$$\begin{aligned} \textsc{Fun} : \alpha_1 &= \alpha \to \alpha_2 \\ \textsc{Op} \ : \alpha_2 &= \mathbf{int} \\ \alpha &= \mathbf{int} \\ \mathbf{int} &= \mathbf{int} \end{aligned}$$

Wir schließen, dass

$$\alpha = \mathbf{int} \qquad \alpha_1 = \mathbf{int} \to \mathbf{int} \qquad \alpha_2 = \mathbf{int}$$

gelten muss. □

Bezeichne $\alpha[e]$ die Typvariable für den Ausdruck e. Jede Regel-Anwendung gibt dann Anlass zu den folgenden Gleichungen:

CONST:	$e \equiv b$	$\alpha[e] = t_b$
NIL:	$e \equiv [\,]$	$\alpha[e] = \alpha$ list $\quad (\alpha$ neu$)$
OP:	$e \equiv e_1 + e_2$	$\alpha[e] = \mathbf{int}$ $\alpha[e_1] = \mathbf{int}$ $\alpha[e_2] = \mathbf{int}$
COMP:	$e \equiv e_1 = e_2$	$\alpha[e_1] = \alpha[e_2]$ $\alpha[e] = \mathbf{bool}$
TUPEL:	$e \equiv (e_1, \ldots, e_m)$	$\alpha[e] = (\alpha[e_1] * \ldots * \alpha[e_m])$
CONS:	$e \equiv e_1 :: e_2$	$\alpha[e_2] = \alpha[e_1]$ list $\alpha[e] = \alpha[e_1]$ list
IF:	$e \equiv \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$	$\alpha[e_0] = \mathbf{bool}$ $\alpha[e] = \alpha[e_1]$ $\alpha[e] = \alpha[e_2]$
MATCH$_1$:	$e \equiv \mathbf{match}\ e_0\ \mathbf{with}\ (x_1, \ldots, x_k) \rightarrow e_1$	$\alpha[e_0] = (\alpha[x_1] * \ldots * \alpha[x_k])$ $\alpha[e] = \alpha[e_1]$
MATCH$_2$:	$e \equiv \mathbf{match}\ e_0\ \mathbf{with}\ [\,] \rightarrow e_1 \mid x :: y \rightarrow e_2$	$\alpha[y] = \alpha[x]$ list $\alpha[e_0] = \alpha[x]$ list $\alpha[e] = \alpha[e_1]$ $\alpha[e] = \alpha[e_2]$
FUN:	$e \equiv \mathbf{fun}\ x \rightarrow e_1$	$\alpha[e] = \alpha[x] \rightarrow \alpha[e_1]$
APP:	$e \equiv e_1\ e_2$	$\alpha[e_1] = \alpha[e_2] \rightarrow \alpha[e]$
LETREC:	$e \equiv \mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots\ \mathbf{and}\ x_m = e_m\ \mathbf{in}\ e_0$	$\alpha[x_1] = \alpha[e_1] \quad \ldots$ $\alpha[x_m] = \alpha[e_m]$ $\alpha[e] = \alpha[e_0]$

Beispiel 4.2.6 Für den Ausdruck id $\equiv$ **fun** $x \rightarrow x$ aus Beispiel 4.2.5 erhalten wir die Gleichung:

$$\alpha[\mathsf{id}] = \alpha[x] \rightarrow \alpha[x]$$

Unterschiediche Lösungen dieser Gleichung ergeben sich, indem wir unterschiedliche Typen t für $\alpha[x]$ wählen. □

In welchem Verhältnis stehen das Gleichungssystem zu einem Ausdruck e und die für diesen Ausdruck ableitbaren Typaussagen? Dazu nehmen wir zuerst einmal an, dass alle in e vorkommenden Bezeichner eindeutig sind. Sei V die Menge der in e vorkommenden Variablen. Im folgenden betrachten wir nur *uniforme* Ableitungen, d.h. Ableitungen von Typaussagen, die alle die gleiche Typumgebung verwenden. Man überzeugt sich, dass jede Ableitung einer Typaussage $\Gamma \vdash e : t$ für eine Typumgebung Γ für die freien Variablen in e zu einer uniformen Ableitung einer Typaussage $\Gamma' \vdash e : t$ für ein Γ' umgebaut werden kann, das mit Γ auf den freien Variablen von e übereinstimmt. Dann gilt:

Satz 4.2.1 Sei e ein Ausdruck, V die Menge der Variablen, die in e vorkommen, und E das Gleichungssystem zu dem Ausdruck e. Dann gilt:

1. Ist σ eine Lösung von E, dann gibt es eine uniforme Ableitung der Aussage:

$$\Gamma \vdash e : t$$

für

$$\Gamma = \{x \mapsto \sigma(\alpha[x]) \mid x \in V\} \qquad \text{und} \qquad t = \sigma(\alpha[e])$$

2. Sei A eine uniforme Ableitung für eine Typaussage $\Gamma \vdash e : t$, bei der für jeden Teilausdruck e' von e eine Typaussage $\Gamma \vdash e' : t_{e'}$ abgeleitet wird. Dann ist die Substitution σ definiert durch:

$$\sigma(\alpha[e']) = \begin{cases} t_{e'} & \text{falls } e' \text{ Teilterm von } e \text{ist} \\ \Gamma(x) & \text{falls } e' \equiv x \in V \end{cases}$$

Satz 4.2.1 sagt uns, dass wir aus den Lösungen des Gleichungssystems zu einem Ausdruck alle gültigen Typaussagen ablesen können. Die Gleichungssysteme, die hier auftreten, stellen Gleichheiten zwischen Typ-*Termen* her. Das Lösen solcher Termgleichungsysteme nennt man *Unifikation*.

Beispiel 4.2.7 1. Betrachten wir die Gleichung

$$Y = X \to X$$

wobei $\to$ ein zwei-stelliger Konstruktor ist, der infix notiert wird. Die Menge der Lösungen dieser Gleichung ist die Substitution

$$\{X \mapsto t, Y \mapsto (t \to t)\}$$

für jeden Term t. Als solchen Term t kann dabei auch die Variable X selbst gewählt werden.

2. Die Gleichung:

$$X \to \mathbf{int} = \mathbf{bool} \to Z$$

hat genau eine Lösung, nämlich die Substitution:

$$\{X \mapsto \mathbf{bool}, Y \mapsto \mathbf{int}\}$$

3. Die Gleichung:

$$\mathbf{bool} = X \to Y$$

hat dagegen *keine* Lösung. □

Wir führen die folgenden Begriffe ein. Eine Substitution σ heißt *idempotent*, wenn $\sigma \circ \sigma = \sigma$ ist. Konkret heißt das, dass keine Variable X mit $\sigma(X) \neq X$ im Bild von σ vorkommt. So ist die Substitution $\{X \mapsto \mathbf{bool}, Y \mapsto Z\}$ idempotent, die Substitution $\{X \mapsto \mathbf{bool}, Y \mapsto X\}$ dagegen nicht.

Bei Term-Gleichungssystemen reicht es, idempotente Lösungen zu betrachten. Eine idempotente Lösung σ eines Term-Gleichungssystems E heißt *allgemeinst*, wenn für jede andere idempotente Lösung τ von E gilt, dass $\tau = \tau' \circ \sigma$ für eine geeignete Substitution τ' ist. Damit ist die Substitution $\{Y \mapsto (X \to X)\}$ eine allgemeinste idempotente Lösung der Gleichung $Y = X \to X$ aus Beispiel 4.2.7. Der folgende Satz charakterisiert die idempotenten Lösungen endlicher Mengen von Term-Gleichungen.

Satz 4.2.2 Jedes System von Gleichungen $s_i = t_i, i = 1, \ldots, m$, zwischen Termen s_i, t_i hat entweder *keine* Lösung oder eine *allgemeinste idempotente* Lösung. □

Die Bestimmung sämtlicher (idempotenter) Lösungen eines Termgleichungssystems reduziert sich aufgrund von Satz 4.2.2 auf die Bestimmung einer allgemeinsten Lösung — sofern überhaupt eine Lösung existiert. Allgemeinste Lösungen können gelegentlich sehr groß werden. Betrachten Sie das folgende Gleichungssystem:

$$X_0 = (X_1 \to X_1), \ldots, X_{n-1} = (X_n \to X_n)$$

Die allgemeinste Lösung dieses Gleichungssystems bildet X_0 auf einen Term ab, in dem es 2^n Vorkommen der Variablen X_n gibt. Nach demselben Prinzip lässt sich auch leicht ein OCAML-Programm konstruieren, das zu exponentiell großen Typausdrücken führt. In den meisten praktisch nützlichen Programmen treten solche Typausdrücke jedoch nicht auf.

Verschiedene Techniken sind bekannt, wie man zu einer endlichen Menge von Termgleichungen eine allgemeinste idempotente Lösung berechnet. Eine solche Methode haben wir bereits im ersten Band *Übersetzerbau: Virtuelle Maschinen* kennen gelernt, als wir eine virtuelle Maschine für die Programmiersprache PROLOG vorstellten. In PROLOG ist Unifikation als Basis-Operation in der

Semantik verankert. In diesem Kapitel stellen wir ein funktionales Programm vor, das für eine Liste von Paaren $(s_i, t_i), i = 1, \ldots, m$, eine allgemeinste idempotente Lösung des Gleichungssystems $s_i = t_i, i = 1, \ldots, m$, berechnet.

Dieses funktionale Programm besteht aus den folgenden Funktionen:

- Die Funktion occurs nimmt ein Paar, bestehend aus einer Variablen X und einem Term t, und überprüft, ob X in t vorkommt.
- Die Funktion unify nimmt ein Paar (s, t) von Termen und eine idempotente Substitution θ und überprüft, ob die Gleichung $\theta(s) = \theta(t)$ lösbar ist. Falls dies nicht der Fall ist, liefert sie Fail zurück. Andernfalls liefert sie eine allgemeinste idempotente Substitution θ' zurück, die dieses Gleichungssystem erfüllt und außerdem eine Spezialisierung von θ ist, d.h. für die $\theta' = \theta' \circ \theta$ gilt.
- Die Funktion unifyList nimmt eine Liste $[(s_1, t_1); \ldots; (s_m, t_m)]$ von Termpaaren und eine idempotente Substitution θ und überprüft, ob das Gleichungssystem $\theta(s_i) = \theta(t_i), i = 1, \ldots, m$, lösbar ist. Falls dies nicht der Fall ist, liefert sie Fail zurück. Andernfalls liefert sie eine allgemeinste idempotente Substitution θ' zurück, die dieses Gleichungs erfüllt und außerdem eine Spezialisierung von θ ist.

Die Funktion occurs ist definiert durch:

$$
\begin{array}{lcl}
\textbf{let rec}\ \mathsf{occurs}\ (X, t) & = & \textbf{match}\ t \\
& & \textbf{with}\ X \qquad\qquad \rightarrow \mathsf{true} \\
& & \quad \mid \quad f(t_1, \ldots, t_k) \rightarrow \mathsf{occurs}\ (X, t_1) \vee \ldots \vee \mathsf{occurs}\ (X, t_k) \\
& & \quad \mid \quad \_ \qquad\qquad \rightarrow \mathsf{false}
\end{array}
$$

Stelvertretend für alle möglichen Konstruktoren, die in einem Term vorkommen können, steht im Programmtext hier der Konstruktor f der Stelligkeit $k \geq 1$. Die Funktionen unify und unifyList sind wechselseitig rekursiv. Sie sind definiert durch:

$$
\begin{array}{lcl}
\textbf{let rec}\ \mathsf{unify}\ (s, t)\ \theta & = & \textbf{if}\ \theta\, s \equiv \theta\, t\ \textbf{then}\ \theta \\
& & \textbf{else match}\ (\theta\, s, \theta\, t) \\
& & \quad \textbf{with}\ (X, t) \rightarrow \textbf{if}\ \mathsf{occurs}\ (X, t)\ \textbf{then}\ \mathsf{Fail} \\
& & \qquad\qquad\qquad\qquad \textbf{else}\ \{X \mapsto t\} \circ \theta \\
& & \qquad \mid \quad (t, X) \rightarrow \textbf{if}\ \mathsf{occurs}\ (X, t)\ \textbf{then}\ \mathsf{Fail} \\
& & \qquad\qquad\qquad\qquad \textbf{else}\ \{X \mapsto t\} \circ \theta \\
& & \qquad \mid \quad (f(s_1, \ldots, s_k), f(t_1, \ldots, t_k)) \rightarrow \mathsf{unifyList}\ [(s_1, t_1), \ldots, (s_k, t_k)]\ \theta \\
& & \qquad \mid \quad \_ \rightarrow \mathsf{Fail} \\
\textbf{and}\ \mathsf{unifyList}\ \mathsf{list}\ \theta & = & \textbf{match}\ \mathsf{list} \\
& & \quad \textbf{with}\ [\,] \rightarrow \theta \\
& & \qquad \mid \quad ((s, t) :: \mathsf{rest}) \rightarrow \textbf{let}\ \theta = \mathsf{unify}\ (s, t)\ \theta \\
& & \qquad\qquad\qquad\qquad\quad \textbf{in if}\ \theta = \mathsf{Fail}\ \textbf{then}\ \mathsf{Fail} \\
& & \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathsf{unifyList}\ \mathsf{rest}\ \theta
\end{array}
$$

Der Algorithmus startet mit dem Aufruf unifyList $[(s_1, t_1), \ldots, (s_m, t_m)]\ \emptyset$, d.h. dem Gleichungssystem zusammen mit einer leeren Substitution. Der Algorithmus terminiert und liefert entweder Fail zurück, wenn das Gleichungssystem keine Lösung hat, oder er liefert eine idempotente allgemeinste Lösung, wenn das Gleichungssystem lösbar ist.

Das bisherige Verfahren, um Typisierungen zu berechnen, hat den Nachteil, dass es nicht *syntaxgerichtet* ist. Wenn das Gleichungssystem zu einem Programm keine Lösung besitzt, erhalten wir keine Information, *wo* der Fehler herkommt. Eine möglichst genaue Eingrenzung der Fehlerursache ist jedoch für den Programmierer von großer Bedeutung. Deshalb modifizieren wir das gegebene Verfahren so, dass wir uns an der Syntax des Programmausdrucks orientieren. Diesen syntax-gerichteten Algorithmus notieren wir wieder als funktionales Programm, das mit Pattern Matching Fallunterscheidungen über die möglichen Formen des Programmausdrucks macht. Um die Syntax des Ausdruck e von der Syntax

des Algorithmus zu unterscheiden, schreiben wir die Schlüsselworte in e in Großbuchstaben und setzen die Operatoren in Hochkommata.

Der Algorithmus $\mathcal{W}$.

Ein Aufruf der Funktion $\mathcal{W}$ wird rekursiv über der Struktur eines Ausdrucks e ausgewertet. In einem zusätzlichen akkumulierenden Parameter reicht er dabei eine Typumgebung Γ und eine Substitution von Typvariablen θ weiter. Als Ergebnis liefert der Aufruf einen Typausdruck t für e und die während der Auswertung akkumulierte Substitution. Bei der nachfolgenden Beschreibung sind die Aufrufe der Hilfsfunktion unify jeweils hervorgehoben. Um die Lesbarkeit zu erhöhen, wird dabei jeweils angenommen, dass die Aufrufe von unify stets eine Substitution zurück liefern. Falls die Unifikation bei einem solchen Aufruf fehlschlagen sollte, sollte stattdessen eine Fehlermeldung generiert werden und die Typüberprüfung entweder abgebrochen oder mit einer sinnvollen Korrektur fortgesetzt werden.

$$
\begin{array}{lll}
\textbf{let rec}\ \mathcal{W}\ e\ (\Gamma,\theta) & = & \textbf{match}\ e \\
\quad \textbf{with}\ c & \rightarrow & (t_c,\theta) \\
\quad \mid\ [\,] & \rightarrow & \textbf{let}\ \alpha = \mathsf{new}() \\
 & & \textbf{in}\ (\alpha\ \mathsf{list},\theta) \\
\quad \mid\ x & \rightarrow & (\Gamma(x),\theta) \\
\quad \mid\ (e_1,\ldots,e_m) & \rightarrow & \quad \textbf{let}\ (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
 & & \qquad\qquad \ldots \\
 & & \textbf{in let}\ (t_m,\theta) = \mathcal{W}\ e_m\ (\Gamma,\theta) \\
 & & \textbf{in}\ \ ((t_1 * \ldots * t_m),\theta) \\
\quad \mid\ (e_1 :: e_2) & \rightarrow & \quad \textbf{let}\ (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
 & & \textbf{in let}\ (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) \\
 & & \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (t_1\ \mathsf{list}, t_2)\ \theta} \\
 & & \textbf{in}\ \ (t_2,\theta) \\
\quad \mid\ (e_1' +' e_2) & \rightarrow & \quad \textbf{let}\ (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
 & & \textbf{in let}\ (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) \\
 & & \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\textbf{int}, t_1)\ \theta} \\
 & & \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\textbf{int}, t_2)\ \theta} \\
 & & \textbf{in}\ \ (\textbf{int},\theta) \\
\quad \mid\ (e_1' =' e_2) & \rightarrow & \quad \textbf{let}\ (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
 & & \textbf{in let}\ (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) \\
 & & \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta} \\
 & & \textbf{in}\ \ (\textbf{bool},\theta) \\
\quad \mid\ (e_1\ e_2) & \rightarrow & \quad \textbf{let}\ (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
 & & \textbf{in let}\ (t_2,\theta) = \mathcal{W}\ e_2\ (\Gamma,\theta) \\
 & & \textbf{in let}\ \alpha = \mathsf{new}\ () \\
 & & \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (t_1, t_2 \rightarrow \alpha)\ \theta} \\
 & & \textbf{in}\ \ (\alpha,\theta) \\
 & \ldots &
\end{array}
$$

Stellvertretend für Operatoren auf Werten aus Basistypen ist hier der zweistellige Operator $+$ angegeben. Entsprechend wurde stellvertretend für Vergleichsoperatoren die Gleichheit behandelt. Beachten Sie, dass die Unifikationsaufrufe hier direkt die Gleichungen umsetzen, die zu den entsprechenden Ausdrücken gehören. Im Falle von Konstanten, der leeren Liste, Anwendungen des Tupel-Konstruktors sowie einzelner Variablen ist keine Unifikation erforderlich, um den Typ des Ausdrucks zu ermitteln. Die

Hilfsfunktion $\mathsf{new}()$ liefert jeweils eine *neue* Typvariable. Der Algorthmus vermeidet jedoch, neue Typvariablen anzufordern, wann immer dies möglich ist. So wird z.B. der Ergebnistyp für den Ausdruck e aus den Typen der Komponenten zusammengesetzt. Dies ist jedoch nicht für den Typ der leeren Liste möglich, weil der Typ eventueller Elemente der Liste an dieser Stelle noch nicht bekannt ist. Das ist auch nicht bei der Funktionsanwendung möglich, bei der der Ergebnistyp eine *Komponente* des Typs der Funktion ist.

$$
\begin{array}{l}
\mid (\textsc{if}\ e_0\ \textsc{then}\ e_1\ \textsc{else}\ e_2) \\
\qquad \rightarrow \quad \textbf{let}\ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\textbf{bool}, t_0)\ \theta} \\
\qquad\quad \textbf{in let}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
\qquad\quad \textbf{in let}\ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma, \theta) \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta} \\
\qquad\quad \textbf{in}\ \ (t_1, \theta) \\
\mid (\textsc{match}\ e_0\ \textsc{with}\ (x_1, \ldots, x_m) \rightarrow e_1) \\
\qquad \rightarrow \quad \textbf{let}\ \alpha_1 = \mathsf{new}() \\
\qquad\qquad\quad \ldots \\
\qquad\quad \textbf{in let}\ \alpha_m = \mathsf{new}() \\
\qquad\quad \textbf{in let}\ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\alpha_1 * \ldots * \alpha_m, t_0)\ \theta} \\
\qquad\quad \textbf{in let}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\}, \theta) \\
\qquad\quad \textbf{in}\ \ (t_1, \theta) \\
\mid (\textsc{match}\ e_0\ \textsc{with}\ [\,] \rightarrow e_1 \mid (x :: y) \rightarrow e_2) \\
\qquad \rightarrow \quad \textbf{let}\ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
\qquad\quad \textbf{in let}\ \alpha = \mathsf{new}() \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\alpha\ \mathsf{list}, t_0)\ \theta} \\
\qquad\quad \textbf{in let}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
\qquad\quad \textbf{in let}\ (t_2, \theta) = \mathcal{W}\ e_2\ (\Gamma \oplus \{x \mapsto \alpha, y \mapsto \alpha\ \mathsf{list}\}, \theta) \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (t_1, t_2)\ \theta} \\
\qquad\quad \textbf{in}\ \ (t_1, \theta) \\
\ldots
\end{array}
$$

Die zweite Gruppe von Fällen setzt die Regeln zur Behandlung von Fallunterscheidungen und Pattern Matching um. Falls mehrere Alternativausdrücke das Ergebnis liefern können, müssen ihre Typen übereinstimmen. Die Typen der Komponenten, in die ein Wert in den *match*-Fällen zerlegt wird, werden mit Hilfe von Unifikation ermittelt.

$$
\begin{array}{l}
\mid (\text{FUN } x \;\rightarrow\; e) \\
\qquad \rightarrow \quad \textbf{let } \alpha = \mathsf{new}() \\
\qquad\quad \textbf{in let } (t,\theta) = \mathcal{W}\ e\ (\Gamma \oplus \{x \mapsto \alpha\}, \theta) \\
\qquad\quad \textbf{in } (\alpha \rightarrow t, \theta) \\
\mid (\textbf{let } x_1 = e_1 \textbf{ in } e_0) \\
\qquad \rightarrow \quad \textbf{let } (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
\qquad\quad \textbf{in let } \Gamma = \Gamma \oplus \{x_1 \mapsto t_1\} \\
\qquad\quad \textbf{in let } (t_0,\theta) = \mathcal{W}\ e_0\ (\Gamma,\theta) \\
\qquad\quad \textbf{in } \ (t_0,\theta) \\
\mid (\text{LET REC } x_1 = e_1 \text{ AND} \ldots \text{ AND } x_m = e_m \text{ IN } e_0) \\
\qquad \rightarrow \quad \textbf{let } \alpha_1 = \mathsf{new}() \\
\qquad\qquad \ldots \\
\qquad\quad \textbf{in let } \alpha_m = \mathsf{new}() \\
\qquad\quad \textbf{in let } \Gamma = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\} \\
\qquad\quad \textbf{in let } (t_1,\theta) = \mathcal{W}\ e_1\ (\Gamma,\theta) \\
\qquad\quad \textbf{in let } \theta = \boxed{\mathsf{unify}\ (\alpha_1, t_1)\ \theta} \\
\qquad\qquad \ldots \\
\qquad\quad \textbf{in let } (t_m,\theta) = \mathcal{W}\ e_m\ (\Gamma,\theta) \\
\qquad\quad \textbf{in let } \theta = \boxed{\mathsf{unify}\ (\alpha_m, t_m)\ \theta} \\
\qquad\quad \textbf{in let } (t_0,\theta) = \mathcal{W}\ e_0\ (\Gamma,\theta) \\
\qquad\quad \textbf{in } \ (t_0,\theta)
\end{array}
$$

Die letzten drei Fälle behandeln Funktionen sowie Definitionen neuer Variablen. Sowohl für den unbekannten Typ des formalen Parameters einer Funktion wie für die unbekannten Typen der simultan definitierten Variablen werden jeweils neue Typvariablen angelegt, deren Bindungen während der Abarbeitung des Ausdrucks e bestimmt werden. Bei der Einführung einer Variablen x durch einen *let*-Ausdruck braucht dagegen keine neue Typvariable angelegt werden: der Typ der Variablen x ergibt sich direkt durch den Typ des definierenden Ausdrucks für x.

Aufgerufen wird die Funktion $\mathcal{W}$ für einen Ausdruck e mit einer Typumgebung Γ_0, die jeder Variable x, die in e vorkommt, eine neue Typvariable α_x zuordnet, und der leeren Substition $\emptyset$. Dann liefert der Aufruf keinen Rückgabewert genau dann, wenn es keine Typumgebung gibt, für die der Ausdruck e einen Typ hat. Liefert dagegen der Aufruf als Rückgabewert das Paar (t,θ), dann gibt es für jede ableitbare Typaussage $\Gamma' \vdash e : t'$ eine Substitution σ, so dass gilt:

$$t' = \sigma(\theta(t)) \qquad \text{und} \qquad \Gamma'(x) = \sigma(\theta(\Gamma(x))) \qquad \text{für alle Variablen } x$$

Der allgemeinste Typ für den Ausdruck e ist deshalb $\theta(t)$.

Polymorphie

Wird für eine Funktion der Typ $\alpha \rightarrow \alpha\,\textbf{list}$ abgeleitet, sollte man erwarten, dass diese Funktion *polymorph* ist, d.h. auf Werte beliebiger Typen angewendet werden darf. Das Typsystem, so wie wir es bisher definiert haben, lässt dies aber gegebenenfalls nicht zu.

Beispiel 4.2.8 Betrachten Sie den folgenden Programmausdruck:

$$
\begin{array}{l}
\textbf{let } \mathsf{single} = \textbf{fun } y \;\rightarrow\; [y] \\
\textbf{in } \ \mathsf{single}\ (\mathsf{single}\ 1)
\end{array}
$$

Für die Funktion single leiten wir den Typ

$$\alpha[\mathsf{single}] = (\gamma \to \gamma\,\mathsf{list})$$

ab. Aufgrund der Funktionsanwendung $(\mathsf{single}\ 1)$ wird die Typvariable γ mit dem Basistyp **int** instantiiert. Für die Funktionsanwendung $(\mathsf{single}\ 1)$ ergibt sich deshalb der Typ

$$\alpha[\mathsf{single}\,1] = \mathbf{int}\,\mathsf{list}$$

Die Typgleichung für die äußere Funktionsanwendung fordert deshalb die Instantiierung von γ mit **int** list. Da die Unifikation von **int** mit **int** list fehl schlägt, wird ein Typfehler gemeldet. □

Eine Möglichkeit, dieses Problem zu lösen, besteht darin, jede *let*-Definition für jede Benutzung der definierten Variable zu kopieren. Im Beispiel erhalten wir dann etwa:

$$\begin{array}{l}(\mathbf{let}\ \mathsf{single} = \mathbf{fun}\ y\ \to\ [y]\ \mathbf{in}\ \mathsf{single})\ (\\ \quad (\mathbf{let}\ \mathsf{single} = \mathbf{fun}\ y\ \to\ [y]\ \mathbf{in}\ \mathsf{single})\ 1)\end{array}$$

Die beiden Vorkommen des Teilausdrucks $(\mathbf{fun}\ \ y\ \ \to\ \ [y])$ werden nun unabhängig voneinander behandelt und erhalten deshalb Typen $\gamma \to \gamma\,\mathbf{list}$ und $\gamma' \to \gamma'\,\mathbf{list}$ für unterschiedliche Typvariablen γ, γ'. Das expandierte Programm ist nun typbar. Im Beispiel kann nun die eine Typvariable mit **int** und die andere mit **int list** instantiiert werden.

Die eben skizzierte Möglichkeit ist jedoch nicht sehr empfehlenswert, da das resultierende Programm nur unter bestimmten Bedingungen die gleiche Semantik hat wie das ursprüngliche Programm. Das so expandierte Programm kann zudem *seeehr* groß werden. Außerdem ist dann Typinferenz nicht mehr *modular*: für eine mehrfach verwendete Funktion einer anderen Übersetzungseinheit muss die Implementierung bekannt sein, um sie kopieren zu können.

Die bessere Idee besteht deshalb darin, nicht Code zu kopieren, sondern nur die Typen! Dazu erweitern wir Typen zu *Typschemata*. Ein Typschema erhält man aus einem Typ t, indem man zusätzlich einige der Typvariablen, die in t vorkommen, *generalisiert*: sie dürfen bei verschiedenen Benutzungen des Typs unterschiedlich instantiiert werden. In dem Typschema:

$$\forall \alpha_1, \ldots, \alpha_m.t$$

sind in t die Variablen $\alpha_1, \ldots, \alpha_m$ generalisiert. Alle weiteren Typvariablen, die in t vorkommen, müssen bei allen Benutzungen des Typschemas gleich instantiiert werden. Der Quantor $\forall$ erscheint nur auf dem obersten Schachtelungsniveau: der Ausdruck t darf keine weiteren $\forall$ enthalten. Typschemata werden für **let**-definierte Variablen eingeführt. Bei deren Benutzung können die Typvariablen im Schema unabängig mit beliebigen Typen instantiiert werden. Der Einfachheit halber fassen wir dabei normale Typausdrücke als Typschemata auf, bei denen eine *leere* Liste von Variaben generalisiert wurde. Als neue Regeln erhalten wir damit:

$$\textsc{Inst:}\quad \frac{\Gamma(x) = \forall\,\alpha_1, \ldots, \alpha_k.t}{\Gamma\ \vdash\ x\ :\ t[t_1/\alpha_1, \ldots, t_k/\alpha_k]} \qquad (t_1, \ldots, t_k \quad \text{beliebig})$$

$$\textsc{Let:}\quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \oplus \{x \mapsto \mathsf{close}\,t_1\,\Gamma\} \vdash e_0 : t_0}{\Gamma\ \vdash\ (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_0)\ :\ t_0}$$

Die Operation close nimmt einen Typausdruck t, eine Typumgebung Γ und generalisiert in t alle Typvariablen, die nicht in Γ vorkommen. Auch die Typen der Variablen, die in einer rekursiven Definition eingeführt werden, können generalisiert werden – aber nur für die Verwendung dieser Variablen im Hauptausdruck:

$$\textsc{Letrec:}\quad \frac{\Gamma' \vdash e_1 : t_1 \quad \ldots \quad \Gamma' \vdash e_m : t_m \quad \Gamma'' \vdash e_0 :\ t}{\Gamma\ \vdash\ (\mathbf{let\ rec}\ x_1 = e_1\ \mathbf{and} \ldots \mathbf{and}\ x_m = e_m\ \mathbf{in}\ e_0)\ :\ t}$$

Dabei ist

$$\begin{aligned}\Gamma' &= \Gamma \oplus \{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\}\\ \Gamma'' &= \Gamma \oplus \{x_1 \mapsto \mathsf{close}\,t_1\,\Gamma, \ldots, x_m, \mapsto \mathsf{close}\,t_m\,\Gamma\}\end{aligned}$$

Generalisiert werden damit alle Variablen in den Typausdrücken $t_1, \ldots, t_m$, die nicht auch in den Typen anderer Variablen der Typumgebung Γ vorkommen, die im Hauptausdruck sichtbar sind. Entscheidend ist, dass die Typen der *rekursiven* Vorkommen der Variablen x_i in den rechten Seiten $e_1, \ldots, e_m$ nicht instantiiert werden dürfen. Typsysteme, in denen solche *polymorphe Rekursion* erlaubt ist, sind, i.a. unentscheidbar.

Wir modifizieren nun den Algorithmus $\mathcal{W}$ so ab, dass er in der Typumgebung Typschemata verwaltet. Für den Fall einer Variablen benötigen wir die folgende Hilfsfunktion, die den Typausdruck in einem Typschema mit frischen Typvariablen instantiiert:

$$
\begin{array}{l}
\textbf{fun}\ \ \mathsf{inst}\ (\forall\, \alpha_1, \ldots, \alpha_k.\, t)\ = \\
\qquad\qquad\quad \textbf{let}\ \ \beta_1 = \mathsf{new}() \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad\ \ \textbf{in let}\ \ \beta_k = \mathsf{new}() \\
\qquad\qquad\ \ \textbf{in}\ \ t[\beta_1/\alpha_1, \ldots, \beta_k/\alpha_k]
\end{array}
$$

Dann ändern wir den Algorithmus $\mathcal{W}$ für Variablen und *let*-Ausdrücke wie folgt ab:

$$
\begin{array}{l}
\ldots \\
\mid x \qquad \rightarrow (\mathsf{inst}\ (\theta(\Gamma(x)))) \\
\mid (\textsc{let}\ x_1 = e_1\ \textsc{in}\ e_0) \\
\qquad\qquad \rightarrow \quad \textbf{let}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma, \theta) \\
\qquad\qquad\qquad\ \ \textbf{in let}\ s_1 = \mathsf{close}\ (\theta\ t_1)\ (\theta \circ \Gamma) \\
\qquad\qquad\qquad\ \ \textbf{in let}\ \Gamma = \Gamma \oplus \{x_1 \mapsto s_1\} \\
\qquad\qquad\qquad\ \ \textbf{in let}\ (\theta\, t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma, \theta) \\
\qquad\qquad\qquad\ \ \textbf{in}\ \ (t_0, \theta)
\end{array}
$$

Entsprechend modifizieren wir den Algorithmus $\mathcal{W}$ für *letrec*-Ausdrücke. Hier müssen wir darauf achten, dass die inferierten Typen für die neu eingeführten Variablen nur für ihre Verwendung im Hauptausdruck generalisert werden:

$$
\begin{array}{l}
\mid (\textsc{let rec}\ x_1 = e_1\ \textsc{and} \ldots\ \textsc{and}\ x_m = e_m\ \textsc{in}\ e_0) \\
\qquad \rightarrow \quad \textbf{let}\ \alpha_1 = \mathsf{new}() \\
\qquad\qquad\ \ \ldots \\
\qquad\quad \textbf{in let}\ \alpha_m = \mathsf{new}() \\
\qquad\quad \textbf{in let}\ \Gamma' = \Gamma \oplus \{x_1 \mapsto \alpha_1, \ldots, x_m \mapsto \alpha_m\} \\
\qquad\quad \textbf{in let}\ (t_1, \theta) = \mathcal{W}\ e_1\ (\Gamma', \theta) \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\alpha_1, t_1)\ \theta} \\
\qquad\qquad\ \ \ldots \\
\qquad\quad \textbf{in let}\ (t_m, \theta) = \mathcal{W}\ e_m\ (\Gamma', \theta) \\
\qquad\quad \textbf{in let}\ \theta = \boxed{\mathsf{unify}\ (\alpha_m, t_m)\ \theta} \\
\qquad\quad \textbf{in let}\ s_1 = \mathsf{close}\ (\theta\, t_1)\ (\theta \circ \Gamma\}) \\
\qquad\qquad\ \ \ldots \\
\qquad\quad \textbf{in let}\ s_m = \mathsf{close}\ (\theta\, t_m)\ (\theta \circ \Gamma) \\
\qquad\quad \textbf{in let}\ \Gamma' = \Gamma \oplus \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\} \\
\qquad\quad \textbf{in let}\ (t_0, \theta) = \mathcal{W}\ e_0\ (\Gamma', \theta) \\
\qquad\quad \textbf{in}\ \ (t_0, \theta)
\end{array}
$$

Beispiel 4.2.9 Betrachten wir erneut den *let*-Ausdruck:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{single} = \textbf{fun}\ y\ \rightarrow\ [y] \\
\textbf{in}\ \ \mathsf{single}\ (\mathsf{single}\ 1)
\end{array}
$$

aus Beispiel 4.2.8. Für die Funktion single findet der Algorithmus $\mathcal{W}$ das Typschema $\forall\,\gamma.\gamma \to \gamma\,\mathbf{list}$. Dieses Typschema wird bei den beiden verschiedenen Vorkommen von single im Hauptausdruck mit jeweils unterschiedlichen Typvariablen γ_1, γ_2 instantiiert, die dann zu den Typen $\mathbf{int\ list}$ und $\mathbf{int}$ spezialisiert werden. Insgesamt liefert deshalb der Algorithmus $\mathcal{W}$ für den *let*-Ausdruck den Typ $\mathbf{int\ list\ list}$. □

Relativ zu einer Typumgebung mit Typschemata für die globalen Variablen eines Ausdrucks berechnet der erweiterte Algorithmus $\mathcal{W}$ den *allgemeinsten* Typ eines Ausdrucks. Die Instantiierung von Typschemata bei jeder Benutzung einer Variablen ermöglicht es, *polymorphe* Funktionen zu definieren, die auf Werte unterschiedlicher Typen angewendet werden können. Typschemata erlauben auch eine *modulare* Typinferenz, weil für Funktionen aus anderen Programmteilen nur ihr Typ(schema) bekannt sein muss, um die Typen von Ausdrücken, in denen sie verwendet werden, zu berechnen.

Die Möglichkeit unterschiedlicher Instantiierung von Typschemata erlaubt jedoch, Programmausdrücke zu konstruieren, deren Typen nicht nur einfach exponentiell, sondern sogar doppelt exponentiell groß sind! Solche Beispiele sind jedoch eher akademisch. Für das praktische Programmieren spielen solche Ausdrücke keine besondere Rolle.

Seiteneffekte

Auch bei im wesentlichen funktionaler Programmierung sind Variablen, deren Werte geändert werden können, gelegentlich nützlich. Um die Probleme zu studieren, die sich aus solchen *änderbaren* Variablen für die Typinferenz ergeben, erweitern wir unsere kleine Programmiersprache um *Referenzen*:

$$e \;::=\; \ldots \;\mid\; \mathbf{ref}\ e \;\mid\; !e \;\mid\; e_1 := e_2$$

Beispiel 4.2.10 Mit Referenzen lässt sich elegant eine Funktion new implementieren, die bei jedem Aufruf einen anderen Wert liefert. Eine solche Funktion benötigten wir etwa zur Implementierung des Algorithmus $\mathcal{W}$. Betrachten Sie das folgende Programm:

```
   let count = ref 0
in let new   = fun () →
                   let ret = !count
                in let _   = count := ret + 1
                in  ret
in  new() + new()
```

Das leere Tupel $()$ ist das einzige Element des speziellen Typs $\mathbf{unit}$. Die Zuweisung eines Werts an eine Referenz ändert den Inhalt der Referenz als *Seiteneffekt*. Die Zuweisung selbst ist ebenfalls ein Ausdruck. Der Wert dieses Ausdrucks ist $()$. Da dieser Wert maximal uninteressant ist, wird in unserem Programm keine spezielle Variable für ihn bereit gestellt, sondern die *anonyme* Variable _ verwendet. □

Typausdrücke erweitern wir nun um den speziellen Typ $\mathbf{unit}$ und, indem wir **ref** als neuen einstelligen Typkonstruktor einführen:

$$t \;::=\; \ldots \;\mid\; \mathbf{unit} \;\mid\; t\,\mathbf{ref} \;\mid\; \ldots$$

Entsprechend erweitern wir die Regeln unseres Typsystems um:

$$\textsc{Ref:}\quad \frac{\Gamma \;\vdash\; e \;:\; t}{\Gamma \;\vdash\; (\mathbf{ref}\ e) \;:\; t\,\mathbf{ref}}$$

$$\textsc{Deref:}\quad \frac{\Gamma \;\vdash\; e \;:\; t\,\mathbf{ref}}{\Gamma \;\vdash\; (!\,e) \;:\; t}$$

$$\textsc{Assign:}\quad \frac{\Gamma \;\vdash\; e_1 \;:\; t\,\mathbf{ref} \qquad \Gamma \;\vdash\; e_2 \;:\; t}{\Gamma \;\vdash\; (e_1 := e_2) \;:\; \mathbf{unit}}$$

Diese Regeln sehen plausibel aus. Leider vertragen sie sich aber nicht mit Polymorphie.

Beispiel 4.2.11 Betrachten Sie den Programmausdruck:

$$\begin{array}{l} \quad \textbf{let}\ y = \textbf{ref}\ [\,] \\ \textbf{in let}\ \_ = y := 1 :: (!\, y) \\ \textbf{in let}\ \_ = y := \textbf{true} :: (!\, y) \\ \textbf{in}\ \ 1 \end{array}$$

Typinferenz führt zu keinerlei Widersprüchen. Für y liefert sie das Typschema $\forall\, \alpha.\, \alpha\,\mathsf{list}\,\textbf{ref}$. Zur Laufzeit wird jedoch eine Liste aufgebaut, die den *int*-Wert 1 zusammen mit dem booleschen Wert **true** enthält. Solche gemischten Listen sollten eigentlich durch das Typsystem verhindert werden. □

Das Problem in Beispiel 4.2.11 kann verhindert werden, wenn die Typen änderbarer Werte nie generalisiert werden. Dies wird durch die *Value Restriction* sichergestellt.

Die Menge der *Wertausdrücke* umfasst alle Ausdrücke, die keine Referenz und keine Funktionsanwendungen enthalten. Insbesondere ist *jede* Funktion $\textbf{fun}\ x \rightarrow e$ ein Wertausdruck. Die *Value Restriction* besagt nun, dass bei einem Ausdruck

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_0$$

nur Typvariablen des Typs von e_1 nur generalisiert werden dürfen, wenn e_1 ein Wertausdruck ist. Eine entsprechende Einschränkung kommt bei *letrec*-Ausdrücken zum Einsatz.

Weil der definierende Ausdruck für y in Beispiel 4.2.11 eine Referenz enthält, darf sein Typ nicht generalisiert werden. Die erste Verwendung von y legt den Typ von y auf $\textbf{int ref}$ fest. Die zweite Verwendung im Beispiel wird deshalb zu einem Typfehler führen.

Polymorphie ist ein sehr nützliches Hilfsmittel bei der Programmierung. In der Form von *Generics* hat es in JAVA 1.5 Einzug gehalten. Auch Typinferenz ist nicht mehr nur auf funktionale Sprachen beschränkt, da man in C# wie in JAVA erkannt hat, dass komplizierte und redundante Typdeklarationen den Code nicht unbedingt lesbarer machen.

Die Entwicklung ausdrucksstarker Typsysteme mit leistungsfähiger Typinferenz ist jedoch mit dem Hindley-Milner-Typsystem nicht zum Erliegen gekommen. In der Programmiersprache HASKELL wird mit verschiedenen Erweiterungen dieses Ansatzes experimentiert. Eine mittlerweile gut etablierte Idee besteht darin, *Bedingungen* an die Typen zu zulassen, die für eine generische Typvariable eingesetzt werden dürfen.

Beispiel 4.2.12 Betrachten Sie die folgende rekursive Funktion

$$\begin{array}{lll} \textbf{fun}\ \mathsf{member} & = & \textbf{fun}\, x \rightarrow \textbf{fun}\, \mathsf{list} \rightarrow \textbf{match}\ \ \mathsf{list} \\ & & \textbf{with}\ [\,] \quad \rightarrow \textbf{false} \\ & & \ \ \mid \quad h :: t \rightarrow \textbf{if}\ \ x\ =\ h\ \ \textbf{then}\ \ \textbf{true} \\ & & \qquad\qquad\quad \textbf{else}\ \ \mathsf{member}\ x\ t \end{array}$$

In OCAML hat die Funktion member den Typ: $\forall\, \alpha.\, \alpha \rightarrow \alpha\,\mathsf{list} \rightarrow \textbf{bool}$. Dies liegt daran, dass die Gleichheit für die Werte *sämtlicher* Typen definiert ist: bei manchen Typen wirft sie jedoch eine *Exception*. Das ist in der funktionalen Sprache SML anders: SML unterscheidet zwischen *Gleichheitstypen*, die Gleichheit unterstützen, und beliebigen Typen. Funktionstypen z.B. unterstützen Gleichheit nicht. Entsprechend dürfen in SML Typ für $member$ für α nur *Gleichheitstypen* eingesetzt werden. □

Überladung

Das Problem aus Beispiel 4.2.12 kann verallgemeinert werden. Oft ist eine Funktion, ist eine Datenstruktur nicht generell polymorph, sondern verlangt Daten, die bestimmte Operationen unterstützen. Die Funktion sort ist z.B. nur auf Listen anwendbar, deren Elemente eine Operation $\leq$ zulassen.

Im Folgenden erläutern wir in den Grundzügen, wie das Typsystem von Hindley-Milner erweitert werden kann, so dass Typparameter mit solchen Bedingungen versehen werden können. Die Vorgehensweise orientiert sich an dem Konzept von *Typklassen* der Programmierklasse HASKELL. Eine Bedingung an eine Typvariable α gibt an, welche Operationen die Typen, die für diese Variable eingesetzt werden dürfen, implementieren müssen. Eine *Typklasse* C versammelt alle Typen, die die Operationen, die zu C gehören, unterstützen. Typklassen zusammen mit den zugehörigen Operationen müssen explizit deklariert werden.

Beispiel 4.2.13 Beispiele für Typklassen sind etwa:

Name	Operation		
Gleichheitstypen	$(=)$	:	$\alpha \to \alpha \to \mathbf{bool}$
Vergleichstypen	$(\leq)$	:	$\alpha \to \alpha \to \mathbf{bool}$
Druckbare Typen	to_string	:	$\alpha \to \mathbf{string}$
Hashbare Typen	hash	:	$\alpha \to \mathbf{int}$

Dabei bezeichne $(\square)$ die zweistellige Funktion zu dem zweistelligen Infix-Operator $\square$. $\square$

Ein bedingtes Typschema für einen beliebigen Ausdruck hat die Form:

$$\forall \alpha_1 : S_1, \ldots, \alpha_m : S_m.\ s$$

wobei $S_1, \ldots, S_m$ endliche *Mengen* von Typklassen darstellen und s ein polymorphes Typschema ist und also selbst ebenfalls generalisierte Variablen enthalten kann. Eine Menge S von Typklassen, die als Bedingung auftritt, nennen wir auch *Sorte*. Wenn eine Sorte $S = \{C'\}$ einelementig ist, sparen wir uns auch die Mengenklammern. Der Einfachheit halber nehmen wir an, zu jeder Typklasse gehöre genau eine Operation. Um eine neue Typklasse C zu deklarieren, müssen die zugehörige Operation op_C und der Typ der Operation op_C spezifiziert werden:

$$\mathbf{class}\ C\ \ \mathbf{where}\ \ \mathsf{op}_C\ :\ \forall \alpha : C\ .\ t$$

für einen Typausdruck t. Das Typschema für op_C darf dabei nur eine generische Variable enthalten, welche durch C qualifiziert ist.

Neben der Deklaration von Klassen werden *Instanzdeklarationen* benötigt. Eine Instanzdeklaration für die Klasse C gibt an, unter welchen Bedingungen an die Argumenttypen die Anwendung eines k-stelligen Typkonstruktors b einen Typ in der Klasse C liefert und stellt eine Implementierung des Operators op_C der Klasse C bereit.

$$\begin{aligned}&\mathbf{inst}\ \ b(S_1, \ldots, S_k)\ :\ \ C\\ &\mathbf{where}\ \ \mathsf{op}_C = e\end{aligned}$$

Ein Operator, der für verschiedene Typen unterschiedliche Implementierungen besitzt, heißt *überladen*. Der Fall $k = 0$ deckt dabei den Fall eines Basistyps ab, bei dem keine Annahmen über Argumenttypen benötigt werden.

Beispiel 4.2.14 Eine Klasse Eq, die Gleichheitstypen zusammenfasst, zusammen mit zwei Instanzdeklarationen dieser Klasse könnte etwa so aussehen:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Eq}\ \textbf{where} \\
\quad (=): \quad \forall\,\alpha : C.\ \alpha \to \alpha \to \textbf{bool} \\
\\
\textbf{inst}\ \ (\mathsf{Eq} * \mathsf{Eq})\ :\ \ \mathsf{Eq} \\
\textbf{where}\ (=) = \textbf{fun}\, x\ \to \textbf{fun}\, y \to\ \textbf{match}\ x\ \textbf{with} \\
\qquad\qquad\qquad\qquad\qquad\qquad (x_1, x_2) \to\ \textbf{match}\ y\ \textbf{with} \\
\qquad\qquad\qquad\qquad\qquad\qquad (y_1, y_2) \to (x_1 = y_1) \wedge (x_2 = y_2) \\
\textbf{inst}\ \ \mathsf{Eq}\ \textbf{list}\ :\ \ \mathsf{Eq} \\
\textbf{where}\ (=) = \ \textbf{let rec}\ \mathsf{eq\_list} = \textbf{fun}\, l_1\ \to \textbf{fun}\, l_2 \to \textbf{match}\ l_1 \\
\qquad\qquad\qquad\qquad \textbf{with} \quad [] \qquad\ \to\ (\textbf{match}\ l_2\ \textbf{with}\ [] \to \textbf{true} \mid \_ \to \textbf{false}) \\
\qquad\qquad\qquad\qquad\ \mid \qquad x :: xs \to\ (\textbf{match}\ l_2\ \textbf{with}\ [] \to \textbf{false} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mid\ y :: ys\ \to\ \textbf{if}\ (=)\, x\, y\ \textbf{then}\ \mathsf{eq\_list}\, xs\, ys \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{else false}) \\
\qquad\qquad\qquad\qquad \textbf{in}\ \ \mathsf{eq\_list}
\end{array}
$$

Die Implementierung der Gleichheit für Paare wird mit Hilfe der Gleichheiten für die Komponententypen implementiert. Entsprechend benutzt die Implementierung der Gleichheit für eine Liste die Gleichheit für den Elementtyp der Liste. Aufgabe der Typinferenz ist damit nicht nur zu überprüfen, dass die verschiedenen Typen der Ausdrücke zusammen passen, sondern auch, für unterschiedliche Vorkommen eines Operators die richtige Implementierung ausfindig zu machen. □

Oft ist es praktisch, mehrere Operationen zu einer Klasse zusammenzufassen. Eine Klasse Number könnte z.B. alle Typen zusammenfassen, die die üblichen vier Grundrechenarten unterstützen. Zusammen mit einer Klassendeklaration können dann direkt *abgeleitete* Operationen implementiert werden. So lässt sich etwa eine Gleichheit implementieren, sofern es nur eine Vergleichsoperation $\leq$ gibt. Eine solche generische Implementierung führt zu einer generischen Unterklassenbeziehung. Zusätzlich zu den vom System bereit gestellten Typen sollte dann die Standardbibliothek der Programmiersprache auch vordefinierte Klassen bereitstellen, in die die vordefinierten Typen eingeordnet sind. In unserem Beispiel sollten etwa die Basistypen **int** und **bool** ebenfalls Instanzen der Klasse Eq sein.

Wenden wir uns nun der Frage zu, wie die richtige Verwendung der überladenen Operatoren überprüft werden kann. Zuerst überzeugen wir uns, dass wir für jeden Typausdruck t bestimmen können, ob er einen Typ aus einer Klasse C repräsentiert. Sei Σ eine Zuordnung von Typvariablen zu Sorten. Typvariablen, die nicht in Σ vorkommen, werden implizit auf die *leere Menge* abgebildet: für sie verlangt Σ keine Einschränkung. Σ heißt dann *Sortenumgebung*. Für die Sortenumgebung Σ und eine Sorte S soll dann die Aussage

$$\Sigma \vdash t : C$$

ausdrücken, dass der Typ t zu der Klasse C gehört, sofern jede Typvariable α, die in t vorkommt, zu allen Klassen aus $\Sigma(\alpha)$ gehört.

Nehmen wir an, Σ sei gegeben. Die Menge $S[t]$ *aller* Klassen C, zu denen t gehört, lässt sich induktiv über die Struktur von t ermitteln. Ist t eine Typvariable α, dann ist $S[t] = \Sigma(\alpha)$. Hat t die Form $b(t_1, \ldots, t_k)$ für einen Typkonstruktor b, dann ist $S[t]$ die Menge aller Klassen C, für die es eine Instanzdeklaration $\textbf{inst}\ b(S_1, \ldots, S_k) : C\ \ldots$ gibt mit $S_i \subseteq S[t_i]$ für alle i.

Wenn es für jede Klasse und jeden Typkonstruktor maximal eine Instanzdeklaration gibt, können umgekehrt, ausgehend von einer Sortenanforderung S an der Wurzel eines Typausdrucks t, Sortenanforderungen für die Teilausdrücke von t abgeleitet werden. Das erlaubt uns, die *minimale* Annahmen an die Typvariablen berechnen, die in t vorkommen, damit t zu allen Klassen aus S gehört.

Beispiel 4.2.15 Nehmen wir an, dass die Basistypen **int** und **bool** zu der Klasse Eq gehören.

- Dann gehören auch die Typen $(\textbf{bool}, \textbf{int list})$ und $(\textbf{bool}, \textbf{int list})\ \textbf{list}$ zu Eq.
- Der Typ $\textbf{bool} \to \textbf{int}$ gehört nicht zu der Klasse Eq, solange keine Instanzdeklaration für Eq und den Typkonstruktor $\to$ bereit gestellt wird.

- Der Typausdruck $(\alpha, \mathbf{int})$ **list** bezeichnet dagegen Typen der Klasse Eq, sofern α zu der Klasse Eq gehört. □

Um Typen für funktionale Programme mit Klassen- und Instanzdeklarationen herzuleiten, könnten wir so vorgehen, dass wir zuerst einmal die Bedingungen in Typschemata ignorieren d.h. Hindley-Milner-Typen inferieren. In einer zweiten Phase werden dann für alle Typvariablen die Sorten ermittelt. Der Nachtteil dieser Vorgehensweise ist, dass damit zwar die Typkorrektheit eines Programms nachgewiesen werden kann, dass nach wie vor aber nicht klar ist, wie solche Programme implementiert werden könnten.

Eine bessere Idee besteht deshalb darin, die polymorphe Typinferenz mit Hilfe des Algorithmus $\mathcal{W}$ so zu modifizieren, dass er für einen Ausdruck e neben Typ- und Sorteninformation auch eine *Übersetzung* von e in einen Ausdruck e' liefert, indem die Auswahl der richtigen Implementierung eines Operators explizit gemacht wird.

Für die Übersetzung stellen wir für jede Sorte S eine Struktur $\alpha\,\mathsf{dict}_S$ bereit, der für jeden Operator op einer Klasse in S in einer Komponente op eine *Implementierung* bereit stellt. Der überladene Operator op_C der Klasse C mit dem Typschema $\forall\,\alpha : C.\ t$ wird dann übersetzt in das Nachschlagen $\alpha.\mathsf{op}_C$ in einer Tabelle α, die eine Komponente op_C besitzt. Aufgabe der Übersetzung ist, zu jeder Verwendung des Operators eine Tabelle zu transportieren, die die jeweils richtige Implementierung des Operators bereit stellt. Eine Variable f, für die der Algorithmus $\mathcal{W}$ ein Typschema $\forall\,\alpha_1 : S_1, \ldots, \alpha_m : S_m.\ s$ angelegt, wird darum in eine Funktion übersetzt, die m Tabellen als zusätzliche Argumente verlangt:

$$f : \forall \alpha_1 \ldots \alpha_m.\ \alpha_1\,\mathsf{dict}_{S_1} \to \ldots \alpha_m\,\mathsf{dict}_{S_m} \to s$$

Um den Algorithmus $\mathcal{W}$ zu modifizieren, benötigen wir eine Unifikationsfunktion, die die Sorteninformationmit verwaltet:

$$\begin{array}{lll}
\mathsf{sort_unify}\ (\tau_1, \tau_2)\ \Sigma & = & \mathbf{match}\ \ \mathsf{unify}\,(\tau_1, \tau_2)\,\emptyset \\
& & \mathbf{with}\ \mathsf{Fail} \to \mathsf{Fail} \\
& & \quad \mid \quad \theta \quad \to (\mathbf{match}\ \theta^{-1}\,\Sigma\ \mathbf{with}\ \mathsf{Fail} \to \ \mathsf{Fail} \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \mid \quad \Sigma' \ \to\ (\theta, \Sigma'))
\end{array}$$

Dabei liefert $\theta^{-1}\,\Sigma$ die minimale Sortenannahme Σ' für die Typvariablen, die im Bild von θ vorkommen, damit die Sortennahmen in Σ erfüllt sind, d.h. dass $\Sigma' \vdash (\theta\,\alpha) : (\Sigma\,\alpha)$ gilt für alle Typvariablen α.

Beispiel 4.2.16 Betrachten Sie Instanzdeklarationen, die zu den folgenden Regeln führen:

$$\begin{array}{ll}
\mathsf{Eq}\ \mathbf{list} & : \mathsf{Eq} \\
\mathsf{Comp}\ \mathbf{set} & : \mathsf{Eq}
\end{array}$$

und sei:

$$\Sigma = \{\alpha \mapsto \mathsf{Eq}\} \qquad \theta = \{\alpha \mapsto\ \beta\,\mathsf{set\,list}\}$$

Umrechnung der Sortenanforderung $\Sigma(\alpha) = \mathsf{Eq}$ für die Typvariable α bzgl. der Typsubstitution θ in eine Sortenanforderung an die Typvariablen (hier: nur β) in dem Typterm $\theta\,\alpha$ liefert die Sortenanforderung:

$$\theta^{-1}\,\Sigma = \{\beta \mapsto \mathsf{Comp}\}$$

Die substituierte Variable α kommt in $\theta^{-1}\,\Sigma$ nicht mehr vor. □

Für die Implementierung des erweiterten Algorithmus $\mathcal{W}$ modifizieren wir ebenfalls die Hilfsfunktionen close und inst.

Der Aufruf $\mathsf{sort\_close}\ (t, e)\ (\Gamma, \Sigma)$ für einen Typ t und einen Ausdruck e bzgl. einer Typumgebung Γ und einer Sortenumgebung Σ macht alle Typvariablen in t generisch, die weder in Γ noch in Σ vorkommen, und *beschränkt generisch* gemäß der Sortenumgebung Σ, wenn sie nicht in Γ, aber in Σ vorkommen. Neben dem Typschema liefert der Aufruf als zweite Komponente die Sortenumgebung Σ

zurück, aus der die Variablen, die in dem Typschema generalisiert werden, nicht mehr vorkommen. Als dritte Komponente wird die Funktion zurück geliefert, die man aus dem Ausdruck e erhält, wenn die generischen Typvariablen des Typschemas als formale Parameter abstrahiert werden:

$$\begin{aligned}\mathsf{sort\_close}\ (t, e)\ (\Gamma, \Sigma) = \quad & \mathbf{let}\ \alpha'_1, \ldots, \alpha'_n = \mathsf{free}\,(t) \setminus (\mathsf{free}\,(\Gamma) \cup \mathsf{dom}\,(\Sigma)) \\ & \mathbf{in\ let}\ s = \forall\, \alpha'_1, \ldots, \alpha'_n.\, t \\ & \mathbf{in\ let}\ \alpha_1, \ldots, \alpha_m = (\mathsf{free}\,(t) \setminus \mathsf{free}\,(\Gamma)) \cap \mathsf{dom}\,(\Sigma) \\ & \mathbf{in\ let}\ s = \forall\, \alpha_1 : \Sigma(\alpha_1), \ldots, \alpha_m : \Sigma(\alpha_m).\, s \\ & \mathbf{in\ let}\ \Sigma = \Sigma \backslash \{\alpha_1, \ldots, \alpha_m\} \\ & \mathbf{in}\ (s, \Sigma, \mathbf{fun}\ \alpha_1 \to \ldots \mathbf{fun}\ \alpha_m \to e)\end{aligned}$$

Die Instantiierung mit *frischen* Typ-Variablen leistet die folgende Funktion:

$$\begin{aligned}\mathbf{fun}\ \ \mathsf{sort\_inst}\ (\forall\, \alpha_1 : S_1, \ldots, \alpha_m : S_m.\, s, x) \ = \quad & \mathbf{let}\ t = \mathsf{inst}\ s \\ & \mathbf{in\ let}\ \beta_1 = \mathsf{new}() \\ & \quad \ldots \\ & \mathbf{in\ let}\ \beta_m = \mathsf{new}() \\ & \mathbf{in\ let}\ t = t[\beta_1/\alpha_1, \ldots, \beta_m/\alpha_m] \\ & \mathbf{in}\ \ (t, \{\beta_1 \mapsto S_1, \ldots, \beta_m \mapsto S_m\}, x\, \beta_1 \ldots \beta_m)\end{aligned}$$

Bei der Transformation werden nur diejenigen Typparameter zu Funktionsparametern, die durch Sorten einschränkt werden. Die Typvariablen, die in die Ausgabeausdrücke eingefügt werden, können im weiteren Verlauf des Algorithmus $\mathcal{W}$ durch Unifikation von Typausdrücken weiter spezialisiert werden. Dann werden sie nicht nur in den Typen, sondern auch in den Ausdrücken substituiert. Wird eine Variable $\alpha : S$ durch einen Typausdruck t ersetzt, wird eine S-Tabelle gemäß dem Typ t generiert und für die Variable α eingesetzt. Die Generierung dieser Tabelle leistet die Transformation $\mathcal{T}$:

$$\begin{aligned}\mathcal{T}[\beta]\ S \qquad\qquad\quad &= \beta \\ \mathcal{T}[b(t_1, \ldots, t_m)]\ S &= \mathbf{forall}\ C \in S \\ & \quad \mathbf{let}\ \mathsf{op}'_C = \mathbf{let}\ d_1 = \mathcal{T}[t_{i_1}]\ S_{C,i_1}\ \ \mathbf{in} \\ & \qquad\qquad\qquad \ldots \\ & \qquad\qquad\qquad \mathbf{let}\ d_k = \mathcal{T}[t_{i_k}]\ S_{C,i_k}\ \ \mathbf{in} \\ & \qquad\qquad\qquad \mathsf{op}_{C,b}\, d_1\ \ldots\ d_k \\ & \quad \mathbf{in}\ \{\mathsf{op}_C = \mathsf{op}'_C \mid C \in S\}\end{aligned}$$

falls $(S_{C,1}, \ldots, S_{C,m})\, b\ :\ C$ gilt und $S_{C,i_1}, \ldots, S_{C,i_k}$ die Teilfolge der nicht-leeren Sorten unter den $S_{C,i}$ ist. Die Implementierung des Operators op_C mit $C \in S$ für den Typ t kann dann in der Tabelle $\mathcal{T}[t]\ S$ nachgeschlagen werden.

Nun sind wir so weit, dass wir den Algorithmus $\mathcal{W}$ erweitern können. Der modifizierte Algorithmus $\mathcal{W}$ erhält als Argumente neben dem Ausdruck e eine Typumgebung Γ, eine idempotente Typsubstitution θ und zusätzlich eine Sortenumgebung Σ. Als Rückgabewert liefert er neben einem Typ für e eine gegebenfalls weiter spezialisierte idempotente Typsubstitution θ', eine Sortenumgebung Σ' sowie die Übersetzung des Ausdrucks e. Hier geben wir nur die wichtigsten Fälle an. Den vollständigen Algorithmus sollten Sie in Aufgabe **??** entwickeln.

$$
\begin{array}{lll}
\ldots & & \\
\mid \mathsf{op}_C & \rightarrow & \textbf{let } \beta = \mathsf{new}() \\
 & & \textbf{in } \quad (t_C[\beta/\alpha], \Sigma \oplus \{\beta \mapsto C\}, \theta, \beta.\mathsf{op}_C) \\
\mid x & \rightarrow & \textbf{let } (t, \Sigma', e') = \mathsf{sort\_inst}\ (\Gamma(x), x) \\
 & & \textbf{in } \quad (t, \Sigma \cup \Sigma', \theta, e') \\
\mid (\text{LET } x = e_1 \text{ IN } e_0) & & \\
 & \rightarrow & \textbf{let } (t_1, \Sigma, \theta, e_1') = \mathcal{W}\ e_1\ (\Gamma, \Sigma, \theta) \\
 & & \textbf{in let } e_1' = \mathcal{T}[e_1']\ (\theta, \Sigma) \\
 & & \textbf{in let } (s_1, \Sigma, e_1') = \mathsf{sort\_close}\ (\theta\ t_1, e_1')\ (\theta\ \Gamma, \Sigma) \\
 & & \textbf{in let } \Gamma = \Gamma \oplus \{x_1 \mapsto s_1\} \\
 & & \textbf{in let } (t_0, \Sigma, \theta, e_0') = \mathcal{W}\ e_0\ (\Gamma, \Sigma, \theta) \\
 & & \textbf{in let } e' = (\text{LET } x_1 = e_1' \text{ IN } e_0') \\
 & & \textbf{in } \quad (t_0, \Sigma, \theta, e')
\end{array}
$$

wobei $\mathcal{T}[e]\ (\theta, \Sigma)$ jedes Vorkommen einer Variable β in e mit $\Sigma(\beta) = S$ durch die Tabelle $\mathcal{T}[\theta\ \beta]\ S$ ersetzt. Die Typinferenz/Transformation startet mit der leeren Sortenumgebung $\Sigma_0 = \emptyset$ und einer leeren Typumgebung $\Gamma_0 = \emptyset$. Sei

$$\textbf{inst } \ b(S_1, \ldots, S_m) : C \ \textbf{ where } \ \mathsf{op}_C = e$$

die Instanz-Deklaration einer Klasse C, deren Operator op_C das Typschema $\forall\, \alpha : C.\, t$ erfüllt, wobei $S_{i_1}, \ldots, S_{i_k}$ die Teilfolge der nicht-leeren Sorten unter den S_i ist. Dann muss überprüft werden, ob die Implementierung einen passenden Typ hat, d.h. ob

$$\mathcal{W}\ e\ (\Gamma_0, \emptyset, \emptyset) = (t', \Sigma, \theta, e') \qquad \text{mit} \qquad \theta\, t' = t[b(\beta_1, \ldots, \beta_m)/\alpha]$$

gilt für geeignete Typvariablen β_i mit:

$$\Sigma(\beta_i) \subseteq S_i$$

gilt. die Implementierung des Operators op_C für den Typkonstruktor b ergibt sich dann zu:

$$\mathsf{op}_{C,b} = \textbf{fun } \beta_{i_1} \rightarrow \ldots \rightarrow \textbf{fun } \beta_{i_k} \rightarrow \mathcal{T}[e']\ (\theta, \Sigma)$$

Beispiel 4.2.17 Betrachten wir die Implementierungen der Gleichheit für Paare und Listen. Gemäß ihrer Deklaration haben sie die Typen:

$$
\begin{array}{l}
(=)_{\mathsf{pair}} : \forall\, \alpha_1 : \mathsf{Eq}, \alpha_2 : \mathsf{Eq}.\ (\alpha_1 * \alpha_2) \rightarrow (\alpha_1 * \alpha_2) \rightarrow \textbf{bool} \\
(=)_{\mathsf{list}} \ : \forall\, \alpha : \mathsf{Eq}.\ \alpha\ \textbf{list} \rightarrow \alpha\ \textbf{list} \rightarrow \textbf{bool}
\end{array}
$$

Nach der Transformation ergeben sich die Typen:

$$
\begin{array}{l}
(=)_{\mathsf{pair}} : \forall\, \alpha_1, \alpha_2.\ \alpha_1\ \mathsf{dict}_{\mathsf{Eq}} \rightarrow \alpha_2\ \mathsf{dict}_{\mathsf{Eq}} \rightarrow (\alpha_1 * \alpha_2) \rightarrow (\alpha_1 * \alpha_2) \rightarrow \textbf{bool} \\
(=)_{\mathsf{list}} \ : \forall\, \alpha.\ \alpha\ \mathsf{dict}_{\mathsf{Eq}} \rightarrow \alpha\ \textbf{list} \rightarrow \alpha\ \textbf{list} \rightarrow \textbf{bool}
\end{array}
$$

Für jeden bedingten Typparameter wird ein weiteres Argument bereit gestellt. Nach der Transformation erhalten wir für die Implementierung:

$$
\begin{array}{lll}
(=)_{\mathsf{pair}} = & \textbf{fun } \beta_1 \rightarrow \textbf{fun } \beta_2 \rightarrow \textbf{fun } x \rightarrow \textbf{fun } y \rightarrow & \\
 & \quad \textbf{match } x \textbf{ with } (x_1, x_2) \ \rightarrow & \\
 & \quad \textbf{match } y \textbf{ with } (y_1, y_2) \ \rightarrow & \\
 & \quad\quad \beta_1.(=)\ x_1\ y_1 \ \wedge \ \beta_2.(=)\ x_2\ y_2 & \\
(=)_{\mathsf{list}} = & \textbf{fun } \beta \rightarrow \textbf{let rec } \mathsf{eq\_list} = \textbf{fun } l_1 \ \rightarrow \textbf{fun } l_2 \rightarrow \textbf{match } l_1 & \\
 & \textbf{with } [] \quad\quad \rightarrow (\textbf{match } l_2 \textbf{ with } [] \rightarrow \textbf{true} \mid \_ \rightarrow \textbf{false}) & \\
 & \quad \mid \quad x :: xs \rightarrow (\textbf{match } l_2 \textbf{ with } [] \rightarrow \textbf{false} & \\
 & \quad\quad\quad\quad \mid \ y :: ys \rightarrow \textbf{if } \beta.(=)\ x\ y \textbf{ then } \mathsf{eq\_list}\ xs\ ys & \\
 & \quad\quad\quad\quad\quad\quad \textbf{else false}) & \\
 & \textbf{in } \ \mathsf{eq\_list} &
\end{array}
$$

Die Programmvariablen β_1, β_2 und β sind neue Programmvariablen, die aus Typvariablen generiert wurden. Ihre Laufzeitwerte sind Tabellen, die die jeweils richtige Implementierung des überladenen Operators $(=)$ bereit stellen. □

In unserer Implementierung tritt der Komponentenname op in allen Tabellen zu Sorten auf, die eine Implementierung des Operators op verlangen. In Programmiersprachen wie OCAML darf jeder Komponentenname jedoch nur in höhchstens einem Verbundtypen verwendet werden. Eine Lösung besteht darin, *nach* abgeschlossener Typinferenz und Transformation die Verbundtypen mit eindeutigen Komponentennamen zu versehen und entsprechend die Zugriffe auf die Komponenten anzupassen. Andere Möglichkeiten diskutiert Aufg. **??**.

Die Einführung von Typklassen ist keineswegs das Ende der Fahnenstange. In der Programmiersprache HASKELL wird mit weiteren Erweiterungen des Typsystems von Hindey-Milner experimentiert. Insbesondere stellt HASKELL neben Typklassen auch *Typkonstruktor*-Klassen zur Verfügung. Diese erlauben eine generische Behandlung von *Monaden*. Mit Monaden lassen sich rein funktional und auf theoretisch befriedigende Weise Ein- und Ausgabe sowie Seiteneffekte modellieren.

4.3 Attributgrammatiken

Es gibt kein Beschreibungssmittel für die statischen semantischen Eigenschaften von Programmen, welches ähnliche Akzeptanz gefunden hätte, wie kontextfreie Grammatiken für die Syntax. Ein elegantes Beschreibungsmittel für Berechnungen auf Syntaxbäumen sind *Attributgrammatiken* (englisch: attribute grammars). Attributgrammatiken erweitern kontextfreie Grammatiken, indem sie mit den Symbolen der zugrundeliegenden Grammatik *Attribute* assoziieren. Die Menge der Attribute eines Symbols X bezeichnen wir mit $\mathcal{A}(X)$. Jedem Attribut a ist eindeutig ein Typ τ_a zugeordnet, der die Menge der möglichen Werte der zugehörigen Attributexemplare festlegt. Betrachten wir eine Produktion $p: \; X_0 \longrightarrow X_1 \ldots X_k$, auf deren rechter Seite $k \geq 0$ Symbole vorkommen. Um die unterschiedlichen Vorkommen von Symbolen in der Produktion p zu unterscheiden, nummerieren wir diese von links nach rechts durch. Das bedeutet, dass die linke Seite X_0 mit $p[0]$ bezeichnet wird, während das i-te Symbol X_i auf der rechten Seite von p mit $p[i]$ für $i = 1, \ldots, k$ bezeichnet wird. Das Attribut a des Symbolvorkommens X_i bezeichnen wir dann mit $p[i].a$.

Zu jeder Produktion werden Vorschriften in einer geeigneten Implementierungssprache angegeben, wie Attribute der Symbolvorkommen in der Produktion aus anderen Attributen von Symbolvorkommen derselben Produktion berechnet werden. Diese Berechnungsvorschriften nennen wir *semantische Regeln*. In unseren Beispielen notieren wir die semantischen Regeln in einer OCAML-artigen Programmiersprache. Das hat den Vorteil, dass wir auf die explizite Angabe von Typen verzichten können.

Ein solcher Mechanismus wird in eingeschränkter Form auch von gängigen LR-Parsern wie YACC oder BISON bereit gestellt: hier ist jedem Symbol der Grammatik jedoch nur ein Attribut zugeordnet. Zu jeder Produktion gibt es eine semantische Regel, die festlegt, wie das Attribut der linken Seite der Produktion aus den Attributen der Symbolvorkommen der rechten Seite berechnet wird.

Beispiel 4.3.1 Betrachten Sie eine kontextfreie Grammatik mit den Nichtterminalen E, T, F für arithmetische Ausdrücke. Die Menge der Terminale bestehe aus Klammersymbolen, Operatorsymbolen und den Symbolen var und const, das *int*-Variablen bzw. Konstanten repräsentiert. Den Nichtterminalen wollen wir ein Attribut $tree$ zuordnen, das eine interne Repräsentation des Ausdrucks enthält.

Zur Berechnung dieses Attributs erweitern wir die Produktionen der Grammatik um semantische Regeln wie folgt:

$$
\begin{aligned}
p_1: \quad & E \longrightarrow E + T \\
& \quad p_1[0].tree = \mathsf{Plus}\,(p_1[1].tree, p_1[2].tree) \\
p_2: \quad & E \longrightarrow T \\
& \quad p_2[0].tree = p_2[1].tree \\
p_3: \quad & T \longrightarrow T * F \\
& \quad p_3[0].tree = \mathsf{Mult}\,(p_3[1].tree, p_3[2].tree) \\
p_4: \quad & T \longrightarrow F \\
& \quad p_4[0].tree = p_4[1].tree \\
p_5: \quad & F \longrightarrow \mathsf{const} \\
& \quad p_5[0].tree = \mathsf{Int}\,(p_5[1].val) \\
p_6: \quad & F \longrightarrow \mathsf{var} \\
& \quad p_6[0].tree = \mathsf{Var}\,(p_6[1].id) \\
p_7: \quad & F \longrightarrow (\ E\ ) \\
& \quad p_6[0].tree = p_6[2].tree
\end{aligned}
$$

Zur Konstruktion der Interndarstellung wurden die Konstruktoren:

$$\mathsf{Plus}, \mathsf{Mult}, \mathsf{Int}, \mathsf{Var}$$

verwendet. Weiterhin wurde angenommen, dass jedes Symbol const über ein Attribut val verfügt, das den Wert der Konstanten enthält, und jedes Symbol var ein Attribut id mit einer eindeutigen Kennung der Variablen. □

Einige Parsergeneratoren addressieren die verschiedenen Vorkommen von Symbolen in einer Produktion, indem sie die Vorkommen sortiert nach jedem Symbol indizieren.

Beispiel 4.3.2 Betrachten Sie erneut die Grammatik aus Beispiel 4.3.1. Gemäß der Konvention, verschiedene Vorkommen des selben Symbols in der selben Produktion durchzunummerieren, werden die semantischen Regeln wie folgt notiert:

$$
\begin{aligned}
p_1: \quad & E \longrightarrow E + T \\
& \quad E[0].tree = \mathsf{Plus}\,(E[1].tree, T.tree) \\
p_2: \quad & E \longrightarrow T \\
& \quad E.tree = T.tree \\
p_3: \quad & T \longrightarrow T * F \\
& \quad T[0].tree = \mathsf{Mult}\,(T[1].tree, F.tree) \\
p_4: \quad & T \longrightarrow F \\
& \quad T.tree = F.tree \\
p_5: \quad & F \longrightarrow \mathsf{const} \\
& \quad F.tree = \mathsf{Int}\,(\mathsf{const}.val) \\
p_6: \quad & F \longrightarrow \mathsf{var} \\
& \quad F.tree = \mathsf{Var}\,(\mathsf{var}.id) \\
p_7: \quad & F \longrightarrow (\ E\ ) \\
& \quad F.tree = E.tree
\end{aligned}
$$

Kommt ein Symbol überhaupt nur einmal vor, wird der Index weggelassen. Kommt ein Symbol X mehrmals vor, bezeichnet der Index 0 ein Vorkommen auf der linken Seite, während alle Vorkommen auf der rechten Seite der Produktion von 1 fortlaufend nummeriert werden. □

In konkreten Beispielen von Attributgrammatiken werden wir uns an die Konvention aus Beispiel 4.3.2 halten, während bei den konzeptuellen Überlegungen die Addressierung der Symbolvorkommen in

einer Produktion $p: \ X_0 \to X_1 \ldots X_k$ durch fortlaufende Indizierung $p[0], \ldots, p[k]$ wie in Beispiel 4.3.1 eleganter ist.

Das eine semantische Attribut an jedem Symbol, das LR-Parsergeneratoren zur Verfügung stellen, kann eingesetzt werden, um während der Syntaxanalyse eine Repräsentation des Syntaxbaums aufzubauen. Attributgrammatiken verallgemeinern diese Idee in zwei Richtungen. Zum einen kann ein Symbol nun mehrere Attribute besitzen. Zum anderen werden die Attribute der linken Seite nicht notwendigerweise immer mit Hilfe der Attribute der Symbole definiert, die auf der rechten Seite vorkommen. Die Werte mancher Attribute der rechten Seite können sich nun auch aus den Werten von Attributen der linken Seite oder von Attributen anderer Symbole der rechten Seite ergeben.

Wir führen folgende Sprechweisen ein. Die einzelnen Attribute $p[i].a$ der verschiedenen Symbolvorkommen in einer Produktion p nennen wir *Attributvorkommen*. Sie gehören der Attributgrammatik an und dienen zur *Spezifikation* des lokalen Verhaltens an einem Knoten. Greift die definierende Vorschrift für ein Attributvorkommen auf ein anderes Attributvorkommen zu, besteht zwischen diesen beiden Attributvorkommen eine *funktionale Abhängigkeit*.

Die Attribute der Symbolvorkommen in einem Syntaxbaum nennen wir dagegen *Attributexemplare*. Sie treten zur Übersetzungszeit auf, nachdem der Quelltext des Programms syntaktisch analysiert ist und weitere Berechnungen auf dem Syntaxbaum vorgenommen werden sollen.

Die funktionalen Abhängigkeiten zwischen Attributvorkommen legen fest, welche Reihenfolgen bei der Berechnung der Attributexemplare an einem Knoten des Syntaxbaums zu beachten sind. Geeignete *Bedingungen* an die funktionalen Abhängigkeiten stellen sicher, dass die *lokalen* semantischen Regeln der Attributgrammatik für die Attributvorkommen in den Produktionen zu einer *globalen* Berechnung der Attributexemplare in einem Syntaxbaum zusammen gesetzt werden können. Die Werte für die Attributexemplare an den einzelnen Knoten des Ableitungsbaums werden von einem globalen Algorithmus bestimmt, der aus der Attributgrammatik generiert wird und sich an jedem Knoten n lokal an die semantischen Regeln hält, die die Produktion für n vorgibt. Wie ein solcher Algorithmus aus einer Attributgrammatik automatisch generiert werden kann, das ist das Thema dieses Kapitels.

Bei den Attributen eines Symbols X unterscheidet man zwischen *ererbten* (engl. inherited) Attributen und *abgeleiteten* (engl. synthesized) Attributen. Die Werte abgeleiteter Attributexemplare an einem Knoten werden aus dem Unterbaum an diesem Knoten berechnet. Deshalb werden die entsprechenden Attributvorkommen der linken Seite einer Produktion in einer Berechnungsvorschrift zu der Produktion definiert. Die Werte der ererbten Attributexemplare eines Knotens werden dagegen aus dem (oberen) Kontext des Knotens berechnet. Deshalb gibt es zu jedem ererbten Attributvorkommen auf der rechten Seite einer Produktion eine Berechnungsvorschrift (siehe Abbildung 4.6).

Die Mengen der ererbten bzw. abgeleiteten Attribute einer Attributgrammatik bezeichnen wir mit $\mathcal{I}$ bzw. $\mathcal{S}$. Die Mengen der ererbten bzw. abgeleiteten Attribute des Symbols X bezeichnen wir entsprechend mit $\mathcal{I}(X)$ bzw. $\mathcal{S}(X)$.

Eine Attributgrammatik ist in *Normalform*, wenn kein Attributvorkommen $p[i].a$ einer Produktion p, das durch eine semantische Regel zu dieser Produktion definiert wird, selbst zur Definition eines Attributvorkommens der Produktion p verwendet wird. Wenn nicht explizit anders gesagt, nehmen wir stets an, dass Attributgrammatiken in Normalform sind.

Wir haben auch bei den Terminalsymbolen der Grammatik abgeleitete Attribute zugelassen. In einem Übersetzer werden Attributgrammatiken zur semantischen Analyse eingesetzt, die auf die lexikalische und syntaktische Analyse folgt. Typische abgeleitete Attribute bei Terminalen sind die Werte von Konstanten, die Externdarstellungen oder eindeutige Identifizierungen von Namen, die Adressen von Stringkonstanten usw. Die Werte dieser Attribute liefert meist der Scanner – zumindest, wenn er um semantische Funktionen erweitert ist. In der Praxis spielen die abgeleiteten Attribute der Terminalsymbole eine entscheidende Rolle.

Auch für die ererbten Attributexemplare an der Wurzel des Syntaxbaums stellt die Attributgrammatik keine semantischen Regeln bereit. Hier muss die Anwendung für eine geeignete Initialisierung sorgen.

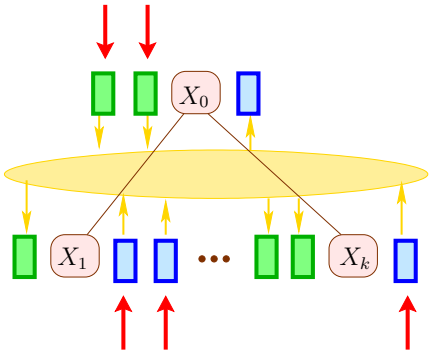


Abb. 4.6. Ein attributierter Knoten im Syntaxbaum mit seinen attributierten Nachfolgern. Exemplare ererbter Attribute sind als Kästchen links von den syntaktischen Symbolen, abgeleitete als Kästchen rechts von Symbolen dargestellt. Rote (dunklere) Pfeile deuten den Informationsfluss von außerhalb an, gelbe (hellere) Pfeile deuten funktionale Abhängigkeiten zwischen Attributexemplaren an, die sich durch die Instantiierung der semantischen Regeln ergeben.

4.3.1 Die Semantik einer Attributgrammatik

Die Semantik einer Attributgrammatik legt für jeden Syntaxbaum t der zugrundeliegenden kontextfreien Grammatik fest, welche Werte die Attribute der Symbole an jedem Knoten in t erhalten.

Für jeden Knoten n in t sei $\mathsf{symb}(n)$ das Symbol der Grammatik, mit dem n beschriftet ist. Ist $\mathsf{symb}(n) = X$, wird n mit den Attributen aus $\mathcal{A}(X)$ ausgestattet. Das Attribut $a \in \mathcal{A}(n)$ des Knoten n selektieren wir durch $n.a$. Weiterhin benötigen wir einen Operator, um von einem Knoten zu seinen Nachfolgern zu navigieren. Sei $n_1, \ldots, n_k$ die Folge der Nachfolger des Knoten n in dem Ableitungsbaum t. Dann bezeichne $n[0]$ den Knoten n selbst und $n[i] = n_i$, $i = 1, \ldots, k$, den i-ten Nachfolger von n im Syntaxbaum t.

Ist $X_0 = \mathsf{symb}(n)$ und für $i = 1, \ldots, k$, $X_i = \mathsf{symb}(n_i)$ die Beschriftung des Nachfolgers n_i von n, dann ist $X_0 \longrightarrow X_1 \ldots X_k$ die Produktion der kontextfreien Grammatik, die am Knoten n angewandt wurde. Aus den semantischen Regeln der Attributgrammatik zu dieser Produktion p generieren wir nun Definitionen von Attributen der Knoten $n, n_1, \ldots, n_k$, indem wir den Platzhalter p mit n instantiieren. Aus der semantischen Regel:

$$p[i].a = f(p[i_1].a_1, \ldots, p[i_r].a_r)$$

der Attributgrammatik für die Produktion p wird dann die semantische Regel

$$n[i].a = f(n[i_1].a_1, \ldots, n[i_r].a_r)$$

für den Knoten n im Syntaxbaum t. Hierbei nehmen wir an, dass die semantische Regel eine *totale* Funktion f spezifiziert. Sei t ein Syntaxbaum und

$$V(t) = \{n.a \mid n \text{ Knoten in } t, a \in \mathcal{A}(\mathsf{symb}(n))\}$$

die Menge aller Attributexemplare in t. Die Teilmenge $V_{in}(t)$ der ererbten Attributexemplare der Wurzel und der abgeleiteten Attributexemplare der Blätter nennen wir die Menge der *Eingabeattributexemplare* von t. Eine Belegung σ aller Attributexemplare für t mit Werten, so dass der Typ des Werts $\sigma(n.a)$ für jedes $n.a \in V(t)$ mit dem Typ von a bzgl. des Symbols von n übereinstimmt, nennen wir eine *Attributierung* des Syntaxbaums t.

Nach der Vorbelegung der Eingabeattributexemplare mit Werten erhalten wir durch die Instantiierung der semantischen Regeln der Atributgrammatik an allen Knoten in t ein Gleichungssystem, das für jede nicht vorbelegte Unbekannte $n.a$ genau eine Gleichung hat. Sei $\mathsf{GLS}(t)$ dieses Gleichungssystem. Ist $\mathsf{GLS}(t)$ rekursiv (zyklisch), kann es mehrere oder auch gar keine Lösung haben. Ist $\mathsf{GLS}(t)$ nicht rekursiv, gibt es für jede Vorbelegung σ der Eingabeattributexemplare genau eine Attributierung

des Syntaxbaums t, die auf den Eingabeattributexemplaren mit σ übereinstimmt und alle Gleichungen des Gleichungssystems erfüllt. Man nennt eine Attributgrammatik deshalb *wohlgeformt*, wenn das Gleichungssystem $\mathsf{GLS}(t)$ für keinen Ableitungsbaum t der zugrundeliegenden kontextfreien Grammatik rekursiv ist. In diesem Fall definieren wir die Semantik der Attributgrammatik als die Abbildung, die jedem Ableitungsbaum t und jeder Vorbelegung σ der Eingabeattributexemplare die Attributierung zuordnet, die auf diesen mit σ übereinstimmt und zusätzlich alle Gleichungen des Gleichungssystems $\mathsf{GLS}(t)$ erfüllt.

4.3.2 Einige Attributgrammatiken

Im Folgenden betrachten wir einige Ausschnitte aus Attributgrammatiken, die wesentliche Teilaufgaben der semantischen Analyse lösen. Die erste Attributgrammatik zeigt, wie die Typen von Ausdrücken mit Hilfe einer Attributgrammatik berechnet werden können.

Beispiel 4.3.3 (Typüberprüfung) Die Attributgrammatik AG_{types} beschreibt die Typberechnung für Ausdrücke mit Wertzuweisungen, nullstelligen Funktionen, den Operatoren $+, -, *, /$ und Variablen und Konstanten vom Typ $\mathbf{int}$ oder $\mathbf{float}$ bei einer C-artigen Programmiersprache mit expliziten Typdeklarationen für Variablen. Die Attributgrammatik hat für die Nichtterminalsymbole E, T und F sowie für das Terminalsymbol const ein Attribut typ, das Typen, in unserem Fall also die Werte Int oder Float annehmen kann. Die Grammatik kann leicht auf allgemeinere Ausdrücke mit Funktionsanwendungen, Selektionen von Komponenten zusammengesetzter Werte oder Zeigern erweitert werden.

$$
\begin{array}{l}
E \longrightarrow \mathsf{var}\ '='\ E \\
E[1].env = E[0].env \\
E[0].typ\ = E[0].env\ \mathsf{var}.id \\
E[0].ok\ \ = \quad \mathbf{let}\ x = \mathsf{var}.id \\
\qquad\quad \mathbf{in\ let}\ \tau = E[0].env\, x \\
\qquad\quad \mathbf{in}\quad (\tau \neq \mathsf{error}) \wedge (E[1].type \sqsubseteq \tau)
\end{array}
$$

$$
\begin{array}{l}
E \longrightarrow E\ \mathsf{aop}\ T \\
E[1].env = E[0].env \\
T.env\ \ \ = E[0].env \\
E[0].typ\ = E[1].typ \sqcup T.typ \\
E[0].ok\ \ = (E[1].typ \sqsubseteq \mathsf{float}) \wedge (T.typ \sqsubseteq \mathsf{float})
\end{array}
\qquad
\begin{array}{l}
E \longrightarrow T \\
T.env = E.env \\
E.typ = T.typ \\
E.ok\ \ = T.ok
\end{array}
$$

$$
\begin{array}{l}
T \longrightarrow T\ \mathsf{mop}\ F \\
T[1].env = T[0].env \\
F.env\ \ \ = T[0].env \\
T[0].typ\ = T[1].typ \sqcup F.typ \\
T[0].ok\ \ = (T[1].typ \sqsubseteq \mathsf{float}) \wedge (F.typ \sqsubseteq \mathsf{float})
\end{array}
\qquad
\begin{array}{l}
T \longrightarrow F \\
F.env = T.env \\
T.typ\ = F.typ \\
T.ok\ \ = F.ok
\end{array}
$$

$$
\begin{array}{l}
F \longrightarrow (E) \\
E.env = F.env \\
F.typ\ = E.typ \\
F.ok\ \ = E.ok
\end{array}
\qquad
\begin{array}{l}
F \longrightarrow \mathsf{const} \\
F.typ = \mathsf{const}.typ \\
F.ok\ \ = \mathsf{true}
\end{array}
$$

$$
\begin{array}{l}
F \longrightarrow \mathsf{var} \\
F.typ = F.env\ \mathsf{var}.id \\
F.ok\ \ = (F.env\ \mathsf{var}.id \neq \mathsf{error})
\end{array}
\qquad
\begin{array}{l}
F \longrightarrow \mathsf{var}\,() \\
F.typ = (F.env\ \mathsf{var}.id)\,() \\
F.ok\ \ = \mathbf{match}\ F.env\ \mathsf{var}.id \\
\qquad\quad \mathbf{with}\ \tau\,() \rightarrow \mathsf{true} \\
\qquad\quad\ \ |\quad \_\quad \rightarrow \mathsf{false}
\end{array}
$$

Das Attribut env der Nichtterminale E, T und F ist ererbt, während alle anderen Attribute der Grammatik AG_{types} abgeleitet sind. Auf die Einführung eines speziellen abgeleiteten Attributs ok wie in der Grammatik aus Beispiel 4.3.4 haben wir hier verzichtet. Allerdings sind die rechten Seiten der semantischen Regeln nun nicht mehr notwendigerweise *total*. So kann das Nachschlagen des Namens eines Bezeichners in der Symboltabelle env fehlschlagen. Auch liefert die rechte Seite der semantischen Regel zu der dritten Produktion für den Operator $'\div'$ den Wert Int nur, wenn die Attributvorkommen $T[1].typ$ und $F.typ$ beide den Wert Int liefern. Andernfalls ist sie nicht definiert. Für bestimmte Werte nicht definiert zu sein, fassen wir implizit auf als einen *Fehlerwert* zurück zu liefern. Da kein Operator für Fehlerwerte definiert ist, propagiert sich dieser weiter. Programme, die zu solchen Fehlern führen, kann der Übersetzer gegebenenfalls zurückweisen. □

Attributgrammatiken beziehen sich auf die konkrete Syntax. Wenn Operatorpräzedenzen durch die kontextfreie Grammatik ausgedrückt werden, führt das dazu, dass die kontextfreie Grammatik eine größere Anzahl von Kettenproduktionen benötigt, bei deren Anwendung die Werte der Attributexemplare im Syntaxbaum vom oberen Knoten zu dem unteren Knoten (im Falle von ererbten Attributen) oder vom unteren Knoten zum oberen Knoten (im Falle von abgeleiteten Attributen) kopiert werden müssen. Dieses Phänomen lässt sich an der Attributgrammatik AG_{types} sehr gut beobachten. Deshalb treffen wir

jetzt eine neue Konvention, die die Aufschreibung von Attributgrammatiken von den meisten identischen Übergaben befreit:

> Bei Fehlen einer semantischen Regel für ein ererbtes Attributvorkommen auf der rechten Seite (abgeleitetes Vorkommen auf der linken Seite) wird eine identische Übergabe von einem gleichnamigen ererbten Attributvorkommen auf der linken Seite (abgeleiteten Attributvorkommen auf der rechten Seite) angenommen.

Natürlich muss im Fall der identischen Übergabe an ein abgeleitetes Attribut der linken Seite genau ein gleichnamiges abgeleitetes Attribut auf der rechten Seite auftreten. Die folgenden Beispiele benutzen diese Konvention – zumindest für Regeln $A \longrightarrow \alpha X \beta$, deren rechte Seite nur ein Symbol X enthalten, dessen Attribute mit den Attributen der linken Seite A übereinstimmen.

Beispiel 4.3.4 (Verwaltung von Symboltabellen) Die Attributgrammatik AG_{scopes} verwaltet Symboltabellen für ein Fragment einer C-artigen imperativen Sprache mit parameterlosen Prozeduren. Dazu werden die Nichtterminale für Deklarationen, Anweisungen, Blöcke und Ausdrücke mit einem ererbten Attribut env assoziiert, welches jeweils die aktuelle Symboltabelle enthält.

Bei der Berechnung der Symboltabellen beachten wir, dass die Neudeklaration eines Bezeichners innerhalb eines Blocks verboten sein soll, während sie in einem neuen Block erlaubt ist. Um dies zu überprüfen, sammeln wir in einem weiteren ererbten Attribut $same$ die Menge der bezeichner, die bereits im aktuellen Block deklariert wurden. Das abgeleitete Attribut ok soll mitteilen, ob alle verwendeten Bezeichner korrekt deklariert und typkorrekt verwendet wurden.

$\langle decl \rangle \longrightarrow \langle type \rangle\ \mathsf{var};$

$\langle decl \rangle.new = (\mathsf{var}.id, \langle type \rangle.typ)$
$\langle decl \rangle.ok = \mathsf{true}$

$\langle decl \rangle \longrightarrow \mathsf{void\ var}\ ()\ \{\ \langle block \rangle\ \}$

$\langle block \rangle.same = \emptyset$
$\langle block \rangle.env = \langle decl \rangle.env \oplus \{\mathsf{var}.id \mapsto \mathsf{void}\,()\}$
$\langle decl \rangle.new = (\mathsf{var}.id, \mathsf{void}\,())$
$\langle decl \rangle.ok = \langle block \rangle.ok$

$\langle block \rangle \longrightarrow \langle decl \rangle\ \langle block \rangle$

$\langle decl \rangle.env = \langle block \rangle[0].env$
$\langle block \rangle[1].same = \mathbf{let}\ (x, \_) = \langle decl \rangle.new\ \mathbf{in}\ \langle block \rangle[0].same \cup \{x\}$
$\langle block \rangle[1].env = \mathbf{let}\ (x, \tau) = \langle decl \rangle.new\ \mathbf{in}\ \langle block \rangle[0].env \oplus \{x \mapsto \tau\}$
$\langle block \rangle[0].ok = \mathbf{let}\ (x, \_) = \langle decl \rangle.new\ \mathbf{in}\ \mathbf{if}\ \neg(x \in \langle block \rangle[0].same)\ \mathbf{then}\ \langle decl \rangle.ok \wedge \langle block \rangle[1].ok\ \mathbf{else}\ \mathsf{false}$

$\langle stat \rangle \longrightarrow E;$

$E.env = \langle stat \rangle.env$
$\langle stat \rangle.ok = E.ok$

$\langle stat \rangle \longrightarrow \{\ \langle block \rangle\ \}$

$\langle block \rangle.env = \langle stat \rangle.env$
$\langle block \rangle.same = \emptyset$
$\langle stat \rangle.ok = \langle block \rangle.ok$

$\langle block \rangle \longrightarrow \langle stat \rangle\ \langle block \rangle$

$\langle stat \rangle.env = \langle block \rangle[0].env$
$\langle block \rangle[1].env = \langle block \rangle[0].env$
$\langle block \rangle[1].same = \langle block \rangle[0].same$
$\langle block \rangle[0].ok = (\langle stat \rangle.ok \wedge \langle block \rangle[1].ok)$

$\langle block \rangle \longrightarrow \epsilon$

$\langle block \rangle.ok = \mathsf{true}$

In der Beispielgrammatik wurde nur eine minimalistische Menge von Produktionen für das Nichtterminalsymbol $\langle stat \rangle$ betrachtet.

Um eine vollständige Grammatik zu erhalten, sind weitere Produktionen für den Aufbau von Ausdrücken wie z.B. in der Grammatik aus Beispiel 4.3.3 und Typausdrücken erforderlich. Für den Fall, dass die Programmiersprache auch Typdeklarationen erlaubt, ist ein weiteres ererbtes Attribut erforderlich, das die aktuelle Typumgebung verwaltet.

Die angegebenen Regeln sammeln die Deklarationen von links nach rechts auf. Das schließt die Verwendung einer Prozedur *vor* dem syntaktischen Auftreten ihrer Deklaration aus. Sollen Prozeduren

dagegen im *gesamten* Block aufgerufen werden dürfen, in dem sie deklariert werden, benötigen wir eine modifizierte Attributgrammatik. Zur Unterscheidung von der Attributgrammatik AG_{scopes} nennen wir diese Grammatik $AG_{scopes+}$. Für die Attributgrammatik $AG_{scopes+}$ muss die Berechnung des Attributs env so modifiziert werden, dass bereits am Anfang eines Blocks sämtliche in dem Block deklarierten Prozeduren zu env hinzu gefügt werden. Um diese Menge zu ermitteln, wird das Nichtterminal $\langle block \rangle$ mit einem zusätzlichen abgeleiteten Attribut $procs$ ausgestattet, wobei zu den Produktionen für das Nichtterminal $\langle block \rangle$ die folgenden semantischen Regeln hinzugefügt werden:

$$
\begin{array}{l}
\langle block \rangle \longrightarrow \varepsilon \\
\langle block \rangle.procs = \emptyset
\end{array}
$$

$$
\begin{array}{l}
\langle block \rangle \longrightarrow \langle stat \rangle\ \langle block \rangle \\
\langle block \rangle[0].procs = \langle block \rangle[1].procs
\end{array}
$$

$$
\begin{array}{l}
\langle block \rangle \longrightarrow \langle decl \rangle\ \langle block \rangle \\
\langle block \rangle[0].procs = \mathbf{match}\ \langle decl \rangle.new \\
\qquad\qquad\qquad\qquad \mathbf{with}\ (x, \mathsf{void}()) \rightarrow \langle block \rangle[1].procs \oplus \{x \mapsto \mathsf{void}()\} \\
\qquad\qquad\qquad\qquad\ \mid\ \_ \qquad\qquad\ \rightarrow \langle block \rangle[1].procs
\end{array}
$$

Die in $\langle block \rangle.procs$ aufgesammelten Prozeduren können dann bei den Produktionen, die Blöcke einführen, zu der Typumgebung $\langle block \rangle.env$ hinzugefügt werden. Damit erhält die Attributgrammatik $AG_{scopes+}$ die folgenden semantische Regeln:

$$
\begin{array}{l}
\langle stat \rangle \longrightarrow \{\ \langle block \rangle\ \} \\
\langle block \rangle.env = \langle stat \rangle.env \oplus \langle block \rangle.procs
\end{array}
$$

$$
\begin{array}{l}
\langle decl \rangle \longrightarrow \mathsf{void\ var}\ ()\ \{\ \langle block \rangle\ \} \\
\langle block \rangle.env = \langle decl \rangle.env \oplus \langle block \rangle.procs
\end{array}
$$

Der Rest der Attributgrammatik $AG_{scopes+}$ stimmt mit der Attributgrammatik AG_{scopes} überein. Die neuen semantischen Regeln sind insofern bemerkenswert, als hier ererbte Attribute eines Nichtterminals auf der rechten Seite von Produktionen von abgeleiteten Attributen desselben Nichtterminals abhängen. □

Attributgrammatiken können auch eingesetzt werden, um Zwischencode oder sogar direkt Code, z.B. für virtuelle Maschinen zu erzeugen, wie wir sie in unserem ersten Band *Übersetzerbau: Virtuelle Maschinen* betrachtet haben. Die Code-Generierungsfunktionen, die wir dort vorstellten, sind rekursiv über der Struktur von Programmen definiert und verwenden Informationen über das Programm wie z.B. die Typen der in einem Programmfragment sichtbaren Variablen, welche sich ebenfalls elegant mit Hilfe von Attributgrammatiken berechnen lassen. Als Beispiel, wie eine solche Codeerzeugung mit Attributgrammatiken implementiert werden kann, betrachten wir als besonders vertracktes Teilproblem die Codeerzeugung für Boolesche Ausdrücke mit *Kurzschlussauswertung*.

Beispiel 4.3.5 Wir betrachten die Codeerzeugung für eine virtuelle Maschine ähnlich der CMA aus dem Buch *Übersetzerbau: Virtuelle Maschinen*. Der von der Attributgrammatik *BoolExp* erzeugte Code für einen Booleschen Ausdruck soll die folgenden Eigenschaften haben:

- es werden nur Lade-Befehle und bedingte Sprünge generiert;
- für die Booleschen Operatoren $\wedge, \vee$ und $\neg$ werden keine Befehle erzeugt;
- die Teilausdrücke des Ausdrucks werden von links nach rechts ausgewertet;
- von jedem (Teil-)Ausdruck werden nur die kleinsten Teilausdrücke ausgewertet, die den Wert des ganzen (Teil-)Ausdrucks eindeutig bestimmen.

Für den Booleschen Ausdruck $(a \wedge b) \vee \neg c$ mit Booleschen Variablen a, b und c wird etwa die folgende Befehlsfolge erzeugt:

	load a	
	jumpf l_1	jump-on-false
	load b	
	jumpt l_2	jump-on-true
l_1:	load c	
	jumpt l_3	
l_2:	Fortsetzung, falls der Ausdruck true liefert	
l_3:	Fortsetzung, falls der Ausdruck false liefert	

Die Attributgrammatik *BoolExp* muss Sprungziele (Marken) für Teilausdrücke erzeugen und diese Marken an die primitiven Teilausdrücke transportieren, von denen aus man diese Marken anspringen kann. Jeder Teilausdruck E oder T erhält die Marke $fsucc$ des Nachfolgers, wenn der Ausdruck sich zu false auswertet, sowie die Marke $tsucc$ des Nachfolgers, wenn er sich zu true auswertet. Außerdem wird im abgeleiteten Attribut $jcond$ berechnet, in welcher Beziehung der Wert des ganzen Ausdrucks zu dem Wert seines am weitesten rechts stehenden Bezeichners steht.

- Hat $jcond$ bei einem Ausdruck den Wert true, heißt das, dass der Wert des Ausdrucks gleich dem Wert des letzten Bezeichners des Ausdrucks ist, der bei der Ausführung geladen wird.
- Hat $jcond$ dagegen den Wert false, ist der Wert des Ausdrucks die Negation des Wertes des letzten geladenen Bezeichners des Ausdrucks.

Entsprechend wird nach dem Erzeugen eines *load*-Befehls für den letzten Bezeichner eines Ausdrucks die Marke $tsucc$ mit einem jumpt angesprungen, falls $jcond =$ true ist, bzw. mit einem jumpf, falls $jcond =$ false ist. Für diese Auswahl verwenden wir die folgende Hilfsfunktion:

$$\mathsf{gencjump}\,(jc, l) = \mathbf{if}\ jc\ \mathbf{then}\ (\mathsf{jumpt}\,l)\ \mathbf{else}\ (\mathsf{jumpf}\,l)$$

Um einen Kontext für Boolesche Ausdrücke zur Verfügung zu stellen, nehmen wir in der Grammatik eine Produktion für zweiseitige bedingte Anweisungen hinzu. Die Marken $tsucc$ und $fsucc$ der Bedingung bezeichnen dann die Anfangsadressen des Codes für den *then*- bzw. *else*-Teil der bedingten Anweisung. Am Ende der Bedingung soll ein bedingter Sprung zum *else*-Teil erzeugt werden. Dieser testet die Bedingung E auf false. Deswegen wird der Funktion gencjump als erster Parameter $\neg jcond$ übergeben. Damit erhalten wir:

$$
\begin{array}{lll}
\langle if\_stat\rangle \longrightarrow \mathsf{if}\ (E)\ \langle stat\rangle\ \mathsf{else}\ \langle stat\rangle & & \\
\quad E.tsucc & = \mathsf{new}() & \\
\quad E.fsucc & = \mathsf{new}() & \\
\quad \langle if\_stat\rangle.code = & \mathbf{let}\ t = E.tsucc & \\
& \mathbf{in\ let}\ e = E.fsucc & \\
& \mathbf{in\ let}\ f = \mathsf{new}() & \\
& \mathbf{in}\ E.code\ \hat{}\ \mathsf{gencjump}\,(\neg E.jcond, e)\hat{} & \\
& \quad t :\ \hat{}\ \langle stat\rangle[1].code\hat{}\ \mathsf{jump}\ f\hat{} & \\
& \quad e :\ \hat{}\ \langle stat\rangle[2].code\hat{} & \\
& \quad f : & \\
E \rightarrow T & & \\
E \rightarrow E\ \mathsf{or}\ T & & \\
\quad E[1].tsucc & = E[0].tsucc & T.tsucc = E[0].tsucc \\
\quad E[1].fsucc & = \mathsf{new}() & T.fsucc = E[0].fsucc \\
\quad E[0].jcond & = T.jcond & \\
\quad E[0].code & = \mathbf{let}\ t = E[1].fsucc & \\
& \mathbf{in}\ E[1].code\hat{}\ \mathsf{gencjump}\,(E[1].jcond, E[0].tsucc)\hat{} & \\
& \quad t :\ \hat{}\ T.code & \\
T \rightarrow F & & \\
T \rightarrow T\ \mathsf{and}\ F & & \\
\quad T[1].tsucc & = \mathsf{new}() & F.tsucc = T[0].tsucc \\
\quad T[1].fsucc & = T[0].fsucc & F.fsucc = T[0].fsucc \\
\quad T[0].jcond & = F.jcond & \\
\quad T[0].code & = \mathbf{let}\ f = T[1].tsucc & \\
& \mathbf{in}\ T[1].code\hat{}\ \mathsf{gencjump}\,(\neg T[1].jcond, T[0].fsucc)\hat{} & \\
& \quad f :\ \hat{}\ F.code & \\
F \rightarrow (E) & & \\
F \rightarrow \mathsf{not}\ F & & \\
\quad F[1].tsucc & = F[0].fsucc & \\
\quad F[1].fsucc & = F[0].tsucc & \\
\quad F[0].code & = F[1].code & \\
\quad F[0].jcond & = \neg F[1].jcond & \\
F \rightarrow \mathsf{var} & & \\
\quad F.jcond & = \mathsf{true} & \\
\quad F.code & = \mathsf{load\ var}.id &
\end{array}
$$

Dabei bezeichnet die Infix-Operation ˆ die Konkatenation von Code-Fragmenten. Diese Attributgrammatik ist *nicht* normalisiert: zur Berechnung des abgeleiteten Attributs $code$ der linken Seite $ifstat$ der ersten Produktion wird z.B. auf die ererbten Attribute $tsucc$ und $fsucc$ des Nichtterminals E auf der rechten Seite zugegriffen. Der Grund ist, dass die beiden ererbten Attribute mit Hilfe von Aufrufen einer Funktion new() berechnet werden, die bei jedem erneuten Aufruf eine *andere* Sprungmarke generiert. Weil sie implizit einen *globalen* Zustand verändert, ist die Hilfsfunktion new() damit streng genommen auf der rechten Seite einer semantischen Regel gar nicht zugelassen! Hier sind prinzipiell zwei Lösungen denkbar:

- Der globale Zustand, also z.B. ein Zähler der bereits vergebenen Marken, wird mit eigenen Attributen durch den Syntaxbaum geschleust. Die Generierung einer neuen Marke greift dann auf diesen Zähler zu und muss nicht mehr auf einen externen globalen Zustand zugreifen. Der Nachteil dieses

Vorgehens ist, dass die eigentliche Logik innerhalb der Attributgrammatik durch die Hilfsprädikate verschleiert wird.

- Wir verwenden die Hilfsfunktion new() wie in der Beispielgrammatik angegeben. Dann müssen wir jedoch auf die Normalisierung einiger semantischer Regeln verzichten, da Funktionsaufrufe mit Seiteneffekten nicht dupliziert werden dürfen. Weiterhin müssen wir uns vergewissern, dass bei unterschiedlichen Auswertungsreihenfolgen der Attributexemplare eines Syntaxbaums, wenn schon nicht die *gleiche*, so doch zumindest stets eine *akzeptable* Attributierung berechnet wird.

□

4.4 Die Generierung von Attributauswertern

In diesem Abschnitt befassen wir uns mit der Auswertung von Attributen, genauer Attributexemplaren, in Syntaxbäumen. Eine Attributgrammatik definiert für jeden Syntaxbaum t der zugrundeliegenden kontextfreien Grammatik ein Gleichungssystem $\mathsf{GLS}(t)$. Die Unbekannten dieses Gleichungsystems sind die Attributexemplare zu den Knoten des Syntaxbaums t. Nehmen wir an, die Attributgrammatik sei wohlgeformt. Dann ist das Gleichungssystem nicht rekursiv. Gleichungssysteme, die nicht rekursiv sind, können durch ein Eliminationsverfahren gelöst werden. In jedem Eliminationsschritt wird ein Attributexemplar ausgewählt, das nur von bereits berechneten Exemplaren abhängt und dann sein Wert berechnet. Ein solcher Attributauswerter ist insofern rein *dynamisch*, als keinerlei Informationen darüber ausgenutzt wird, woraus das Gleichungssystem $\mathsf{GLS}(t)$ generiert wurde. Einen solchen Auswerter beschreiben wir im nächsten Abschnitt.

4.4.1 Bedarfsgetriebene Auswertung der Attribute

Einen einigermaßen effizienten dynamischen Attributauswerter für wohlgeformte Attributgrammatiken erhalten wir, indem wir Atributexemplare *bedarfsgetrieben* auswerten.

Bedarfsgetriebene Auswertung bedeutet, dass auf die vorgängige Berechnung der Werte sämtlicher Attributexemplare verzichtet wird. Stattdessen wird die Auswertung durch eine *Wertanfrage* an ein Attributexemplar angestoßen. Dazu implementieren wir eine rekursive Funktion solve, die für einen Knoten n und ein Attribut a des Symbols am Knoten n aufgerufen wird. Die Auswertung überprüft zuerst, ob für das nachgefragte Attributexemplar $n.a$ bereits ein Wert berechnet wurde. Ist dies der Fall, wird dieser Wert zurück geliefert. Andernfalls wird die Berechnung des Werts für $n.a$ angestoßen. Diese Berechnung wird möglicherweise die Werte weiterer Attributesemplare nachfragen, deren Auswertung dann rekursiv angestoßen wird. Diese Strategie hat zur Folge, dass zu jedem Attributexemplar des Syntaxbaums höchstens einmal die rechten Seite einer definierenden semantischen Regel ausgewertet wird. Die Auswertung von Attributexemplaren, die *nie* nachgefragt werden, wird ganz vermieden.

Um diese Idee umzusetzen, werden vor der ersten Wertanfrage alle Attributexemplare, die nicht vordefiniert sind, mit dem Wert Undef initialisiert. Jedes mit einem Wert d vorbelegte Attributexemplar wird auf den Wert Value d gesetzt. Für die Navigation im Syntaxbaum greifen wir auf die Postfixoperatoren $[i]$ zurück, mit denen wir von einem Knoten n zu seinen i-ten Nachfolgern navigieren können (bzw. für $i = 0$ bei n bleiben). Weiterhin benötigen wir einen Operator father, der für einen Knoten n das Paar (n', j) zurück liefert, das aus dem Vater n' des Knotens n besteht zusammen mit der Information, in welcher Richtung, von n' aus betrachtet, n zu finden ist, d.h. die angibt, das wievielte Kind der Knoten vom Vaterknoten n' ist. Um die Funktion solve zur rekursiven Auswertung zu implementieren, benötigen wir eine Funktion eval. Ist p die Produktion, die am Knoten n angewandt wurde, und

$$f(p[i_1].a_1, \ldots, p[i_r].a_r)$$

die rechte Seite der semantischen Regel für das Attributvorkommen $p[i].a$, dann liefert eval $n\ (i, a)$ den Wert von f zurück, wobei für jedes benötigte Attributexemplar erst die Funktion solve aufgerufen wird, d.h.

$$\mathsf{eval}\ n\ (i, a) = f(\mathsf{solve}\ n[i_1]\ a_1, \ldots, \mathsf{solve}\ n[i_r]\ a_r)$$

Simultan rekursiv mit der Funktion eval wird die Funktion solve implementiert durch:

```
solve n a = match n.a
    with Value d → d
      | Undef   → if b ∈ S(symb(n))
                  then  let d = eval n (0, a)
                        in let _ = n.a ← Value d
                        in d
                  else  let (n', j') = father n
                        in let d' = eval n' (j', a)
                        in let _ = n.a ← Value d'
                        in d'
```

Die Funktion solve überprüft, ob das Attibutexemplar $n.a$ im Syntaxbaum bereits einen Wert hat. Ist das so, wird dieser Wert zurückgeliefert. Hat das Attributexemplar $n.a$ noch keinen Wert, ist $n.a$ mit Undef markiert. In diesem Fall wird die zugehörige semantische Regel für $n.a$ gesucht.

Ist a ein abgeleitetes Attribut des Symbols am Knoten n, gibt es eine semantische Regel zu der Produktion p am Knoten n. Die rechte Seite f dieser Regel wird dabei so modifiziert, dass nicht direkt auf die entsprechenden Attributexemplare zugegriffen, sondern stattdessen für diese jeweils erneut die Funktion solve für den Knoten n aufgerufen wird. Ist der Wert d für das Attributexemplar $n.a$ ermittelt, wird er in dem Attributexemplar $n.a$ abgelegt und zusätzlich als Ergebnis zurück geliefert.

Ist a ein ererbtes Attribut des Symbols am Knoten n, wird die semantische Regel für $n.a$ nicht von n, sondern vom Vater von n bereitgestellt. Sei n' der Vater von n und n das j'-te Kind von n'. Dann wird bei der Produktion p' am Knoten n' die semantische Regel für das Attributvorkommen $p'[j'].a$ ausgewählt. Deren rechte Seite wird erneut so modifiziert, dass vor jedem Zugriff auf das entsprechende Attributexemplar die Funktion solve für den Knoten n' aufgerufen wird. Der berechnete Wert wird wieder im Attributexemplar $n.a$ abgespeichert und als Rückgabewert zurückgeliefert.

Ist die Attributgrammatik wohlgeformt, berechnet der bedarfsgetriebene Auswerter in jedem Syntaxbaum für jedes Attributexemplar stets den richtigen Wert. Ist die Attributgrammatik nicht wohlgeformt, gibt es Syntaxbäume, für die das zugehörige Gleichungssystem rekursiv ist. Ist t so ein Syntaxbaum, gibt es in t einen Knoten n und ein Attribut a an n, so dass $n.a$ mittelbar oder unmittelbar von sich selbst abhängt. Der Aufruf solve n a wird dann möglicherweise nicht terminieren. Diese Nichtterminierung kann vermieden werden, indem an einem Attributexemplar, dessen Auswertung angestoßen, aber noch nicht beendet ist, eine gesonderte Markierung Called angebracht wird. Trifft die Funktion solve auf ein Attributexemplar, das mit Called markiert ist, kann die Berechnung sofort abgebrochen und eine Fehlermeldung zurück geliefert werden (siehe Aufgabe **??**).

4.4.2 Statische Vorberechnungen für Attributauswerter

Dynamische Attributauswertung wie das Verfahren des letzten Abschnitts sind sehr flexibel, nutzen jedoch keinerlei Wissen über die vorliegende Attributgrammatik aus.

Statische Attributauswertungsverfahren dagegen versuchen, sich die Kenntnis der funktionalen Abhängigkeiten innerhalb der semantischen Regeln einer Produktion zu Nutze zu machen. Ein Attributvorkommen $p[i].a$ bei einer Produktion p hängt von einem Vorkommen $p[j].b$ an p funktional ab, wenn $p[j].b$ ein Argument für die semantische Regel von p für $p[i].a$ ist.

Diese Abhängigkeiten bestimmen die Abhängigkeiten zwischen Attributexemplaren im Gleichungssystem GLS(t). Werden die funktionalen Abhängigkeiten zwischen Attributvorkommen berücksichtigt, können gegebenenfalls die Attributexemplare gemäß einer statisch bestimmten *Besuchsreihenfolge* so auswertet werden, dass bei der Auswertung eines Attributexemplars die Werte der Attributexemplare, auf welche die zugehörige semantische Regel zugreift, bereits vorliegen. Betrachten wir noch einmal Abbildung 4.6. Die Attributauswertung erfordert ein Zusammenspiel von Berechnungen, die *lokal* an einem Knoten n und seinen Nachfolgern $n_1, \ldots, n_k$ stattfinden, und solchen in der Umgebung. Eine lokale Berechnung eines (Exemplars eines) definierenden Vorkommens an dem Knoten n, der mit X_0 beschriftet ist, stellt der lokalen Berechnung am Knoten oberhalb einen neuen Wert zur Verfügung.

Eine Berechnung eines Attributexemplars am gleichen Knoten n, die oberhalb stattfindet, macht einen neuen Wert verfügbar, der eventuell wieder neue lokale Berechnungen gemäß den semantischen Regeln für die Produktion $p: \; X_0 \longrightarrow X_1 \ldots X_k$ ermöglicht. Ein ähnlicher Datenaustausch findet über die Attributexemplare an den Knoten $n_1, \ldots, n_k$ mit den darunter vorgenommenen Berechnungen statt. Soll dieses Zusammenspiel statisch geplant werden, müssen die *globalen* funktionalen Abhängigkeiten zwischen Attributexemplaren analysiert werden. Ausgangspunkt für die Berechnung der globalen funktionalen Abhängigkeiten sind die produktionslokalen funktionalen Abhängigkeiten. Wir führen folgende Begriffe ein.

Für eine Produktion p sei $V(p)$ die Menge der Attributvorkommen in p. Die semantischen Regeln zur Produktion p definieren auf der Menge $V(p)$ die Relation $Dp(p) \subseteq V(p) \times V(p)$ der *produktionslokalen* funktionalen Abhängigkeiten. Die Relation $Dp(p)$ enthält ein Paar $(p[j].b, p[i].a)$ von Attributvorkommen genau dann, wenn $p[j].b$ in der rechten Seite einer semantischen Regel für $p[i].a$ vorkommt.

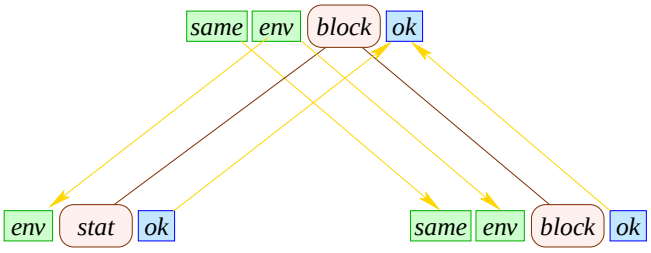


Abb. 4.7. Die produktionslokale Abhängigkeitsrelation zur Produktion $block \rightarrow stat \;\; block$ aus AG_{scopes}.

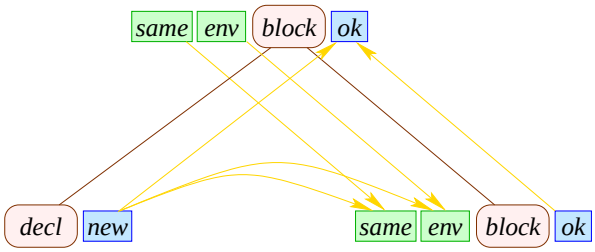


Abb. 4.8. Die produktionslokale Abhängigkeitsrelation zur Produktion $block \rightarrow decl \;\; block$ aus AG_{scopes}.

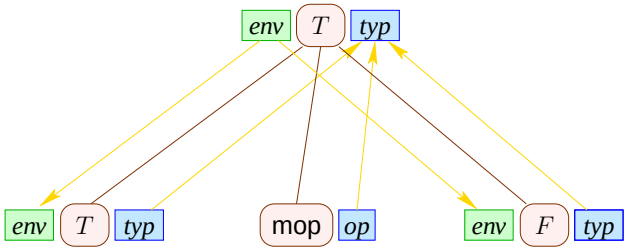


Abb. 4.9. Die produktionslokale Abhängigkeitsrelation zur Produktion $T \rightarrow T \;\mathsf{mop}\; F$ aus AG_{types}.

Beispiel 4.4.1 (Fortführung der Beispiele 4.3.4 und 4.3.3) Zur besseren Lesbarkeit stellen wir Attributabhängigkeitsrelationen immer zusammen mit der zugrundeliegenden syntaktischen Struktur,

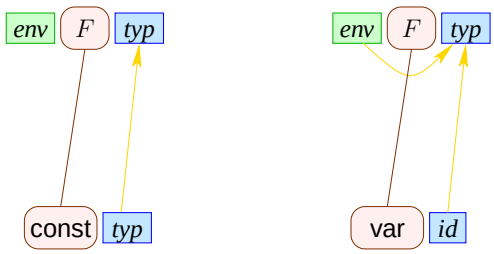


Abb. 4.10. Die produktionslokalen Abhängigkeitsrelationen der Produktionen $F \longrightarrow \mathsf{const}$ und $F \longrightarrow \mathsf{var}$ aus AG_{types}.

d.h. der Produktion oder dem Syntaxbaum dar. Die Abhängigkeitsrelationen für die Produktionen $block \longrightarrow stat\ \ block$ und $block \longrightarrow decl\ \ block$ aus AG_{scopes} zeigen die Abb. 4.7 und 4.8. Die produktionslokalen Abhängigkeitsrelationen zur Attributgrammatik AG_{types} sind alle sehr einfach: es gibt Abhängigkeiten zwischen dem Vorkommen des ererbten Attributs env der linken Seite und den ererbten Vorkommen der Attribute env der rechten Seite sowie zwischen den abgeleiteten Attributen typ und op der rechten Seite zu dem abgeleiteten Attribut typ der linken Seite (siehe Abb. 4.9). Nur bei der Produktion $F \longrightarrow \mathsf{var}$ ergibt sich eine Abhängigkeit zwischen den Attributen env und typ des Nichtterminals F (siehe Abb. 4.10). □

In Attributgrammatiken in Normalform sind die Argumente in den semantischen Regeln der definierenden Vorkommen immer angewandte Attributvorkommen. Deshalb haben alle Wege in allen produktionslokalen Abhängigkeitsrelationen die Länge 1, und es gibt auch keine Zykel der Form $(p[i].a, p[i].a)$. Das Vorliegen der Normalform erleichtert damit einige Betrachtungen. Wenn nicht anders gesagt, gehen wir im Folgenden immer von Attributgrammatiken in Normalform aus.

Die produktionslokalen Abhängigkeiten zwischen Attributvorkommen in Produktionen führen zu Abhängigkeiten zwischen Argumentexemplaren in den Syntaxbäumen der Grammatik. Sei t ein Baum der kontextfreien Grammatik, die unserer Attributgrammatik zu Grunde liegt. Die *individuelle* Abhängigkeitsrelation auf der Menge $V(t)$ der Attributexemplare von t, $Dt(t)$, erhält man durch *Instantiierung* der produktionslokalen Abhängigkeitsrelationen der in t angewendeten Produktionen. Für jeden Knoten n in t, an dem die Produktion p angewendet wurde, enthält die Relation $Dt(t)$ genau die Paare $(n[j].b, n[i].a)$ mit $(p[j].b, p[i].a) \in Dp(p)$.

Beispiel 4.4.2 (Fortführung von Beispiel 4.3.4) Die Abhängigkeitsrelation zu dem Syntaxbaum der Anweisung $\{$ **int** $x;\ x = 1;\ \}$ gemäß der Attributgrammatik AG_{scopes} zeigt Abbildung 4.11. Der Einfachheit halber wurde angenommen, dass sich das Nichtterminal $type$ direkt zu dem Basistyp int und sich das Nichtterminal E für Ausdrücke direkt zu dem Terminal const ableiten lassen. □

Eine Relation R auf einer Menge A nennen wir zyklisch, wenn ihre transitive Hülle ein Paar (a, a) enthält. Andernfalls nennen wir die Relation R *azyklisch*. Eine Attributgrammatik nennen wir *zyklenfrei*, wenn alle individuellen Abhängigkeitsrelationen der Attributgrammatik azyklisch sind. Eine individuelle Abhängigkeitsrelation $Dt(t)$ ist genau dann azyklisch, wenn das Gleichtungssystem $\mathsf{GLS}(t)$, das wir in 4.3.1 für einen Ableitungsbaum t eingeführt hatten, nicht rekursiv ist. Unter dieser letzten Bedingung nannten wir eine Attributgrammatik wohlgeformt. Also ist eine Attributgrammatik genau dann wohlgeformt, wenn sie zyklenfrei ist.

Betrachten wir einen Ableitungsbaum t mit Wurzelmarkierung X wie in Abbildung 4.12. Die Exemplare der ererbten Attribute an der Wurzel fassen wir als die Eingabe für t auf und die Exemplare der abgeleiteten Attribute an der Wurzel als die Ausgabe von t. Das Exemplar von d an der Wurzel hängt (transitiv) nur von dem Exemplar von c an der Wurzel ab. Ist der Wert des Exemplars von c bekannt, kann ein Attributauswerter in t hinabsteigen und mit dem Wert für das Exemplar von d zurückkehren, da es weitere Abhängigkeiten von Exemplaren von außerhalb t, die nicht durch c gehen, nicht gibt. Das Exemplar von e an der Wurzel hängt von den Exemplaren von a und b an der Wurzel ab. Liegen beide Werte vor, kann die Auswertung des Exemplars von e angestoßen werden. Diese Situation beschreibt die von t induzierte *untere charakteristische Abhängigkeitsrelation* von X.

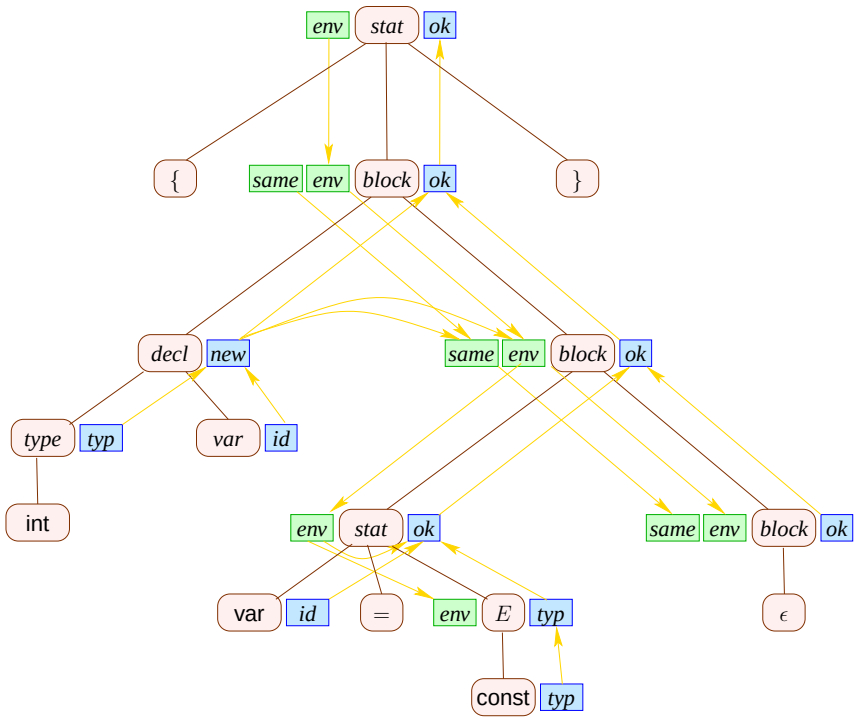


Abb. 4.11. Die individuelle Abhängigkeitsrelation für den Syntaxbaum zu { int x; $x = 1$; } bzgl. der Attributgrammatik AG_{scopes}.

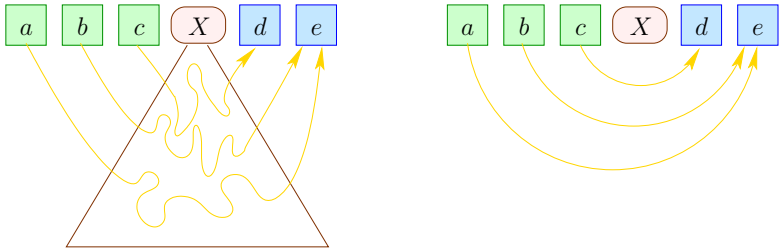


Abb. 4.12. Attributabhängigkeiten in einem Syntaxbaum für X und die induzierte untere charakteristische Abhängigkeitsrelation.

Sei t ein Syntaxbaum für ein Symbol X mit Wurzel n. Dann besteht die von t induzierte untere charakteristische Abhängigkeitsrelation $R_t(X)$ für X auf der Menge $\mathcal{A}(X)$ der Attribute von X aus allen Paaren (a, b) von Attributen, für die das Paar $(n.a, n.b)$ von Attributexemplaren an der Wurzel n von t in der transitiven Hülle der individuellen Abhängigkeitsrelation $Dt(t)$ liegt. Insbesondere ist damit

$$R_t(X) \subseteq \mathcal{I}(X) \times \mathcal{S}(X).$$

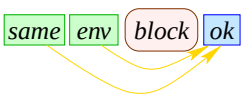


Abb. 4.13. Untere charakteristische Abhähngigkeitsrelation zu $block$.

Beispiel 4.4.3 (Fortführung von Beispiel 4.4.2) Die von dem Unterbaum der Wurzel von t mit Wurzel $block$ aus Beispiel 4.4.2 induzierte untere charakteristische Relation für das Nichtterminal $block$ ist in Abbildung 4.13 dargestellt. □

Lemma 4.1. Für eine Attributgrammatik sind die folgenden Aussagen äquivalent:

1. Für jeden Syntaxbaum t ist die untere charakteristische Abhängigkeitsrelation $U_t(X)$ (X das Symbol an der Wurzel von t) azyklisch;
2. Für jeden Syntaxbaum t ist die Abhängigkeitsrelation $Dt(t)$ azyklisch.

Im Unterschied zu der Menge der Abhängigkeitsrelationen einer Attributgrammatik ist die Menge der *unteren* Abhängigkeitsrelationen notwendigerweise endlich, weil sie sich nur auf die Menge der Attribute einzelner Nichtterminale bezieht. Indem wir die Menge aller unteren Abhängigkeitsrelationen berechnen, können wir damit entscheiden, ob für jeden Syntaxbaum t der Abhängigkeitsrelationen $Dt(t)$ azyklisch ist und damit der bedarfsgetriebene Attributauswerter immer terminiert.

Sei X ein Symbol mit einer Menge $\mathcal{A}$ von Attributen. Für eine Relation $R \subseteq \mathcal{A}^2$ und $i \geq 0$ und eine Produktion p definieren wir die Relation $R[p,i]$ durch

$$R[p,i] = \{(p[i].a, p[i].b) \mid (a,b) \in R\}$$

Betrachten wir nun eine Produktion $p : \quad X \longrightarrow X_1 \ldots X_k$ Für eine binäre Relation $S \subseteq V(p)^2$ auf der Menge der Attributvorkommen der Produktion p definieren wir weiterhin die folgenden zwei Operationen:

$$\begin{array}{ll} S^+ = \bigcup\{S^j \mid j \geq 1\} & \text{(transitive Hülle)} \\ \pi_i(S) = \{(a,b) \mid (p[i].a, p[i].b) \in S\} & \text{(Projektion)} \end{array}$$

Die Projektion gestattet uns, aus einer Abhängigkeitsrelation für Attributvorkommen einer Produktion p die induzierten Abhängigkeiten zwischen den einzelnen Attributen eines Symbols zu extrahieren, das in der Produktion p vorkommt. Damit können wir den Effekt $[\![p]\!]^\sharp$ der Anwendung der Produktion p auf Abhängigkeitsrelationen $R_1, \ldots, R_k$ für die Symbolvorkommen $p[i]$ auf der rechten Seite von p definieren durch:

$$[\![p]\!]^\sharp(R_1, \ldots, R_k) = \pi_0((Dp(p) \cup R_1[p,1] \cup \ldots \cup R_k[p,k])^+)$$

Die Operation $[\![p]\!]^\sharp$ nimmt die lokale Abhängigkeitsrelation der Produktion p und fügt die (geeignet instantiierten) Abhängigkeitsrelationen für die Symbolvorkommen in der rechten Seite hinzu. Von dieser Relation wird der transitive Abschluss gebildet, welcher dann auf die Attribute der linken Seite von p projiziert wird. Wird die Produktion p an der Wurzel eines Syntaxbaums t angewendet, und sind die Relationen $U_1, \ldots, U_k$ die unteren Abhängigkeitsrelationen für die Teilbäume unter der Wurzel von t, dann ergibt sich die untere charakteristische Abhängigkeitsrelation für t gerade durch

$$U_t(X) = [\![p]\!]^\sharp(U_1, \ldots, U_k)$$

Die Mengen $\mathcal{U}(X), X \in V$, aller unteren Abhängigkeitsrelationen für Nichtterminalsymbole X ergeben sich als die kleinste Lösung des Gleichungssystems:

$$(\mathcal{U}) \qquad \begin{array}{ll} \mathcal{U}(a) = \{\emptyset\}\,, & a \in V_T \\ \mathcal{U}(X) = \{[\![p]\!]^\sharp(U_1, \ldots, U_k) \mid p : X \to X_1 \ldots X_k \in P, U_i \in \mathcal{U}(X_i)\}\,, & X \in V_N \end{array}$$

Dabei sind V_T, V_N und P die Mengen der Terminal- und Nichtterminalsymbole bzw. der Produktionen der kontextfreien Grammatik, die der Attributgrammatik zu Grunde liegt. Jede rechte Seite dieser Gleichungen ist *monoton* in jeder Unbekannten $\mathcal{U}(X_i)$, von der sie abhängt. Weil die Menge aller transitiven binären Relationen auf einer endlichen Menge endlich ist, ist auch die Menge ihrer Teilmengen endlich. Deshalb können wir iterativ die kleinste Lösung des Gleichungssystems und damit für jedes Symbol X die Menge aller unteren Abhängigkeitsrelationen für X berechnen. Indem wir die auftretenden Relationen auf Zyklenfreiheit überprüfen, können wir entscheiden, ob die Abhängigkeitsrelationen aller Syntaxbäume zyklenfrei sind und die Attributgrammatik damit wohlgeformt ist.

Satz 4.4.1 Es ist entscheidbar, ob eine Attributgrammatik wohlgeformt ist oder nicht. □

Um Wohlgeformtheit zu entscheiden, berechnen wir sämtliche unteren Abhängigkeitsrelationen der Attributgrammatik. Diese Menge ist zwar endlich, kann jedoch exponentiell in der Anzahl der verwendeten Attribute wachsen. Praktisch ist die Überprüfung der Zyklenfreiheit damit nur durchführbar, wenn entweder die Anzahl der verwendeten Attribute klein ist oder für jedes Symbol nur wenige untere Abhängigkeitsrelationen auftreten. Im Allgemeinen ist der exponentielle Aufwand allerdings unvermeidbar, da das Problem, für eine Attributgrammatik Zyklenfreiheit nachzuweisen, *EXPTIME*-vollständig ist.

In vielen Attributgrammatiken gibt es zwar gegebenenfalls mehrere untere charakteristische Abhängigkeitsrelationen zu einem Nichtterminalsymbol X, diese sind jedoch alle in einer gemeinsamen transitiven azyklischen Abhängigkeitsrelation enthalten.

Beispiel 4.4.4 Betrachten wir dazu etwa die Attributgrammatik AG_{scopes} aus Beispiel 4.3.4. Dann gibt es für das Nichtterminal $block$ die folgenden unteren charakteristischen Abhängigkeitsrelationen:

$$\begin{array}{ll} (1) & \emptyset \\ (2) & \{(same, ok)\} \\ (3) & \{(env, ok)\} \\ (4) & \{(same, ok), (env, ok)\} \end{array}$$

Hier sind die ersten drei Abhängigkeitsrelationen sämtlich in der vierten enthalten. □

Um für jedes Symbol X eine transitive Relation zu berechnen, die alle unteren charakteristischen Abhängigkeiten für X enthält, stellen wir das folgende Gleichungsystem über transitiven Relationen auf:

$$\begin{array}{lll} (\mathcal{R}) & \mathcal{R}(a) = \emptyset\,, & a \in V_T \\ & \mathcal{R}(X) = \bigsqcup\{[\![p]\!]^\sharp(\mathcal{R}(X_1), \ldots, \mathcal{R}(X_k)) \mid p : X \to X_1 \ldots X_k \in P\}\,, & X \in V_N \end{array}$$

Die Ordnungsrelation auf transitiven Relationen ist die Teilmengenrelation $\subseteq$. Beachten Sie, dass die kleinste obere Schranke der transitiven Relationen $R \in \mathcal{S}$ jedoch nicht einfach deren Vereinigung ist. Vielmehr gilt:

$$\bigsqcup \mathcal{S} = (\bigcup \mathcal{S})^+$$

d.h. nach der Vereinigung der Relationen muss noch einmal die transitive Hülle gebildet werden. Für jede Produktion p ist die Operation $[\![p]\!]^\sharp$ monoton in jedem ihrer Argumente. Deshalb besitzt das Gleichungssystem eine kleinste Lösung. Weil es nur endlich viele transitive Relationen auf der Menge der Attribute gibt, lässt sich diese Lösung durch einen iterativen Algorithmus berechnen. Seien $\mathcal{U}(X), X \in V$, und $\mathcal{R}(X), X \in V$, die kleinsten Lösungen der Gleichungssysteme $(\mathcal{U})$ bzw. $(\mathcal{R})$. Mit Induktion über die einzelnen Iterationen des Fixpunktalgorithmus lässt sich zeigen, dass für alle $X \in V$

$$\mathcal{R}(X) \supseteq \bigcup \mathcal{U}(X)$$

gilt. Wir schließen, dass alle charakteristischen unteren Abhängigkeitsrelationen der Attributgrammatik azyklisch sind, falls nur alle Relationen $\mathcal{R}(X), X \in V$, azyklisch sind. Eine Attributgrammatik, bei der alle Relationen $\mathcal{R}(X), X \in V$, azyklisch sind, heißt *absolut zyklenfrei*. Jede absolut zyklenfreie Attributgrammik ist folglich wohlgeformt. Das bedeutet, dass für absolut zyklenfreie Attributgrammatiken der Algorithmus zur bedarfsgetriebenen Attributauswertung stets terminiert. Mit dem Lösen des Gleichungssystems $(\mathcal{R})$ haben wir damit ein polynomielles Kriterium identifiziert, das die Anwendbarkeit der bedarfsgetriebenen Attributauswertung garantiert.

Analog zu der *unteren* charakteristischen Abhängigkeitsrelation des Vorkommens eines Symbols X an einem Knoten n in einem Ableitungsbaum t lässt sich auch die *obere* charakteristische Abhängigkeitsrelation zu X definieren. Diese wird durch die Attributabhängigkeiten des oberen Baumfragments zu n vermittelt. Zur Erinnerung: das obere Baumfragment von t an n ist der Baum, den wir erhalten, indem wir aus t den Teilbaum mit Wurzel n durch den Knoten n ersetzen. Dieses obere Baumfragment

bezeichnen wir mit $t\backslash n$. Sei $Dt(t\backslash n)$ die individuelle Abhängigkeitsrelation des oberen Baumfragments, d.h. die Menge aller Paare $(n_1.a, n_2, b)$ der individuellen Abhängigkeitsrelation $Dt(t)$, für die sowohl n_1 wie n_2 im oberen Baumfragment $t\backslash n$ liegen. Dann besteht die obere charakteristische Abhängigkeitsrelation $O_{t,n}(X)$ für X am Knoten n in t aus allen Paaren $(a, b) \in \mathcal{A}(X) \times \mathcal{A}(X)$, für die das Paar $(n.a, n.b)$ in der transitiven Hülle von $Dt(t\backslash n)$ liegt (siehe Abb. 4.14). Auch für die Menge $\mathcal{O}(X)$ aller möglichen oberen charakteristischen Abhängigkeitsrelationen des Symbols X lässt sich ein Gleichungssystem über Mengen transitiver Relationen aufstellen (siehe Aufgabe **??**).

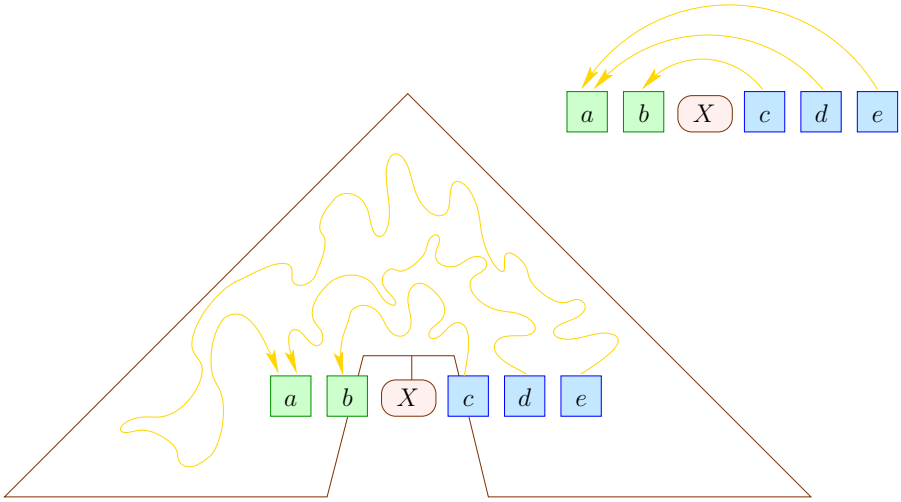


Abb. 4.14. Attributabhängigkeiten in einem oberen Baumfragment für X und die induzierte obere charakteristische Abhängigkeitsrelation.

4.4.3 Besuchsgesteuerte Attributauswertung

Der Vorteil eines statisch generierten Attributauswerters über den bedarfsgetriebenen dynamischen Auswerter des letzten Abschnitts ist, dass das Verhalten des Auswerters an jedem Knoten bereits zur Generierungszeit festgelegt wird: der Test zur Auswertungszeit an jedem Attributexemplar, ob es bereits ausgewertet ist, entfällt.

Die größte Klasse von Attributgrammatiken, für die wir die Generierung statischer Attributauswerter beschreiben, ist die Klasse der *l-geordneten* oder *simple-multi-visit* Attributgrammatiken. Eine Attributgrammatik heißt *l-geordnet*, wenn es eine Abbildung $\mathcal{T}$ gibt, die jedem Symbol X eine *totale* Ordnung $\mathcal{T}(X) \subseteq \mathcal{A}^2$ auf der Menge $\mathcal{A}$ der Attribute von X zuordnet, die mit allen Produktionen *kompatibel* ist. Das bedeutet, dass für jede Produktion $p: \; X_0 \longrightarrow X_1 \ldots X_k$ der zugrundeliegenden Grammatik die Relation

$$D_{\mathcal{T}}(p) = (Dp(p) \cup \mathcal{T}(X_0)[p, 0] \cup \ldots \cup \mathcal{T}(X_k)[p, k])^+$$

azyklisch ist. Diese Aussage ist äquivalent dazu, dass

$$\mathcal{T}(X_i) = \pi_i((Dp(p) \cup \mathcal{T}(X_0)[p, 0] \cup \ldots \cup \mathcal{T}(X_k)[p, k])^+)$$

gilt für alle i. Deshalb gilt insbesondere:

$$\mathcal{T}(X_0) \supseteq [\![p]\!]^\sharp(\mathcal{T}(X_1), \ldots, \mathcal{T}(X_k))$$

Indem wir diese Ungleichung mit der Gleichung für die Unbekannte X_0 im Gleichungssystem $(\mathcal{R})$ aus dem letzten Abschnitt vergleichen, können wir folgern, dass die totale Ordnung $\mathcal{T}(X_0)$ die Abhängigkeitsrelation $\mathcal{R}(X_0)$ umfasst. Da $\mathcal{T}(X_0)$ eine totale Ordnung und damit azyklisch ist, ist die

Attributgrammatik absolut zyklenfrei, wobei sämtliche lokalen unteren Abhängigkeitsrelationen an X_0 in $\mathcal{T}(X_0)$ enthalten sind. Analog kann man zeigen, dass $\mathcal{T}(X_0)$ alle *oberen* Abhängigkeitsrelationen an X_0 umfasst.

Beispiel 4.4.5 (Fortführung von Beispiel 4.3.4) Bei der Attributgrammatik AG_{scopes} bieten sich für die Symbole $stat, block, decl, E$ und var die folgenden totalen Ordnungen auf den Attributmengen an:

$stat$	$env \rightarrow ok$
$block$	$same \rightarrow env \rightarrow ok$
$decl$	new
E	$env \rightarrow ok$
var	id

□

Sei $B_{\mathcal{T}}(X) \in \mathcal{A}(X)^*$ die lineare Abfolge der Attribute von X gemäß der totalen Ordnung $\mathcal{T}(X)$. Diese lineare Abfolge lässt sich faktorisieren in Teilfolgen aus rein ererbten bzw. rein abgeleiteten Attributen. In unserem Beispiel ist diese Faktorisierung für alle betrachteten Symbole sehr einfach: alle ererbten Attribute stehen stets vor allen abgeleiteten Attributen. Im allgemeinen können einige ererbte Attribute auch von anderen (zuvor) abgeleiteten Attributen abhängen. Dann erhalten wir eine Faktorisierung

$$B_{\mathcal{T}}(X) = I_{X,1}S_{X,1} \ldots I_{X,r_X}S_{X,r_X}$$

wobei $I_{X,i} \in \mathcal{I}(X)^*$ und $S_{X,i} \in \mathcal{S}(X)^*$ gilt für alle $i = 1, \ldots, r_X$ und weiterhin $I_{X,i} \neq \epsilon$ für $i = 2, \ldots, r_X$ und $S_{X,i} \neq \epsilon$ für $i = 1, \ldots, r_X - 1$.

Intuitiv bedeutet diese Faktorisierung der Folge $B_{\mathcal{T}}(X)$, dass die abgeleiteten Attribute an jedem Knoten eines Ableitungsbaums, der mit X markiert ist, durch maximal r_X Besuche berechnet werden können: bei der ersten Ankunft an dem Knoten vom Vaterknoten liegen die Werte der ererbten Attribute aus $I_{X,1}$ vor, während vor der Rückkehr zum Vater die Werte der abgeleiteten geleiteten Attribute aus $S_{X,1}$ ausgewertet werden. Entsprechend liegen beim i-ten Besuch des Knotens die Werte der ererbten Attribute aus $I_{X,1} \ldots I_{X,i}$ vor und die abgeleiteten Attribute aus $S_{X,i}$ werden berechnet. Eine Teilfolge $I_{X,i}S_{X,i}$ von $B_{\mathcal{T}}(X)$ nennen wir einen *Besuch* von X. Um zu bestimmen, welche Berechnung während des i-ten Besuchs an einem Knoten n und an den Nachfolgern des Knotens n passieren soll, betrachten wir die Abhängigkeitsrelation $D_{\mathcal{T}}(p)$ für die Produktion $X_0 \longrightarrow X_1 \ldots X_k$, die an n angewendet wird. Weil die Relation $D_{\mathcal{T}}(p)$ azyklisch ist, lässt sich zu $D_{\mathcal{T}}(p)$ eine lineare Anordnung finden. In unserem Fall wählen wir eine Anordnung $B_{\mathcal{T}}(p)$, die sich in *Besuche* faktorisieren lässt. Insgesamt erhalten wir so zu der Relation $D_{\mathcal{T}}(p)$ eine Besuchsreihenfolge

$$B_{\mathcal{T}}(p) = B_{\mathcal{T},1}(p) \ldots B_{\mathcal{T},r_{X_0}}(p)$$

Die i-te Teilfolge $B_{\mathcal{T},i}(p)$ beschreibt detailliert, was bei dem i-ten Besuch eines Knotens n, an dem die Produktion $p : X_0 \longrightarrow X_1 \ldots X_k$ angewendet wurde, passieren soll. Für jedes Vorkommen ererbter Attribute der X_j ($j > 0$) in der Teilfolge muss nacheinander das entsprechende Attributexemplar berechnet werden. Nach der Berechnung der aufgelisteten ererbten Attributexemplare des i'-ten Besuchs des j-ten Nachfolgers wird rekursiv dieser Nachfolger besucht, um die Werte für die abgeleiteten Attribute, die dem i'-ten Besuch zugeordnet sind, zu berechnen. Liegen die Werte der abgeleiteten Attribute aller Nachfolger vor, die mittelbar oder unmittelbar zur Berechnung der abgeleiteten Attribute des i-ten Besuchs der linken Seite X_0 erforderlich sind, werden deren Werte berechnet.

Um die Teilfolge $B_{\mathcal{T},i}(p)$ elegant beschreiben zu können, führen wir folgende Abkürzung ein. Sei $w = a_1 \ldots a_l$ eine Folge von Attributen des Nichtterminals X_j. Dann bezeichne $p[j].w = p[j].a_1 \ldots p[j].a_l$ die zugehörige Folge von Attributvorkommen in p. Der i'-te Besuch $I_{X_j,i'}S_{X_j,i'}$ des j-ten Symbols der Produktion p bezeichnen wir so durch die Folge $p[j].(I_{X_j,i'}\ S_{X_j,i'})$. Die Folge $B_{\mathcal{T},i}(p)$, aufgefasst als Folge von Attributvorkommen von p, hat dann die Form:

$$\begin{aligned} B_{\mathcal{T},i}(p) = \; & p[0].I_{X_0,i} \\ & \quad p[j_1].(I_{X_{j_1},i_1}\ S_{X_{j_1},i_1}) \\ & \quad \ldots \\ & \quad p[j_r].(I_{X_{j_r},i_r}\ S_{X_{j_r},i_r}) \\ & p[0].S_{X_0,i} \end{aligned}$$

für eine geeignete Folge von Paaren $(j_1, i_1), \ldots, (j_r, i_r)$. Sie besteht aus den Besuchen der Nichtterminalvorkommen $X_{j_1}, \ldots, X_{j_r}$ der rechten Seite der Produktion p, die in den i-ten Besuch der linken Seite von p eingebettet sind.

Sei p eine Produktion und $f(p[j_1].a_1, \ldots, p[j_r].a_r)$ die rechte Seite der semantischen Regel für das Attributvorkommen $p[j].a$ für eine totale Funktion f. Für einen Knoten n im Syntaxbaum, an dem die Produktion p angewandt wurde, definieren wir dann

$$\mathsf{eval}_{p,j,a}\ n = f(n[j_1].a_1, \ldots, n[j_r].a_r)$$

Die Funktionen $\mathsf{eval}_{p,j,a}$ verwenden wir, um aus der i-ten Teilfolge $B_{\mathcal{T},i}(p)$ der Produktion p eine Funktion $\mathsf{solve}_{p,i}$ zu generieren:

$$\begin{aligned} \mathsf{solve}_{p,i}\ n = \; & \textbf{forall}\ (a \in I_{X_{j_1},i_1}) \\ & \quad n[j_1].a \leftarrow \mathsf{eval}_{p,j_1,a}\ n; \\ & \mathsf{visit}_{i_1}\ n[j_1]; \\ & \ldots \\ & \textbf{forall}\ (a \in I_{X_{j_r},i_r}) \\ & \quad n[j_r].a \leftarrow \mathsf{eval}_{p,j_r,a}\ n; \\ & \mathsf{visit}_{i_r}\ n[j_r]; \\ & \textbf{forall}\ (a \in S_{X_0,i}) \\ & \quad n.a \leftarrow \mathsf{eval}_{p,0,a}\ n; \end{aligned}$$

Beispiel 4.4.6 (Fortführung von Beispiel 4.3.4) Die produktionslokale Abhängigkeitsrelation für die Produktion $block \rightarrow decl\ block$ erhalten wir aus der Relation in Abbildung 4.8, indem wir die totalen Ordnungen auf den Attributvorkommen zu den einzelnen Symbolen hinzufügen. Insgesamt lässt sich diese Relation in die folgende totale Ordnung einbetten:

$$\begin{aligned} & block[0].same \rightarrow block[0].env \rightarrow \\ & \quad decl.new \rightarrow \\ & \quad block[1].same \rightarrow block[1].env \rightarrow block[1].ok \rightarrow \\ & block[0].ok \end{aligned}$$

Gemäß dieser totalen Ordnung steigt der Auswerter zuerst in den Teilbaum für das Nichtterminal $decl$ ab, um den Wert des Attributs new zu berechnen. Damit können die ererbten Attribute des Nichtterminal $block$ der rechten Seite der Produktion berechnet werden. Ein Abstieg in den Teilbaum für dieses Nichtterminal erlaubt dann, den Wert des abgeleiteten Attributs ok dieses Nichtterminals zu bestimmen. Damit liegen alle Werte vor, um den Wert des abgeleiteten Attributs ok der linken Seite der Produktion zu berechnen. □

Die Auswertungsreihenfolgen in visit_i werden so gewählt, dass der Wert jedes Attributexemplars $n[j'].b$ bereits vor jedem lesenden Zugriff berechnet wurde. Die Funktionen $\mathsf{solve}_{p,i}$ sind simultan rekursiv untereinander und mit den Funktionen visit_i. Für einen Knoten n sei $\mathsf{get\_prod}\ n$ die Produktion, die an n angewandt wurde oder Null, falls n ein Blatt ist, das mit einem Terminalsymbol oder ϵ beschriftet ist. Ist $p_1, \ldots, p_m$ eine Anordnung der Produktionen der Grammatik, dann ist die Funktion visit_i gegeben durch:

$$
\begin{aligned}
\mathsf{visit}_i\ n \quad &= \mathbf{match}\ \mathsf{get\_prod}\ n \\
&\quad\ \mathbf{with}\ \mathsf{Null} \rightarrow\ () \\
&\quad\ \mid\quad p_1\ \rightarrow\ \mathsf{solve}_{p_1,i}\ n \\
&\qquad\qquad \ldots \\
&\quad\ \mid\quad p_m\ \rightarrow\ \mathsf{solve}_{p_m,i}\ n
\end{aligned}
$$

Für einen Knoten n überprüft die Funktion visit_i, ob n ein Blatt ist oder durch Anwendung einer Produktion erzeugt wurde. Ist der Knoten n des Syntaxbaums ein Blatt, braucht der Auswerter nichts zu tun, wenn wir annehmen, dass die abgeleiteten Attribute der Blätter vorgängig mit einem Wert initialisiert wurden. Dass n ein Blatt ist, erkennt der Auswerter daran, dass der Aufruf $\mathsf{get\_prod}\ n$ den Wert Null zurückliefert. Ist der Knoten n des Syntaxbaums kein Blatt, liefert der Aufruf $\mathsf{get\_prod}\ n$ die Produktion p_j, die am Knoten n im Syntaxbaum angewendet wurde. In diesem Fall wird die Funktion $\mathsf{solve}_{p_j,i}$ für den Knoten n aufgerufen.

Sei S das Startsymbol der kontextfreien Grammatik, das der Attributgrammatik zu Grunde liegt. Besitzt S keine ererbten Attribute, dann besteht $B_{\mathcal{T}}(X)$ aus einer Anordnung alleine von abgeleiteten Attributen, die in einem einzigen Besuch berechnet werden. Die Auswertung sämtlicher Attributexemplare in einem Syntaxbaum t mit Wurzel n_0 für das Startsymbol S wird dann durch den Aufruf $\mathsf{visit}_1\ n_0$ erledigt.

Der Auswerter, den wir hier vorgestellt haben, lässt sich aus der Attributgrammatik zusammen mit den totalen Ordnungen $\mathcal{T}(X), X \in V$, in polynomieller Zeit generieren. Nicht jede Attributgrammatik besitzt jedoch ein solches System kompatibler totaler Ordnungen. Die Frage, ob eine beliebige Attributgrammatik l-attributiert ist, ist sicherlich in NP, da geeignete totale Ordnungen $\mathcal{T}(X), X \in V$, in polynonieller Zeit *geraten* und dann auf Kompatibilität geprüft werden können. Ein wesentlich besserer Algorithmus ist allerdings nicht bekannt: das Problem liegt nicht nur in NP, sondern ist sogar NP-vollständig.

Praktisch gesehen, wird deshalb meist nur eine Unterklasse der l-geordneten Attributgrammatiken betrachtet, bei denen ein bestimmtes einfaches Verfahren kompatible totale Ordnungen $\mathcal{T}(X), X \in V$, liefert. Ausgangspunkt für die Konstruktion ist das Gleichungssystem:

$$
\begin{aligned}
(\mathcal{R}') \qquad \mathcal{R}'(X) \ = \bigsqcup \{ & \pi_i((Dp(p) \cup \mathcal{R}'(X_0)[p,0] \cup \ldots \cup \mathcal{R}'(X_k)[p,k])^+) \mid \\
& p:\ X_0 \rightarrow X_1 \ldots X_k \in P, \quad X = X_i \} \ , \qquad X \in V
\end{aligned}
$$

über den *transitiven* Relationen auf Attributen, geordnet durch die Teilmengenbeziehung $\subseteq$. Wir erinnern uns, dass die kleinste obere Schranke transitiver Relationen $R \in \mathcal{S}$ gegeben ist durch:

$$\bigsqcup \mathcal{S} = (\bigcup \mathcal{S})^+$$

Die kleinste Lösung des Gleichungssystems $(\mathcal{R}')$ existiert, weil die Operationen auf den rechten Seiten der Gleichungen monoton sind. Die kleinste Lösung lässt sich mit dem iterativen Verfahren berechnen, das wir in Kapitel 3.2.5 etwa zur Berechnung von first_k-Mengen eingesetzt haben, weil die Anzahl möglicher transitiver Relationen endlich ist.

Sei $\mathcal{R}'(X)$, für $X \in V$, die kleinste Lösung des Gleichungssystems. Weil auch jedes System $\mathcal{T}(X), X \in V$, kompatibler *totaler* Ordnungen eine Lösung des Gleichungssystems $(\mathcal{R}')$ darstellt, gilt $\mathcal{R}'(X) \subseteq \mathcal{T}(X)$ für alle Symbole $X \in V$. Existiert also ein solches System $\mathcal{T}(X), X \in V$, kompatibler totaler Ordnungen, dann müssen die Relationen $\mathcal{R}'(X)$ sämtlich azyklisch sein. Die Relationen $\mathcal{R}'(X)$ sind deshalb ein guter Startpunkt, um totale Ordnungen $\mathcal{T}(X)$ zu konstruieren.

Diese Konstruktion wird so durchgeführt, dass sich für jedes X eine Anordnung mit minimal vielen Besuchen ergibt. Für ein Symbol X mit $\mathcal{A}(X) \neq \emptyset$ wird eine Folge $I_1 S_1 \ldots I_r S_r$ berechnet, wobei I_i und S_i jeweils Folgen von ererbten bzw. abgeleiteten Attributen sind. Alle bereits angeordneten Attribute sammeln wir in einer Menge D. Am Anfang setzen wir D auf die leere Menge. Nehmen wir an, $I_1, S_1, \ldots I_{i-1}, S_{i-1}$ seien bereits berechnet und D enthalte alle Attribute, die in diesen Folgen vorkommen. Dann werden zwei Schritte durchgeführt.

1. Als erstes wird eine maximal große Menge *ererbter* Attribute von X bestimmt, die nicht in D sind und nur voneinander oder von Attributen aus D abhängen. Diese Menge wird topologisch sortiert. Das liefert die Folge I_i. Anschließend wird die Menge zu D hinzu gefügt.

2. Nun wird eine maximal große Menge *abgeleiteter* Attribute bestimmt, die nicht in D sind und nur voneinander oder Attributen in D abhängen. Diese Menge wird ebenfalls zu D hinzu gefügt und eine topologische Sortierung als S_i ausgegeben.

Mit diesem Verfahren werden so viele Teilfolgen $I_i S_i$ konstruiert, bis alle Attribute angeordnet sind, d.h. bis D gleich der Gesamtmenge $\mathcal{A}(X)$ der Attribute des Nichtterminals X ist.

Seien $\mathcal{T}'(X), X \in V$, die totalen Ordnungen auf den Attributen der Symbole von X, die auf diese Weise berechnet werden. Dann nennen wir die Attributgrammatik *geordnet*, wenn die totalen Ordnungen $\mathcal{T}'(X), X \in V$, bereits kompatibel sind, d.h. ebenfalls das Gleichungssystem $(\mathcal{R}')$ erfüllen. Bei dieser Vorgehensweise wurden die Relationen $\mathcal{R}'(X)$ *isoliert* zu totalen Ordnungen ergänzt, ohne zu berücksichtigen, ob die hinzugefügten künstlichen Abhängigkeiten mit den Produktionen Zyklen verursachen. Die polynomielle Komplexität der Konstruktion wurde deshalb mit einer Einschränkung behandelbarer Attributgrammatiken erkauft.

In unseren Beispielen 4.3.4, 4.3.3 und 4.3.5 aus Abschnitt 4.3.2 werden durch unser Verfahren Attributauswerter generiert, die jeden Knoten eines Syntaxbaums genau einmal besuchen. Nicht alle praktisch interessanten Attributgrammatiken sind so einfach auszuwerten. Eine Attributgrammatik zur Berechnung der Symboltabelle für JAVA muss z.B. den Rumpf einer Klasse mehrmals besichtigen. Das liegt daran, dass in JAVA für Methoden keine *forward*-Deklaration erforderlich ist wie etwa in C für Funktionen: eine JAVA-Methode, die später deklariert wird, kann trotzdem im Rumpf einer früher deklarierten Methode aufgerufen werden. Ein ähnliches Problem tritt bei funktionalen Sprachen wie OCAML auf, wenn wechselseitig rekursive Funktionen eingeführt werden (siehe Aufg. **??**).

4.4.4 Parsergesteuerte Attributauswertung

In diesem Abschnitt betrachten wir einige stark eingeschränkte, aber praktisch nützliche Attributgrammatikklassen. Bei ihnen können die Attribute während der Syntaxanalyse gesteuert durch den Parser ausgewertet werden. Die Attributwerte werden jeweils kellerartig verwaltet, sei es auf einem eigenen Attributkeller, sei es zusammen mit den Parserzuständen und Grammatiksymbolen auf dem Syntaxkeller. Der Aufbau des Syntaxbaums, zumindest zum Zwecke einer späteren Attributauswertung, ist überflüssig. Solche Verfahren sind deshalb zur Implementierung besonders effizienter Übersetzer interessant. Da die Attributauswertung parsergesteuert erfolgen soll, müssen die Werte von abgeleiteten Attributen an Terminalsymbolen direkt vom Scanner geliefert werden.

L-attributierte Grammatiken

Alle Parser, die wir zur Steuerung der Attributauswertung betrachten, verarbeiten ihre Eingabe von links nach rechts. Deshalb ist die als erstes eingeführte Grammatikklasse eine Oberklasse aller parsergesteuert auswertbaren Attributgrammatikklassen. Sie enthält alle Attributgrammatiken in Normalform, bei denen alle Attribute in *einem* links-rechts-Tiefendurchlauf über den Syntaxbaum ausgewertet werden können. Formal nennen wir eine Attributgrammatik *L-attributiert* (oder kurz: eine $L-AG$), wenn für jede Produktion $p: \ X_0 \to X_1 \ldots X_k$ der zugrunde liegenden Grammatik das Vorkommen $p[j].b$ eines ererbten Attributs nur von Attributvorkommen $p[i].a$ abhängt mit $i < j$. Die Attributauswertung in einem links-rechts-Tiefendurchlauf erfolgt mit dem Algorithmus aus Abschnitt 4.4.3, der jeden Knoten im Syntaxbaum nur einmal und die Kinder eines Knotens jeweils in einer festen links-rechts-Reihenfolge besucht. Für eine Produktion $p: \ X_0 \to X_1 \ldots X_k$ generieren wir eine Funktion solve_p, die definiert ist durch:

$$
\begin{aligned}
\mathsf{solve}_p\ n = \quad & \textbf{forall}\ (a \in I_{X_1}) \\
& \quad n[1].a \leftarrow \mathsf{eval}_{p,1,a}\ n; \\
& \mathsf{visit}\ n[1]; \\
& \ldots \\
& \textbf{forall}\ (a \in I_{X_k}) \\
& \quad n[k].a \leftarrow \mathsf{eval}_{p,k,a}\ n; \\
& \mathsf{visit}\ n[k]; \\
& \textbf{forall}\ (a \in S_{X_0}) \\
& \quad n.a \leftarrow \mathsf{eval}_{p,0,a}\ n;
\end{aligned}
$$

Dabei sind I_X und S_X jeweils die Mengen der ererbten bzw. abgeleiteten Attribute des Symbols X, und der Ausdruck $\mathsf{eval}_{p,j,a}\ n$ liefert den Wert der rechten Seite der semantischen Regel für das Attributexemplar $n[j].a$. Der Besuch eines Knoten n wird durch die Funktion visit realisiert:

$$
\begin{aligned}
\mathsf{visit}\ n \quad = \ & \textbf{match}\ \mathsf{get\_prod}\ n \\
& \textbf{with}\ \mathsf{Null} \rightarrow () \\
& \quad \mid \quad p_1 \ \rightarrow \mathsf{solve}_{p_1}\ n \\
& \qquad\qquad \ldots \\
& \quad \mid \quad p_m \ \rightarrow \mathsf{solve}_{p_m}\ n
\end{aligned}
$$

Hier liefert erneut die Hilfsfunktion $\mathsf{get\_prod}\ n$ die Produktion, die am Knoten n angewandt wurde (oder Null, falls n ein Blatt ist). Die Attributgrammatiken AG_{scopes}, AG_{types} und AG_{bool} aus den Beispielen 4.3.4, 4.3.3 und 4.3.5 sind alle L-attributiert, wobei die letzte allerdings nicht in Normalform ist.

LL-attributierte Grammatiken

Betrachten wir die notwendigen Aktionen für eine parsergesteuerte Attributauswertung:

- Bei Lesen eines Terminalsymbols a Übernehmen der abgeleiteten Attribute von a aus dem Scanner;
- Bei Beginn der Analyse eines Worts für X Auswertung der ererbten Attribute von X;
- Bei Beendigung der Analyse des Worts für X Auswertung der abgeleiteten Attribute von X;

Ein $LL(k)$-Parser, wie er im Kapitel Syntaktische Analyse beschrieben ist, kann diese Aktionen jeweils bei Expansion, bei Reduktion bzw. bei Lesen eines Terminals anstoßen. Eine Attributgrammatik in Normalform nennen wir deshalb *LL-attributiert*, wenn

- sie L-attributiert ist, und
- die zugrundeliegende kontextfreie Grammatik eine starke $LL(k)$-Grammatik ist (für irgend ein $k \geq 1$).

Die LL-Attributiertheit einer Attributgrammatik besagt, dass die syntaktische Analyse mithilfe eines LL-Parsers geschehen kann, und dass, wann immer der LL-Parser die Expansion eines Nichtterminals vornimmt, alle Argumente für seine ererbten Attribute berechnet sind, genauer gesagt, berechnet sein können.

In Kapitel 3.3 konstruierten wir zu einer starken $LL(k)$-Grammatik einen Parser, indem wir den Item-Kellerautomaten zu der Grammatik mit k-Vorausschau ausstatteten. Auf seinem Keller verwaltet dieser Parser Items $[A \rightarrow \alpha.\beta]$, wobei der Punkt kennzeichnet, dass der Teil der Eingabe, der aus α abgeleitet wurde, bereits verarbeitet wurde. Diesen Kellerautomaten erweitern wir, so dass er zu jedem Item ι zu einer Produktion $p: \ X_0 \rightarrow X_1 \ldots X_k$ eine Struktur $S(\iota)$ verwaltet, in welcher die ererbten Attribute der linken Seite X_0 sowie die abgeleiteten Attribute der Symbole $X_1, \ldots, X_k$ der rechten Seite abgelegt werden. Steht der Punkt in dem Item ι vor dem i-ten Symbol, liegen die Werte der ererbten Attribute von X_0 sowie die Werte der abgeleiteten Attribute der Symbole $X_1, \ldots, X_{i-1}$ in $S(\iota)$ bereits vor.

Abbildung 4.15 visualisiert die Aktionen der LL-Parser-gesteuerten Attributauswertung. Nehmen wir an, das Item ι gehöre zu der Produktion $p: \ X_0 \to X_1 \ldots X_k$ und der Punkt befinde sich hinter dem $(i-1)$-ten Symbol.

- Bei einem *shift*-Übergang für ι zu einem Terminalsymbol $X_i = a$ wird in ι der Punkt über das Symbol X_i weitergerückt. Die Attributstruktur zu dem modfizierten Item ergibt sich aus $S(\iota)$, indem die abgeleiteten Attribute, die der Scanner für a lieferte, in $S(\iota)$ eingetragen werden.
- Bei einem *expand*-Übergang für ι, der das Symbol X_i durch eine Produktion $p': \ X_i \to \gamma$ expandiert, wird das Item $[X_i \to .\gamma]$ gekellert. Für dieses Item ι' wird eine neue Struktur $S(\iota')$ angelegt, in der jedes ererbte Attribut b der linken Seite X_i mit Hilfe der semantischen Regel der Produktion p für das Attributvorkommen $p[i].b$ berechnet wird.
- Bei einem *reduce*-Übergang befindet sich auf dem Keller über dem Item ι ein vollständiges Item $\iota' = [X_i \to \gamma.]$ zu einer Produktion p' zusammen mit seiner zugehörigen Attributstruktur $S(\iota')$. Nun werden die abgeleiteten Attribute der linken Seite X_i mithilfe der semantischen Regeln der Produktion p' berechnet und in der darunter liegenden Attributstruktur $S(\iota)$ für X_i eingetragen. Dann wird das vollständige Item ι' zusammen mit seiner Attributstruktur vom Keller entfernt und der Punkt in ι um eine Position versetzt.

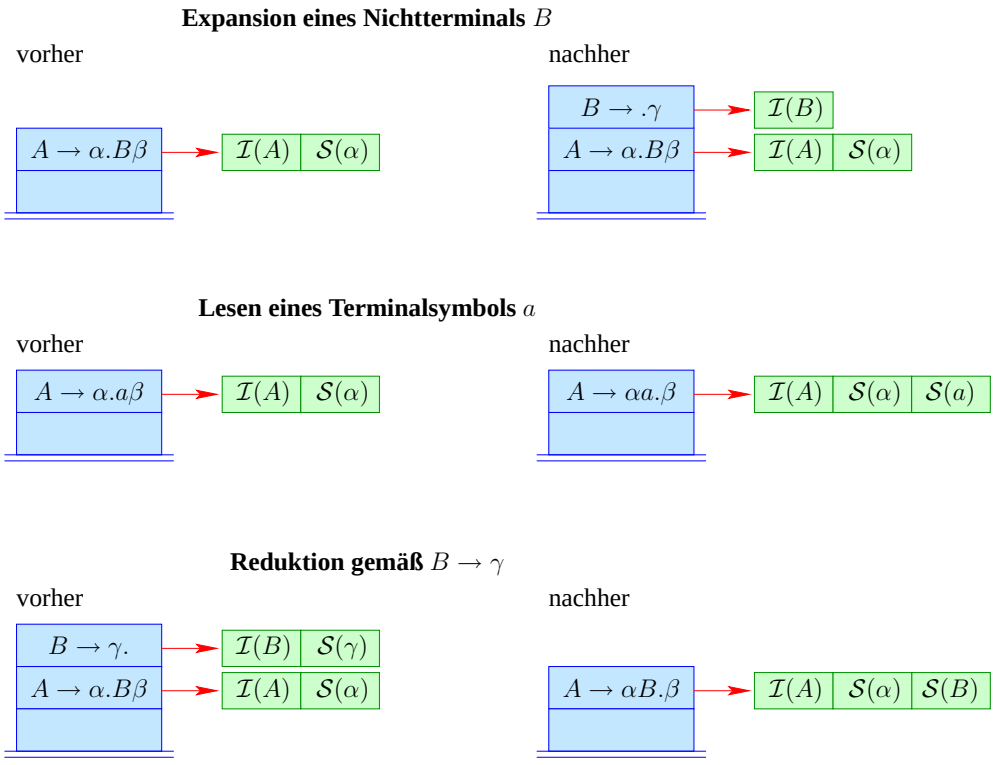


Abb. 4.15. Aktionen einer LL-Parser-gesteuerten Attributauswertung. Dabei bezeichnen $\mathcal{I}(A)$ bzw. $\mathcal{S}(\alpha)$ die Folgen der Werte der ererbten Attribute eines Symbols A bzw. der abgeleiteten Attribute der Symbole in α.

Obwohl die Attributgrammatiken AG_{types} und AG_{bool} L-attributiert sind, sind beide nicht LL-attributiert. In beiden Fällen ist die zugrundeliegende kontextfreie Grammatik linksrekursiv und deshalb nicht $LL(k)$ für irgend ein k. Im Falle der Attributgrammatik AG_{bool} kann man sogar zeigen, dass es keine LL-attributierte Grammatik gibt, die das Codegenerierungsproblem auf diese Weise, d.h. durch Propagation von zwei Sprungzielen an jeden Teilausdruck löst.

LR-attributierte Grammatiken

Jetzt entwickeln wir ein Verfahren, wie ein LR-Parser die Auswertung von Attributen steuern kann. Ein LR-Parser verwaltet auf seinem Keller Zustände. Jeder Zustand, der verschieden vom Endzustand f ist, besteht aus einer Menge von Items, gegebenenfalls dekoriert mit Vorausschaumengen. Jeden solchen Zustand q statten wir mit einer Attributstruktur $\mathcal{S}(q)$ aus. Die Attributstruktur des Anfangszustands ist leer. Für jeden anderen Zustand $q \notin \{q_0, f\}$, der von einem Symbol X erreicht wird, enthält $\mathcal{S}(q)$ die Werte für die abgeleiteten Attribute des Symbols X. Zusätzlich statten wir den LR-Parser mit einer globalen Attributstruktur $\mathcal{I}$ aus, die für jedes ererbte Attribut b den zuletzt berechneten Wert bereit hält bzw. $\perp$, falls der Wert des Attributs b nicht zur Verfügung steht. Am Anfang enthält die globale Attributstruktur $\mathcal{I}$ nur die Werte der ererbten Attribute des Startsymbols.

Die Werte der abgeleiteten Attribute eines Terminalsymbols werden vom Scanner geliefert. Zwei Probleme müssen gelöst werden, wenn wir die Attributwerte für die Attributstruktur $\mathcal{S}(q)$ eines Zustands q berechnen wollen:

- Wir müssen die semantische Regel identifizieren, mit der der Attributwert berechnet werden soll.
- Wir müssen die Werte für die Attributvorkommen in der semantischen Regel im Keller auffinden.

Die Werte der abgeleiteten Attribute eines Nichterminals X_0 können wir berechnen, wenn der LR-Parser einen Reduktionsübergang macht: dann ist die Produktion $p : \quad X_0 \rightarrow X_1 \ldots X_k$ bekannt, mit der X_0 abgeleitet wurde. Zur Berechnung eines abgeleiteten Attributs b von X_0 kommt deshalb die semantische Regel für das Attributvorkommen $p[0].b$ dieser Produktion zum Einsatz. Zu dem Zeitpunkt der Reduktion liegt eine Folge $q' q_1, \ldots, q_k$ von Zuständen oben auf dem Keller, wobei $q_1, \ldots, q_k$ durch die Symbole $X_1, \ldots, X_k$ der rechten Seite von p erreicht werden. Nehmen wir an, die Werte für die Attributstrukturen $\mathcal{S}(q_1), \ldots, \mathcal{S}(q_k)$ seien bereits berechnet. Dann kann die semantische Regel für ein abgeleitetes Attribut von X_0 ausgewertet werden, indem die Werte für die Vorkommen $p[0].b$ ererbter Attribute der linken Seite X_0 in $\mathcal{I}$ und die Werte für Vorkommen $p[j].b$ abgeleiteter Attribute von X_j der rechten Seite in $\mathcal{S}(q_j)$ nachgeschlagen werden. Damit können wir vor dem *reduce*-Übergang auch für den Zustand $q = \delta(q', X_0)$, der damit von X_0 erreicht wird, die Werte der abgeleiteten Attribute ausrechnen. Ungelöst bleibt dabei die Frage, wie die aktuellen Werte der ererbten Attribute von X_0 bestimmt werden können.

Für den Fall, dass es *keine* ererbte Attribute gibt, haben wir damit bereits ein Verfahren zur Attributauswertung in der Hand. Eine Attributgrammatik nennen wir *S-attributiert*, wenn sie *nur* abgeleitete Attribute besitzt. Beispiel 4.3.1 ist eine solche Grammatik, mit der die Werte von Ausdrücken berechnet werden können. Allgemeiner kann die Spezifikation der Berechnung eines semantischen Werts, die von Anaysegeneratoren wie YACC oder BISON für verschiedene Programmiersprachen angeboten werden, als S-Grammatik aufgefasst werden. Jede S-attributierte Grammatik ist auch L-attributiert. Ist eine LR-Grammatik S-attributiert, lassen sich die Attributstrukturen der Zustände im Keller und damit insbesondere die Werte für die Exemplare der abgeleiteten Attribute des Startsymbols berechnen.

Abgeleitete Attribute allein sind jedoch für etwas anspruchsvollere Aufgaben nicht ausdrucksstark genug. Bereits die Berechnung der Typen eines Ausdrucks relativ zu einer Symboltabelle *env* aus Beispiel 4.3.3 erfordert ein ererbtes Attribut, das nach unten im Syntaxbaum weitergereicht wird. Unser Ziel ist darum, den Ansatz für S-attributierte Grammatiken um Techniken zum Umgang mit ererbten Attributen zu erweitern. Im Allgemeinen ist dem LR-Parser jedoch das obere Baumfragment, in dem die Transportwege für ererbte Attributwerte liegen, noch nicht bekannt. Insbesondere wenn eine Grammatik linksrekursiv ist, kann die Anwendung beliebig vieler semantischer Regeln erforderlich sein, um den Wert eines ererbten Attributs zu berechnen. Hier kommt uns zur Hilfe, dass die Werte ererbter Attribute sehr oft unverändert im Syntaxbaum von oben nach unten weitergereicht werden. Das ist zum Beispiel bei der Attributgrammatik AG_{types} aus Beispiel 4.3.3 zur Berechnung des Typs eines Ausdrucks der Fall, bei welcher der Wert des Attributs env von der linken Seite der Produktionen in Attribute gleichen Namens der Nichtterminalvorkommen der rechten Seite kopiert wird. Das beobachten wir auch bei der Produktion $block \rightarrow stat \;\; block$ der Attributgrammatik AG_{scopes} aus Beispiel 4.3.4, bei der das ererbte Attribut $same$ der linken Seite in ein gleichnamiges Attribut des Nichtterminalvorkommens $block$ der rechten Seite und das ererbte Attribut env der linken Seite in gleichnamige Attribute beider Nichtterminalvorkommen der rechten Seite kopiert werden.

Formal nennen wir ein Vorkommen $p[j].b$ eines ererbten Attributs b am j-ten Symbol einer Produktion $P: \ X_0 \to X_1 \ldots X_k$ *kopierend*, wenn es ein $i < j$ gibt, so dass gilt:

1. $p[j].b = p[i].b$, und
2. $p[i].b$ ist das letzte Vorkommen des Attributs b vor $p[j].b$, d.h. $b \notin \mathcal{A}(X_{i'})$ für alle $i < i' < j$.

In diesem Sinne sind sämtliche Vorkommen des ererbten Attributs env auf rechten Seiten der Attributgrammatik AG_{types} kopierend. Das Gleiche gilt für die Vorkommen der ererbten Attribute $same$ und env der Attributgrammatik AG_{scopes} in der Produktion $block \to stat \ block$.

Nehmen wir für einen Moment an, sämtliche Vorkommen von ererbten Attributen in rechten Seiten seien kopierend. Dann ändern sich die Werte der ererbten Attribute nie. Enthält folglich die globale Attributstruktur $\mathcal{I}$ einmal den richtigen Wert für ein ererbtes Attribut, muss dieser nie wieder verändert werden.

Nicht alle Vorkommen ererbter Attribute einer L-attributierten Grammatik sind jedoch kopierend. Für ein nicht kopierendes Vorkommen $p[j].b$ eines ererbten Attributs b muss der Attributauswerter die Produktion $p: \ X_0 \to X_1 \ldots X_k$ und die Position j in der rechten Seite von p kennen, um die richtige semantische Regel für das Attributvorkommen auswählen zu können. Dazu verwenden wir einen Trick. Wir führen ein neues Nichtterminal $N_{p,j}$ mit der einzigen Regel $N_{p,j} \to \epsilon$ ein. Dieses Nichtterminal $N_{p,j}$ wird vor dem Symbol X_j in die rechte Seite von p eingefügt. Das Nichtterminalsymbol $N_{p,j}$ statten wir mit allen ererbten Attributen b von X_j aus, die in p nicht kopierend sind. Jedes Attribut b von $N_{p,j}$ statten wir mit einer semantischen Regel aus, die den gleichen Wert berechnet wie vorher die semantische Regel für $p[j].b$. Beachten Sie, dass durch das Einfügen von Hilfssymbolen $N_{p,j_1}, \ldots, N_{p,j_r}$ sich die Nummerierung der ursprünglichen Symbolvorkommen der Produktion p in der transformierten Seite gegebenenfalls verschiebt.

Beispiel 4.4.7 Betrachten wir die Produktion $block \to decl \ \ block$ der Attributgrammatik AG_{scopes} aus Beispiel 4.3.4. Die Attributvorkommen $block[1].same$ und $block[1].env$ auf der rechten Seite der Produktion sind nicht kopierend. Deshalb fügen wir vor dem Nichtterminal $block$ der rechten Seite ein neues Nichtterminal N ein:

$$\begin{aligned} block &\longrightarrow decl \ N \ block \\ N \quad &\longrightarrow \epsilon \end{aligned}$$

Das neue Nichtterminalsymbol N erhält die Menge $\{same, env\}$ als Menge der ererbten Attribute. Abgeleitete Attribute benötigt es dagegen nicht. Die neuen semantischen Regeln für die transformierte Produktion sind:

$$\begin{aligned} N.same &= \textbf{let } (x,\tau) = decl.new \\ &\quad \textbf{in } block[0].same \cup \{x\} \\ N.env &= \textbf{let } (x,\tau) = decl.new \\ &\quad \textbf{in } block[0].env \oplus \{x \mapsto \tau\} \\ block[1].same &= N.same \\ block[1].env &= N.env \\ block[1].ok &= \textbf{let } (x,\tau) = decl.new \\ &\quad \textbf{in if } x \notin block[0].same \\ &\quad\quad \textbf{then } block[1].ok \\ &\quad\quad \textbf{else } \mathsf{false} \end{aligned}$$

Da N nur ererbte Attribute besitzt, benötigt es keine eigenen semantischen Regeln. Wir bemerken, dass die ererbten Attribute $same$ und env des Nichtterminals $block$ nach der Transformation beide kopierend sind. □

Durch das Einfügen der Nichtterminale $N_{p,j}$ wird die Menge der akzeptierten Wörter nicht verändert. Möglicherweise geht jedoch die $LR(k)$-Eigenschaft verloren. In unserem Beispiel Beispiel 4.4.7 ist dies nicht der Fall. Ist die zu Grunde liegende kontextfreie Grammatik nach der Transformation immer noch eine $LR(k)$-Grammatik, nennen wir die Attributgrammatik *LR-attributiert*.

Nach der Transformation sind die einzigen nicht-kopierenden ererbten Attributvorkommen diejenigen an den neu eingefügten Nichtterminalen $N_{p,j}$. Bei einem *reduce*-Übergang für $N_{p,j}$ hat der LR-Parser die Produktion p und und die Position j in der rechten Seite von p identifiziert, an der er sich gerade befindet. Bei der Reduktion wird der neue Wert für das ererbte Attribut b berechnet in der globalen Attributstruktur $\mathcal{I}$ abgelegt. Die Zustände q', in die man durch einen Übergang unter dem Nichtterminal $N_{p,j}$ gelangen kann, statten wir mit einer Attributstruktur $old(q')$ aus. Diese Attributstruktur enthält jedoch nicht Werte für abgeleitete Attribute. Stattdessen werden in ihr die *vorherigen* Werte der bei der Reduktion *überschriebenen* ererbten Attribute aus $\mathcal{I}$ abgelegt. Diese vorherigen Werte sind erforderlich, um die ursprünglichen Werte der ererbten Attribute vor dem Abstieg in den Syntaxbaum für X zu rekonstruieren.

Betrachten wir genauer, wie der Wert eines ererbten Attributs b des Nichtterminals $N_{p,j}$ berechnet werden kann. Sei $\bar{p}: \; X \to \alpha.N_{p,j}\beta$ die Produktion, die durch die Transformation aus p entstanden ist, wobei α die Länge m hat. Vor dem *reduce*-Übergang für $N_{p,j}$ liegt oben auf dem Keller eine Folge $q'q_1 \ldots q_m$, wobei $q_1, \ldots, q_m$ der Folge der Symbole in α entspricht. Die Auswertung der semantischen Regel für das ererbte Attribut b von $N_{p,j}$ findet die Werte der abgeleiteten Attribute der Symbole in α in den Attributstrukturen der Zustände $q_1, \ldots, q_m$. Den Wert eines ererbten Attributs a der linken Seite X kann der Attributauswerter in der globalen Struktur $\mathcal{I}$ nachschlagen, falls das Attribut a durch kein $N_{p,i}$ mit $i < j$ in der bisherigen Abarbeitung der rechten Seite von $\bar{p}$ neu definiert wurde. Falls das jedoch der Fall war, kann der Wert von a in der Struktur $old(q_{i'})$ zu dem Zustand $q_{i'}$ nachgeschlagen werden, welcher der ersten Neudefinition von a in der rechten Seite von p entspricht.

Betrachten wir im Detail, was bei dem *reduce*-Übergang für eine transformierte Produktion $\bar{p}$ passieren muss. Sei $N_{p,j_1}, \ldots, N_{p,j_r}$ die Folge der neuen Nichtterminale, die bei der Transformation in die rechte Seite der Produktion p eingefügt wurden, und sei m die Länge der transformierten rechten Seite. Vor dem *reduce*-Übergang befindet sich oben auf dem Keller eine Folge $q'q_1 \ldots q_m$ von Zuständen, wobei die Zustände $q_{j_1}, \ldots, q_{j_r+r-1}$ den Vorkommen der Nichtterminale $N_{p,j_1}, \ldots, N_{p,j_r}$ entsprechen. Mit Hilfe der zugehörigen Attributstrukturen $old(q_{j_1}), \ldots, old(q_{j_r+r-1})$ wird die Belegung der ererbten Attribute vor dem Abstieg in den Syntaxbaum für X rekonstruiert. Kommt ein Attribut b in keiner der Strukturen $old(q_{j_i+i-1})$ vor, enthält $\mathcal{I}$ bereits den aktuellen Wert von b. Andernfalls setzen wir den Wert von b auf den Wert von b in der ersten Struktur $old(q_{j_i+i-1})$, in der b vorkommt. Diese Rekonstruktion der globalen Datenstruktur $\mathcal{I}$ für die ererbten Attribute veranschaulicht Abbildung 4.16. Ist die Attributstruktur $\mathcal{I}$ vor Abarbeitung der rechten Seite der Produktion $\bar{p}$ wiederhergestellt, können die semantischen Regeln zur Berechnung der abgeleiteten Attribute der linken Seite X ausgewertet werden. Ein dafür benötigtes abgeleitetes Attribut des i-ten Symbolvorkommens der rechten Seite von $\bar{p}$ kann in der Attributstruktur zu q_i nachgeschlagen werden.

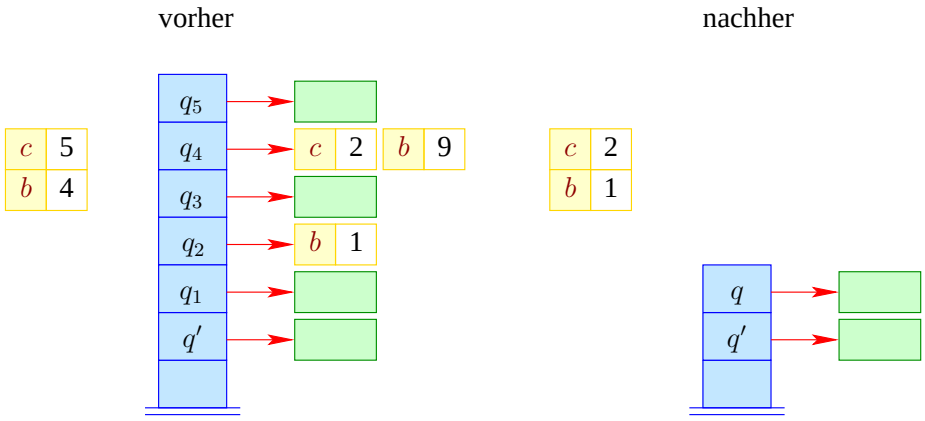


Abb. 4.16. Die Rekonstruktion der ererbten Attribute bei einem *reduce*-Übergang für eine Produktion $X \to \gamma$ mit $|\gamma| = 5$ und $\delta(q', X) = q$. Die Attributstrukturen $old(q_2)$ und $old(q_4)$ enthalten die überschriebenen Werte der ererbten Attribute b und c aus $\mathcal{I}$.

Damit haben wir ein Verfahren an der Hand, um einen LR-Parser so zu erweitern, dass er für LR-attributierte Grammatiken nicht nur die abgeleiteten Attribute zu jedem Zustand im Keller, sondern auch die jeweils benötigten ererbten Attribute berechnen kann.

Beispiel 4.4.8 Die Attributgrammatik *BoolExp* aus Beispiel 4.3.5 ist L-attributiert, aber weder LL- noch LR-attributiert. In der linksrekursiven Produktion für das Nichtterminal E muss ein neues ε-Nichtterminal an den Anfang der rechten Seite gesetzt werden, weil das ererbte Attribut $fsucc$ des Nichtterminals E der rechten Seite nicht kopierend ist. Entsprechend muss in der linksrekursiven Produktion für das Nichtterminal T ebenfalls ein neues ε-Nichtterminal an den Anfang der rechten Seite gesetzt werden, weil hier das ererbte Attribut $tsucc$ des Nichtterminals T nicht kopierend ist. Damit ist die transformierte Grammatik nicht mehr $LR(k)$ für irgend ein $k \geq 0$. □

4.5 Übungen

1. **1.1**
   Wie ist der Inhalt der Symboltabelle im Rumpf der Prozedur q hinter der Deklaration der Prozedur r in Beispiel 4.1.6?
2. **1.2**
   Gegeben seien die folgenden Operatoren:
   $+$:*int* $\longrightarrow$ *int*
   $+$:*real* $\longrightarrow$ *int*
   $+$:*int* $\times$ *int* $\longrightarrow$ *int*
   $+$:*real* $\times$ *real* $\longrightarrow$ *real*
   $/$:*int* $\times$ *int* $\longrightarrow$ *int*
   $/$:*int* $\times$ *int* $\longrightarrow$ *real*
   $/$:*real* $\times$ *real* $\longrightarrow$ *real*
   Benutzen Sie den Algorithmus aus Abschnitt 4.1.3, um die Überladung in der Zuweisung $A := 1/2 + 3/4$ an die *real*-Variable A aufzulösen.
3. **1.3**
   Leiten Sie mit Hilfe der Inferenzregeln den Typ des folgenden LaMa-Ausdrucks her:
   letrec *length* $=$ λl.**if** *null* l **then** 0 **else** *succ* $($*length* $(tl\ l))$
4. **1.4**
   Modifizieren Sie den in Abschnitt 4.1.2 angegebenen Algorithmus zur Identifizierung von Bezeichnern so, dass er (a) die Pascal-Gültigkeitsregeln, (b) die Algol 60-Gültigkeitsregeln realisiert.

2.1: Gegeben sei die folgende Grammatik:

P $\longrightarrow$ *B*
B $\longrightarrow$ **begin** *Ds*; *Sts* **end**
Ds $\longrightarrow$ *Ds*; *D* | *D*
Sts $\longrightarrow$ *Sts*; *St* | *St*
D $\longrightarrow$ **id**: *T*
T $\longrightarrow$ *int* | **list** T | **id** | **type** T
St $\longrightarrow$ **id** | *B*

Geben Sie Attributierungen an, welche die Identifizierung von Bezeichnern für Algol- und Pascal-Gültigkeitsregeln beschreiben. Organisieren Sie die Symboltabellen als lineare Listen, an welche neue Einträge durch die Operation *conc* vorne angefügt werden.

5. **2.2**
   Formulieren Sie den im Abschnitt 4.1.3 beschriebenen Algorithmus zur Auflösung von Überladungen als Attributgrammatik.