Typisierte Merkmalstrukturen und HPSG. Eine Erweiterung von UBS in SEPIA.

Diplomarbeit von Frieder Stolzenburg
Betreut von Martin Volk, M. Sc., und Prof. Dr. Ulrich Furbach
Universität Koblenz-Landau

November 1992

Inhaltsverzeichnis

1 Theoretische Grundlagen **1**
1.1 Der Inhalt dieser Arbeit 1
1.1.1 Motivation 1
1.1.2 Zielsetzungen und Aufgabenstellung 1
1.1.3 Überblick über den Rest der Arbeit 3
1.2 Grundlegende Begriffe 4
1.2.1 Typisierte Merkmalstrukturen 4
1.2.2 Darstellung von Merkmalstrukturen 5
1.2.3 Subsumption und Äquivalenz 7
1.3 Die Einbettung von Termen 8
1.3.1 Terme und Listen 8
1.3.2 Angemessenheit von Attributen 12
1.3.3 Gleichheit und Identität 13
1.4 Merkmalterme und Operationen 15
1.4.1 Die Konstruktion eines Verbandes 15
1.4.2 Negation und Implikation 17
1.4.3 Merkmalterme und ihre Eigenschaften 19
1.5 Logikprogramme und Hornklausellogik 21
1.5.1 Logische Formeln und Programme 21
1.5.2 Ableitungen in Logikprogrammen 21
1.5.3 Die kürzeste Definition von UBS 23
1.6 Anwendungen in HPSG 23
1.6.1 Die Bestandteile einer Grammatik 23
1.6.2 Mehrere Begriffe von Implikation und Negation 25
1.6.3 Die Verwendung von Listen und Mengen 28

2 Die Sprache UBS **30**
2.1 Was man unbedingt wissen muß 30
2.1.1 Der Start 30
2.1.2 . . . und wie es weitergeht 30
2.2 Die Syntax 31
2.2.1 Die Struktur von Argumenten 31
2.2.2 Die Struktur eines Programms 34
2.3 Wie funktioniert UBS? 36
2.3.1 Programmtransformation 36

2.3.2 Anfragen an UBS 36
2.4 Eine Beispielgrammatik 37
2.4.1 Das Programm 37
2.4.2 Ein Testdurchlauf 44
2.4.3 Ein Zeitvergleich 48
2.5 Fehlermeldungen 48
2.5.1 Überblick 48
2.5.2 Die Meldungen im einzelnen 48

3 Anmerkungen zur Implementation 51
3.1 Typen und ihre interne Repräsentation 51
3.1.1 Merkmalstrukturen 51
3.1.2 Listen 51
3.1.3 Mengen 51
3.1.4 Terme 51
3.2 Typisierte Merkmalstrukturen 51
3.2.1 Was ist ein Typ? 51
3.2.2 Vererbung 52
3.2.3 Repräsentation durch Wertlisten 54
3.2.4 Typen und Wertlisten 55
3.2.5 Unifikation in allgemeinen Typenverbänden 57
3.3 Unifikation von Mengen 59
3.3.1 Begriffsdefinitionen 59
3.3.2 Was sind Mengen? 60
3.3.3 Ein einfaches Verfahren 61
3.3.4 Das Verfahren in UBS 62
3.3.5 Eine Handvoll Beispiele 63
3.3.6 Vergleich der Verfahren 64
3.3.7 Andere Ansätze 66
3.4 Funktionen und Negation 68
3.4.1 Funktionen 68
3.4.2 Relationen 69
3.4.3 Die Negation 71
3.5 Das Problem der Disjunktion 72
3.5.1 Disjunktive Normalform 72
3.5.2 Eine verbesserte Methode 73
3.5.3 Die Vorgehensweise in UBS 76

3.5.4 Verteilte oder benannte Disjunktionen . . . 77
3.5.5 Vermeidung struktureller Disjunktion . . . 78
3.5.6 Weitere Ideen . . . 78

4 Diskussion und Ausblick **80**
4.1 Vergleich mit anderen Systemen . . . 80
4.1.1 Das System AVAG . . . 80
4.1.2 Die PROLOG-Erweiterung GULP . . . 81
4.1.3 Der Formalismus STUF . . . 82
4.1.4 Ein Parser für HPSG . . . 83
4.1.5 Typisierte Merkmalstrukturen in TFS . . . 85
4.2 Alles auf einen Blick . . . 87
4.2.1 Eine Tabelle 87
4.2.2 . . . und ihre Interpretation . . . 87
4.3 Ausblick . . . 88
4.3.1 Wo noch etwas getan werden kann . . . 88
4.3.2 Schlußbemerkungen . . . 89

Abkürzungen **90**

Danksagung **91**

Erklärung **91**

Literaturverzeichnis **92**

Abbildungsverzeichnis

1 Graphdarstellung einer Merkmalstruktur . . . 6
2 Typen für Terme und Listen . . . 10
3 Veranschaulichung der Konstruktion für Terme . . . 11
4 Graphdarstellung von Termen . . . 14
5 Beispiele zur Unifikation . . . 17
6 Theoreme über Merkmalterme . . . 20
7 Die Syntax von Argumenten . . . 33
8 Typen in UBS . . . 34
9 Die Arbeitsweise von UBS . . . 38
10 Typenhierarchie von HPSG . . . 53
11 Ein Typenverband . . . 58
12 Zuordnung von Elementen . . . 63
13 Unifikation von Mengen . . . 64
14 Relationen zwischen Mengen . . . 65

1 Theoretische Grundlagen

1.1 Der Inhalt dieser Arbeit

1.1.1 Motivation

Typisierte Merkmalstrukturen spielen bei der Entwicklung von formalen Grammatiken für natürliche Sprachen eine immer größere Rolle. Sie finden in einer ganzen Reihe von zeitgenössischen Theorien der Computerlinguistik Verwendung. Merkmalstrukturen haben andererseits sehr enge Verwandtschaft mit dem Datentyp *Record*. Deshalb scheinen typisierte Merkmalstrukturen auch für die Informatik interessant zu sein, stellen sie doch einen Ansatz zur Vereinigung zweier wichtiger Paradigmen dar, nämlich

- objektorientierte Programmierung und
- logische Programmierung [Zajac 92, 159-161].

Das Konzept der Vererbung aus der objektorientierten und die Deklarativität bei der logischen Programmierung erlauben zusammen die Erstellung von kompakten und transparenten Programmen bzw. Grammatiken.

Eine linguistische Theorie, die ganz besonders Gebrauch von typisierten Merkmalstrukturen macht, ist die HPSG (Head-Driven Phrase Structure Grammar) [PolSag 87] und [Pollard 91]. Sie gehört gleichzeitig zur Gruppe der sogenannten unifikationsbasierten Grammatikformalismen. Unifikationsbasiert bedeutet hierbei, daß die Unifikation die grundlegende Operation auf einer Merkmalstruktur darstellt. Doch umfaßt der HPSG-Formalismus noch einige Elemente mehr als typisierte Merkmalstrukturen und Unifikation: Zu nennen sind hier insbesondere die Verwendung von Mengen, die Negation und die funktionalen Abhängigkeiten. All diese Elemente können – gegebenenfalls miteinander kombiniert – in Werten von Merkmalstrukturen auftreten.

In der Studienarbeit [Stolzenburg 91] ist der Versuch unternommen worden, die Elemente des HPSG-Formalismus in einer Programmiersprache zur Verfügung zu stellen. Das Ergebnis dieser Arbeit ist die unifikationsbasierte Sprache UBS[1], die als Erweiterung von regulärem PROLOG implementiert ist. Sie bietet aber noch eine ganze Reihe von Möglichkeiten zu ihrer Erweiterung bzw. Verbesserung. Das soll in dieser Diplomarbeit geschehen. Der Hauptansatzpunkt bildet dabei die Einführung einer Typisierung von Merkmalstrukturen.

1.1.2 Zielsetzungen und Aufgabenstellung

Die Ideen bei der Entwicklung von UBS sollen in dieser Arbeit weitergeführt werden, d.h. UBS soll weiterhin als Erweiterung von PROLOG implementiert werden. Im folgenden soll mit UBS* die alte Version von UBS gemeint sein; die Neuimplementierung in SEPIA, einer mächtigen Erweiterung von PROLOG, wird einfach mit UBS bezeichnet.

Beim Übergang von UBS* nach UBS sind einige wesentliche Änderungen zu beobachten; sie sollen hier kurz zusammengefaßt werden. Die folgende Auflistung stellt gleichzeitig

[1] Ein Verzeichnis aller in dieser Arbeit verwendeten Abkürzungen befindet sich im Anhang.

die Zielsetzungen und die Aufgabenstellung dieser Arbeit dar. Die ersten beiden Punkte werden auch in [Stolzenburg 92, 32-33] vorgeschlagen.

Zielsetzung 1: Der Benutzer von UBS soll die Möglichkeit erhalten, Typen von Merkmalstrukturen definieren zu können. Das erlaubt zum einen eine Vorgehensweise bei der Formulierung von Grammatiken, die der von HPSG näher kommt, als das in UBS* der Fall ist. Zum anderen bringt die Einführung typisierter Merkmalstrukturen einen Effizienzgewinn: Sowohl Speicherplatz als auch Rechenzeit kann eingespart werden. Die interne Darstellung von typisierten Merkmalstrukturen benutzt Wertlisten, die schon im System GULP [Covington 89] eingeführt worden sind. Die Hinzufügung von Typinformation zu den Wertlisten in UBS ermöglicht eine kompaktere interne Repräsentation von Merkmalstrukturen als in GULP.

Zielsetzung 2: Um mit Negation, Mengen und Funktionswerten umgehen zu können, werden in UBS* drei interne Größen eingesetzt. Sie werden jedem transformierten Prädikat als weitere Argumente hinzugefügt. Das bringt für einen Benutzer das Problem mit sich, zwischen sogenanntem markierten und unmarkierten Code unterscheiden zu müssen. Außerdem ist die Behandlung der drei Größen recht aufwendig und langsam in regulärem PROLOG. In UBS soll deshalb auf die drei internen Parameter verzichtet werden. Das wird ermöglicht durch die Verwendung der Programmiersprache SEPIA.

SEPIA [SEPIA 91] ist eine Entwicklung des ECRC, eines europäischen Forschungsinstituts in München. Es besitzt umfangreiche Erweiterungen gegenüber regulärem PROLOG: Programme in SEPIA werden grundsätzlich compiliert; trotzdem ist ein effektives Debugging möglich. Globale Variablen und Arrays lassen sich definieren. Der Benutzer kann explizit Fehlerbehandlung und die Behandlung von synchron oder asynchron auftretenden Ereignissen vornehmen. SEPIA besitzt außerdem ein einfaches Modulkonzept. Auch Programme in C können leicht angebunden werden.

Für die Entwicklung von UBS besonders interessant ist das Definieren nebenläufiger, verzögerter Prädikate, die sogenannten Metaterme und die in SEPIA verfügbare konstruktive Negation. Die verzögerten Prädikate sind hilfreich für die korrekte Implementation von Funktionsaufrufen. Metaterme sind Variablen, die mit einem bestimmten Wert attributiert bzw. assoziert sind; sie erlauben das Speichern von Beschreibungen von Mengen zur Laufzeit. Die Negation im HPSG-Formalismus kann auf die konstruktive Negation zurückgeführt werden; in der gegenwärtigen Implementation von UBS geschieht dies jedoch nicht, sondern die Negation wird durch ein spezielles verzögertes Prädikat realisiert.

Zielsetzung 3: Das System UBS* soll vervollständigt werden: Im neuen UBS ist die Unifikation von Mengen nicht nur korrekt, sondern auch vollständig implementiert.[2] Die Syntax für die Berechnung von Funktionswerten ist geändert; Funktionsaufrufe können jetzt flexibler verarbeitet werden. Bei der Ausgabe sind die Koreferenzen sichtbar gemacht. Weitere kleinere Erweiterungen des Systems kommen hinzu. Sie sollen hier nicht alle im einzelnen aufgeführt werden.

[2] Das ist eine Verbesserung gegenüber UBS* [Stolzenburg 92, 22].

1.1.3 Überblick über den Rest der Arbeit

Diese Diplomarbeit setzt sich zusammen aus dem Programm für UBS, diesem Text und einer kleinen HPSG-Beispielgrammatik, geschrieben in UBS. Der Text soll die bei der Implementation verwendeten Ideen und Verfahren dokumentieren und erläutern. Die dabei berührten Probleme werden theoretisch aufgearbeitet. Der Vergleich mit anderen Arbeiten und Vorgehensweisen wird angestrebt.

Nun soll kurz der Inhalt aller Kapitel dieser Arbeit skizziert werden:

Kapitel 1: Das erste Kapitel enthält neben dieser allgemeinen Einführung eine abstrakte Definition der Elemente des Formalismus. Es ist angereichert mit Beispielen aus der linguistischen Theorie von HPSG (letzter Abschnitt von Kapitel 1). Sie sollen zum einen das Verständnis des Formalismus fördern; zum anderen zeigen sie, wofür die einzelnen Elemente des Formalismus gebraucht werden können. Wichtige Quellen zu diesem Kapitel stellen die Arbeiten von Carpenter [Carpenter 90], was die formalen Grundlagen, und Pollard und Sag [PolSag 87], [Pollard 91], was die linguistische Anwendung anbelangt, dar.

Kapitel 2: Nach dieser theoretischen Fundierung wird das neue System UBS vorgestellt. Seine Syntax und Funktionsweise wird beschrieben. Wie das System zu benutzen ist, wird dann konkret anhand eines Beispiels erläutert. Immer wieder finden sich Hinweise, inwiefern sich die alte und die neue Version von UBS voneinander unterscheiden.

Kapitel 3: Der Schwerpunkt dieser Arbeit liegt auf der Implementation des HPSG-Formalismus. Daher nimmt die Beschreibung von Verfahren und Algorithmen einen größeren Raum in dieser Arbeit ein. Die Erörterung, wie z.B. typisierte Merkmalstrukturen, die Unifikation von Mengen oder die Disjunktion implementiert werden können, ist durchaus nicht trivial. Denn die im ersten Kapitel gegebenen Definitionen liefern meistens keinen Hinweis darauf, wie das betreffende Element (effizient) implementiert werden kann.

Im ersten Teil von Kapitel 3 wird die Implementation von Datenstrukturen, im zweiten die von Operationen diskutiert. Oft werden mehrere mögliche Verfahren zu ihrer Behandlung angegeben. Die einzelnen Verfahren werden begründet, an Beispielen erläutert und zum Teil auch bewiesen. Nicht immer wird das effizienteste Verfahren in UBS benutzt. Die hier vorgestellten, effizienteren Verfahren können in einer späteren Version von UBS realisiert werden. In diesem Zusammenhang soll es nur darum gehen, einen Einblick in unterschiedliche Vorgehensweisen zu geben.

Kapitel 4: Das letzte Kapitel liefert einen Überblick über zur Zeit bestehende Systeme, die für den HPSG-Formalismus geeignet sind. Es beginnt mit der Darstellung des relativ alten Systems AVAG [Sedogbo 86] und endet mit der des aktuellen TFS [EmZa 90]. Zu jedem vorgestellten System ist aufgezählt, von wem, wo und in welchem Zusammenhang es entwickelt worden ist, welche Möglichkeiten es bietet, inwieweit es tatsächlich für HPSG geeignet ist und welche weiteren Besonderheiten es hat.

Ein Vergleich der Systeme untereinander und mit UBS wird angestrebt. Die wesentlichen Merkmale der verschiedenen Systeme sind in einer Tabelle festgehalten. Damit kann eine abschließende Beurteilung des Systems UBS in seinem jetzigen Zustand gegeben werden. Zuletzt erhält der Leser einen Ausblick, was noch getan werden kann.

Anhang: Im Anhang befindet sich

- eine Auflösung der in dieser Arbeit verwendeten Abkürzungen,
- einige Formalien und
- das Literaturverzeichnis.

Der interessierte Leser kann aus diesem Teil Anregungen für weitere Arbeit auf dem Gebiet der unifikationsbasierten Grammatikformalismen erhalten, indem er das System UBS studiert oder in weiterführender Literatur nachliest.

1.2 Grundlegende Begriffe

Im folgenden soll kurz, knapp und präzise beschrieben werden, mit was für Elementen es der HPSG-Formalismus zu tun hat. Die Datenstrukturen und Operationen des – leicht erweiterten – Formalismus werden rein abstrakt und mathematisch dargestellt. Es soll hier nur darum gehen, allgemeingültige Eigenschaften des Formalismus festzuhalten. Die Frage, wie die Datenstrukturen bzw. Operationen möglichst effizient repräsentiert werden können, wird dadurch in keiner Weise gelöst. Das ist der Gegenstand von Kapitel 3.

Die kommenden Ausführungen orientieren sich an mehreren Arbeiten, im wesentlichen [Aït-Kaci 86], [Carpenter 90], [PolMos 90] und [Smolka 89]. Die Begriffe sind mal der einen, mal der anderen Arbeit entnommen, gegebenenfalls mit leichten Modifikationen. Darum ist eine Bemerkung angebracht: Zitatsangaben bei Definitionen besagen nur, daß eine sinngemäße Übernahme aus der jeweiligen Quelle erfolgt ist. Eventuelle Fehler sind im Zweifelsfalle nicht den Quellen, sondern dem Autor dieser Arbeit anzulasten.

1.2.1 Typisierte Merkmalstrukturen

Im HPSG-Formalismus unterliegen die Merkmalstrukturen – die grundlegende Datenstruktur im Formalismus – einer Typenhierarchie. Daher soll zunächst der Begriff Typenhierarchie definiert werden, bevor formal eingeführt wird, was eine Merkmalstruktur ist.

Definition 1: Eine *Typenhierarchie* ist eine geordnete Menge $(\mathcal{S}, \preceq)$. $\mathcal{S}$ ist eine (endliche) Menge von Typnamen. Die Ordnungsrelation $\preceq$ ist die reflexive und transitive Hülle der *Vererbungsrelation* $\prec \subseteq \mathcal{S} \times \mathcal{S}$, die minimale Teilmenge von $\preceq$ ist.

Falls $s_1 \preceq s_2$ $(s_1 \prec s_2)$ gilt, dann heißt s_1 *Subtyp* (*direkter Subtyp*) von s_2 bzw. s_2 *Supertyp* (*direkter Supertyp*) von s_1. Eine Typenhierarchie heißt *flach*, falls $\preceq = \emptyset$, und *einfach*, falls aus $s \prec s_1$ und $s \prec s_2$ folgt, daß $s_1 = s_2$. □

Definition 2: Eine (*erweiterte* oder typisierte) *Merkmalstruktur* über einer (endlichen) Menge von Attributnamen $\mathcal{L}$ und einem Typenverband $(\mathcal{S}, \preceq)$ ist ein Tupel $(Q_{ind}, Q_{set}, q_0, \delta, \alpha, \eta)$ mit:

1. $Q = Q_{ind} \cup Q_{set}$ ist eine nicht-leere, endliche Knotenmenge. Dabei sind Q_{ind} und Q_{set} disjunkte Mengen von Individuen- bzw. Mengen-Knoten.
2. $q_0 \in Q$ heißt Wurzel(knoten) der Merkmalstruktur.
3. $\delta : Q_{ind} \times \mathcal{L} \rightarrow Q$ ist die (partiell definierte) Übergangsfunktion.
4. $\alpha : Q_{ind} \rightarrow \mathcal{S}$ ist eine (totale) Funktion, Typzuweisung genannt.
5. $\eta \subseteq Q_{set} \times Q$ ist die Element-Relation (für Mengen).
6. Alle Knoten $q \in Q$ sind von der Wurzel q_0 aus erreichbar (Konnektivität).

Ein Knoten $q = q_n$ heißt *erreichbar* von einem Knoten q_1 aus, falls eine Sequenz von Knoten $q_1 \ldots q_n$ existiert, so daß für $1 < i \leq n$ gilt: Entweder gibt es ein $l \in \mathcal{L}$ mit $\delta(q_{i-1}, l) = q_i$ oder es gilt $q_{i-1} \eta q_i$. Falls $n > 1$ ist, heißt q *echt erreichbar* von q_1 aus. [PolMos 90, 306]

Eine Merkmalstruktur mit $Q_{set} = \emptyset$ heißt *einfach* (oder typisiert). Falls in einer Merkmalstruktur kein Knoten von sich selbst aus echt erreichbar ist, so heißt sie *azyklisch*. Eine Merkmalstruktur besitzt eine *Koreferenz* im Knoten $q \neq q_0$, falls $\delta(q_1, l_1) = q = \delta(q_2, l_2)$ für zwei Paare $(q_1, l_1),\ (q_2, l_2) \in Q \times \mathcal{L}$ gilt, die sich in mindestens einer Komponente unterscheiden; Koreferenz im Knoten q_0 liegt dann vor, wenn es ein $q \in Q$ und ein $l \in \mathcal{L}$ gibt mit $\delta(q, l) = q_0$. □

1.2.2 Darstellung von Merkmalstrukturen

Merkmalstrukturen können als gerichtete Graphen [Shieber 86, 20-21] oder als endliche Automaten [Kasper 87, 259] aufgefaßt werden, wobei jedoch zusätzliche Annotationen zur Darstellung von Merkmalstrukturen nötig sind.

Beispiel 1: Sei z.B. die folgende Merkmalstruktur B gegeben:

1. Typenhierarchie:
   $\mathcal{S} = \{Adele, Fritz, Geschlecht, männlich, Name, Person, weiblich\}$ mit
   $Adele \prec Name$, $Fritz \prec Name$, $männlich \prec Geschlecht$, $weiblich \prec Geschlecht$
2. Attributnamen:
   $\mathcal{L} = \{\text{NAME}, \text{GESCHL}, \text{VERH}\}$
3. $B = (Q_{ind}, Q_{set}, q_o, \delta, \alpha, \eta)$ mit
   (a) Knotenmengen:
   $Q_{ind} = \{q_1, q_2, q_3, q_4, q_5, q_6\}$ $\quad Q_{set} = \{q_0\}$
   (b) Wurzelknoten: q_0

(c) Übergangsfunktion:
$\delta(q_1, \text{NAME}) = q_3$, $\delta(q_1, \text{GESCHL}) = q_4$, $\delta(q_1, \text{VERH}) = q_2$,
$\delta(q_2, \text{NAME}) = q_5$, $\delta(q_2, \text{GESCHL} = q_6$, $\delta(q_2, \text{VERH}) = q_1$

(d) Typzuweisung:
$\alpha(q_1) = \alpha(q_2) = Person$,
$\alpha(q_3) = Fritz$, $\alpha(q_4) = männlich$, $\alpha(q_5) = Adele$, $\alpha(q_6) = weiblich$

(e) Element-Relation:
$\eta = \{(q_0, q_1), (q_0, q_2)\}$

Die Merkmalstruktur kann als gerichteter Graph veranschaulicht werden: Die Abbildung 1 zeigt die Graphdarstellung der Merkmalstruktur B. Wie zu erkennen ist, modelliert die Merkmalstruktur eine Menge, bestehend aus zwei Personen namens Fritz und Adele, die miteinander verheiratet sind.

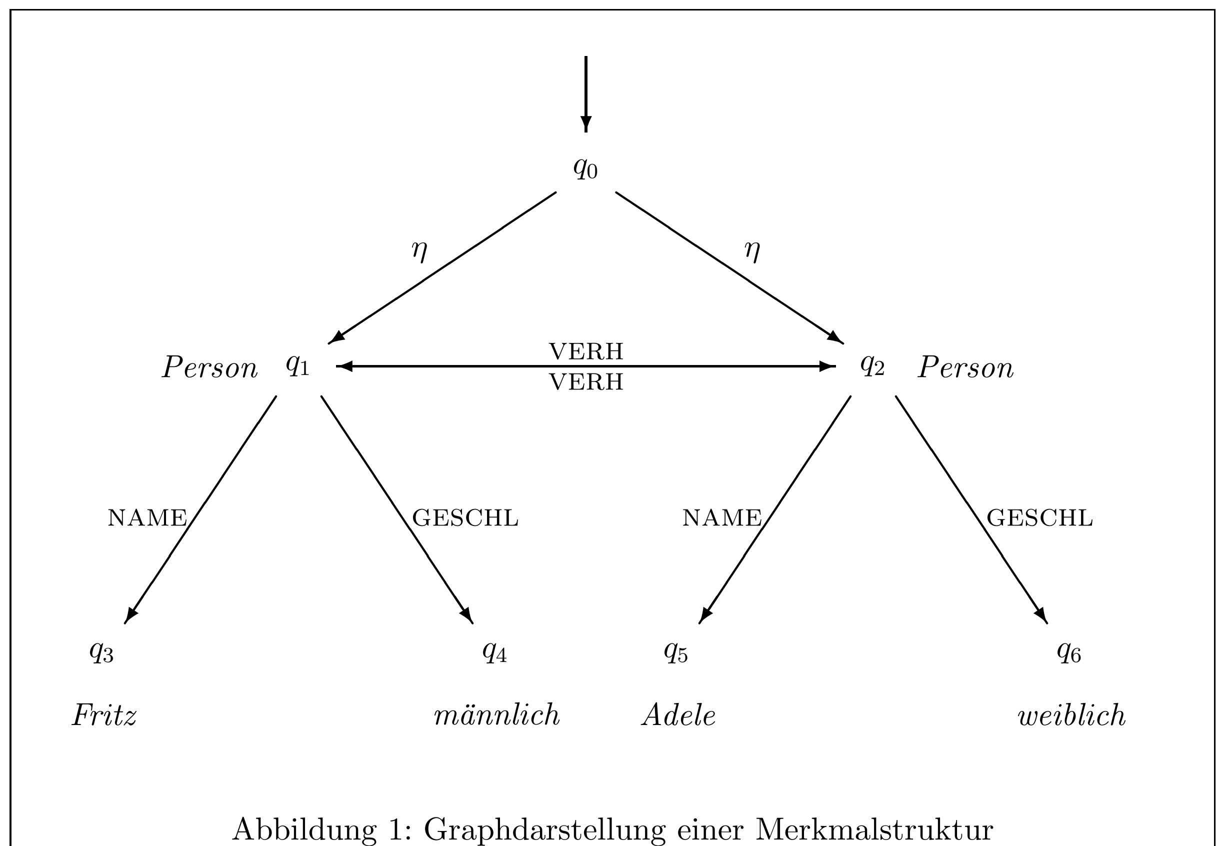


Abbildung 1: Graphdarstellung einer Merkmalstruktur

Werden die Knotenbezeichnungen q_0, q_1, ..., q_6 weggelassen, so ergibt sich die Darstellung der sogenannten abstrakten Merkmalstruktur $[B]$ zu B. Die Knoten werden dann einfach als Punkte gezeichnet. Was abstrakte Merkmalstrukturen sind, wird im nächsten Abschnitt formal eingeführt.

Abstrakte Merkmalstrukturen können auch in Matrixnotation (als AVM) geschrieben werden. Mengen werden dabei in geschweiften Klammern notiert, wie es auch in der Mathematik üblich ist. Zur Darstellung von (einfachen) Merkmalstrukturen werden eckige Klammern verwendet. Typnamen werden den Merkmalstrukturen in Kursivschrift vorangestellt; zum Teil werden sie einfach weggelassen, falls die Typinformation schon aus

dem Kontext hervorgeht. Die Darstellung von $[B]$ in Matrixnotation ist:

$$\left\{ Person \boxed{1} \begin{bmatrix} \text{NAME } \mathit{Fritz} \\ \text{GESCHL } \mathit{männlich} \\ \text{VERH } \boxed{2} \end{bmatrix}, Person \boxed{2} \begin{bmatrix} \text{NAME } \mathit{Adele} \\ \text{GESCHL } \mathit{weiblich} \\ \text{VERH } \boxed{1} \end{bmatrix} \right\}$$

Erläuterung: Zwischen den eckigen Klammern sind Paare von Attributen und ihren zugehörigen Werten aufgelistet. Ein solches Paar wird Merkmal genannt. Gewöhnlicherweise werden seine beiden Bestandteile durch einen Doppelpunkt oder – wie hier – einfach durch einen kleinen Zwischenraum voneinander getrennt. Eingerahmte Zahlen kennzeichnen identische (oder koreferente) Werte (ein einziger Knoten im Graph). Der genaue Wert wird dabei nur einmal für jede Zahl hinter einem der Kästen, der die Zahl einrahmt, angegeben. Die eingerahmten Zahlen können als Variablen für Merkmalstrukturen angesehen werden.

1.2.3 Subsumption und Äquivalenz

Nun sollen zwei Relationen für Merkmalstrukturen definiert werden: die Subsumption und die Äquivalenz. Merkmalstrukturen gelten als partielle Beschreibungen von Objekten. Daher können die beiden Relationen intuitiv wie folgt definiert werden:

1. Eine Merkmalstruktur A subsumiert eine Merkmalstruktur A', wenn die Menge der durch A beschriebenen Objekte eine Obermenge der durch A' beschriebenen ist, oder anders ausgedrückt: die Beschreibung A' ist informativer als Beschreibung A.
2. Äquivalent heißen zwei Merkmalstrukturen dann, wenn sie den gleichen Informationsgehalt haben bzw. die gleiche Menge von Objekten beschreiben (können).

Definition 3: Eine Merkmalstruktur A *subsumiert* eine Merkmalstruktur A', in Zeichen $A' \sqsubseteq_F A$, falls es eine Relation $R \subset (Q_{ind} \times Q'_{ind}) \cup (Q_{set} \times Q'_{set})$ gibt mit:

1. $q_0 R q'_0$.
2. Falls $\delta(q,l)\downarrow$ und qRq' gilt, so folgt daraus, daß $\delta'(q',l)\downarrow$ und $\delta(q,l)R\delta'(q',l)$.[3]
3. qRq' impliziert $\alpha'(q') \preceq \alpha(q)$.
4. Falls $q \in Q_{set}$, $q' \in Q'_{set}$ und qRq', dann gibt es zu jedem $x \in Q$ mit $q\eta x$ ein $y \in Q'$ mit $q'\eta' y$, so daß xRy zutrifft, und umgekehrt.
5. R ist eine totale Funktion, eingeschränkt auf $Q_{ind} \times Q'_{ind}$.
6. Zu jedem $q \in Q$ existiert ein $q' \in Q'$ mit qRq'. [PolMos 90, 307] □

Definition 4: Zwei erweiterte Merkmalstrukturen A, A' heißen *äquivalent* (oder gleich oder isomorph), in Zeichen $A \equiv A'$, falls $A \sqsubseteq_F A'$ und $A' \sqsubseteq_F A$ gilt. Die Relation $\equiv$ ist

[3]Die Schreibweise $\downarrow$ hinter einem Funktionswert bedeutet, daß der Wert an der betreffenden Stelle definiert ist; $\uparrow$ besagt, daß er undefiniert ist.

eine Äquivalenzrelation in der Menge der erweiterten Merkmalstrukturen. Ihre Äquivalenzklassen $[A]$, wobei A eine erweiterte Merkmalstruktur ist, bilden die Grundlage der folgenden Definition. □

Definition 5: Eine Äquivalenzklasse $[A]$ bezüglich der Relation $\equiv$ in der Menge der erweiterten Merkmalstrukturen heißt *abstrakte Merkmalstruktur*. Eine abstrakte Merkmalstruktur $[A]$ *subsumiert* eine abstrakte Merkmalstruktur $[A']$, in Zeichen $[A'] \sqsubseteq_{AF} [A]$, genau dann, wenn $A' \sqsubseteq_F A$ gilt. Die Relation $\sqsubseteq_{AF}$ ist eine Ordnungsrelation, während $\sqsubseteq_F$ nur reflexiv und transitiv, aber nicht antisymmetrisch ist. □

Beispiel 2: Es folgt eine Subsumptions-Kette nach [Shieber 86, 16] zur Illustration dessen, was Subsumption praktisch bedeutet:

$$\begin{bmatrix} \text{CAT: } np \\ \text{AGR: } \boxed{2} \begin{bmatrix} \text{NUM: } sng \\ \text{PER: } third \end{bmatrix} \\ \text{SUBJ: } \boxed{2} \end{bmatrix} \sqsubseteq_{AF} \begin{bmatrix} \text{CAT: } np \\ \text{AGR: } \begin{bmatrix} \text{NUM: } sng \\ \text{PER: } third \end{bmatrix} \\ \text{SUBJ: AGR: } \begin{bmatrix} \text{NUM: } sng \\ \text{PER: } third \end{bmatrix} \end{bmatrix}$$

$$\sqsubseteq_{AF} \begin{bmatrix} \text{CAT: } np \\ \text{AGR: } \begin{bmatrix} \text{NUM: } sng \\ \text{PER: } third \end{bmatrix} \end{bmatrix}$$

1.3 Die Einbettung von Termen

1.3.1 Terme und Listen

Neben Mengen, die in erweiterten Merkmalstrukturen schon darstellbar sind, sieht der HPSG-Formalismus auch Listen innerhalb von Merkmalstrukturen vor. In UBS sind zusätzlich noch Terme im Formalismus vorgesehen. Wie können aber Terme und Listen in Merkmalstrukturen eingebettet werden? – Bevor diese Frage beantwortet werden kann, sind zunächst die Begriffe Term und Liste zu definieren.

Definition 6: Ein *Term* ist entweder ein Variablensymbol, ein Konstantensymbol oder ein Funktionssymbol, gefolgt von einer Sequenz von Termen, getrennt durch Kommata, eingeschlossen in runden Klammern. Im letzteren Fall spricht man von einem zusammengesetzten Term; die Länge der Sequenz heißt auch *Stelligkeit* des Funktionssymbols.

Eine *Substitution* σ ist eine Funktion, die Variablen in Terme abbildet. Das Ergebnis der Anwendung einer Substitution σ auf einen Term t, geschrieben als $t\sigma$, wird erhalten, indem alle Variablen x in t ersetzt werden durch $\sigma(x)$. Die *Kompositon* zweier Substitutionen σ und τ, geschrieben als $\sigma\tau$, ist die Hintereinanderausführung der Substitutionen, zuerst σ, dann τ.

Ein Term t subsumiert einen Term t', in Zeichen $t' \leq t$, falls es eine Substitution σ gibt mit $t\sigma = t'$. Man sagt dann auch: t *matcht* t'. Das obige Gleichheitszeichen bedeutet syntaktische (zeichenweise) Gleichheit. [Knight 89, 95]

Eine Substitution σ heißt *Unifikator* zweier Terme s und t, falls $s\sigma = t\sigma$. Eine Substitution σ heißt *allgemeiner* als eine Substitution τ, in Zeichen $\sigma \geq \tau$, falls es eine Substitution θ gibt mit $\tau = \sigma\theta$. σ heißt *allgemeinster Unifikator* (MGU) von s und t, falls für alle anderen Unfikatoren τ von s und t gilt: $\sigma \geq \tau$. □

Substitutionen können durch endliche Mengen von Variablenbindungen spezifiziert werden. Die Schreibweise für eine Substitution $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ steht für die nachfolgende Funktion. Der Pfeil $\leftarrow$ entspricht einer Gleichsetzung bzw. Unifikation.

$$\sigma(x) = \begin{cases} t_i, & \text{falls } x = x_i \ (1 \leq i \leq n) \\ x, & \text{sonst} \end{cases}$$

Definition 7: Eine *Liste* ist entweder die leere Liste [] oder eine zusammengesetzte Liste der Form $[First|Rest]$, wobei $First$ ein beliebiger Ausdruck und $Rest$ entweder eine Liste oder ein Platzhalter für eine Liste ist. Die Schreibweise $[t_1, \ldots, t_n]$ wird abkürzend für $[t_1| \ldots [t_n|[]]]$ verwendet; es handelt sich dann um eine geschlossene Liste. Eine Liste der Form $[t_1| \ldots [t_n|x]]$, abgekürzt $[t_1, \ldots, t_n|x]$, heißt offene Liste, wobei x ein Platzhalter (Variable) für eine Liste ist. □

Listen können auch als spezielle Terme angesehen werden: Die leere Liste entspricht dann der Konstanten *nil*; zusammengesetzte Listen werden mit Hilfe des zweistelligen Funktionssymbols *cons* aufgebaut. Eine Liste $[t_1, \ldots, t_n]$ kann somit durch den Term $cons(t_1, \ldots cons(t_n, nil))$ repräsentiert werden.

In der Matrixnotation können die oben definierten Schreibweisen für Terme und Listen übernommen werden. Um Listen jedoch klar von (einfachen) Merkmalstrukturen unterscheiden zu können, werden für Listen spitze statt eckige Klammern in der Matrixnotation verwendet.

Beispiel 3: Die folgende Matrix soll den Sachverhalt modellieren, daß der Satz

Fritz liebt Adele.

den durch den Term $lieben(Fritz, Adele)$ ausgedrückten propositionalen Gehalt hat und gleichzeitig die Verwendung von Listen und Termen in der Matrixnotation illustrieren:

$$\begin{bmatrix} \text{PHONOLOGY: } \langle Fritz, liebt, Adele \rangle \\ \text{SEMANTICS: } lieben(Fritz, Adele) \end{bmatrix}$$

Um die Einbettung von Termen und Listen in Merkmalstrukturen formal vornehmen zu können, werden einfach einige Typen und Attribute für die Repräsentation von Termen und Listen vordefiniert. Es gelte $\{\text{FIRST}, \text{REST}, 1, \ldots, N\} \subseteq \mathcal{L}$ für ein hinreichend großes N (größte vorkommende Stelligkeit eines Funktionssymbols). Die Typenhierarchie $(\mathcal{S}, \preceq)$ umfaßt (mindestens) die Typenhierarchie in Abbildung 2, als Hasse-Diagramm dargestellt.

Zu allen Termen oder Listen können nun einfache Merkmalstrukturen A_t konstruiert werden, die t repräsentieren. Die Merkmalstrukturen A_t können induktiv wie folgt definiert werden:

1. **Terme:**
   (a) Variablen: Sei $t = x$ eine Variable, dann ist $A_t = (\{x\}, \emptyset, x, \delta, \alpha, \emptyset)$ mit $\delta(x, l)\uparrow$ für alle $l \in \mathcal{L}$ und $\alpha(x) = Variable$. Variablen können nicht nur mit Termen, sondern auch mit Listen instanziiert werden.
   (b) Konstanten: Sei $t = c$ eine Konstante, dann ist $A_t = (\{c\}, \emptyset, c, \delta, \alpha, \emptyset)$ mit $\delta(c, l)\uparrow$ für alle $l \in \mathcal{L}$ und $\alpha(c) = c$.
   (c) Zusammengesetzter Term: Sei $t = f(t_1, \ldots, t_n)$ ein Term und $A_{t_i} =$

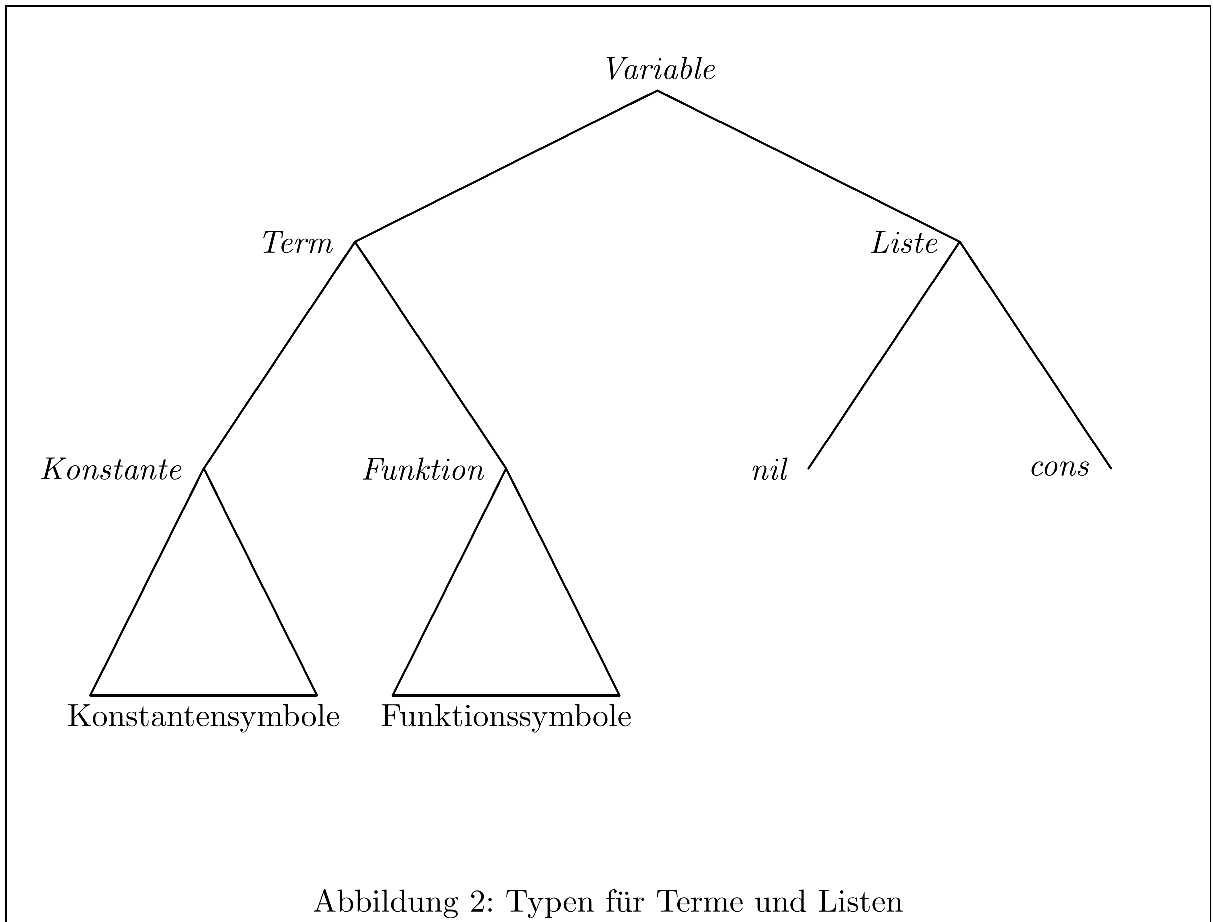


Abbildung 2: Typen für Terme und Listen

$(Q_i, \emptyset, q_i, \delta_i, \alpha_i, \emptyset)$ für $1 \leq i \leq n$. Dann ist

$$A_t = (\{q_{f_n}\} \cup \bigcup_{i=1}^{n} Q_i, q_{f_n}, \delta, \alpha, \emptyset)$$

mit dem neuen Knoten $q_{f_n} \notin \bigcup_{i=1}^{n} Q_i$ und für alle $1 \leq i \leq n$ und $q \in Q_i$ gilt:

i. $\delta(q_{f_n}, i) = q_i$
ii. $\delta(q, l) = \delta_i(q, l)$
iii. $\alpha(q) = \alpha_i(q)$
iv. $\alpha(q_{f_n}) = f_n$

Die Konstruktion von A_t kann wie in Abbildung 3 veranschaulicht werden. Zu beachten ist, daß die Graphen für die A_{t_i} (hier als Dreiecke gezeichnet) durch Kanten miteinander verbunden sein können, da ein und dieselbe Variable oder Konstante durch jeweils einen einzigen Knoten im Graphen repräsentiert wird. Für jedes Vorkommen eines Funktionssymbols wird allerdings ein neuer Knoten eingeführt. [Stolzenburg 91, 70-72]

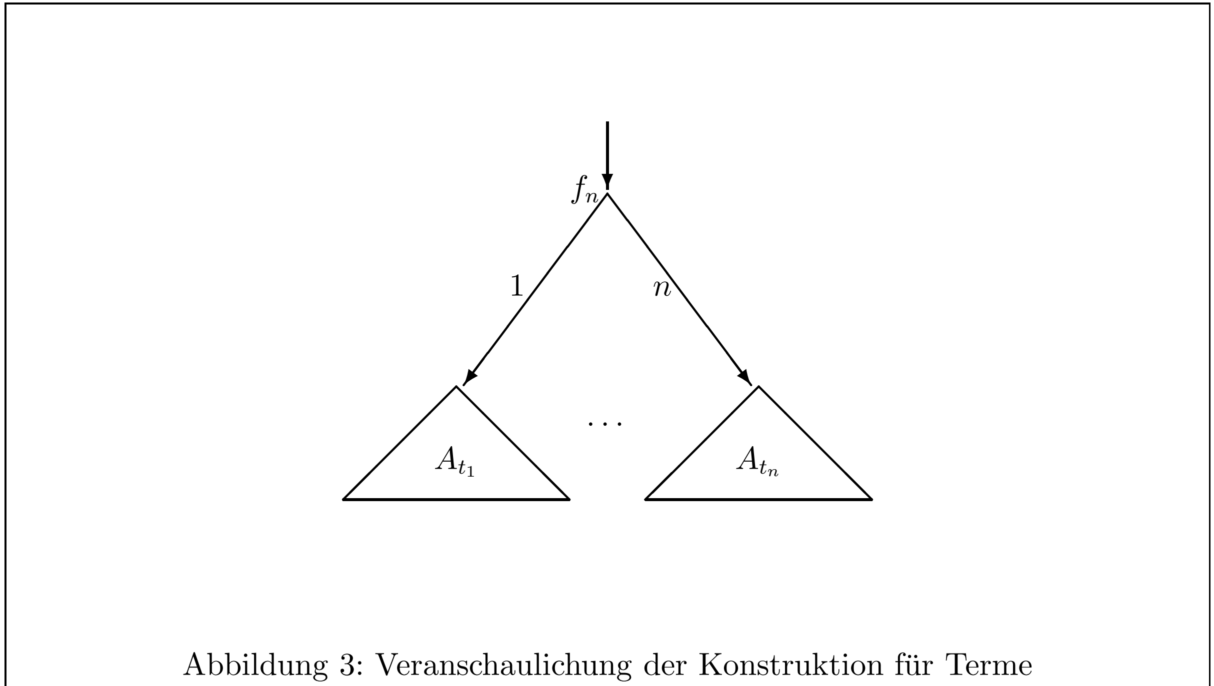


Abbildung 3: Veranschaulichung der Konstruktion für Terme

2. **Listen:**

(a) leere Liste: Es ist $A_{[]} = (\{nil\}, \emptyset, \delta, \alpha, \emptyset)$ mit $\delta(nil, l) \uparrow$ für alle $l \in \mathcal{L}$ und $\alpha(nil) = nil$.

(b) nicht-leere Liste: Es gilt

$$A_{[First|Rest]} = (\{q_0\} \cup Q_{First} \cup Q_{Rest}, \emptyset, q_0, \delta, \alpha, \emptyset)$$

mit $A_i = (Q_i, \emptyset, q_i, \delta_i, \alpha_i, \emptyset)$ für $i \in \{First, Rest\}$. Dann repräsentiert $A_{[First|Rest]}$ die Liste $[First|Rest]$, wenn für $i \in \{First, Rest\}$ und $q \in Q_i$ gilt:

i. $\delta(q_0, \text{FIRST}) = q_{First}$ und $\delta(q_0, \text{REST}) = q_{Rest}$

ii. $\delta(q, l) = \delta_i(q, l)$

iii. $\alpha(q) = \alpha_i(q)$

iv. $\alpha(q_0) = cons$

(c) Platzhalter: Ein Platzhalter x für Listen wird repräsentiert durch $A_x = (\{x\}, \emptyset, x, \delta, \alpha, \emptyset)$ mit $\delta(x, l)\uparrow$ für alle $l \in \mathcal{L}$ und $\alpha(x) = list$. Letzteres garantiert, daß x nur mit einer Liste instanziiert werden kann. [Shieber 86, 29-30]

Für Terme und Listen t und t' gilt $t \leq t'$ genau dann wenn $A_t \sqsubseteq_F A_{t'}$ (ohne Beweis). Es folgt die Darstellung von Beispiel 3 vollständig durch einfache Merkmalstrukturen in Matrixnotation:

$$\begin{bmatrix} \text{PHONOLOGY: } cons \begin{bmatrix} \text{FIRST: } \boxed{1}\ Fritz \\ \text{REST: } cons \begin{bmatrix} \text{FIRST: } liebt \\ \text{REST: } cons \begin{bmatrix} \text{FIRST: } \boxed{2}\ Adele \\ \text{REST: } nil \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{SEMANTICS: } lieben_2 \begin{bmatrix} 1\ \boxed{1} \\ 2\ \boxed{2} \end{bmatrix} \end{bmatrix}$$

1.3.2 Angemessenheit von Attributen

Ein aufmerksamer Leser könnte an dieser Stelle einwenden, daß die Darstellung von Konstanten nach dem obigen Verfahren nicht ganz sauber ist. Denn für Konstanten, die hier durch spezielle Typen dargestellt werden, sollte gelten, daß keines ihrer Attribute einen definierten Wert erhalten kann. Das wird jedoch – zumindest bis jetzt – nicht durch irgendwelche Regeln ausdrücklich verboten. Im Prinzip kann jedes Attribut in jedem Typ einen Wert zugewiesen bekommen.

Tatsächlich ist aber für einen bestimmten Typ nur eine Teilmenge aller Attribute relevant. Die Angabe eines Wertes für das Attribut PHONOLOGY bei einer Merkmalstruktur vom Typ *Person* ist sicherlich unpassend. Daher sollte geregelt werden können, welche Attribute mit welchen Wertebereichen (Typangabe) bei welchem Typ erlaubt sind.

In GPSG dienen die Feature Cooccurrence Restrictions (FCR) diesem Zweck [Sells 85, 102-103]. Ein anderer Vorschlag von [EisDör 90, 125] ist es, einen besonderen Typ *none* einzuführen, für den keine Merkmale definiert werden können. Wenn ein Attribut den Wert *none* hat, so bedeutet dies, daß dieses Attribut keinen definierten Wert haben kann. Für alle Konstanten c ist dann z.B. $\delta(c, l) = none$ für alle $l \in \mathcal{L}$ zu setzen.

Da die Menge der Attribute, die in einem bestimmten Typ unpassend sind, aber meistens groß ist gegenüber der Anzahl der passenden Attribute, scheint es günstiger, positiv zu definieren, welche Attribute in einem Typ erlaubt sind. Das geschieht in der folgenden Definition.

Definition 8: Die Angemessenheit von Attributen in Merkmalstrukturen über der Typenhierarchie $(\mathcal{S}, \preceq)$ und der Menge von Attributnamen $\mathcal{L}$ wird durch die partielle Funktion $Approp : \mathcal{S} \times \mathcal{L} \to \mathcal{S}$ definiert. Es gilt:

1. Zu jedem Attribut $l \in \mathcal{L}$ gibt es einen (eindeutigen) allgemeinsten Typ $Intro(l)$ mit $Approp(Intro(l), l)\downarrow$, d.h. $Intro(l) = sup\{s \in \mathcal{S} \mid Approp(s, l)\downarrow\}$ (Supremum).

2. Falls $Approp(s,l)\downarrow$ und $t \preceq s$, dann gilt $Approp(t,l)\downarrow$ und $Approp(t,l) \preceq Approp(s,l)$.

Falls $Approp(t,l) = w$ ist, so sagt man: Das Attribut l ist in Merkmalstrukturen vom Typ t *angemessen* (oder zulässig oder erlaubt oder passend) mit einem Wert vom Typ w. [Carpenter 90, 107-108] □

Definition 9: Eine Merkmalstruktur heißt *wohlgetypt*, falls für jeden Knoten q und alle $l \in \mathcal{L}$ mit $\delta(q,l)\downarrow$ folgt, daß $Approp(\alpha(q),l)\downarrow$ und $\alpha(\delta(q,l)) \preceq Approp(\alpha(q),l)$ gilt.

Die Funktion *Approp* heißt *konstant*, falls für alle $l \in \mathcal{L}$ und $s,t \in \mathcal{S}$ mit $Approp(s,l)\downarrow$ und $Approp(t,l)\downarrow$ gilt, daß $Approp(s,l) = Approp(t,l)$. Falls die Funktion *Approp* konstant ist, so ist das Ergebnis der Unifikation zweier wohlgetypter Merkmalstrukturen auch wohlgetypt. [Carpenter 90, 131]

Eine Merkmalstruktur A heißt *typisierbar*, wenn es eine wohlgetypte Merkmalstruktur $A' \sqsubseteq_F A$ gibt. Es gibt dann eine allgemeinste solche Merkmalstruktur A', die durch ein Typinferenzverfahren hergeleitet werden kann. [Carpenter 90, 110] □

Beispiel 4: Nun soll ein Beispiel betrachtet werden, in dem die Funktion *Approp* verwendet wird. Für Listen und ihre Subtypen kann *Approp* wie folgt definiert werden:

1. Für alle $l \in \mathcal{L}$ gilt $Approp(Liste,l)\uparrow$ und $Approp(nil,l)\uparrow$.

2. Es gilt $Approp(cons, \textsc{first}) = Variable$, $Approp(cons, \textsc{rest}) = Liste$, andernfalls gilt $Approp(cons,l)\uparrow$.

Das Typkonzept kann noch weiter verfeinert werden, z.B. dahingehend, daß in einem Programm mit Merkmalstrukturen – dieser Begriff ist noch zu definieren – die Typisierung statisch erfolgen kann, d.h. zur Compilezeit. Zur Laufzeit sind dann keine Typüberprüfungen mehr nötig. Ausführungen zur statischen und zur sogenannten totalen Typisierung von Merkmalstrukturen finden sich in [Carpenter 90, 118-135].

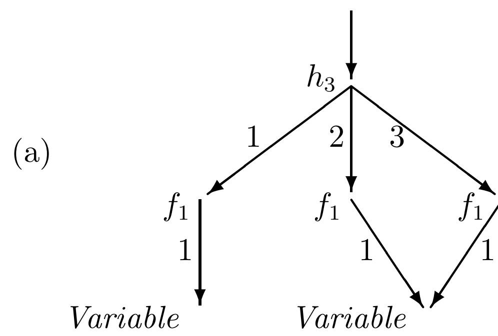
1.3.3 Gleichheit und Identität

Durch die Einführung der Funktion *Approp* ist die Behandlung von Konstanten und Variablen in befriedigender Weise gelöst. Sie werden dadurch so modelliert, daß sie sich wie die Konstanten bzw. Variablen in PROLOG verhalten. Wiederum gibt es aber noch etwas an der Behandlung von Termen auszusetzen. Dazu soll der zusammengesetzte Term $h(f(x), f(y), f(y))$ betrachtet werden, der durch die abstrakte Merkmalstruktur aus Abbildung 4 (a) dargestellt wird.

In der graphischen Darstellung ist deutlich zu erkennen, daß ein und dieselbe Variable auch nur durch einen einzigen Knoten im Graphen repräsentiert wird: Der Variablen-Knoten links unten repräsentiert die Variable x, der rechts unten y. Man schreibt für die *Identität* $y = y$, für die bloße *Gleichheit* $x \equiv y$.

Terme im Sinne von PROLOG gelten aber allgemein als identisch, wenn Funktionssymbol und Stelligkeit und alle Argumente auf gleichen Positionen miteinander übereinstimmen. So gilt z.B., daß die beiden Vorkommen von $f(y)$ in obigem Term miteinander identisch sind, während die Terme $f(x)$ und $f(y)$ zwar gleich, aber nicht identisch miteinander sind.

(a)



(b)

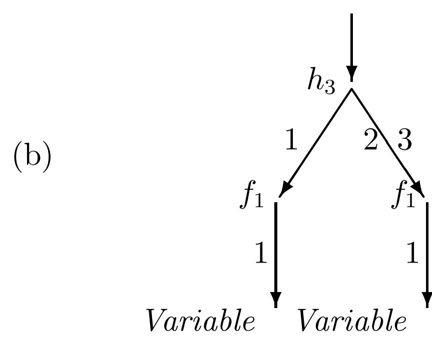


Abbildung 4: Graphdarstellung von Termen

Das Problem ist jedoch, daß die beiden Terme $f(y)$ durch verschiedene Knoten im Graphen repräsentiert sind. Eine angemessene Darstellung des obigen Terms müßte aber die beiden Knoten miteinander identifizieren, etwa wie in Abbildung 4 (b).

Um diese Identifizierung zu gewährleisten, sollen nun sogenannte extensionale Typen eingeführt werden. Alle anderen Typen heißen intensional. Zunächst eine informale Definition, dann eine genauere formale:

extensional: Ein Typ heißt extensional, falls alle Knoten in einer Merkmalstruktur von diesem Typ mit identischen Merkmalen miteinander zu identifizieren sind.

intensional: Alle anderen Typen heißen intensional. Intensionale Typen erlauben die Unterscheidung mehrerer Objekte trotz gleicher Merkmale.

Definition 10: Sei $(\mathcal{S}, \preceq)$ eine Typenhierarchie. $\mathcal{S}_{ext} \subseteq \mathcal{S}$ ist die nach unten abgeschlossenen Menge der extensionalen Typen, d.h. aus $t \in \mathcal{S}_{ext}$ und $s \preceq t$ folgt $s \in \mathcal{S}_{ext}$. Die Menge $\mathcal{S}_{int} = \mathcal{S} \backslash \mathcal{S}_{ext}$ enthält die intensionalen Typen.

Eine Merkmalstruktur heißt *extensional*, wenn für alle $q, q' \in Q$ und $t \in \mathcal{S}_{ext}$ mit $\alpha(q) \preceq t$ und $\alpha(q') \preceq t$ und alle $l \in \mathcal{L}$ mit $\delta(q,l)\downarrow$, $\delta(q',l)\downarrow$, $Approp(t,l)\downarrow$ und $\delta(q,l) = \delta(q',l)$ gilt, daß $q = q'$ ist. [Carpenter 90, 158-159]

Eine Merkmalstruktur A heißt *extensionalisierbar*, wenn es eine extensionale Merkmalstruktur $A' \sqsubseteq_F A$ gibt. Zur Konstruktion von A' läßt sich ein Inferenzverfahren angeben, das – zumindest vom Prinzip her – die Unifikation (siehe unten) verwendet. □

1.4 Merkmalterme und Operationen

1.4.1 Die Konstruktion eines Verbandes

Nun sollen die beiden Operationen Unifikation und Disjunktion formal eingeführt werden. Zu diesem Zweck wird im folgenden ein Verband über der Menge der abstrakten Merkmalstrukturen definiert. Das Infimum (größte untere Schranke) $\sqcap$ zweier Elemente in diesem Verband entspricht der Unifikation, das Supremum (kleinste obere Schranke) $\sqcup$ der Disjunktion. Zunächst werden einige Schreibweisen eingeführt nach [Aït-Kaci 86, 343].[4]

Schreibweisen: Sei $\mathcal{A}$ die Menge aller abstrakten Merkmalstrukturen über der Menge von Attributnamen $\mathcal{L}$ und der Typenhierarchie $(\mathcal{S}, \preceq)$. Dann gilt:

1. $2^{(\mathcal{A})}$ ist die Menge aller endlichen Teilmengen aus $\mathcal{A}$ von paarweise nicht miteinander durch $\sqsubseteq_{AF}$ vergleichbaren Elementen, *beschränkte Potenzmenge* genannt.

2. $\underline{x} = \{y \in \mathcal{A} \mid y \sqsubseteq_{AF} x\}$ ist die *Menge aller unteren Schranken* von x in $\mathcal{A}$.

3. Die Schreibweise $\lceil X \rceil$(*Maximum einer Menge* X), $X \subseteq \mathcal{A}$, bezeichnet die Menge aller bezüglich $\sqsubseteq_{AF}$ maximalen Elemente in X;

4. Die Relation $\sqsubseteq$ in $2^{(\mathcal{A})}$ heißt *Subsumption* und ist wie folgt definiert: $X \sqsubseteq Y$ genau dann, wenn für alle $x \in X$ ein $y \in Y$ existiert mit $x \sqsubseteq_{AF} y$. Die Subsumption $\sqsubseteq$ ist eine Ordnungsrelation auf $2^{(\mathcal{A})}$. □

[4] Vergleiche mit [Zajac 92, 163].

Definition 11: Die algebraische Struktur $(2^{(\mathcal{A})}, \sqsubseteq, \sqcap, \sqcup)$ wird definiert durch:

1. Subsumption: (siehe oben)

$$X \sqsubseteq Y \iff \forall x \in X\ \exists y \in Y (x \sqsubseteq_{AF} y)$$

2. Unifikation:

$$X \sqcap Y = \lceil \bigcup_{x \in X, y \in Y} (\underline{x} \cap \underline{y}) \rceil$$

3. Disjunktion:

$$X \sqcup Y = \lceil X \cup Y \rceil$$

Diese algebraische Struktur ist ein distributiver, aber nicht vollständiger Verband [Aït-Kaci 86, 319]. Er heißt *Merkmalstruktur-Verband.* Der Verband besitzt das Nullelement $\bot = \emptyset$, genannt Bottom, und das Einselement $\top = \lceil \mathcal{A} \rceil$, genannt Top. Letzteres ist gleich der Disjunktion aller maximalen Typen in der Typenhierarchie $(\mathcal{S}, \preceq)$ (ohne Beweis). □

Der Vorteil dieser Konstruktion ist es, daß die bestehende Ordnung erhalten bleibt, d.h. $x \sqsubseteq_{AF} y$ gilt genau dann, wenn $\{x\} \sqsubseteq \{y\}$ ist. Entsprechend gilt auch für die Unifikation (aber nicht für die Disjunktion): Falls das Infimum von $x, y \in \mathcal{A}$ existiert und gleich z ist, dann gilt $\{x\} \sqcap \{y\} = \{z\}$. Ein weiterer Vorteil dieser Konstruktion ist es, daß sie ausreicht, um die Semantik von Merkmaltermen zu beschreiben. Was unter einem Merkmalterm verstanden werden soll, wird noch im Abschnitt 1.4.3 definiert.

Die Unifikation kann auch weniger abstrakt für einfache azyklische Merkmalstrukturen wie nachfolgend operational definiert werden. Dabei ist eine flache Typenhierarchie unterstellt. Es gibt nur genau einen Typ für Merkmalstrukturen, bei dem alle Attribute zulässig sind mit beliebigen Werten. Außerdem gibt es weitere atomare Typen (Konstanten), für die überhaupt keine Attribute erlaubt sind.

Definition 12: Zwei Merkmalstrukturen $x, y \in \mathcal{A}$ seien miteinander zu *unifizieren.* Normalerweise sind sie vom Typ Merkmalstruktur, aber im Verlaufe der Unifikation kann es vorkommen, daß atomare Werte unifiziert werden müssen. Daher erklärt sich die folgende Fallunterscheidung:

1. Falls x und y beide atomar sind, müssen sie identisch sein. Sonst schlägt die Unifikation fehl.

2. Falls x und y beides Merkmalstrukturen sind, dann wird die Unifikation z durch folgende Schritte erhalten:

   (a) Wenn ein Attribut l in beiden Merkmalstrukturen einen definierten Wert hat, dann ist der Wert von l in z die Unifikation beider Werte aus x und y (Rekursion). Wenn die Werte nicht miteinander unifizierbar sind, dann sind x und y insgesamt nicht miteinander unifizierbar.

   (b) Jedes Attribut, das nur in einer Merkmalstruktur erscheint, wird einfach mit seinem Wert in z übernommen.

3. Andernfalls, falls x atomar und y eine Merkmalstruktur ist oder umgekehrt, dann schlägt die Unifikation fehl. [ShPeKaKa 86, 20-21] □

Anhand der obigen Definition wird es dem Leser sicherlich nicht schwerfallen, die Beispiele in Abbildung 5 nachzuvollziehen. Sie sind aus [Stolzenburg 92, 6-7] übernommen und demonstrieren die Unifikation von Merkmalstrukturen und die von Termen.

Unifikation von Merkmalstrukturen:

1. $\begin{bmatrix} \text{PER: } \mathit{third} \\ \text{NUM: } \mathit{sng} \end{bmatrix} \sqcap \begin{bmatrix} \text{GEN: } \mathit{fem} \\ \text{NUM: } \mathit{sng} \end{bmatrix} = \begin{bmatrix} \text{PER: } \mathit{third} \\ \text{NUM: } \mathit{sng} \\ \text{GEN: } \mathit{fem} \end{bmatrix}$

2. $\begin{bmatrix} \text{AGR: } [\text{NUM: sng}] \\ \text{SUBJ: } [\text{NUM: sng}] \end{bmatrix} \sqcap \begin{bmatrix} \text{SUBJ: } [\text{PER: } \mathit{third}] \end{bmatrix} = \begin{bmatrix} \text{AGR: } [\text{NUM: sng}] \\ \text{SUBJ: } \begin{bmatrix} \text{NUM: } \mathit{sng} \\ \text{PER: } \mathit{third} \end{bmatrix} \end{bmatrix}$

3. $\begin{bmatrix} \text{AGR: } \boxed{1}[\text{NUM: sng}] \\ \text{SUBJ: } \boxed{1} \end{bmatrix} \sqcap \begin{bmatrix} \text{SUBJ: } [\text{PER: } \mathit{third}] \end{bmatrix} =$
$\begin{bmatrix} \text{AGR: } \boxed{1} \begin{bmatrix} \text{NUM: } \mathit{sng} \\ \text{PER: } \mathit{third} \end{bmatrix} \\ \text{SUBJ: } \boxed{1} \end{bmatrix}$

Unifikation von Termen:

1. $f(a,b,x) \sqcap f(y,b,c) = f(a,b,c) \qquad MGU = \{x \leftarrow c, y \leftarrow a\}$
2. $f(x,y) \sqcap f(h(y),g(a)) = f(h(g(a)),g(a)) \qquad MGU = \{x \leftarrow h(g(a)), y \leftarrow g(a)\}$
3. $f(x) \sqcap f(y) = f(z) \qquad MGU = \{x \leftarrow z, y \leftarrow z\}$
4. $f(x,b) \sqcap f(b,a)$ (nicht unifizierbar)

Abbildung 5: Beispiele zur Unifikation

1.4.2 Negation und Implikation

Eine weitere interessante Operation in $2^{(\mathcal{A})}$ ist die Negation. Sie wird meistens als $\neg X$ notiert. Ihre Semantik ist jedoch je nach Autor unterschiedlich definiert. Das soll an dieser Stelle kurz erörtert werden.

Negation als Nicht-Subsumierbarkeit: Einige Autoren gehen im Prinzip von klassischer Negation aus, so [Kasper 88, 233] und [Smolka 89, 28]. Der Ausdruck $\neg X$ steht dann für alle die Merkmalstrukturen, die nicht von X subsumiert werden. Negation in diesem Sinne entspricht also Nicht-Subsumierbarkeit. Der Vorteil der klassischen Negation ist, daß dadurch eine Boolesche Algebra entsteht. Es gelten dann die Komplementgesetze

$X \sqcap \neg X = \bot$ und $X \sqcup \neg X = \top$.

Daraus folgt, daß auch die Regeln von De Morgan gelten, die nützlich zur Vereinfachung von Formeln mit Merkmalstrukturen sind. Das Problem ist jedoch, daß diese Art der Negation nicht in der Menge $2^{(\mathcal{A})}$ definiert werden kann. Denn alle Mengen $X \in 2^{(\mathcal{A})}$ charakterisieren im Prinzip Mengen $\bigcup_{x \in X} \underline{x} \subseteq \mathcal{A}$. Jedes Element aus $x \in X$ meint sozusagen alle von x subsumierten Elemente mit. Es ist nicht möglich, einen Teil der subsumierten Elemente wieder herauszunehmen.

Negation als Nicht-Unifizierbarkeit: Weitere Vorschläge, die Negation zu definieren, finden sich in [DawVij 89] und [Carpenter 90, 136-143]. In dem erstgenannten Papier ist eine dreiwertige Logik vorgesehen, im anderen wird die Negation mittels Ungleichungen realisiert. In den Ungleichungen ist festgehalten, welche Knoten nicht miteinander identifiziert bzw. unifiziert werden dürfen. Negation in diesem Sinne entspricht Nicht-Unifizierbarkeit. Ein Nachteil bei den letzteren Vorgehensweisen ist jedoch, daß nur das erste, nicht aber das zweite Komplementgesetz gilt und somit die Regeln von De Morgan nicht angewendet werden können. Dafür ist aber die Formalisierung in der Menge $2^{(\mathcal{A})}$ möglich.

Wünschenswert wäre eine Definition für die Negation, die die Vorteile beider Begriffe vereint. Dies soll im folgenden versucht werden. Dabei wird zunächst von der Negation im Sinne von Nicht-Unifizierbarkeit ausgegangen. Die Geltung der Komplementgesetze kann dann durch eine Annahme erreicht werden.

Definition 13: Die Negation eines Elementes $X \in 2^{(\mathcal{A})}$, geschrieben als $\neg X$, ist definiert als:

$$\neg X = \bigsqcup_{X \sqcap Y = \bot} Y$$

Es ist $\neg X \in 2^{(\mathcal{A})}$ nur dann, wenn X nur einfache abstrakte Merkmalstrukturen ohne Koreferenzen enthält. □

Die Geltung der Komplementgesetze wird erreicht, indem angenommen wird, daß die Typenhierarchie $(\mathcal{S}, \preceq)$ *vollständig* ist (ohne Beweis). Das soll heißen, jede Merkmalstruktur gehört letztendlich zu genau einem minimalen Typ $t \in \mathcal{S}$. Das wird im folgenden nun genauer definiert.

Definition 14: Die Vollständigkeit einer Typenhierarchie $(\mathcal{S}, \preceq)$ anzunehmen heißt, die folgenden Identitäten zu akzeptieren:

1. $t = s_1 \sqcup \ldots \sqcup s_k$, falls $t, s_1, \ldots, s_k \in \mathcal{S}$ und die Menge $\{s_1, \ldots, s_k\}$ genau die direkten Subtypen von t umfaßt. Das entspricht einer Art Closed World Assumption (CWA): Jeder Typ wird erschöpfend durch seine Subtypen abgedeckt [Zajac 92, 166].

2. $s[l\ t] = s$ (Matrixnotation) mit $s, t \in \mathcal{S}$ und $l \in \mathcal{L}$, falls $\mathit{Approp}(s, l) = t$. Mit anderen Worten: Redundante Merkmale ohne Information sind zu ignorieren.

Jede Merkmalstruktur kann durch die obigen Gleichheiten *vereinfacht* werden. Man kann leicht ein Inferenzverfahren angeben, das Disjunktionen der angegebenen Art und redundante Merkmale entfernt. Ersetzt werden sie in beiden Fällen durch einen Typ. □

Definition 15: Die *Implikation* wird mit Hilfe der Negation und Disjunktion definiert. Als Operator-Symbol wird $\Rightarrow$ verwendet. Es gilt:

$$X \Rightarrow Y = \neg X \sqcup Y$$

Das entspricht der klassischen Definition der Implikation. □

Die Anwendung von Negation und Implikation wirft in der Praxis noch eine ganze Reihe von Problemen auf. Sie hängen mit Intensionalität und Extensionalität zusammen. Das soll aber erst im letzten Abschnitt dieses Kapitels erörtert werden (siehe Seite 27).

1.4.3 Merkmalterme und ihre Eigenschaften

Nun soll eine neue Form von Ausdrücken eingeführt werden: die Merkmalterme. Es handelt sich dabei um Formeln, in denen alle Elemente des HPSG-Formalismus vorkommen können. Sie beschreiben Mengen von Merkmalstrukturen, genauer gesagt bezeichnen sie Mengen $X \in 2^{(\mathcal{A})}$ von Merkmalstrukturen, die allesamt

- wohlgetypt (nach Definition 9),
- extensional (nach Definition 10) und
- vereinfacht (nach Definition 14)

sind. Falls ein Element in X nicht die oben genannten Bedingungen erfüllt, kann es durch Inferenzverfahren auf die gewünschte Form gebracht werden. Eine der ersten Arbeiten, die Merkmalterme formal definieren ist [Smolka 89, 27-34]. Merkmalterme sind wesentliches Ausdrucksmittel zur Formulierung von HPSG-Grammatiken.

Definition 16: *Merkmalterme* entsprechen der bereits eingeführten Matrixnotation für Merkmalstrukturen. In ihnen dürfen auch beliebig verschachtelt Operatoren (z.B. Disjunktion, Negation) auftreten. Merkmalterme unterscheiden sich von der Matrixnotation jedoch in den folgenden Punkten:

1. Es wird im allgemeinen kein Unterschied gemacht zwischen der Operation Unifikation und der Aneinanderfügung von Merkmalen zu einer Merkmalstruktur.
2. Statt eingerahmten Zahlen zur Kennzeichnung von Koreferenzen können Variablen X, Y, Z usw. verwendet werden. Variablen gleichen Namens stehen für ein und denselben Merkmalterm, überall wo sie vorkommen.
3. Werte in Merkmalstrukturen können funktional voneinander abhängig sein. Die Abhängigkeiten können durch Funktionen oder Relationen beschrieben werden. Das wird an geeigneter Stelle noch genauer definiert. □

Es lassen sich nun Theoreme für Merkmalterme formulieren, die nützlich beim Umgang mit Merkmaltrukturen sind. Eine Übersicht zeigt die Abbildung 6. Sie ist aus [Stolzenburg 92, 13] übernommen. Dabei steht a für einen Attributnamen.

Idempotenz:

$s \sqcap s = s \qquad s \sqcup s = s$

Kommutativität:

$s \sqcap t = t \sqcap s \qquad s \sqcup t = t \sqcup s$

Assoziativität:

$s \sqcap (t \sqcap u) = (s \sqcap t) \sqcap u \qquad s \sqcup (t \sqcup u) = (s \sqcup t) \sqcup u$

Definition der Subsumption $\sqsubseteq$:

$s \sqsubseteq t \iff s \sqcap t = s \iff s \sqcup t = t$

Generalisierung:

$[a : s] \sqcap [a : t] = [a : (s \sqcap t)] \qquad [a : s] \sqcup [a : t] = [a : (s \sqcup t)]$

Distributivität:

$(s \sqcup t) \sqcap u = (s \sqcap u) \sqcup (t \sqcap u) \qquad (s \sqcap t) \sqcup u = (s \sqcup u) \sqcap (t \sqcup u)$

Absorption:

$(s \sqcap t) \sqcup s = s \qquad (s \sqcup t) \sqcap s = s$

Regeln von De Morgan:

$\neg(s \sqcup t) = \neg s \sqcap \neg t \qquad \neg(s \sqcap t) = \neg s \sqcup \neg t$

Doppelte Negation:

$\neg\neg s = s$

Null- und Einselement:

$s \sqcap \top = s \qquad s \sqcup \bot = s$

Komplement-Regeln:

$s \sqcup \neg s = \top \qquad s \sqcap \neg s = \bot$

Definition der Implikation:

$s \Rightarrow t = \neg s \sqcup t$

Abbildung 6: Theoreme über Merkmalterme

1.5 Logikprogramme und Hornklausellogik

1.5.1 Logische Formeln und Programme

Im folgenden sollen Logikprogramme im Sinne von PROLOG definiert werden. Die Sprache UBS kann dann als erweitertes Logikprogramm angesehen werden. Einem Logikprogramm zugrunde liegen logische Formeln, die das zu lösende Problem beschreiben.

Definition 17: Eine *logische Formel* wird induktiv wie folgt definiert:

1. Falls p ein n-stelliges Prädikatensymbol ist und $t_1, \ldots, t_n$ Terme sind, ist $p(t_1, \ldots, t_n)$ eine Formel, auch *Atom* genannt. Dabei heißen $t_1, \ldots, t_n$ auch *Argumente* des Prädikats.
2. Falls F und G Formeln sind, so auch $\neg F$ (Negation), $F \wedge G$ (Konjunktion), $F \vee G$ (Disjunktion) und $F \leftarrow G$ (Implikation).
3. Falls F eine Formel und x eine Variable ist, dann sind $\forall x F$ (Allquantifizierung) und $\exists x F$ (Existensquantifizierung) ebenfalls Formeln. [Lloyd 84, 6] □

Durch gewisse Umformungsschritte, die in [Schöning 87, 59-68] ausführlich dargelegt werden, kann jede logische Formel in die sogenannte *Klauselform* überführt werden. Einfache Logikprogramme bestehen aus einer ganz bestimmten Art von Klauseln, Hornklauseln genannt.

Definition 18: Eine *Hornklausel* ist eine logische Formel mit folgendem Aufbau:

$$\forall x_1 \ldots \forall x_m (A \leftarrow B_1 \wedge \ldots \wedge B_n)$$

Dabei sind A und die B_i Atome. Außerdem gilt $m, n \geq 0$. Die abkürzend verwendete Schreibweise

$$A \leftarrow B_1, \ldots, B_n$$

heißt *Programmklausel*. Falls $n = 0$ ist, heißt sie *Einheitsklausel*. Ein *Logikprogramm* ist eine endliche Sequenz von Programmklauseln. [Lloyd 84, 8] □

In PROLOG wird statt des Operators $\leftarrow$ das Symbol `:-/2` verwendet, das bei Einheitsklauseln ganz wegfällt. Außerdem werden Programmklauseln immer mit einem Punkt abgeschlossen. Der Benutzer kann Anfragen an das Programm stellen, indem er eine sogenannte *Zielklausel* G der Form

$$\leftarrow A_1, \ldots, A_n$$

vorgibt. Dadurch wird die Anfrage gestellt, ob die Formel

$$\exists x_1 \ldots \exists x_m (A_1 \wedge \ldots \wedge A_n)$$

eine logische Konsequenz des Programms ist. Dabei sind $x_1, \ldots, x_m$ genau die in den Atomen $A_1, \ldots, A_n$ vorkommenden Variablen.

1.5.2 Ableitungen in Logikprogrammen

Wie bestimmmt nun ein Logikprogramm die Antwort zu einer Anfrage? Was soll überhaupt unter einer Antwort verstanden werden? – Das Verfahren, die Lösungen zu ermitteln, heißt SLD-Resolution und führt die Ableitung der Lösung in mehreren Schritten

durch. Es soll nun durch die folgenden Definitionen nach [Lloyd 84, 36-38] beschrieben werden.

Definition 19: Sei $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_n$ eine Zielklausel, $C_{i+1} = A \leftarrow B_1, \ldots, B_l$ eine Programmklausel und R eine Berechnungsregel. Dann heißt G_{i+1} aus G_i und C_{i+1} unter Anwendung der Substitution θ_{i+1} mittels R in einem Schritt *abgeleitet*, falls gilt:

1. A_m ist das durch die Berechnungsregel R ausgewählte Atom aus der Zielklausel.

2. θ_{i+1} ist der allgemeinste Unifikator von A und A_m.

3. $G_{i+1} = \leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_l, A_{m+1}, \ldots, A_n)\theta_{i+1}$ ist die neue Zielklausel.

G_{i+1} heißt auch *Resolvent* von G_i und C_{i+1}. □

Definition 20: Sei P ein Programm, G eine Zielklausel und R eine Berechnungsregel. Eine (endliche) *SLD-Ableitung* von $P \cup \{G\}$ mittels R besteht aus den Sequenzen

1. $(G = G_0), G_1, \ldots, G_n$ (Zielklauseln),

2. $C_1, \ldots, C_n$ (Varianten von Programmklauseln aus P) und

3. $\theta_1, \ldots, \theta_n$ (allgemeinste Unifikatoren),

so daß jedes G_i $(0 < i \leq n)$ in einem Schritt aus G_{i-1} und C_i unter der Anwendung der Substitution θ_i mittels R abgeleitet ist. Eine *SLD-Widerlegung* von $P \cup \{G\}$ mittels R ist eine endliche SLD-Ableitung, das die leere Klausel als letzte Zielklausel besitzt. Die *leere Klausel* □ entspricht der leeren Menge $\emptyset$. □

Definition 21: Die mittels R berechnete *Antwortsubstitution* θ von $P \cup \{G\}$ ist die Komposition aller Substitutionen $\theta_1 \ldots \theta_n$ aus einer SLD-Widerlegung von $P \cup \{G\}$ mittels R, eingeschränkt auf die in G vorkommenden Variablen. Das Kalkül ist korrekt und vollständig, unabhängig von der Berechnungsregel R. □

Die Programmiersprache PROLOG kann als Logikprogramm aufgefaßt werden, wobei die Berechnungsregel R immer das vorderste Atom in der Zielklausel auswählt. In SEPIA kann an der Berechnungsregel R manipuliert werden, indem Prädikate mit `delay`-Klauseln versehen werden. Dazu folgen nähere Ausführungen in Kapitel 3 unter dem Stichwort Relationen (siehe Abschnitt 3.4.2).

Wir wollen nun ein Beispiel betrachten, und zwar die vielzitierte Funktion `append`/3.[5] Sie dient zur Konkatenation zweier Listen, und zwar soll `append(FIRST,SECOND,RESULT)` genau dann gelten, wenn die beiden Listen `FIRST` und `SECOND`, in dieser Reihenfolge aneinandergefügt, die Liste `RESULT` ergeben (können). Im folgenden wird eine Definition von `append`/3 in PROLOG angegeben, anschließend eine Ableitung dazu, ausgehend von der Zielklausel `G = append([a],[b],L)`.

```
append([],LIST,LIST).
append([ELEM|LIST1],LIST2,[ELEM|RES]) :-
     append(LIST1,LIST2,RES).
```

[5]Die Schreibweise $/n$ hinter einem Prädikatsnamen gibt dessen Stelligkeit an.

```
G0 = append([a],[b],L)
G1 = append([],[b],L')  θ1 = {L←[a|L']}
G2 = □                  θ2 = {L'←[b]}

                        θ = θ1θ2 = {L←[a,b]}
```

1.5.3 Die kürzeste Definition von UBS

Die Sprache UBS läßt sich nun in aller Kürze wie folgt definieren:

Definition 22: Ein Programm in UBS ist ein Logikprogramm, bei dem die Atome neben einfachen Termen beliebige Merkmalterme als Argumente haben können. □

Die Ausführungen in den nächsten Kapiteln sind dem Problem gewidmet, wie die Integration von Merkmaltermen in Logikprogramme tatsächlich implementiert werden kann.

1.6 Anwendungen in HPSG

Nun soll ein kurzer Überblick erfolgen, wofür die eingeführten Begriffe überhaupt nützlich sind und wo sie tatsächlich in der linguistischen Theorie von HPSG zum Einsatz kommen. Die Nennung konkreter Beispiele – aus der Literatur oder selbst ausgedacht – dient zur Illustration sowohl der linguistischen Theorie als auch des Formalismus von HPSG. Dadurch kann der Leser beurteilen, inwieweit die linguistischen Phänomene durch den Formalismus dem jeweiligen Problem angemessen, unzureichend oder gar zu kompliziert beschrieben werden.

1.6.1 Die Bestandteile einer Grammatik

Typenhierarchie: HPSG ist eine der ersten Theorien in der Computerlinguistik – möglicherweise die einzige [Zajac 92, 159] – die ausdrücklich eine Typisierung von Merkmalstrukturen vorschreibt. Das führt dazu, daß eine Grammatik immer strukturiert ist und dadurch Übersichtlichkeit gewährleistet werden kann. Es handelt sich im weitesten Sinne um eine objektorientierte Entwicklung von Grammatiken. Jede Merkmalstruktur gehört zu einem bestimmten Typ, der angibt, welche Art von Objekt durch die Merkmalstruktur beschrieben werden soll.

Jeder Typ besitzt ganz bestimmte Merkmale. Auch stehen die Typen in vielfältigen Relationen zueinander. Die wichtigste ist sicherlich die Vererbungsrelation. Aber auch andere Beziehungen sind denkbar, etwa die, daß zwei Typen disjunkt sind. – Pollard und Sag erlauben ganz allgemein Relationen zwischen Typen, die durch Gleichungen beschrieben werden können, in denen neben den Typen selbst auch Verknüpfungen der Typen durch Operationen wie die Unifikation und Disjunktion zugelassen sind [PolSag 87, 197].

Die Typenhierarchie von HPSG ist im wesentlichen eine einfache Hierarchie. Nur im Lexikon wird intensiv Gebrauch gemacht von multipler Vererbung. *Multiple Vererbung* ist dann gegeben, wenn keine einfache Typenhierarchie vorliegt, d.h. mindestens ein

Typ hat mehr als einen direkten Supertyp. Das soll im folgenden an einem Beispiel verdeutlicht werden:

Der Lexikoneintrag zur Wortform *likes* aus dem Englischen wird realisiert durch eine Merkmalstruktur vom Typ *likes*. Für die Wortform gilt:

- Die Wortform steht in der dritten Person Singular.
- Es handelt sich um ein Vollverb.
- Das Verb ist strikt transitiv.

Darum besitzt der Typ *likes* (mindestens) drei Supertypen, die die obigen drei Eigenschaften besagen. Sie werden in dem Typ *likes* zusammengefaßt. Man spricht in diesem Zusammenhang von Vererbung.

Merkmalterme: Zentral für HPSG ist die Verwendung von Merkmalstrukturen bzw. Merkmaltermen. Fast alle Aspekte einer Grammatik werden mit ihrer Hilfe beschrieben: sprachliche Universalien, sprachspezifische Prinzipien, Grammatikregeln und Lexikoneinträge. Das ist der Hauptunterschied zu verwandten Theorien [Cooper 90, 1-5]. Andere Theorien wie LFG, GPSG oder GB [Sells 85] benutzen zusätzlich noch weitere Mittel zur Formulierung von Grammatiken, nämlich Konstituentenbäume bzw. Transformationsregeln.

Pollard behauptet in [Pollard 91, (1) 2-4], daß Baumstrukturen zur Darstellung von syntaktischen Zusammenhängen linguistisch nicht signifikant sind. Damit lehnt er die Geltung des von Chomsky aufgestellten c-Kommandos ab. Das c-Kommando hilft bei der Suche der Bezugswörter eines Pronomens, z.B. in dem Satz

Weil <u>er</u> krank ist, kann Fritz Adele nicht besuchen.

Es wird in der Bindungstheorie von HPSG ersetzt durch das o-Kommando, das Bezug nimmt auf die Reihenfolge der Konstituenten in der Liste, die Angaben zur Subkategorisierung enthält [Pollard 91, (6) 15-17].

Koreferenzen statt Transformationsregeln: Ebenfalls kritisiert Pollard die Verwendung von Transformations- oder Metaregeln. Mit der Regel *move-*α aus der GB-Theorie soll z.B. erklärt werden, daß Fragepronomen im Vergleich zur normalen Satzstellung vorgezogen sind, etwa in dem englischen Fragesatz

What$_i$ *does Peter see* $___i$*?*

Man kann annehmen, daß der Satz aus dem folgenden Aussagesatz entstanden ist:

Peter sees $___i$.

Der Index i bezeichnet hierbei den Referenten der jeweilige Konstituente. Koindizierung besagt, daß das gleiche Individuum bzw. Objekt gemeint ist. Pollard findet, daß in solchen oder anderen Beispielen das Stattfinden einer Bewegung suggeriert wird, was empirisch nur schwer gerechtfertigt werden kann. In HPSG werden statt Transformationsregeln Koreferenzen verwendet.

Eine ähnliche Argumentation bringt Pollard für das Phänomen des Agreement in natürlichen Sprachen, d.h. Übereinstimmung von mehreren Wortformen in bestimmten Merkmalen, vor [Pollard 91, (2) 1-7]. Ein Beispiel dafür ist die Kongruenz von Subjekt und zugehörigem Verb bezüglich Numerus und Person im Deutschen, Englischen und Französischen. Im Zusammenhang von HPSG spricht man hier am besten von einem Informations*abgleich*, der durch Unifikation der relevanten Merkmale erfolgt. Z.B. teilen

sich Subjekt und Verb in dem Satz

Sie kommen.

die Information, daß es sich um Formen der dritten Person im Plural handelt. Es entsteht also eine Koreferenz. Die Sichtweise, daß hier ein Informations*transport* stattfindet – so Chomsky und seine Vertreter –, etwa vom Subjekt zum Verb, scheint nicht ganz einleuchtend. Diese Ansicht führt außerdem künstlich Mehrdeutigkeiten ein, die im Subjekt *sie* vor einem solchen Informationstransport bestehen.

Grammatikregeln: Eine Sprache kann als Menge von Merkmalstrukturen aufgefaßt werden, die alle wohlgeformten Zeichen der Sprache enthält. Eine Grammatik beschreibt nun eine Sprache mit Hilfe von Merkmaltermen. Z.B. umfaßt eine Sprache alle Merkmalstrukturen vom Typ *sign*. Diese Zeichen unterliegen gewissen Eigenschaften, die wiederum durch einen Merkmalterm spezifiziert werden können. Das bedeutet, jedem Typ ist eine Struktur (Merkmalterm) zugeordnet. Diese Struktur muß auch für alle Subtypen gelten; das wird durch Unifikation erreicht.

In [PolMos 90, 300-306] wird formal präzise beschrieben, was unter einer Grammatik verstanden werden soll. Sie ist definiert als

$$G = \{(t, m) \mid t \in \mathcal{S}\}$$

mit m ist ein Merkmalterm. Die Merkmalterme m beschreiben erschöpfend in disjunktiver Form, was es für Möglichkeiten gibt, eine wohlgeformte Struktur vom Typ s zu sein. Die Ausdrücke m sind zu unifizieren mit allen m', für die gilt: $(t', m') \in G$ und $t \preceq t'$. Eine Grammatik kann Rekursionen enthalten, d.h. Typen können unter Zuhilfenahme anderer Typen beschrieben werden.

Meist reichen Merkmalterme allein aber nicht aus, um eine Grammatik in befriedigender Weise vollständig zu formulieren. Man benötigt zusätzlich so etwas wie Funktionen oder Relationen, etwa um das Aneinanderfügen zweier Listen (*append*) formulieren zu können; dies wird nämlich im Rahmen des Subkategorisierungsprinzips verlangt. Das führt nun dazu, daß die Merkmalterme in eine funktionale, relationale oder logische Programmiersprache eingebettet werden sollten. Das genau geschieht in UBS. In [DörSei 91, 10-16] wird sogar ganz auf Typen(hierarchien) verzichtet zugunsten von Relationen. Typen und Relationen sind tatsächlich in einem gewissen Sinne komplementäre Begriffe.

1.6.2 Mehrere Begriffe von Implikation und Negation

So unterschiedlich die formalen Definitionen der Negation – und meist damit zusammenhängend – der Implikation je nach Autor ausfallen, so verschiedenartig sind auch die Anwendungen der beiden Operationen. Offensichtlich liegen zum Teil auch verschiedene Begriffe zugrunde. Das soll der Gegenstand dieses Abschnitts sein.

Implikation: Die Implikation ist mit Rückgriff auf Negation und Disjunktion definiert worden. Das entspricht einem *klassischen* Begriff der Implikation. Dabei ist natürlich zu beachten, daß die Negation nicht klassisch definiert ist. Die Implikation verknüpft zwei Merkmalterme miteinander. Das ist das Gemeinsame der nun vorzustellenden drei Begriffe von Implikation in HPSG:

1. Die Implikation gilt normalerweise nur *lokal* in einer Merkmalstruktur, nämlich dort, wo sie auftritt. Man betrachte nun das folgende Beispiel. Es ist aus

[DawVij 89, 19] übernommen, in dem im Prinzip der in UBS verwendete Begriff von Negation und Implikation definiert wird.

$$\begin{bmatrix} \text{SUBJ:REF: } X \\ \text{OBJ:REF: } X \end{bmatrix} \Rightarrow \begin{bmatrix} \text{OBJ:TYPE: } reflexive \end{bmatrix}$$

Diese Implikation besagt, daß immer dort – und nur dort –, wo sie auftritt, eine Merkmalstruktur mit zumindest dem Attribut OBJ vorliegt. Dabei muß das Objekt ein Reflexivpronomen sein, falls Subjekt und Objekt den gleichen Referenten haben. Das etwa ist der Inhalt dieser Implikation, natürlichsprachlich ausgedrückt.

2. Das Gebiet, in dem die Implikation in HPSG hauptsächlich angewendet wird, sind die universalen oder auch nur sprachspezifischen Prinzipien. Es handelt sich dabei um *typgebundene Bedingungen*, d.h. auf der linken Seite der Implikation steht ein Typ, auf der rechten ein beliebiger Merkmalterm. Das trifft z.B. auf das HFP zu, hier zitiert nach [Cooper 90, 10]. Auf die Bedeutung der Attribute innerhalb der linguistischen Theorie von HPSG soll an dieser Stelle nicht näher eingegangen werden; der interessierte Leser sei verwiesen auf [Stolzenburg 92, 3], [PolSag 87, 51-80] und [Pollard 91, (1) 14-18].

$$phrase\begin{bmatrix} \\ \end{bmatrix} \Rightarrow \begin{bmatrix} \text{SYNSEM:LOC:CAT:HEAD: } \boxed{3} \\ \text{DTRS:HEAD-DTR:SYNSEM:LOC:CAT:HEAD: } \boxed{3} \end{bmatrix}$$

Im Gegensatz zum obigen Begriff der Implikation gilt das HFP und alle anderen typgebundenen Bedingungen *global*: Jede Merkmalstruktur vom Typ *phrase*, egal wo sie vorkommt, muß die oben ausgedrückte Übereinstimmung von HEAD-Merkmalen aufweisen. Solche typgebundenen Bedingungen $s \Rightarrow m$ sind als Bestandteile einer Grammatik G – wie weiter oben definiert – aufzufassen. Es ist dann $(s, m) \in G$ für alle typgebundenen Bedingungen $s \Rightarrow m$.

3. Implikationen werden außerdem in den sogenannten *Lexikonregeln* verwendet. Sie vermeiden redundante Angaben in den Lexikoneinträgen. Zwei Arten von Redundanz können unterschieden werden nach [PolSag 87, 209]:

vertikal: Lexikoneinträge teilen Eigenschaften mit ganzen Wortklassen: Sie sind Vollverben oder transitiv etc. Diese Art von Redundanz wird durch multiple Vererbung und dem Aufbau einer geeigneten Typenhierarchie beseitigt (siehe Seite 24).

horizontal: Zwischen Wortformen können auch andere Beziehungen bestehen: Sie können durch Flexion oder Derivation auseinander hervorgangen sein. Das kann mit Hilfe der noch einzuführenden Lexikonregeln ausgedrückt werden (siehe unten).

Die folgende Lexikonregel für das Englische nach [PolSag 87, 210] beschreibt den Zusammenhang zwischen der Grundform eines Verbs und der entsprechenden Form in der dritten Person Singular. Dabei ist $flex$ eine einstellige Funktion,

die die richtige Wortform erzeugt. In den meisten Fällen ist einfach nur ein *s* für die dritte Person anzuhängen, z.B. *take* wird zu *takes*.

$$base \begin{bmatrix} \text{PHON: } \boxed{1} \\ \text{SYN:LOC:SUBCAT: } \boxed{2} \\ \text{SEM:CONT: } \boxed{3} \end{bmatrix} \mapsto 3rdsng \begin{bmatrix} \text{PHON: } flex(\boxed{1}) \\ \text{SYN:LOC:SUBCAT: } \boxed{2} \\ \text{SEM:CONT: } \boxed{3} \end{bmatrix}$$

Zu beachten ist, daß diese Regel vereinfacht dargestellt ist. Sie ist zu modifizieren, um die unregelmäßigen Verben korrekt behandeln zu können. Dazu werden den Lexikoneinträgen neue Merkmale hinzugefügt, die die abweichenden Formen, z.B. *has* von *have*, beinhalten [PolSag 87, 213]. Wegen der neuen Merkmale sind konsequenterweise neue Typen zu kreieren.

Diese dritte Art von Implikation unterscheidet sich wiederum grundlegend von den beiden vorhergehenden: Sie leitet aus bestehenden Lexikoneinträgen die Existenz weiterer, nicht explizit erwähnter Einträge ab. Das entspricht mehr der Implikation in der Logik bzw. der Logischen Programmierung. Wie zu sehen ist, wird auch ein anderes Zeichen für den Operator benutzt, nämlich $\mapsto$ statt $\Rightarrow$.

Negation: Obgleich Pollard die Negation in [PolSag 87, 45] vorstellt, findet sie praktisch keine Anwendung in HPSG. Sie dient höchstens zur Schreibvereinfachung bei sonst komplexen Disjunktionen (wie im Beispiel unten). Trotz dieser Seltenheit der Negation soll an dieser Stelle nochmals auf sie eingegangangen werden.

In Definition 13 ist die Negation als Nicht-Unifizierbarkeit festgelegt worden. Das entspricht den Vorgehensweisen in [DawVij 89] und [Carpenter 90, 136-154]. Und doch unterscheiden sich beide Begriffe. Dazu soll nun das folgende – fast klassische – Beispiel für die Negation betrachtet werden:

$$\neg index \begin{bmatrix} \text{PER: } 3rd \\ \text{NUM: } sng \end{bmatrix}$$

In UBS wird ähnlich wie bei [DawVij 89, 21] die obige Negation in eine Disjunktion überführt. Eine Merkmalstruktur, die mit dem obigen Merkmalterm unifizierbar ist, muß entweder von einem Typ sein, der nicht mit *index* unifizierbar ist oder sonst in mindestens einem Wert der beiden Attribute PER bzw. NUM nicht mit der obigen Struktur übereinstimmen. Das läßt sich formal so angeben:

$$\neg index \sqcup index \begin{bmatrix} \text{PER: } \neg 3rd \sqcup \text{NUM: } \neg sng \end{bmatrix}$$

Wenn der Typ *index* aber als intensional definiert ist, so ergibt sich ein Problem: Die Intensionalität wird nämlich gar nicht berücksichtigt bei der Auflösung der Negation in die obige Disjunktion.

Die Intensionalität einer Merkmalstruktur kann durch eine Variable (eingerahmte Zahl in der Matrixnotation) erfaßt werden. Zajac sieht darum eine typisierte Merkmalstruktur als besthend aus drei Bestandteilen an: einer Variablen zur Identifikation der Struktur, einem Typ und den eigentlichen Merkmalen [Zajac 92, 164].

$$\boxed{\text{X}}\ typ \begin{bmatrix} a_1 : t_1 \\ \ldots \\ a_n : t_n \end{bmatrix}$$

Jede Merkmalstruktur ist genau dann gleich einer anderen, wenn beide durch ein und dieselbe Variable identifiziert werden; sonst liegt noch keine Gleichheit vor. Carpenter beschreibt die Negation darum durch eine Ungleichungsrelation, die zwischen Knoten, die den Variablen hier entsprechen, gilt [Carpenter 90, 139-140]. Daher ist der obigen Disjunktion noch eine Ungleichung anzufügen, etwa $Y \neq X$, falls die negierte Merkmalstruktur mit der Variablen X und die gesamte Negation mit der Variablen Y identifiziert wird.

Im Grunde genommen erübrigt sich dann die obige umformung in die Disjunktion sogar ganz, da die Ausdrücke eindeutig und vollständig durch die Variablen X und Y charakterisiert sind. Merkmalstrukturen sind sozusagen bezüglich ihrer zugeordneten Variable extensional. Diese Art der Negation könnte ganz anders implementiert werden, als dies gegenwärtig in UBS geschieht.

1.6.3 Die Verwendung von Listen und Mengen

Listen: Listen finden in HPSG vornehmlich Verwendung im Wert des Attributs SUBCAT. Bei Verben gibt dieser Wert die Valenzpartner an, etwa für das Verb *to give*

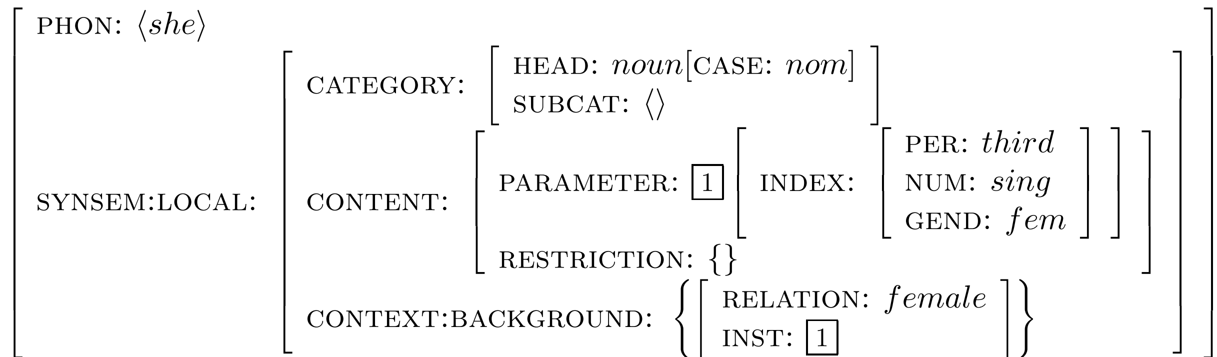
$$\langle \mathrm{NP}_{\textsc{subject}}, \mathrm{NP}_{\textsc{direct object}}, \mathrm{NP}_{\textsc{object 2}} \rangle.$$

Die Reihenfolge der Elemente ist bei Listen im Gegensatz zu Mengen signifikant. Sie spiegelt im obigen Fall die sogenannte Obliqueness-Hierarchie wieder [PolSag 87, 117-121]. Diese Hierarchie wird herangezogen, um bestimmte Wortstellungsphänomene zu erklären [Sag 86]. Listen werden auch im Attribut DTRS benutzt; sie sind dann eine Folge von Konstituenten, die direkten Nachfolger in einer Baumstruktur [Stolzenburg 92, 3-5].

Mengen: In der linguistischen Theorie von HPSG gibt es einige mengenwertige Attribute, speziell die Attribute RESTRICTION und BACKGROUND. Das Attribut RESTRICTION enthält eine Reihe von Angaben zur Semantik eines Wortes, etwa daß das Wort *Buch* eine Objektinstanz bezeichnet, die die Eigenschaft hat, ein Buch zu sein. Das wird üblicherweise durch logische Formeln, gegebenenfalls mit Konjunktionen $\wedge$ ausgedrückt, etwa $book(X)$. In HPSG werden hier Mengen von Merkmalstrukturen des Typs *parametrized-states-of-affairs* verwendet, im Falle des Wortes *Buch* nach [Pollard 91, (1) 17] etwa:

$$\left\{ \begin{bmatrix} \textsc{relation} & \mathit{book} \\ \textsc{instance} & X \end{bmatrix} \right\}$$

Das Attribut BACKGROUND hat als Wert eine Menge von gleicher Struktur wie oben, die aber hier als Angaben zur Pragmatik zu verstehen sind. Die Angaben lassen sich auch unter die Begriffe Präsupposition oder konventionelle Implikatur einordnen. Z.B. impliziert das Pronomen *she* im Englischen, das sein Referent weiblich ist. Das ergibt sich aus der Tatsache, daß Englisch eine Sprache mit natürlichem Geschlecht ist. Im Deutschen oder Französischen – beides Sprachen mit grammatischem Geschlecht – läßt sich diese Schlußfolgerung nicht unbedingt ziehen. Hier ein Ausschnitt der Merkmalstruktur zum Wort *she* [Pollard 91, (1) 13]:



Eine weitere, wichtige Anwendung von Mengen ist im Zusammenhang mit einer sogenannten Unbounded Dependency Construction (UDC). Das ist z.B. bei der Analyse von topikalisierten, Frage- oder Relativsätzen von Bedeutung. Eine UDC liegt vor, wenn ein oder mehrere Konstituenten in einem Satz an einer anderen Stelle auftreten, als es die üblichen Syntaxregeln erwarten lassen. Diese Konstituenten werden in einer Menge gemerkt. Besonders gut paßt dieser Ansatz für das Englische, da hier im allgemeinen ein strenger Satzbau vorliegt. Ein Beispiel ist der Satz

This is a problem which$_1$ *John*$_2$ *is difficult to talk to* $___2$ *about* $___1$.

Hier treten gleich zwei Konstituenten an anderer Stelle auf.

Eine interessante Beobachtung ist es, daß, wenn Mengen verwendet werden, sie meist nur wenige Elemente enthalten (höchstens zwei). Die Operation Unifikation von Mengen wird im Vergleich zur Vereinigung von Mengen auch nur sehr selten wirklich benötigt. Darum stellt sich die Frage, inwieweit Mengen überhaupt für HPSG notwendig sind, stellen sie einen doch vor schwierige theoretische und praktische Probleme.

Mengen könnten aber verwendet werden bei Sprachen mit freier Wortstellung, da bei Mengen im Gegensatz zu Listen die Reihenfolge der Elemente keine Rolle spielt. Als Rahmen ist ein ID/LP-Formalismus denkbar, bei dem die rechte Seite der ID-Regeln Mengen von Konstituenten sind, deren Reihenfolge durch LP-Regeln festgelegt wird [Sells 85, 84-86]. Möglicherweise stellen aber auch in diesem Fall Listen die bessere Datenstruktur dar; so ist dies zum Beispiel im Ansatz von Reape vorgesehen [ReaHep 89].

In HPSG werden Mengen noch im Zusammenhang mit linguistischen Quantoren (Attribut QSTORE) benötigt.

2 Die Sprache UBS

2.1 Was man unbedingt wissen muß

2.1.1 Der Start ...

Die neue Version von UBS ist in SEPIA geschrieben. Diese stark erweiterte Version von PROLOG wird in [SEPIA 91] beschrieben. Es läuft – unter anderem – auf SUN-Rechnern. Weil Programme in UBS auf SEPIA abgebildet werden, benötigt man zum Starten von UBS

- das System SEPIA (Version 3.2) und
- einen SUN-Rechner (Version 3 oder 4).

Das System SEPIA muß nun installiert und gestartet werden. Sodann ist UBS in das Modul `ubs` zu compilieren durch den untenstehenden Aufruf. Er compiliert die Datei `ubs.pl` in die Wissensbasis von SEPIA und schaltet in den Coroutining Modus [SEPIA 91, (1) 97-103].

```
compile(ubs).
```

2.1.2 ... und wie es weitergeht

Jetzt stehen dem Benutzer einige Prädikate global zur Verfügung: Er kann in UBS geschriebene Programme in die Wissensbasis laden, und es stehen alle UBS-spezifischen Konstrukte zur Verfügung. In UBS müssen Dateinamen auf das Suffix `.ubs` enden.[6] Der folgende Aufruf lädt das in UBS geschriebene Programm namens `hpsg.ubs` ein:

```
ld(hpsg).

hpsg.ubs   4176 bytes transformed into 6850 bytes prolog
hpsg.pl    compiled traceable 21164 bytes in 0.63 seconds
```

Das System liefert die oben mit angegebenen Meldungen. Sie besagen, daß die die erstgenannte Zahl an Bytes umfassende Datei `hpsg.ubs` transformiert worden ist nach PROLOG (genauer SEPIA). Das Ergebnis der Transformation wird in der Datei `hpsg.pl` festgehalten, deren Länge in Bytes ebenfalls gemeldet wird (zweite Angabe). Im nächsten Schritt wird diese Zwischendatei durch das System SEPIA compiliert und schließlich in die Wissensbasis eingeladen.

Die Datei `hpsg.pl` kann nun vom Benutzer angeschaut und gegebenenfalls weiter bearbeitet werden. Diese Zwischenstufe war in UBS* nicht vorgesehen. Dort wurde das Programm direkt und uncompiliert in die Wissensbasis geladen. Mit dem Befehl `cp/1` (anstelle von `ld/1`) ist es sogar möglich, nur die Zwischendatei zu erzeugen. Der letzte Schritt von oben entfällt dann.

Eine Bemerkung zu dem obigen Beispiel sei noch gestattet: Es ist deutlich zu erkennen, daß sich die Programmlänge durch die Transformation und nachfolgende Compilation

[6] Im Gegensatz zu UBS* darf das Suffix im Ladebefehl auch gar nicht mehr angegeben werden.

in Instruktionen einer abstrakten Maschine, die emuliert wird [SEPIA 91, (1) 23], stark vergrößert. Im allgemeinen gilt für die Längen die Relation

```
UBS < SEPIA < ABSTRAKT.
```

Diese Beobachtung läßt die folgende Interpretation zu: Die Sprache UBS ist eine sehr abstrakte Programmiersprache, die eine äußerst kompakte Formulierung von Programmen erlaubt. Das beschleunigt die Entwicklung von komplexen Programmen bzw. Grammatiken.[7]

2.2 Die Syntax

Die Herangehensweise, UBS als Erweiterung von PROLOG, hier genauer gesagt SEPIA, zu implementieren, ist beibehalten worden. Programme in UBS haben deshalb im Prinzip die gleiche Form wie Programme in SEPIA, nur daß in den Argumenten der Prädikate die Elemente des HPSG-Formalismus verwendet werden können.

2.2.1 Die Struktur von Argumenten

Die Syntax für Argumente in UBS ist in Abbildung 7 nach [Stolzenburg 92, 11] zusammengefaßt. Die Syntax ist im wesentlichen die gleiche wie in UBS*. Sie soll hier trotzdem noch einmal ausführlich beschrieben werden.

Terme und Listen: Die Syntax für Terme und Listen (`TERM` bzw. `LIST`) ist identisch mit der üblichen PROLOG-Syntax. Daher ist in UBS nun auch die Benutzung offener Listen möglich. Die Einklammerung von Listen in spitze Klammern aus UBS* [Stolzenburg 92, 10], die teilweise nur Verwirrung stiftete, entfällt.

Mengen: Für Mengen ist die Syntax in ähnlicher Weise wie für Listen vereinfacht worden. Die zusätzliche Klammerung der Elemente der Menge durch eckige Klammern in UBS* ist nun überflüssig. Die Elemente werden durch Kommata getrennt, zwischen geschweiften Klammern aufgelistet; der Ausdruck `{}` bezeichnet die leere Menge. Es können nur endliche Mengen definiert werden. Mit dem Aufruf

```
set(LIST,SET)
```

können zur Laufzeit Mengen in Listen umgewandelt werden und auch umgekehrt. Dabei darf `LIST` keine offene Liste sein.

Merkmalstrukturen: Die Syntax für Merkmalstrukturen (`AVM`) ist nach wie vor gleichgeblieben. Wie schon in GULP [Covington 89] werden Attribut (`ATTR`) – das ist ein atomarer Ausdruck im Sinne von PROLOG – und Wert durch Doppelpunkt voneinander abgesetzt. Merkmale werden durch zwei Punkte voneinander getrennt.

[7] Im übrigen ist aber sichergestellt, daß sich die Länge eines Programms bei der Transformation von UBS nach SEPIA nicht exponentiell aufbläht.

Es ist darauf zu achten, daß alle vorkommenden Attribute definiert sind und typverträglich sind. Jedes Attribut darf zudem nur einmal einen Wert zugewiesen bekommen. Es wird der minimal mögliche Typ der Merkmalstruktur angenommen. Eine Merkmalstruktur hat die Gestalt:

$\mathtt{l}_1$`:` $\mathtt{t}_1$
...
$\mathtt{l}_n$`:` $\mathtt{t}_n$

Die verwendeten Attribute $l_1, \ldots, l_n$ stammen hierbei aus der Menge $L \subseteq \mathcal{L}$; die Angemessenheit der Attribute muß gemäß Definition 8 festgelegt sein. Dann wird für sie der Typ

$$\sqcap_{l \in L} Intro(l)$$

inferiert.

Typausdruck: In UBS ist es möglich zu fordern, daß ein Argument oder eine Struktur darin von einem bestimmten Typ sein soll. Dazu sind die Typausdrücke da. Sie werden durch das Präfix `@` gekennzeichnet, das von einem Typnamen gefolgt sein muß. Man beachte, daß der Operator `@` in UBS* eine andere Bedeutung hatte. Dort wurde er nämlich für Makroaufrufe eingesetzt.

Als Typnamen sind nur ganz bestimmte Bezeichner erlaubt. Generell gilt: Typbezeichner sind entweder

- vordefiniert oder
- benutzerdefiniert.

Im ersten Fall handelt es sich um einen der Typen `avm` (Merkmalstruktur), `nil` (leere Liste), `cons` (zusammengesetzte Liste), `set` (Menge) oder `term` (Term). Alle anderen, benutzerdefinierten Typen sind sämtlich Subtypen des Typs `avm`, d.h. spezielle Merkmalstrukturen. Sie werden durch eine Typdeklaration (siehe Seite 35). eingeführt. Abbildung 8 zeigt die in UBS angesetzte Typenhierarchie.

Funktion: Ein Funktionsaufruf in UBS hat ab sofort die Form `$FUNCTION` mit

`FUNCTION = PRED(ARG`$_1$`,...,ARG`$_n$`).`

Zur Laufzeit wird dann das Prädikat

`PRED(ARG`$_1$`,...,ARG`$_n$`,RES)`

aufgerufen, dessen letztes Argument `RES` als Resultat der Berechnung gilt und an die Stelle des Aufrufs in den Programmtext eingesetzt wird.

In UBS* mußte der Benutzer Prioritäten angeben und so die Reihenfolge der Abarbeitung der Funktionsaufrufe kontrollieren [Stolzenburg 92, 22-24]. Dadurch konnte der Benutzer auf Termination und Korrektheit eines Programms Einfluß

nehmen. Um jedoch die richtigen Prioritäten zu wählen, mußte der Benutzer Überblick über das gesamte Programm haben.

Die Vorgehensweise in UBS setzt jetzt ganz auf die in SEPIA vorhandene Möglichkeit, Aufrufe von Prädikaten verzögert zu implementieren [SEPIA 91, (1) 97-103]. Dazu ist eventuell eine `delay`-Klausel für das durch den Funktionsaufruf angesprochene Prädikat zu definieren. Der Benutzer braucht nur noch die Prädikate einzeln, für sich zu betrachten, um richtiges Programmverhalten zu erzielen. Das soll an geeigneter Stelle noch genauer erklärt werden (siehe Seite 69).

Logische Operatoren: Für die logischen Operatoren Unifikation, Disjunktion, Negation und Implikation sind in UBS die gleichen Operatoren wie in UBS* verwendet worden (siehe Abbildung 7).

Argumentstruktur:

- gewöhnlicher Term, Liste, Menge:

```
STRUKT ⟶ TERM | LIST | SET
```

- Merkmalstruktur, Typausdruck, Funktionsaufruf:

```
STRUKT ⟶ AVM | @ TYP | $ FUNCTION
```

- Unification, Disjunktion, Negation, Implikation:

```
STRUKT ⟶
     STRUKT & STRUKT | STRUKT # STRUKT |
     ^ STRUKT | STRUKT => STRUKT
```

Merkmalstrukturen:

- einzelnes Merkmal:

```
AVM ⟶ ATTR : STRUKT
```

- komplexe Struktur:

```
AVM ⟶ AVM .. AVM
```

Abbildung 7: Die Syntax von Argumenten

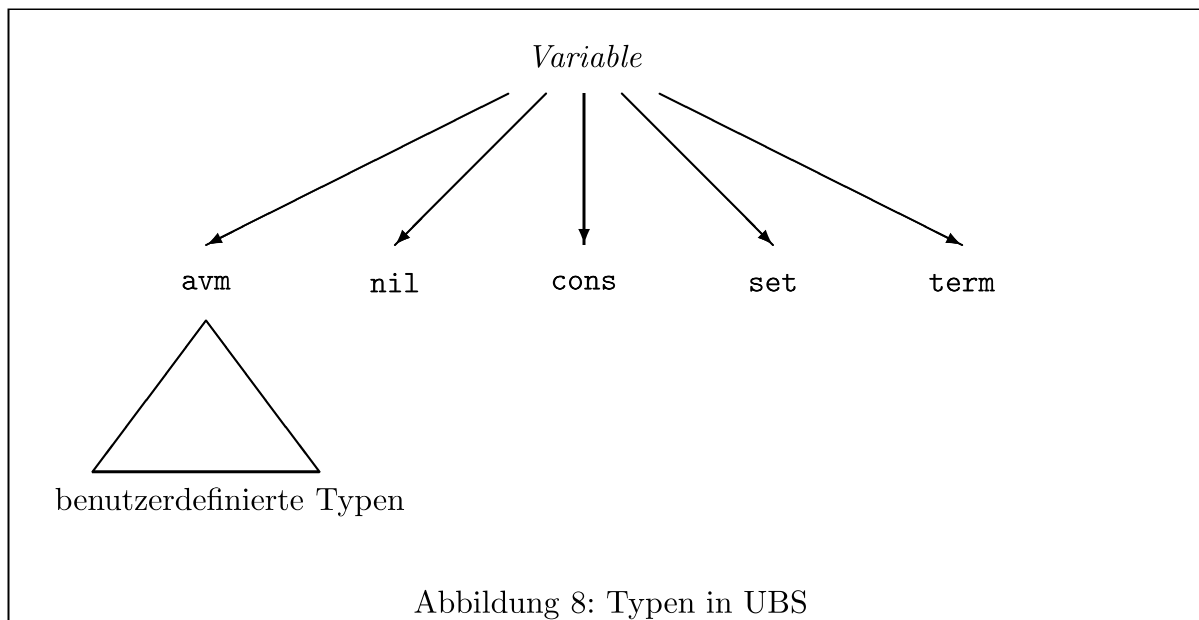


Abbildung 8: Typen in UBS

2.2.2 Die Struktur eines Programms

Wie ein gewöhnliches Programm in PROLOG bzw. SEPIA besteht auch ein Programm in UBS aus einer Folge von Klauseln. Mehrere Arten von Klauseln sind zu unterscheiden. Im Vergleich zu UBS* haben sich hier einige Änderungen ergeben: Einige Konstruktionen haben sich als überflüssig, andere als nötig erwiesen und sind neu in UBS aufgenommen worden. Es gilt ab sofort grundsätzlich für jede Art von Klausel in UBS, daß sie der Programmtransformation unterzogen wird.

In der folgenden Auflistung werden grundlegende Kenntnisse der Programmiersprache PROLOG vorausgesetzt, wie sie etwa in dem Buch [ClocMell 87] vermittelt werden. Der Leser, der das alte UBS* nicht kennt, kann die Absätze über markierten Code und Makros getrost überspringen, denn beides spielt im neuen UBS keine Rolle mehr. Trotzdem sei [Stolzenburg 92] als Lektüre empfohlen; es ist eine knappe Darstellung von UBS*, auf die in diesem Kapitel oft Bezug genommen wird.

Regeln, Fakten und DCG-Notation: Die Behandlung dieser Art von Klauseln ist gleichgeblieben. Auf den Argumentpositionen der Prädikate sind alle in UBS eingeführten Elemente des HPSG-Formalismus erlaubt. Sie werden in Ausdrücke und Code in SEPIA transformiert.

Beispiele:

```
lexicon(
     'dog',
     @noun & case:(nom#acc) & agr:(per:third..num:sng),
     [head: @det..subcat:[]]).

word(phon:[WORD]..synloc:(head:VALUE..subcat:SUBCAT)) :-
     lexicon(WORD,VALUE,SUBCAT).

np(N) --> det, n(N).
```

Markierter Code: Die Einführung von sogenanntem markierten, d.h. nicht zu transformierenden Code ist in UBS jetzt überflüssig, da bei der Programmtransformation den Prädikaten keine Parameter mehr hinzugefügt werden müssen dank der Verwendung von SEPIA, wie noch zu sehen sein wird.

Makros: Die Makros wurden ebenfalls aus der Sprache herausgenommen. Sie können durch Funktionsaufrufe ersetzt werden. Eine Makrodefinition in UBS*

MACRO(ARG$_1$,...,ARG$_n$) := DEF

ist fortan zunächst als Prädikat mit der einzigen Klausel

MACRO(ARG$_1$,...,AGR$_n$,DEF)

umzusetzen. Statt eines Makroaufrufs ist dann der Funktionsaufruf

MACRO(ARG$_1$,...,AGR$_n$)

zu tätigen. Ein kleiner Nachteil ist, daß die Makros in UBS* negiert werden konnten, während die Negation von Funktionen in UBS (noch) nicht implementiert ist. Das ist aber wegen der Seltenheit der Anwendung der Negation nicht sehr gravierend.

Direktiven: Im Gegensatz zu UBS* werden Direktiven in einem Programm vor ihrer Ausführung der Programmtransformation unterzogen. Diese Transformation erfolgt auch, wenn eine Anfrage vom Interpreter aus gestellt wird. Zu den Direktiven sind in UBS jetzt auch die `delay`-Klauseln zu rechnen.

Beispiele:

```
:- module(hpsg).
:- sign(SIGN & @word).

delay append(LIST1,_,LIST2) if var(LIST1), var(LIST2).
```

Typdeklarationen: Echt neu in UBS sind die Typdeklarationen. Sie machen die Definition von Typen von Merkmalstrukturen durch den Benutzer möglich. In einer Liste sind alle für diesen Attribut neuen Attribute aufgezählt. Die Angabe von Wertebereichen ist leider (noch) nicht vorgesehen.

Zu jedem Typ ist sein direkter Supertyp zu nennen. Das kann auch der Typ `avm` sein, der keinerlei Attribute besitzt. Die Spezifikation von Werten zu Attributen, die in Supertypen des neu zu definierenden Typs definiert sind, bleibt natürlich weiterhin möglich. Eine Typdeklaration hat die allgemeine Form

```
TYP := ATTRLIST - SUPTYP.
```

Beispiele:

```
sign := [phon,synsem,qstore] - avm.
word := [] - sign.
phrase := [dtrs] - sign.
```

2.3 Wie funktioniert UBS?

2.3.1 Programmtransformation

Am Beispiel einer Regel soll jetzt erklärt werden, wie die Programmtransformation im Prinzip funktioniert. Man vergleiche mit der Beschreibung in [Stolzenburg 92, 10-12]. Der folgende Aufruf führt die Transformation eines Argumentes `STRUKT` durch.

```
trans(STRUKT,STRUKT',CODE)
```

Dieser Aufruf bewirkt die rekursive Transformation des Ausdrucks `STRUKT` in seine interne Repräsentation `STRUKT'`. Manchmal muß zusätzlich der PROLOG-Code `CODE` erzeugt werden, um den Ausdruck `STRUKT` zu berechnen. Die Behandlung von Negation, Mengen und Funktionen erfordert in UBS keine zusätzlichen Parameter mehr.[8]

Der Ablauf der Transformation von UBS nach PROLOG ist im wesentlichen gleichgeblieben. Dazu soll die folgende Regel, wie sie in einem Programm in UBS auftreten kann, betrachtet werden:

$$
\begin{array}{l}
\texttt{PRED}_0\texttt{(ARG}_{01}\texttt{,...,ARG}_{0n_0}\texttt{) :-} \\
\quad \texttt{PRED}_1\texttt{(ARG}_{11}\texttt{,...,ARG}_{1n_1}\texttt{),} \\
\quad \texttt{...,} \\
\quad \texttt{PRED}_l\texttt{(ARG}_{l1}\texttt{,...,ARG}_{ln_l}\texttt{).}
\end{array}
$$

$\texttt{PRED}_0$, $\texttt{PRED}_1$, ..., $\texttt{PRED}_l$ sind hierbei Namen von Prädikaten. Die Argumente des Prädikats $\texttt{PRED}_i$ sind $\texttt{ARG}_{ij}$ mit $1 \leq j \leq n_i$. $n_i \geq 0$ ist die Stelligkeit des Prädikats $\texttt{PRED}_i$. Es gelte für alle möglichen i und j:

$$\texttt{trans(ARG}_{ij}\texttt{,ARG'}_{ij}\texttt{,CODE}_{ij}\texttt{)}$$

Die obige Klausel in UBS wird nun in die folgende Form in SEPIA überführt. Es sei noch einmal betont, daß keine Argumente bei der Transformation hinzugefügt werden und deshalb die Stelligkeit der Prädikate erhalten bleibt im Gegensatz zu UBS*. Das ist auch die Grundlage dafür, daß nicht mehr zwischen normalem und markiertem Code unterschieden werden muß.

$$
\begin{array}{l}
\texttt{PRED}_0\texttt{(ARG'}_{01}\texttt{,...,ARG'}_{0n_0}\texttt{) :-} \\
\quad \texttt{CODE}_{01}\texttt{, ..., CODE}_{0n_0}\texttt{,} \\
\quad \texttt{CODE}_{11}\texttt{, ..., CODE}_{1n_1}\texttt{, PRED}_1\texttt{(ARG'}_{11}\texttt{,...,ARG'}_{1n_1}\texttt{),} \\
\quad \texttt{...,} \\
\quad \texttt{CODE}_{l1}\texttt{, ..., CODE}_{ln_l}\texttt{, PRED}_l\texttt{(ARG'}_{l1}\texttt{,...,ARG'}_{ln_l}\texttt{).}
\end{array}
$$

2.3.2 Anfragen an UBS

Ein so transformiertes und compiliertes Programm ist nun in SEPIA ausführbar. Es können Anfragen an das System gestellt werden. Ihre Durchführung in UBS erfordern nur noch drei Phasen, die zudem im Vergleich mit UBS* [Stolzenburg 92, 15-16] vereinfacht wurden. Zu diesem Zweck ist in UBS jetzt kein besonderes Prädikat (Meta-Interpreter) mehr nötig, das Anfragen ausführt und die Lösungen ausgibt.

[8] In UBS* waren hierfür die drei internen Größen `NEG`, `SET` und `FUN` nötig.

In UBS* waren insgesamt vier Prädikate nötig, um mit Anfragen umgehen zu können, nämlich `call_/2` und `print_/2`, sowie `ubs_/1` und `sub_/2`. Die letzten beiden Prädikate waren allein der Behandlung der drei internen Größen in UBS* gewidmet [Stolzenburg 92, 30]. Sie erübrigen sich in UBS nun ganz.

Es folgt jetzt eine Erläuterung der Schritte, die bei der Verarbeitung von Anfragen in UBS zu bewältigen sind. Abbildung 9 zeigt die Arbeitsweise von UBS noch einmal schematisch. Man vergleiche sie mit der grafischen Darstellung in [Stolzenburg 91, 49].

1. **Vorwärtsübersetzung:** Die Elemente von UBS in einer Anfrage müssen in die interne Repräsentation nach SEPIA transformiert werden. Merkmalstrukturen werden in spezielle Terme abgebildet. Die logischen Operationen erfordern das Einfügen von zusätzlichem Programmcode.

   Dieser Schritt wurde vollständig für den Benutzer unsichtbar gemacht, indem die Transformation ganz der Ereignisbehandlung von SEPIA übergeben wird. Hierzu mußte die Schleife zur Bearbeitung von Anfragen (Top-Level Loop) [MeiSchRos 90, 5-8] manipuliert werden. Das ist möglich durch das Konzept, Ereignisse in SEPIA durch den Benutzer steuern zu können [SEPIA 91, (1) 65-72].

2. **Verarbeitung:** Die transformierte Anfrage kann ganz der Behandlung durch das System SEPIA überlassen werden. Die Verarbeitung von Negation, Mengen und Funktionsaufrufen erfordert keine gesonderte Nachbehandlung mehr. Alle Schritte, die in UBS* dazu nötig waren, erübrigen sich bzw. sind in das transformierte Programm integriert.

3. **Ausgabe:** Das Ergebnis, d.h. die Bindung der (nicht anonymen) Variablen aus der Anfrage wird formatiert ausgegeben. Das erfolgt bei der normalen Ausgabe von Lösungen durch den Interpreter (Top-Level Loop). Die interne Repräsentation von Merkmalstrukturen wird nur für die Ausgabe wieder zurücktransformiert. Sie wird dabei nicht verändert. Insbesondere werden keine Variablen instanziiert, so daß die ausgegebene Struktur nach der Ausgabe auch weiterverarbeitet werden kann.

   Die Ausgabe wird von dem Prädikat `show/1` bewerkstelligt. Dabei wird immer mit ausgegeben, welche Ziele noch verzögert sind. Der Benutzer kann `show/1` auch selbst explizit aufrufen. Es wird dann nur eine einzige, nämlich die gerade berechnete Lösung angezeigt. Das Argument von `show/1` muß eine Liste sein, deren Elemente die Gestalt `[NAME|VALUE]` haben. Das soll heißen, die Variable mit Namen `NAME` hat den Wert `VALUE`. Diese Bindung wird formatiert in der Form `NAME=VALUE` ausgegeben.

2.4 Eine Beispielgrammatik

2.4.1 Das Programm

Im folgenden soll ein Beispiel vorgeführt werden, das die Funktionsweise von UBS demonstriert. Gleichzeitig soll es zeigen, inwieweit UBS tatsächlich für HPSG geeignet ist.

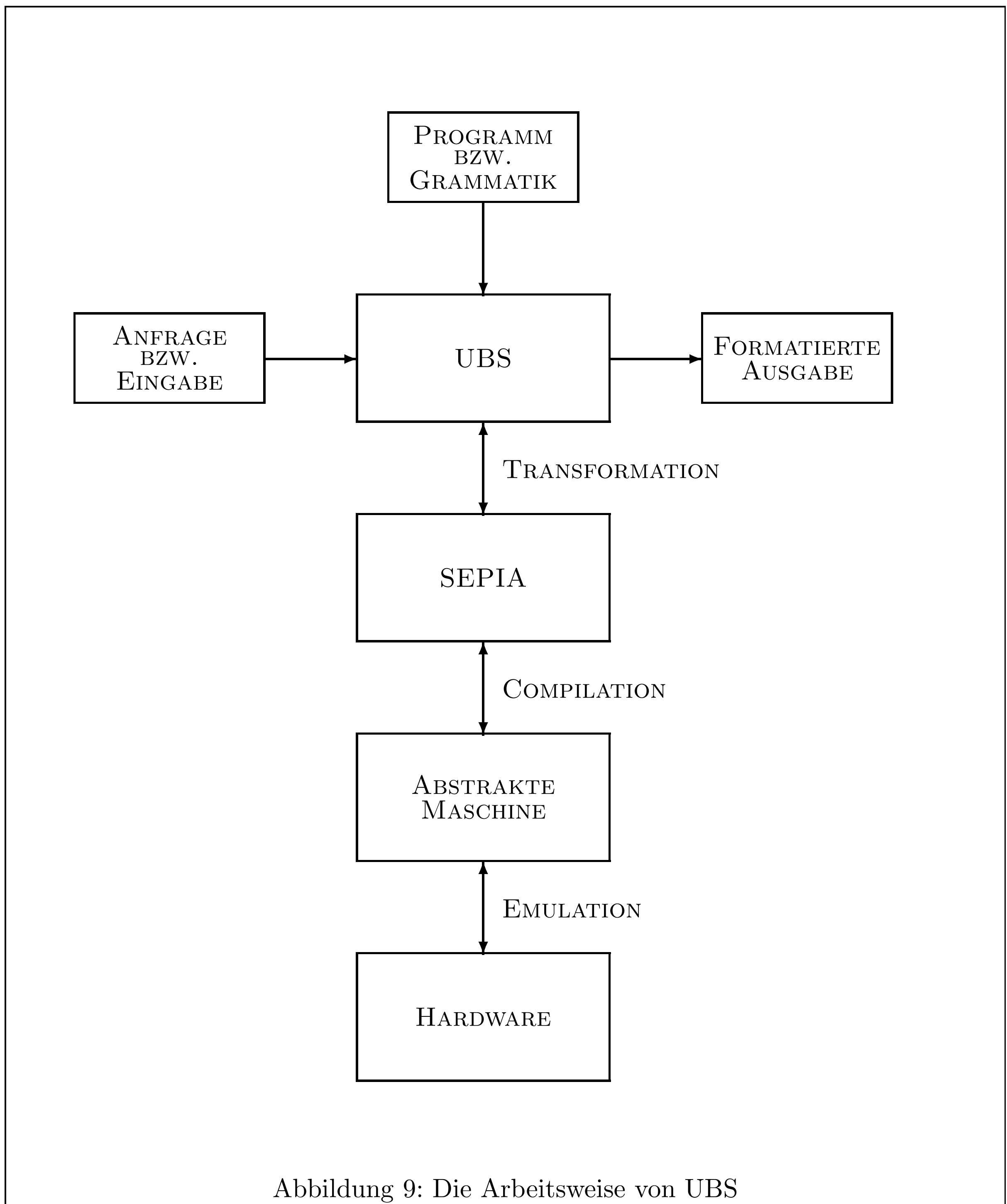


Abbildung 9: Die Arbeitsweise von UBS

Darum ist die für das Beispiel entwickelte Grammatik im Sinne der linguistischen Theorie von HPSG entwickelt worden. Sie ist stark an das Beispiel in [Stolzenburg 92, 26-29] angelehnt. Sie ist sehr fragmentarisch und erlaubt im wesentlichen nur das Parsen der beiden englischen Sätze *The dog barks.* und *The dogs bark.* sowie ihrer Konstituenten.

Es folgt nun das Listing der Beispielgrammatik. Sie liegt in der Datei `hpsg.ubs` vor. Der Leser möge sich die Grammatik anschauen. Anschließend folgen noch ein paar Bemerkungen dazu.

```
/*
    1. TYPENHIERARCHIE

    Definition von (syntaktischen) Typen.
    In dem Attribut 'synloc' wird der Pfad 'synsem:loc:cat'
    zusammengefasst; es handelt sich um die lokalen syntaktischen
    Eigenschaften. Das ist eine Vereinfachung.
*/

sign := [phon,synloc] - avm.
word := [] - sign.
phrase := [dtrs] - sign.

synsem := [head,subcat] - avm.

struct := [head_dtr,comp_dtrs] - avm.

speech := [agr] - avm.
noun := [case] - speech.
det := [] - speech.
verb := [] - speech.

index := [per,num,gen] - avm.




/*
    2. ALLGEMEINES

    Zeichen (Typ 'sign') sind entweder Woerter (Typ 'word')
    oder Phrasen (Typ 'phrase').
*/

sign($word & @word).
sign($phrase & @phrase).

word(phon:[WORD]..
     synloc:(head:VALUE..
             subcat:SUBCAT))
:- lexicon(WORD,VALUE,SUBCAT).

phrase(dtrs:(head_dtr:HEAD..
             comp_dtrs:COMPLEMENT) &
       $principles(HEAD,COMPLEMENT) &
       $rule(HEAD,COMPLEMENT)).
```

```
principles(HEAD,
           COMPLEMENT,
           $constituent_order_principle(HEAD,COMPLEMENT) &
           $subcat_principle(HEAD,COMPLEMENT) &
           $head_feature_principle(HEAD)).




/*
    3. PRINZIPIEN

    Sprachspezifisch: Constituent Order Principle
    Universal: Subcat Principle
               Head Feature Principle
*/


constituent_order_principle(HEAD & phon:PHON,
                            COMPLEMENT,
                            phon: $order(COMPLEMENT,PHON)).

subcat_principle(synloc:subcat: $conc(SUBCAT,$extract(COMPLEMENT)),
                 COMPLEMENT,
                 synloc:subcat:SUBCAT).

head_feature_principle(synloc:head:VALUE,
                       synloc:head:VALUE).




/*
    4. GRAMMATIKREGELN

    Die Regeln haben das Format
        'rule(HEAD,COMPLEMENT,PHRASE)'
    und entsprechen Phrasenstrukturregeln
        PHRASE --> COMPLEMENT HEAD,
    und zwar (in dieser Reihenfolge):
        S  --> NP VP
        NP --> DET N
        VP --> V
*/

rule($phrase & synloc:(head:agr:AGR..subcat:[_]),
     [$phrase & synloc:(head:(@noun & agr:AGR)..subcat:[])],
```

```
     @phrase & synloc:(head: @verb..subcat:[])).
rule($word,
     [$word & synloc:head: @det],
     @phrase & synloc:head: @noun).
rule($word,
     [],
     @phrase & synloc:head: @verb).




/*
    5. LEXIKONEINTRAEGE

    Die Lexikoneintraege sind parametrisiert:
        'lexicon(WORD,VALUE,SUBCAT)'
*/

lexicon('the',
        @det,
        []).
lexicon('dog',
        @noun &
        case:(nom#acc) &
        agr:(per:third..
             num:sng),
        [head: @det..
         subcat:[]]).
lexicon('dogs',
        @noun &
        case:(nom#acc) &
        agr:(per:third..
             num:plu),
        [head: @det..
         subcat:[]]).
lexicon('bark',
        @verb &
        agr:(AGR & @index &
            ^per:third..
             num:sng),
        [head:(@noun &
               case:nom..
               agr:AGR)..
         subcat:[]]).
lexicon('barks',
        @verb &
        agr:(AGR &
```

```
           per:third..
           num:sng),
       [head:(@noun &
              case:nom..
              agr:AGR)..
        subcat:[]]).
```

```
/*
    6. SONSTIGES

    'conc(LIST1,LIST2,LIST3)'
        entspricht 'append(LIST1,LIST2,LIST3)' mit dem Unterschied,
        dass 'conc' gegebenenfalls verz"ogert aufgerufen wird.

    'extract(LIST)'
        extrahiert aus der Liste LIST von Konstituenten (Komplementen)
        die SYNSEM-Werte (genau genommen hier die Werte des Attributs
        'synloc').

    'order(SIGNS,LIST,PHON)'
        wird im Rahmen des Constituent Order Principles benoetigt;
        fuegt die Werte von 'phon' der Liste von Konstituenten
        (Komplemente) SIGNS und die Liste LIST zur Liste PHON zusammen.
*/

delay conc(LIST1,_,LIST2) if var(LIST1), var(LIST2).
conc([ELEM|LIST1],LIST2,[ELEM|LIST3]) :-
       conc(LIST1,LIST2,LIST3).
conc([],LIST,LIST).

delay extract(LIST,SYNSEM) if var(LIST), var(SYNSEM).
extract([],[]).
extract([synloc:SYNSEM|LIST],[SYNSEM|REST]) :-
        extract(LIST,REST).

delay order(SIGNS,_,PHON) if var(SIGNS), var(PHON).
order([],PHON,PHON).
order([phon:PHON1|SIGNS],
      PHON2 & @cons,
      $conc(PHON1 & @cons, $order(SIGNS,PHON2)) & @cons).
```

Im Beispielprogramm wird reger Gebrauch von Funktionsaufrufen gemacht. Es handelt sich um eine parametrisierte Version von HPSG, wie in [DörSei 91, 13-16] vorgeschlagen. Merkmalterme werden in Relationen eingebettet. Alle eingeführten Prinzipien (`principles`) haben zwei (oder im Falle des HFP nur einen) Parameter, nämlich den Head (`HEAD`) der Phrase und die Liste der Komplemente der Phrase `COMPLEMENT`.

Die beiden Parameter sind gleichzeitig die Werte der Attribute HEAD-DTR und COMP-DTRS. Diese Attribute treten im Wert des Attributs DTRS auf und spiegeln die Konstituentenstruktur der Phrase wieder. Im Vorschlag von [DörSei 91, 14] sind diese Attribute ganz weggelassen. Die Autoren begründen das mit dem Lokalitätsprinzip von HPSG [PolSag 87, 143-144]. Hier sind die Attribute beibehalten worden, damit die Information über die Konstituentenstruktur für die Ausgabe einer Lösung zur Verfügung steht.

Das obige winzige Fragment einer Grammatik des Englischen enthält einige Vereinfachungen der linguistischen Theorie von HPSG. An diesen Stellen ist die Grammatik zu erweitern und präzisieren in späteren Versionen. Nun folgen einige Anmerkungen zu den vorgenommenen Vereinfachungen:

- Die Typenhierarchie von HPSG ist stark verkürzt und unvollständig implementiert. Insbesondere ist der Pfad SYNSEM:LOC:CAT zu dem einen Attribut SYNLOC komprimiert. Dadurch sind die Typen der Elemente in der SUBCAT-Liste eigentlich nicht vom Typ *synsem*, wie in HPSG vorgesehen.[9]

- Die Grammatikregeln (`rule`/3) sind hier spezieller als die in [PolSag 87, 149-157] vorgeschlagenen sehr abstrakten Schemata. Sie reichen aber zur Analyse der hier möglichen Konstituentenstrukturen aus.

- Das Constituent Order Principle geht von der Annahme aus, daß der (syntaktische) Head der Phrase phonologisch als letztes realisiert wird. Davor stehen die PHON-Werte der Komplemente. Diese Annahme gilt schon nicht mehr für transitive Verben. Das Verb *to bark* aus dem Beispiel ist aber intransitiv.

Ein Hinweis zum Erstellen eigener Grammatiken mit UBS soll noch gegeben werden: Es ist wichtig, daß die in Funktionsaufrufen angesprochenen Prädikate mit einer `delay`-Klausel versehen werden. Vorsicht ist außerdem bei Linksrekursionen geboten, die trotz einer solchen Klausel Endlosschleifen bewirken können.

2.4.2 Ein Testdurchlauf

Mit dem Beispielprogramm von oben kann die eine Analyse von Sätzen – genau genommen von Phrasen und Wörtern – vorgenommen werden, aber auch Generierung. Z.B. kann analysiert werden, ob die Sätze *The dog bark.* und *The dogs bark.* (syntaktisch) wohlgeformt sind. Dazu ist einfach eine Anfrage mit dem Prädikat `sign`/1 zu stellen.

Die Lösungen werden formatiert ausgegeben. Sie werden durch Backtracking aufgezählt, falls es mehrere gibt. Die Ausgabe erfolgt eingerückt. Je tiefer die Verschachtelung ist,

[9] Letzteres wurde erst später in HPSG eingeführt und trägt auch dem Lokalitätsprinzip Rechnung [Pollard 91, (1) 16]. In der ursprünglichen Version von HPSG war eine Liste von Zeichen (Typ *sign*) als Wert des Attributs SUBCAT vorgesehen [PolSag 87, 115-117].

desto größer ist die Einrückung. Die maximale Tiefe kann an der Größe `print_depth` mit dem Befehl `set_flag`/2 eingestellt werden [SEPIA 91, (2) 249]. Falls sie erreicht ist, wird ein Stern anstelle des eigentlichen Wertes ausgegeben.

Bei Merkmalstrukturen werden alle Merkmale, die einen spezifizierten Wert haben, aufgezählt. Die Attribute werden mit gleicher Einrückung übereinander gesetzt. Ergänzend wird darunter der Typ der Merkmalstruktur und direkt dahinter sein Identifikator (Variable) genannt. Das erlaubt die Unterscheidung von gleichen versus identischen Merkmalstrukturen bei intensionalen Typen.

Schließlich werden die noch in Verzögerung befindlichen Prädikate ausgegeben. Es wird immer nur die interne Bezeichnung von Variablen angegeben, nicht die vom Benutzer vergebenen Variablennamen. Der folgende Ausdruck stammt aus einer Sitzung, bei dem Anfragen an die Grammatik `hpsg.ubs` gestellt werden.

```
[sepia 55]: set_flag(print_depth,9).

yes.
[sepia 56]: sign(SIGN & phon:[the,dog,bark]).

no (more) solution.
[sepia 57]: sign(SIGN & phon:[the,dogs,bark]).
SIGN=phon:[the,
           dogs,
           bark]..
     synloc:head:agr:per:third..
                     num:plu..
                     index_g1109..
                 verb_g1005..
            subcat:[]..
            synsem_g640..
     dtrs:head_dtr:phon:[bark]..
                   synloc:head:agr:per:third..
                                   num:plu..
                                   index_g1109..
                               verb_g1005..
                          subcat:[head:agr:per:third..
                                           num:plu..
                                           index_g1109..
                                       case:nom..
                                       noun_g792..
                                  subcat:[]..
                                  synsem_g1021]..
                          synsem_g604..
                   dtrs:head_dtr:phon:[bark]..
                                 synloc:head:agr:per:third..
                                                 num:plu..
                                                 index_g1109..
                                             verb_g1005..
                                        subcat:[head:agr:*..
                                                     case:*..
                                                     noun_g792..
                                                subcat:[]..
                                                synsem_g1021]..
                                        synsem_g950..
                                 sign_g924..
                        comp_dtrs:[]..
                        struct_g818..
                   phrase_g798..
          comp_dtrs:[phon:[the,
                           dogs]..
                     synloc:head:agr:per:third..
```

```
                                num:plu..
                                index_g1109..
                            case:nom..
                            noun_g792..
                       subcat:[]..
                       synsem_g1021..
                dtrs:head_dtr:phon:[dogs]..
                              synloc:head:agr:per:*..
                                              num:*..
                                              index_g1109..
                                          case:nom..
                                          noun_g792..
                                     subcat:[head:*..
                                             subcat:*..
                                             synsem_g1671]..
                                     synsem_g1613..
                              sign_g1523..
                     comp_dtrs:[phon:[the]..
                                synloc:head:det_g1695..
                                       subcat:[]..
                                       synsem_g1671..
                                sign_g1545]..
                     struct_g1417..
                phrase_g1397]..
          struct_g400..
     phrase_g374

     More? (;)

no (more) solution.
[sepia 57]:
```

2.4.3 Ein Zeitvergleich

Die folgende Tabelle zeigt die Antwortzeiten der alten und der neuen Version von UBS. Wie zu sehen ist, besteht ein gewaltiger Unterschied. Das liegt natürlich hauptsächlich daran, daß unterschiedliche Rechner benutzt wurden. Nichtsdestotrotz hat beim Übergang von UBS* nach UBS eine wesentliche Steigerung in der Effizienz stattgefunden. Die notierten Zeiten können je nach Eingabe und Auslastung des Rechners etwas schwanken.

Version	Rechner	Zeit
UBS*	386er PC	ca. 13.00s
UBS	SUN 3/80	ca. 0.03s

2.5 Fehlermeldungen

2.5.1 Überblick

Bei der Programmentwicklung mit UBS – wie auch mit jeder anderen Programmiersprache – können Fehler auftreten. Niemand ist perfekt. Die Fehler können verschiedene Ursachen haben.[10]

UBS-Syntax: Die Transformation eines Ausdrucks von UBS nach SEPIA kann fehlschlagen. Als Gründe dafür kommen fehlerhafte Syntax oder das Verwenden einer unerlaubten Struktur in Frage.

SEPIA-System: Da die meiste Arbeit das System SEPIA für UBS übernimmt, können auch von dorther Fehlermeldungen kommen, und zwar sowohl zur Compile- als auch zur Laufzeit des Programms. Die Fehler bzw. Ereignisse in Zusammenhang von SEPIA sind in [SEPIA 91, (1) 241-251] aufgelistet.

Sonstiges: Am schwierigsten sind logische Fehler, die keine Fehlermeldungen nach sich ziehen, zu beheben. In einem solchen Fall muß das Programm gegebenenfalls anhand einiger Testbeispiele einer eingehenden Analyse unterzogen werden.

2.5.2 Die Meldungen im einzelnen

Die Fehlermeldungen von UBS beziehen sich eigentlich alle auf die Transformationsphase. Die Meldungen sind in UBS vollständig in Englisch gehalten. Der Ausdruck, bei dem der Fehler erkannt wird, wird angezeigt. Die Einführung von Typen in UBS bewirkt eine Vergrößerung der Anzahl an Fehlermeldungen. Gleichzeitig wird aber das Schreiben korrekter Programme gefördert.

Die Fehler sind in Klassen ähnlicher Fehlerarten unter einem Schlüsselwort vereint. Die Klassen fassen unterschiedliche, aber irgendwie ähnliche Fehler zusammen. Es folgt die Auflistung aller Klassen mit Angabe von Beispielen von Klauseln, die die jeweilige Fehlermeldung nach sich ziehen.

[10] Man vergleiche mit der Auflistung in [Stolzenburg 92, 31].

negation: Die Negation von (1) Mengen und (2) Funktionen ist in UBS (noch) nicht möglich. Dazu muß erst einmal die Extensionalität von Typen allgemein, speziell die von Mengen, in der Implementation realisiert werden.

Zu beachten ist, daß die Negation nicht immer direkt vor der Menge oder dem Funktionsaufruf steht, sondern eventuell weiter außen. Der Grund dafür ist, daß Negationen umgeformt werden und möglichst weit nach innen gebracht werden, nämlich vor Konstanten, Variablen oder Typbezeichnern.

Beispiele:

```
negation(^{NP1,NP2}).
negation(^head:(@verb)..subcat:($append(S1,S2))).
```

twice: Es wird versucht, eine schon einmal definierte Größe nochmals zu definieren, obwohl das verboten ist. Das ist der Fall, wenn (1) mehrere Typdeklarationen den gleichen Typ definieren oder (2) ein Attribut in mehr als einer Typdeklaration eingeführt wird oder (3) ein Attribut in einer Merkmalstruktur mehrfach einen Wert zugewiesen bekommt.

Beispiele:

```
typ1 := [a,b,c] - avm.
typ2 := [c,d,e] - avm.
typ1 := [f,g,h] - avm.
test(a:x..b:y..a:z).
```

type: Es liegt ein Typfehler im Ausdruck vor. Mögliche Ursachen dieser Meldung sind: (1) Die Liste von Attributen in Typdeklarationen ist nicht geschlossen. (2) Der Konstruktor .. taucht im anderen Zusammenhang als bei Merkmalstrukturen auf. (3) Es werden mehrere Attribute in einer Merkmalstruktur verwendet, die zu verschiedenen, nicht miteinander zu vereinbarenden Typen gehören.

Beispiele:

```
sign := [phon,synsem,qstore|_] - avm.
error1(a:x..error1).
```

undefined: Ein Ausdruck wird vor seiner Definition verwendet. Gründe dafür können sein: (1) In einer Typdeklaration tritt ein undefinierter Supertyp auf. (2) Ein Typname ist undefiniert in einem Typaufruf. (3) In einer Merkmalstruktur kommt ein nicht definiertes Attribut vor.

Beispiele:

```
typ := [] - undef.
example1(@undef).
example2(undef:X).
```

variable: Hier darf nichts Variables stehen. Dieser Fehler kann durch einen der folgenden Punkte bedingt sein: (1) Eine Klausel oder Direktive oder ein Ziel darin ist eine Variable. (2) Ein Funktionsaufruf hat keinen Funktor, sondern ist eine Variable. (3) Ein Attribut in einer Typdeklaration oder in einer Merkmalstruktur ist nicht atomar. (4) Der Konstruktor .. wird mit einer Variablen kombiniert.

Beispiele:

```
P :- Q.
:- R.
typ := [X,Y,Z] - avm.
more1($F).
more2(A:x).
more3(a:x..B).
```

3 Anmerkungen zur Implementation

3.1 Typen und ihre interne Repräsentation

UBS erlaubt die Unterscheidung verschiedener Datentypen. Alle Typen werden intern mit Hilfe von einfachen PROLOG-Termen repräsentiert, die jeweils anderen Aufbau haben. Im wesentlichen gibt es vier Haupttypen: Merkmalstrukturen, Listen Mengen und (gewöhnliche) Terme.

3.1.1 Merkmalstrukturen

Ihre interne Repräsentation wird ausführlich im Abschnitt 3.2 behandelt. Merkmalstrukturen sind der einzige Datentyp, der die Definition von Subtypen durch den Benutzer erlaubt.

3.1.2 Listen

Die Notation für Listen ist direkt von PROLOG übernommen worden. Es können geschlossene sowie offene Listen formuliert werden. Leere und nicht-leere Listen werden unterschieden durch die beiden Typen `nil` und `cons`.

3.1.3 Mengen

Für Mengen ist die aus der Mathematik gewohnte Schreibweise eingeführt worden, mit geschweiften Klammern. Es können nur endliche Mengen beschrieben werden.

Eine Mengenbeschreibung `{T1,...,Tn}` wird bei der Programmtransformation durch `{X}` ersetzt, wobei `X` eine attributierte Variable (*Metaterm*, siehe [SEPIA 91, (1) 105-108])in SEPIA ist. Die Variable `X` ist mit der durch `sort`/2 sortierten Liste der Elemente der Menge assoziiert; identisch doppelte Elemente werden entfernt.

Die Unifikation zweier Mengen wird immer dann angestoßen, wenn zwei mit Mengen assoziierte Variablen unifiziert werden müssen. Auf das Problem der Unifikation von Mengen wird in Abschnitt 3.3 noch genauer eingegangen.

3.1.4 Terme

Terme brauchen nicht übersetzt zu werden. Sie werden durch sich selbst, als PROLOG-Terme repräsentiert. Alles, was nicht zu den drei vorigen Datentypen gehört, ist von diesem Typ.

3.2 Typisierte Merkmalstrukturen

3.2.1 Was ist ein Typ?

Eine Möglichkeit, den Begriff Typ zu definieren, ist es, einen Typ (oft auch als Sorten bezeichnet) als Teilmenge aller Merkmalstrukturen aufzufassen. So geschieht dies z.B.

in [Smolka 89, 35]. Dann entsprechen den Typen genau die Makros aus UBS*. Folglich ist dann z.B. auch das HFP, definiert durch

```
head_feature_principle :=
     synsem:loc:cat:head:HEAD..
     dtrs:head_dtr:synsem:loc:cat:head:HEAD.
```

ein Typ, der alle die Merkmalstrukturen lizensiert, bei denen die HEAD-Merkmale mit denen ihrer Haupttochter (HEAD-DTR) identisch sind.

Dieser Begriff stimmt jedoch nicht mit dem überein, was in HPSG unter einem Typ verstanden wird. Dort wird der Begriff wie folgt eingeführt nach [PolSag 87, 39-40], [Pollard 91, (1) 11-12] und [Henschel 92]:

1. Jeder Typ legt eine bestimmte Menge von zu ihm zugehörigen Attributen fest, die auch leer sein kann (atomare Typen).

2. Darüberhinaus legt jeder Typ für jedes seiner Attribute einen Wertebereich (Typangabe) fest.

3. Die Typen bilden einen Subsumptionsverband bezüglich einer Vererbungsrelation.

3.2.2 Vererbung

Man betrachte die Abbildung 10, einen Ausschnitt aus der Typenhierarchie in HPSG. Es gelten die folgenden Konventionen: Attributnamen sind groß, Typnamen durchgängig klein und kursiv geschrieben. Der Pfeil bezeichnet die Vererbungsrelation. Der allgemeinste Typ ist *avm*; er subsumiert alle anderen Typen. Zu jedem Typ ist in eckigen Klammern angegeben, welche (neuen) Attribute er hat samt den dazugehörigen Wertebereichen.

In welcher Beziehung steht nun ein Typ t mit seinem direkten Subtypen s? Welche Eigenschaften werden vererbt? Hierfür kommt mehreres in Betracht:

1. **Attribute:** Welche Attribute eine Merkmalstruktur haben kann, ist durch ihren Typ festgelegt. Durch die Funktion *Approp* ist jedem Typ einen Menge von zulässigen Attributen zugeordnet. Sie werden vererbt, d.h. falls $s \preceq t$, impliziert $Approp(t, l)\downarrow$, daß $Approp(s, l)\downarrow$ (nach Definition 8).

   Beispiel: Merkmalstrukturen vom Typ *sign* erlauben Wertangaben nur für die Attribute PHON, SYNSEM und QSTORE. Im direkten Subtyp *phrase* darf zusätzlich noch ein Wert für das Attribut DTRS spezifiziert sein.

2. **Wertebereiche:** Der Wertebereich eines Attributs wird durch eine Typangabe festgelegt. Dieser wird in Subtypen übernommen oder gegebenenfalls weiter eingeschränkt. Das ergibt sich ebenfalls aus Definition 8. Die Wertebereiche werden durch die jeweiligen Funktionswerte von *Approp* bestimmt.

   Beispiel: Der Typ *head-adjunct-structure* erbt die beiden Attribute HEAD-DTR und COMP-DTRS vom Typ *constituent-structure*. Der Wertebereich des ersten Attributs wird einfach übernommen, der des zweiten wird eingeschränkt von Listen

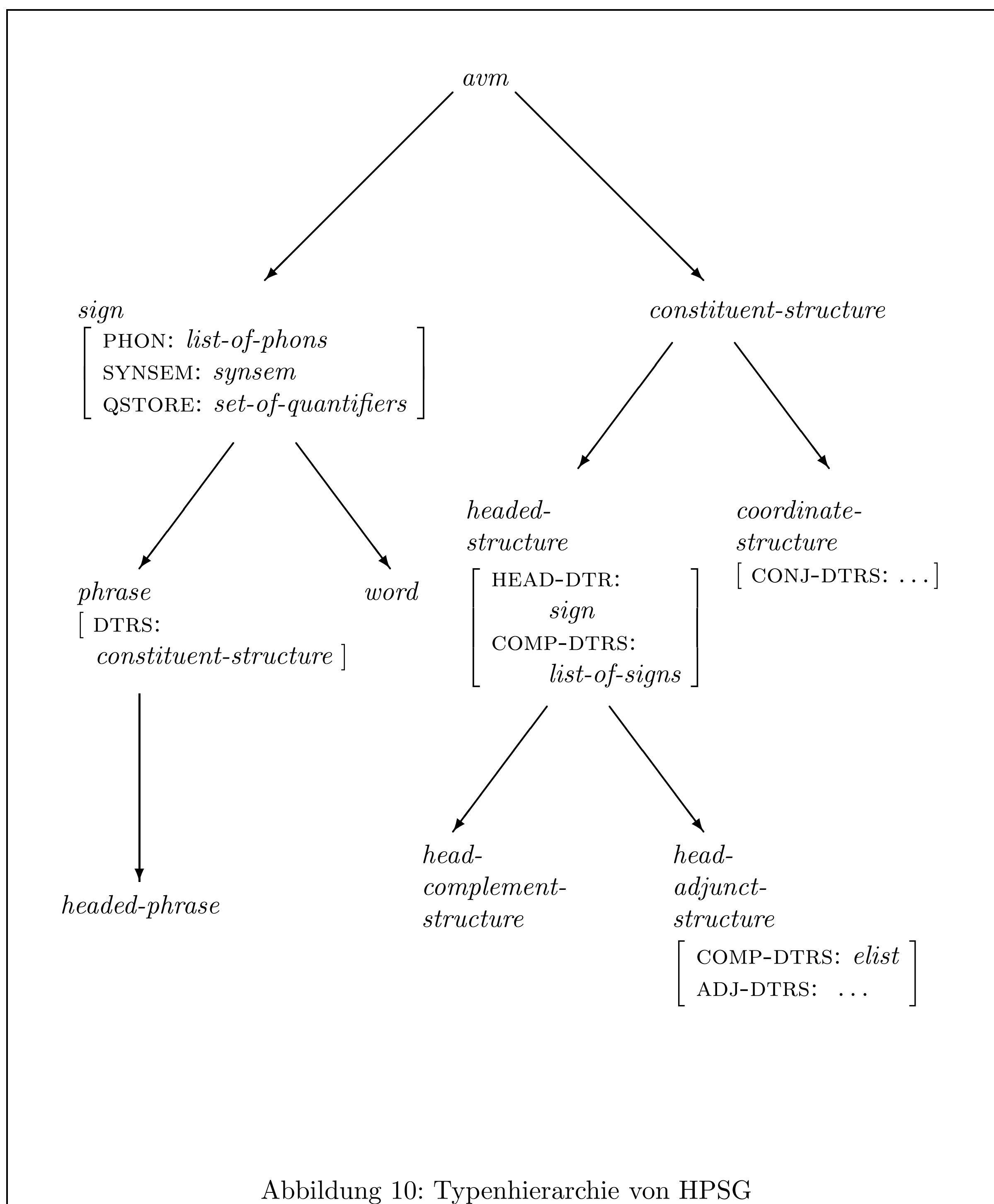


Abbildung 10: Typenhierarchie von HPSG

von Zeichen allgemein (*list-of-signs*) nur auf die leere Liste (*elist*). Der Typ *elist* muß als Subtyp von *list-of-signs* definiert sein.

3. **Constraints:** Bestimmte Typen können Identitäten (auch Koreferenzen genannt) von Werten an verschiedenen Stellen in der Merkmalstruktur verlangen oder disjunktive Angaben enthalten. Solche Constraints können durch Heranunifizieren an den Typ geltend gemacht werden und an eventuelle Subtypen vererbt werden. Diese Art wird nicht mehr durch die Funktion *Approp* erfaßt.

   Beispiel: Das HFP (Definition siehe oben) gilt für alle Merkmalstrukturen vom Typ *headed-phrase*. Die darin bewirkte Koreferenz der HEAD-Merkmale ist an Subtypen von *headed-phrase* zu vererben.

Die gerade genannten Vererbungsbeziehungen sind in den verschiedenen Implementationen für den HPSG-Formalismus zu unterschiedlichen Graden realisiert. In UBS gibt es eine ganze Reihe von Einschränkungen, die aber – wie im weiteren Verlauf gezeigt werden soll – relativ einfach und effizient implementiert werden können:

1. Nur Attribute werden vererbt, keine Wertebereiche oder gar Constraints. Für alle $s \in \mathcal{S}$ und $l \in \mathcal{L}$ mit $Approp(s,l)\downarrow$ gilt $Approp(s,l) = \top$. Die Funktion *Approp* ist somit konstant, so daß die Unifikation zweier Merkmalstrukturen dann immer wohlgetypt ist (siehe Definition 9).

2. Es sind nur einfache Typenhierarchien zugelassen. Die Typenhierarchie ist also ein Baum (läßt man einmal den inkonsistenten Typ $\perp$ außer acht). Jeder Typ besitzt dann einen eindeutigen Pfad von der Wurzel des Typenhierarchiebaumes aus. Das läßt sich für die Implementation ausnutzen (siehe Abschnitt 3.2.4).

3.2.3 Repräsentation durch Wertlisten

In GULP [Covington 89] sowie in UBS* [Stolzenburg 92, 17] werden Merkmalstrukturen durch Wertlisten ersetzt. Wertlisten sind offene Listen, wobei jede Position in der Liste reserviert ist für den Wert eines bestimmten Attributs. Falls ein Attribut keinen spezifizierten Wert hat, dann enthält die entsprechende Position in der Liste eine anonyme Variable. Das offene Ende der Liste, ebenfalls eine anonyme Variable, faßt alle Positionen am Ende der Liste hinter der letzten Position mit einem spezifizierten Wert zusammen. Diese Maßnahme spart Speicherplatz und Rechenzeit ein.

Um spezifizierte Werte von unspezifizierten Werten unterscheiden zu können, werden erstere mit Hilfe des Funktors `gulp`/1 markiert. Dadurch können Variablen, die noch an anderer Stelle auftreten (Koreferenzen) und anonyme Variablen ohne Bedeutung auseinandergehalten werden. Wenn z.B. die Attribute `phon`, `synsem`, `qstore`, `dtrs`, `head-dtr` und `comp-dtrs` in dieser Reihenfolge den ersten sechs Positionen in der Wertliste zugeordnet werden, dann wird die in UBS-Syntax gegebene Merkmalstruktur

```
phon:PHON..synsem:SYNSEM..dtrs:head_dtr:HEAD
```

transformiert in die folgende verschachtelte offene Liste (wobei die Listenstruktur in Wirklichkeit nicht mit dem Punktoperator, sondern mit dem Operator `gulp`/2 konstruiert wird):

```
[gulp(PHON),gulp(SYNSEM),_,gulp([_,_,_,_,gulp(HEAD)|_])|_]
```

Ein Nachteil dieser Darstellung ist es, daß jedes Attribut, das an irgendeiner Stelle im Programm verwendet wird, eine Position in der Wertliste beansprucht. Das macht Merkmalstrukturen umfangreicher und langsamer in der Verarbeitung als nötig. Oft fallen zwar unbenutzte Attribute in das offene Ende der Wertliste und weder Speicherplatz noch Rechenzeit wird verschwendet, aber nicht alle unbenutzten Attribute genießen diese Eigenschaft. Tatsächlich enthält fast jede Wertliste Lücken, d.h. Positionen, die niemals mit einem Wert belegt werden. Um die Zahl der Lücken zu verkleinern, ist das System dahingehend zu verändern, daß verschiedene Typen von Merkmalstrukturen unterschieden werden. Dieser Vorschlag von [Covington 89, 33] ist im neuen UBS in die Tat umgesetzt worden.

3.2.4 Typen und Wertlisten

Die Idee bei der Codierung von Merkmalstrukturen über einer einfachen Typenhierarchie ist, die Wertliste einer Merkmalstruktur vom Typ mit dem Pfad im Typenhierarchiebaum in einem Term zusammenzufassen. Dadurch wird die Vererbung in der Typenhierarchie und Angemessenheit von Attributen gleichzeitig in den Griff bekommen.

Was die Codierung von Typinformation betrifft, findet sich die gleiche Idee auch in [SchmWern 89, 58-59] skizziert. Aus den Ausführungen in [Walther 89, 31-32] folgt, daß die Unifikation von Typen nur dann auf die Unifikation von (PROLOG-)Termen reduzierbar ist, wenn die Typenhierarchie Waldstruktur hat, d.h. der Typenhierarchiegraph eine Ansammlung von Bäumen ist. (Formale Definitionen dieser Begriffe finden sich in [Ebert 81, 72].) Die Verwendung von Attributen in Verbindung mit Typen wird in beiden Aufsätzen jedoch nicht untersucht.

Die interne Repräsentation von Merkmalstrukturen hat die Form

```
REPR = avm(PATH,GULP),
```

wobei `PATH` und `GULP` beides offene Listen sind: `PATH` ist der Pfad in der Typenhierarchie und `GULP` ist die Wertliste der Merkmalstruktur, die im Vergleich zur untypisierten Version oft stark verkürzt ist, weil nur für die angemessenen Attribute des jeweiligen Typs eine Position in der Wertliste reserviert werden muß.

Die Zuordnung von Attributen und Positionen in der Wertliste erfolgt aufgrund der Typdeklarationen. Die Typdeklarationen

```
sign := [phon,synsem,qstore] - avm.
phrase := [dtrs] - sign.
headed := [head_dtr,comp_dtrs] - avm.
```

bewirken z.B., daß

1. die ersten drei Positionen für Wert der Attribute `phon`, `synsem` und `qstore` reserviert werden für Merkmalstrukturen vom Typ `sign`;

2. die vierte Position in Wertlisten für Merkmalstrukturen vom Typ `phrase` vorgesehen ist für Werte des Attributs `dtrs`; die ersten drei haben die gleiche Bedeutung wie beim Typ `sign`, denn `phrase` ist direkter Subtyp von `sign`;

3. die ersten beiden Positionen in Wertlisten für Merkmalstrukturen vom Typ `headed` für die Attribute `head_dtr` und `comp_dtrs` reserviert werden.

Folglich ist die Merkmalstruktur von oben nun so in die interne Repräsentation zu übersetzen:

```
avm(
      [sign,phrase|_],
      [gulp(PHON),gulp(SYNSEM),_,
       gulp(avm([headed|_],[gulp(HEAD)|_]))|_]
)
```

Die Vorwärts- wie Rückwärtsübersetzung von Merkmalstrukturen in Terme bzw. umgekehrt erfolgt mit Hilfe von Übersetzungsschemata:

1. für jedes Attribut ein Vorwärtssübersetzungsschema: `forward_schema(ATTR,VALUE,REPR)`;
2. zu jedem Typ ein Rückwärtsübersetzungsschema: `backward_schema(TYP,AVM,REPR,END1,END2).`

Die Schemata werden in der Reihenfolge der Typdeklarationen angelegt. Die Typdeklarationen von oben bewirken die folgenden Einträge in der Datenbasis:

```
forward(phon,PHON,avm([sign|_],[PHON|_])).
forward(synsem,SYNSEM,avm([sign|_],[_,SYNSEM|_])).
forward(qstore,QSTORE,avm([sign|_],[_,_,QSTORE|_])).

backward(sign,
          [qstore:QSTORE,synsem:SYNSEM,phon:PHON],
          avm([sign|END1],[PHON,SYNSEM,QSTORE|END2]),
          END1,END2).


forward(dtrs,DTRS,avm([sign,phrase|_],[_,_,_,DTRS|_])).

backward(phrase,
          [dtrs:DTRS,qstore:QSTORE,synsem:SYNSEM,phon:PHON],
          avm([sign,phrase|END1],[PHON,SYNSEM,QSTORE,DTRS|END2]),
          END1,END2).


forward(head_dtr,HEAD_DTR,avm([headed|_],[HEAD_DTR|_])).
forward(comp_dtrs,COMP_DTRS,avm([headed|_],[_,COMP_DTRS|_])).

backward(headed,
          [comp_dtrs:COMP_DTRS,head_dtr:HEAD_DTR],
          avm([headed|END1],[HEAD_DTR,COMP_DTRS|END2]),
          END1,END2).
```

3.2.5 Unifikation in allgemeinen Typenverbänden

Auch allgemeine Typenhierarchien lassen sich sehr effizient implementieren. Das wird in dem Aufsatz [AïBoLiNa 89] gezeigt. Dazu werden die Typen in einer ganz speziellen Weise codiert, so daß die Operationen Unifikation und Disjunktion in nahezu konstanter Zeit erledigt werden können. Das soll im folgenden weiter ausgeführt werden.

Gegeben sei die in Abbildung 11 (a) dargestellte Typenhierarchie $(\mathcal{S}, \preceq)$ nach [AïBoLiNa 89, 121]. Ein naives Verfahren zur Berechnung der Unifikation zweier Typen, z.B. $b \sqcap e$, bestimmt einfach alle unteren Schranken der beiden Typen $\underline{b} = \{a, b\}$ und $\underline{e} = \{a, b, c\}$. Die Schnittmenge $\underline{b} \cap \underline{e} = \{a\}$ ist dann das Ergebnis der Unifikation. Die Typenhierarchie ist in einen Verband gemäß Definition 11 einzubetten. Es ist leicht einzusehen, daß diese Suche im Typenhierarchiegraphen ein zeitraubendes – wenn auch korrektes – Verfahren ist.

Die Idee in [AïBoLiNa 89, 115-116] ist es, einen zu $(\mathcal{S}, \sqcap, \sqcup)$ isomorphen Verband $(\mathcal{T}, \wedge, \vee)$ zu finden, in dem sich die größten unteren und kleinsten oberen Schranken – also die Operationen $\wedge$ und $\vee$ – einfach und effizient berechnen lassen. Falls $\gamma : \mathcal{S} \rightarrow \mathcal{T}$ ein solcher Isomorphismus, d.h. eine Bijektion, ist, dann läßt sich die Unifikation $s = s_1 \sqcap s_2$ in $\mathcal{S}$ reduzieren auf die Berechnung von $t = \gamma(s_1) \wedge \gamma(s_2)$ in $\mathcal{T}$. Es gilt dann $s = \gamma^{-1}(t)$, wobei diese Rücktransformation nur für die Ausgabe wirklich benötigt wird. Alle internen Berechnungen können vollständig in $\mathcal{T}$ erfolgen.

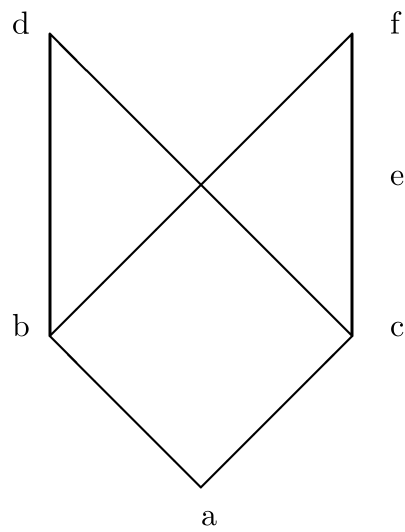
Die Autoren geben nun eine Codierung durch sogenannte *Bitstrings* an, die alle gewünschten Eigenschaften erfüllt [AïBoLiNa 89, 121-124]. Dazu ist die Matrix für die Ordnungsrelation $\preceq$ aufzustellen. Sie ist die reflexive und transitive Hülle der Vererbungsrelation $\prec$ (siehe Definition 1). Falls $s_1 \preceq s_2$ gilt, ist eine 0 in das Feld in Zeile s_1 und Spalte s_2 einzutragen, andernfalls eine 1. Die Matrix für unser Beispiel ist in Abbildung 11 (b) dargestellt. Sie kann z.B. durch Matrizenmultiplikation errechnet werden [AïBoLiNa 89, 122].

Die Typen können nun folgendermaßen codiert werden: Der Typ s ist durch die entsprechende Zeile in der Matrix zu repräsentieren. Es ist z.B. $\gamma(b) = 110000$ und $\gamma(e) = 101010$. Die Operationen $\wedge$ und $\vee$ korrespondieren einfach mit der bitweisen *and*- bzw. *or*-Verknüpfung [AïBoLiNa 89, 122]. Folglich ist $b \sqcap e = \gamma^{-1}(110000 \wedge 101010) = \gamma^{-1}(100000) = a$. Die Verknüpfungen stehen meist als Maschineninstruktionen zur Verfügung und können daher sehr effizient implementiert werden.

Die Decodierung ist im allgemeinen komplizierter als im obigen Beispiel. Es können nämlich disjunktive Typen entstehen, bedingt durch die Konstruktion des Verbands gemäß Definition 11. Für die Funktion γ^{-1} gilt im allgemeinen nach [AïBoLiNa 89, 120] $\gamma^{-1}(t) = \lceil\{s \in \mathcal{S} \mid \gamma(s) \preceq t\}\rceil$, also z.B. $d \sqcap f = \gamma^{-1}(111000) = \lceil\{a, b, c\}\rceil = b \sqcup c$.

Obwohl das Verfahren sehr schnell ist, vergeudet es doch einiges an Speicherplatz: Angenommen, die Typenhierachie umfaßt N Typen, d.h. $|\mathcal{S}| = N$. Dann werden zur Codierung aller dieser Typen N^2 Bits benötigt, für jeden Typ ein Bitstring der Länge N. Im Vergleich dazu ist die in UBS verwendete Codierung – wenn sie geschickt durchgeführt wird – kürzer; sie eignet sich natürlich auch nur für einfache Typenhierarchien. In UBS können die N Typen nämlich durch jeweils $\log(N)$ Bits dargestellt werden. Unter der Annahme, daß die Pfadlänge im Typenhierarchiebaum im Mittel ebenfalls $\log(N)$ beträgt, werden zur Codierung aller Typen $N \cdot \log^2(N)$ Bits benötigt.

(a) Typenhierachie



(b) Reflexive und transitive Hülle der Vererbungsrelation

	a	b	c	d	e	f
a	1	0	0	0	0	0
b	1	1	0	0	0	0
c	1	0	1	0	0	0
d	1	1	1	1	0	0
e	1	0	1	0	1	0
f	1	1	1	0	1	1

Abbildung 11: Ein Typenverband

Die gerade vorgestellte Codierung für allgemeine Typenhierarchien kann aber noch entscheidend verbessert werden, so daß insgesamt schon $N \cdot \log(N)$ Bits ausreichen können. Eine erste Verbesserung ist die, daß nicht für jeden Typen ein Bit reserviert wird. Wie das möglich ist, steht in [AïBoLiNa 89, 126-130]. Das wird als *kompakte* Codierung bezeichnet.

Weiter können die Typen in Module aufgeteilt werden. Der Bitstring für jeden Typ ist dann zweigeteilt: Die vorderen Bits geben das Modul an, die hinteren legen den Platz in der Hierarchie innerhalb des Moduls fest. Diese Vorgehensweise nennt sich *modulierte* Codierung. Sie ist jedoch nicht immer anwendbar und erfordert auch mehr Aufwand zur Durchführung der Operationen $\wedge$ und $\vee$.

3.3 Unifikation von Mengen

Betrachtet werden soll nun die Unifikation zweier endlicher Mengen A und B, die nur einfache Terme als Elemente besitzen. Die Möglichkeit, daß Mengen selbst wieder Mengen als Elemente besitzen, ist in den folgenden Betrachtungen also ausgeschlossen. Das gilt jedoch nicht für einfache Merkmalstrukturen über einer einfachen Typenhierarchie, da diese, wie im letzten Abschnitt gezeigt, auf Terme reduziert werden können.

Das Problem ist mehrfach in der Literatur behandelt worden, wobei manchmal jedoch das allgemeinere Problem der sogenannten ACI-Unifikation [Büttner 86], [LinChr 88], manchmal das speziellere Problem des Matchings von Mengen [ShmTsuZan 92], [Jayaraman 92] erörtert wird. Manchmal ist die Formulierung des Problems rein mathematisch [Rounds 88], [PolMos 90], ohne Angabe eines brauchbaren Algorithmus. In diesem Abschnitt geht es aber gerade um das Entwickeln von – nicht allzu ineffizienten – Verfahren zur Unifikation von Mengen. Bevor es dazu kommt, sind erst noch einige Zusammenhänge zu erklären.

3.3.1 Begriffsdefinitionen

Falls σ Unifikator zweier Terme s und t ist, so heißt die Unifikationsgleichung $s = t$, im folgenden einfach nur *Gleichung* genannt, erfüllbar mit der *Lösung* σ. Eine endliche Menge von Gleichungen S, im folgenden *System* genannt, heißt *erfüllbar* mit der Lösung σ, falls σ alle Gleichungen in S erfüllt.

Satz 1: Aus der Existenz eines allgemeinsten Unifikators für zwei Terme folgt, daß jedes erfüllbare System eine (eindeutige) allgemeinste Lösung σ besitzt. [ShmTsuZan 92, 96] □

Satz 2: Sei S ein System und σ dessen allgemeinste Lösung. Dann besitzen auch alle Teilsysteme $T \subseteq S$ eine allgemeinste Lösung $\tau \geq \sigma$.

Beweis: Da σ eine Lösung von S ist, erfüllt sie alle Gleichungen in S, insbesondere also die aus T wegen $T \subseteq S$. Darum hat T ebenfalls σ als Lösung. Für die allgemeinste Lösung τ von T, die nach Satz 1 existiert, muß aber $\tau \geq \sigma$ gelten, weil τ allgemeiner als σ ist. □

3.3.2 Was sind Mengen?

Mengen in diesem Zusammenhang sind partielle Beschreibungen von Mengen von Objekten. Sie besitzen im Prinzip die gleichen Eigenschaften wie die aus der Mathematik bekannten Mengen:

Kommutativität*: Die Reihenfolge, in der die Elemente einer Menge aufgezählt werden, spielt keine Rolle.

Idempotenz*: Elemente, die ein und dasselbe Objekt beschreiben, werden nur einmal aufgeführt.

Die Unifikation zweier Mengen A und B beschreibt nun eine Menge von Objekten, die durch beide Mengen A und B beschrieben werden kann. Für die Beziehung zwischen einer Menge(nbeschreibung) und einer durch sie beschriebenen Menge von Objekten gilt nach [PolSag 87, 47], daß

1. jedes Objekt der beschriebenen Menge erfaßt wird durch mindestens ein Element der Mengenbeschreibung und
2. jedes Element der Mengenbeschreibung Informationen über genau ein Element der Menge von Objekten liefert,

mit anderen Worten: es besteht eine Surjektion von der Mengenbeschreibung in die Menge der beschriebenen Objekte.

Ein Unifikator zweier Mengen A und B ist eine Substitution σ, für die $A\sigma = B\sigma$ gilt. Aus der Definition der Gleichheit für zwei Mengen ($A = B$ genau dann, wenn $A \subseteq B$ und $B \subseteq A$ gilt), läßt sich die folgende Definition für die Unifikation von Mengen ableiten:

Definition: σ heißt *Unifikator zweier Mengen* A und B, wenn es zu jedem $x \in A$ ein $y \in B$ gibt – und umgekehrt –, so daß $x\sigma = y\sigma$ gilt. [KapuNare 86, 492] Der Begriff entspricht dem der Bisimulation in [Rounds 88, 11]. Ebenfalls verträglich ist die obige Definition mit den Ausführungen in [PolMos 90]. □

Unifikatoren von Mengen lassen sich berechnen, indem Systeme von Unifikationsgleichungen gelöst werden. Wie sich aus der obigen Definition schon erahnen läst, besteht ein solches System aus Gleichungen, in denen immer ein Element aus A mit einem aus B gleichgesetzt wird. Diese Zuordnungen von Elementen kann mathematisch am besten durch Relationen beschrieben werden. Eine andere Möglichkeit der Darstellung bieten Matrizen, die nur mit 0 oder 1 besetzt sind. Matrizen werden oft für die Beschreibung von Verfahren zur ACI-Unifikation benutzt.

Schreibweisen: Sei $R \subseteq A \times B$ eine zweistellige Relation mit der Eigenschaft, sowohl links- als auch rechtstotal zu sein, d.h. zu jedem $x \in A$ gibt es ein $y \in B$ – und umgekehrt – mit xRy. Diese Eigenschaft einer Relation wird im folgenden mit $\mathcal{T}(R)$ abgekürzt. Das aus einer Relation $R \subseteq A \times B$ abgeleitete System $S(R)$ sei definiert als $\{x = y \mid xRy\}$. Die Eigenschaft eines abgeleiteten Systems $S(R)$, erfüllbar zu sein, wird im folgenden abgekürzt mit $\mathcal{S}(R)$. σ_R bezeichnet dann seine allgemeinste Lösung. □

3.3.3 Ein einfaches Verfahren

Mit den obigen Schreibweisen läßt sich ein einfaches Verfahren zur Bestimmung der Unifikatoren zweier Mengen A und B angeben:

Verfahren 1: Man bestimme unter allen Relationen $R \subseteq A \times B$ (davon gibt es genau $2^{|A| \cdot |B|}$, also nur endlich viele) alle diejenigen mit $\mathcal{T}(R)$ und versuche dann, das System S(R) zu lösen, indem man sukzessive alle zueinander in Relation stehenden Elemente aus A und B miteinander unifiziert. Gibt es eine Lösung, so hat man einen Unifikator σ von A und B gefunden. □

Das Verfahren ist natürlich sehr ineffizient, was wegen der NP-Vollständigkeit der Unifikation von Mengen, nachgewiesen in [KapuNare 86, 492-493], aber nicht ganz verwunderlich ist. Die Menge aller durch dieses Verfahren berechneten Lösungen ist $M_1 = \{\sigma_R \mid \mathcal{T}(R) \wedge \mathcal{S}(R)\}$.

Satz 3: Im allgemeinen ist die Unifikation zweier Mengen nicht eindeutig, d.h. es gibt keinen eindeutigen allgemeinsten Unifikator, der allgemeiner ist als alle anderen Unifikatoren der beiden Mengen.

Beweis: Es genügt, ein Beispiel anzugeben. Sei $A = \{x_1, x_2\}$ eine Menge von Variablen, $B = \{c_1, c_2\}$ eine Menge von Konstanten. Es gilt für $\tau_1 = \{x_1 \leftarrow c_1, x_2 \leftarrow c_2\}$, aber auch für $\tau_2 = \{x_1 \leftarrow c_2, x_2 \leftarrow c_1\}$, daß sie Unifikatoren von A und B sind. Kein Unifikator ist jedoch allgemeiner als der andere, wie sich leicht nachweisen läßt. □

Eine Konsequenz aus Satz 3 ist, daß die Unifikation von Mengen A und B nicht auf die in PROLOG eingebaute Unifikation von Termen, die immer eindeutig ist, zurückgeführt werden kann. Statt einem allgemeinsten Unifikator σ ist eine Menge von allgemeinsten Unifikatoren M zu berechnen, für die nach [Siekmann 84, 16] folgendes gelten muß:

1. **Korrektheit:** Jedes $\sigma \in M$ ist ein Unifikator von A und B.
2. **Vollständigkeit:** Zu jedem überhaupt möglichen Unifikator τ von A und B gibt es ein $\sigma \in M$ mit $\sigma \geq \tau$.
3. **Minimalität:** Falls σ_1, σ_2 aus M sind, dann muß $\sigma_1 \geq \sigma_2$ genau dann gelten, wenn $\sigma_2 \geq \sigma_1$ ist, d.h. entweder ist $\sigma_1 = \sigma_2$ (bis auf Variablenumbenennungen) oder keine Lösung ist allgemeiner als die andere.

Satz 4: Die durch das Verfahren 1 berechnete Menge M_1 ist zwar korrekt und vollständig, aber nicht minimal.

Beweis: Die Korrektheit folgt direkt aus den weiter oben eingeführten Definitionen und Schreibweisen. Der Beweis der Vollständigkeit wird an dieser Stelle ausgelassen. Er wird aber durch den entsprechenden Beweis für das noch vorzustellende Verfahren 2, das die Menge von Unifikatoren M_2 berechnet, abgedeckt wegen $M_2 \subseteq M_1$.

Zur Widerlegung der Minimalität genügt die Angabe eines Beispiels: Seien $A = B = \{x, y\}$ identische Mengen von Variablen. Wegen der Idempotenz der Unifikation allgemein ist die Unifikation zweier identischer Mengen A und B identisch mit A bzw. B, d.h. die Menge der allgemeinsten Unifikatoren M von A und B enthält in diesem Fall nur die leere Substitution ϵ. Verfahren 1 liefert aber $M_1 = \{\epsilon, \tau\}$ als Lösungsmenge, wobei $\tau = \{x \leftarrow y\}$ (bzw. $\tau = \{y \leftarrow x\}$) ist. Es gilt $\epsilon \geq \tau$, aber nicht $\tau \geq \epsilon$, was zu zeigen war. □

3.3.4 Das Verfahren in UBS

Verfahren 2: Der Suchraum nach Lösungen läßt sich im Vergleich zu Verfahren 1 noch wesentlich verkleinern, indem unter den Relationen $R \subseteq A \times B$ mit $\mathcal{T}(R)$ nur die minimalen aufgezählt werden. Minimal soll hierbei bedeuten, daß für alle $R' \subseteq R$ mit $\mathcal{T}(R')$ folgt, daß $R' = R$ ist; diese Eigenschaft wird im folgenden mit $\mathcal{M}(R)$ abgekürzt. Die Lösungen werden in der Menge $M_2 = \{\sigma_R \mid \mathcal{T}(R) \wedge \mathcal{M}(R) \wedge \mathcal{S}(R)\}$ gesammelt. □

Satz 5: Die durch das Verfahren 2 berechnete Menge von Unifikatoren ist korrekt und vollständig.

Beweis: Wegen $M_2 \subseteq M_1$ und der Korrektheit von M_1 folgt sofort die Korrektheit von M_2. Die Vollständigkeit ist schwieriger zu zeigen: Sei τ ein Unifikator der Mengen A und B. Dann folgt aus der Definition der Unifikation von Mengen, daß $\mathcal{T}(R_\tau)$ gilt, wobei $xR_\tau y$ genau dann gelte, wenn $x\tau = y\tau$ zutrifft. Jede Relation R_τ mit $\mathcal{T}(R_\tau)$ besitzt aber (mindestens) eine Teilmenge $R \subseteq R_\tau$ mit $\mathcal{T}(R)$ und $\mathcal{M}(R)$; das kann aus der Definition für minimal geschlossen werden. Mit Satz 2 folgt jetzt $\sigma_R \geq \sigma_{R_\tau} \geq \tau$ und damit die Behauptung. □

Das Verfahren 2 läßt sich relativ einfach implementieren, indem die in Frage kommenden minimalen Relationen R durch ein Verfahren mit Backtracking nacheinander erzeugt werden. Es ist natürlich darauf zu achten, daß jede mögliche Relation R nur genau einmal gebildet wird. Beschleunigt wird die Vorgehensweise dadurch, daß sofort, wenn ein Paar $(x, y) \in A \times B$ in R aufgenommen wird, die Unifikation von x mit y durchgeführt wird. Falls die Unifikation fehlschlägt, kann direkt Backtracking ausgelöst werden. Das Verfahren ist in der neuen UBS-Version implementiert.

Algorithmus: Es handelt sich um ein nicht-deterministisches Verfahren, das jede Lösung von Verfahren 2 nur genau einmal ermittelt.

1. Man wähle ein Element $x \in A$ aus und unifiziere es mit mindestens zwei (verschiedenen) Elementen $y_1, y_2 \in B$. Die miteinander unifizierten Elemente werden aus den Mengen A und B entfernt.

2. Der erste Schritt wird für eine gewisse Anzahl von Elementen von A wiederholt. Es ist auch möglich, daß diese Zahl gleich null ist.

3. Die restlichen Elemente $y \in B$ werden mit jeweils mindestens einem der verbliebenen Elemente $x \in A$ unifiziert. Auch hier werden die miteinander unifizierten Elemente aus den Mengen A und B entfernt.

4. Zum Schluß müssen beide Mengen leer sein. Dann ist ein Unifikator gegeben durch das System, das alle miteinander unifizierten Elemente aus A und B gleichsetzt. □

Auffällig an dem Algorithmus ist, daß in Schritt 1 und 2 jeweils einem Element aus A mindestens zwei Elemente aus B zugeordnet werden müssen, während für die Zuordnung in umgekehrter Richtung in Schritt 3 schon ein Element ausreicht. Diese Asymmetrie im Verfahren ist nötig, um zu garantieren, daß jede Lösung, d.h. jeder Unifikator, tatsächlich nur einmal berechnet wird. Das soll nun an einem Beispiel klargemacht werden.

Seien wie im Beweis zu Satz 3 die Mengen $A = \{x_1, x_2\}$ und $B = \{c_1, c_2\}$ gegeben. Sie besitzen die zwei angegebenen Unifikatoren, die der Algorithmus auch nur jeweils genau

einmal berechnet. Falls nun im obigen Algorithmus der Schritt 1 dahingehend modifiziert wird, daß nur mindestens ein Element aus B mit dem Element aus A unifiziert werden braucht, dann wird jede der beiden Lösungen gleich viermal berechnet. Diese Modifikation wollen wir als Verfahren 2* bezeichnen.

Abbildung 12 verdeutlicht das für die zuerst gefundene Lösung: Die Elemente der beiden Mengen sind dabei als Punkte dargestellt – links die von A und rechts die von B. Ein Pfeil von links nach rechts bedeutet eine Zuordnung von Elementen im Sinne von Schritt 1 und 2, ein Pfeil in umgekehrter Richtung eine Zuordnung im Sinne von Schritt 3 des Algorithmus. Jede der vier möglichen Zuordnungen bestimmt ein und dasselbe System, bestehend aus den Gleichungen $x_1 = c_1$ und $x_2 = c_2$. Es genügt also, nur eine der Zuordnungen zu betrachten. Der Algorithmus in der ursprünglichen Form berechnet auch nur die in der Abbildung letzte Zuordnung.

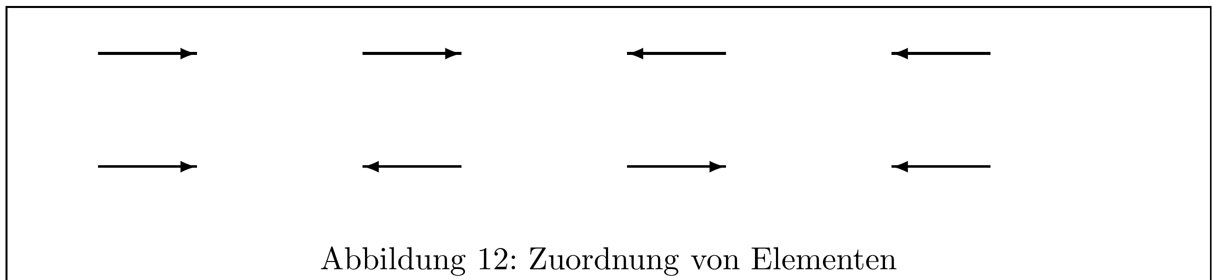


Abbildung 12: Zuordnung von Elementen

3.3.5 Eine Handvoll Beispiele

Abbildung 13 zeigt einige Beispiele zur Unifikation von Mengen mit Merkmalstrukturen als Elementen. Sie sind aus [Stolzenburg 92, 22] übernommen. In UBS können im Gegensatz zu UBS* alle Unifikationen berechnet werden.

Leider erfüllt auch die Menge M_2 nicht für jede zwei miteinander zu unifizierenden Mengen A und B die Bedingung der Minimalität, denn das Verfahren 2 liefert für die Mengen aus dem Beweis zu Satz 4 die gleiche Lösungsmenge wie Verfahren 1. Ein verbessertes Verfahren müßte berücksichtigen, welche Gleichheiten zusätzlich zu den durch eine Relation R mit $\mathcal{T}(R)$ und $\mathcal{S}(R)$ explizit geforderten gelten bzw. impliziert werden. Hier scheint es nützlich, den Begriff der erweiterten Relation $\hat{R}$ zu R einzuführen (unter der Vorbedingung, daß $\mathcal{S}(R)$ gilt): $x\hat{R}y$ gelte genau dann, wenn $x\sigma_R = y\sigma_R$ zutrifft. Es ist klar, daß $R \subseteq \hat{R}$ ist.

Man betrachte nun das folgende Beispiel: Sei

$$A = \{x, f(y_1), g(y_1), g(z_1)\} \text{ und } B = \{x, f(y_2), g(y_2), g(z_2)\}.$$

Dann gilt $x\hat{R}x$ immer, unabhängig vom gewählten R, und das Vorhandensein von $f(y_1)Rf(y_2)$ würde $g(y_1)\hat{R}g(y_2)$ implizieren (und umgekehrt).

Die Menge der allgemeinsten Unifikatoren für diese beiden Mengen enthält nur 3 Elemente, wohingegen Verfahren 2 gleich 17 Lösungen (Relationen) angibt. Abbildung 14 zeigt die berechneten Lösungen; die allgemeinsten sind durchgezogen eingerahmt. Die Elemente der Menge sind einfach als Punkte dargestellt, die Relationen R durch Linien, die Elemente verbinden.

Ein auf den erweiterten Relationen basierendes Verfahren bringt das Problem mit sich,

1. $\{[\textsc{type}\colon \textit{car}], [\textsc{colour}\colon \textit{purple}]\} \sqcap \{[\textsc{size}\colon \textit{big}]\} = \left\{ \begin{bmatrix} \textsc{type}\colon \textit{car} \\ \textsc{colour}\colon \textit{purple} \\ \textsc{size}\colon \textit{big} \end{bmatrix} \right\}$

2. $\{[\textsc{type}\colon \textit{bus}], [\textsc{colour}\colon \textit{blue}]\} \sqcap \{[\textsc{colour}\colon \textit{blue}], [\textsc{colour}\colon \textit{yellow}]\} =$
$\left\{ \begin{bmatrix} \textsc{type}\colon \textit{bus} \\ \textsc{colour}\colon \textit{yellow} \end{bmatrix}, [\textsc{colour}\colon \textit{blue}] \right\}$

3. $\{[\textsc{type}\colon \textit{car}], [\textsc{type}\colon \textit{bike}]\} \sqcap \{[\textsc{colour}\colon \textit{grey}], [\textsc{colour}\colon \textit{red}]\} =$
$\left\{ \begin{bmatrix} \textsc{type}\colon \textit{car} \\ \textsc{colour}\colon \textit{grey} \end{bmatrix}, \begin{bmatrix} \textsc{type}\colon \textit{bike} \\ \textsc{colour}\colon \textit{red} \end{bmatrix} \right\} \vee$
$\left\{ \begin{bmatrix} \textsc{type}\colon \textit{car} \\ \textsc{colour}\colon \textit{red} \end{bmatrix}, \begin{bmatrix} \textsc{type}\colon \textit{bike} \\ \textsc{colour}\colon \textit{grey} \end{bmatrix} \right\}$

4. $\{[\textsc{type}\colon \textit{bike}], [\textsc{type}\colon \textit{bike}], [\textsc{colour}\colon \textit{green}]\} \sqcap$
$\{[\textsc{type}\colon \textit{bike}], [\textsc{colour}\colon \textit{green}], [\textsc{colour}\colon \textit{green}]\} =$
$\{[\textsc{type}\colon \textit{bike}], [\textsc{colour}\colon \textit{green}]\}$

Abbildung 13: Unifikation von Mengen

fortwährend die Identität von Elementen aus A und B untersuchen zu müssen, was die Effizienz des Verfahren aber wohl stark mindern wird. Hier sind weitere Nachforschungen nötig.

Eine Verbesserung des Verfahrens 2 ist jedoch noch möglich: Dazu ist der Schritt 1 des Algorithmus wie folgt zu verändern: Falls das gewählte Element $x \in A$ ebenfalls in B vorkommt, dann ist x sofort aus B zu entfernen. In analoger Weise ist auch Schritt 3 zu optimieren. Wir wollen diese Modifikation das Verfahren 2** nennen. Der Beweis der Korrektheit und Vollständigkeit für Verfahren 2 ist auf diese Modifikation zu übertragen.

Das Verfahren 2** ist in UBS implementiert. Es werden dann nur noch die 9 überhaupt eingerahmten Lösungen aus Abbildung 14 berechnet für die beiden Mengen A und B. Das bedeutet eine weitere, deutliche Verbesserung. Aber auch diese Verbesserung erzeugt noch keine Lösungsmenge, die die Bedingung der Minimalität erfüllt, wie man sieht.

3.3.6 Vergleich der Verfahren

In der Tabelle unten werden alle Verfahren miteinander verglichen. Dabei ist angegeben, wie viele Relationen die Verfahren jeweils berechnen. Als Beispiel wird die Unifikation zweier identischer Mengen betrachtet, die ein (1:1), zwei (2:2), drei (3:3) bzw. vier (4:4) verschiedene Variablen enthalten. Die letzte Spalte (A:B) zeigt die Anzahlen für das Beispiel aus Abbildung 14. In der letzten Zeile ist angegeben, wie viele Elemente die Menge der allgemeinsten Unifikatoren für die jeweiligen Mengen (minimal) enthält.

x x

$f(y_1)$ $f(y_2)$

$g(y_1)$ $g(y_2)$

$g(z_1)$ $g(z_2)$

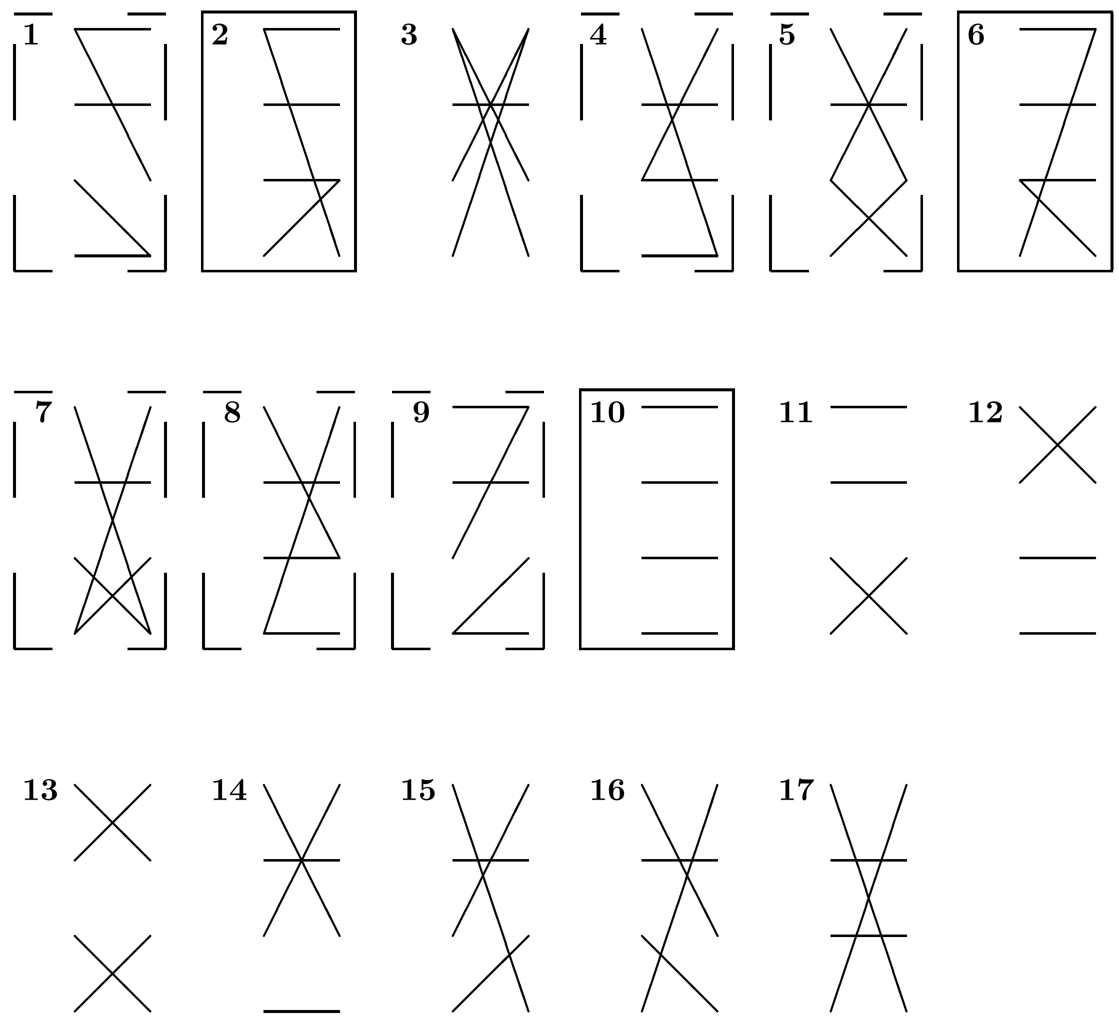


Abbildung 14: Relationen zwischen Mengen

Verfahren	1:1	2:2	3:3	4:4	A:B
1	2	16	512	65536	65536
2*	2	8	57	688	146
2	1	2	15	184	17
2**	1	1	7	49	9
∞	1	1	1	1	3

3.3.7 Andere Ansätze

Ein Blick in die Literatur zeigt, daß die Unifikation von Mengen ein Gebiet ist, auf dem zur Zeit noch aktiv geforscht wird. Die folgenden Ausführungen sollen einen kleinen Einblick in andere Ansätze zur Behandlung des Problems gewähren. Man kann die Herangehensweisen im wesentlichen unter drei Schlagwörtern einordnen: ACI-Unifikation, Termersetzungssysteme und Matching von Mengen.

ACI-Unifikation: Viele Aufsätze beschäftigen sich mit dem Problem der Unifikation von Funktionen, für die das Assoziativ-, das Kommutativ- und das Idempotenzgesetz oder eine Teilmenge dieser Gesetze gelten. Diese Eigenschaften einer (zweistelligen) Funktion f lassen sich mit den folgenden drei Axiomen nach [Siekmann 84, 19] festhalten:

Assoziativität: $f(f(x,y),z) = f(x,f(y,z))$

Kommutativität: $f(x,y) = f(y,x)$

Idempotenz: $f(x,x) = x$

Die Unifikation von Mengen kann nun nach [ShmTsuZan 92, 93-94] folgendermaßen als Spezialfall der ACI-Unifikation behandelt werden: Der Funktor f entspricht der Vereinigungsbildung von Mengen $\cup$. Dann stehen aber Variablen, die als Argumente des Funktors f auftreten für Teilmengen und nicht für einzelne Elemente der Menge, wie eigentlich gewünscht.

Um dem abzuhelfen, wollen wir das einstellige Funktionssymbol g verwenden, an das keine Axiome geknüpft sind. Der Funktor g fungiert als Konstruktor einelementiger Mengen $\{\cdot\}$. Weiterhin ist es notwendig, ein neutrales Element n bezüglich der Funktion f einzuführen. Es bezeichnet die leere Menge $\emptyset$. Dann gelten z.B. die folgenden Entsprechungen:

$$\begin{aligned} \emptyset &\,\hat{=}\, n \\ \{a\} &\,\hat{=}\, g(a) \\ \{x, t(x), y\} &\,\hat{=}\, f(g(x), f(g(t(x)), g(y))) \end{aligned}$$

Damit ist gezeigt, daß die Unifikation von Mengen auf die ACI-Unifikation reduziert werden kann. Es ist jedoch von vornherein anzunehmen, daß ein Algorithmus zur Bewältigung des zweiten, allgemeineren Problems eher ineffizienter ist als ein auf das speziellere Problem zugeschnittener. Trotzdem sollen nun einige Verfahren zur ACI-Unifikation aus der Literatur auf ihre Eignung für die Unifikation von Mengen hin untersucht werden.

In [Büttner 86] wird ein kompliziertes Verfahren entwickelt, das aber in [BaaBüt 88, 346] als unvollständig entlarvt wird. In dem neueren Aufsatz wird ein Rechenverfahren

angegeben, das nur Funktionen f zuläßt, die als Argumente Konstanten, Variablen oder Terme aussschließlich mit dem Funktor f besitzen. Sie werden dort – ein bißchen verwirrend – als Mengen, bestehend aus Konstanten und Funktionen, bezeichnet.

Wie oben zu sehen ist, müssen zur Behandlung der Unifikation von Mengen durch die ACI-Unifikation aber auch andere Terme als Argumente zugelassen sein, insbesondere Terme mit dem einstelligen Funktor g. Darum sind die in [Büttner 86] und [BaaBüt 88] vorgestellten Verfahren meiner Ansicht nach nicht geeignet für die Unifikation von Mengen in unserem Zusammenhang.

Ein anderer Vorschlag findet sich in [LinChr 88]. Die Autoren zerlegen die ACI-Unifikation in mehrere Phasen: in der ersten wird die Idempotenz, in der zweiten die Assozitivität und Kommutativität abgehandelt. Es ergibt sich die folgende Prozedur nach [LinChr 88, 360]:

1. Terme mit dem Funktionssymbol f werden flach gemacht. Es wird die folgende Termersetzungsregel angewendet:

$$f(t_1, \ldots, f(s_1, \ldots, s_m), \ldots, t_n) \rightarrow f(t_1, \ldots, s_1, \ldots, s_m, \ldots, t_n)$$

2. danach werden Subterme entfernt, die paarweise, in beiden Unifikanden vorkommen.

3. Um der Idempotenz gerecht zu werden, sind identisch doppelte Vorkommen von Konstanten, Termen oder Variablen in den einzelnen Unifikanden zu löschen.

4. Zum Schluß wird die Unifikation unter Berücksichtigung von Assoziativität und Kommutativität durchgeführt. Das Verfahren ist in der Arbeit genauer beschrieben [LinChr 88, 364-365].

Wir wollen nun ein Beispiel betrachten: Zu unifizieren seien die beiden Terme $t_1 = f(a, f(h(x), h(y)))$ und $t_2 = f(f(a, a), h(z))$. Die obige Prozedur ergibt nach den ersten drei Schritten $t_1 \rightarrow f(h(x), h(y))$ und $t_2 \rightarrow f(h(z))$. Diese beiden Termen können im vierten Schritt aber nicht miteinander unifiziert werden, weil unter Assoziativität und Kommutativität die Unifikation nur bei Unifikanden mit gleicher Stelligkeit überhaupt gelingen kann.

Tatsächlich sind die beiden Terme jedoch unifizierbar, und zwar mit der Substitution $\tau = \{x \leftarrow z, y \leftarrow z\}$. Das in [LinChr 88] beschriebene Verfahren ist daher meiner Ansicht nach nicht vollständig.

Termersetzungssysteme: Diese Herangehensweise setzt voraus, daß Mengen als Terme dargestellt werden können, für die in endlicher Zeit eine eindeutige Normalform abgeleitet werden kann. Das wird mit Hilfe von sogenannten konfluenten und terminierenden Termersetzungssystemen bewerkstelligt.

Eine Möglichkeit, Mengen durch Terme zu repräsentieren, geht folgendermaßen nach [ShmTsuZan 92, 94]: Der Term $g(e, s)$ steht für eine Menge, bestehend aus dem Element e vereinigt mit der Menge s. Für die leere Menge wird wiederum die Konstante n benutzt. Es gelten die folgenden Axiome:

Kommutativität*: $g(e, g(f, s)) = g(f, g(e, s))$

Idempotenz*: $g(e, g(e, s)) = g(e, s)$

Leider wurde aber für diese Axiome bisher kein Termersetzungssystem mit den gewünschten Eigenschaften gefunden [ShmTsuZan 92, 94]. Mit dem Problem der Termination von Termersetzungssystemen beschäftigen sich auch die Papiere [Weisweber 89] und [Weisweber 92]. Dort werden Termersetzungssysteme allgemein als Hilfsmittel für die Maschinelle Übersetzung angesehen.

Matching von Mengen: Matching ist ein Spezialfall der Unifikation. Die beiden Begriffe fallen nur dann zusammen, falls einer der zu unifizierenden Terme (Mengen) ein Grundterm ist, d.h. es kommen keine Variablen in ihm vor. Die Unifikation von Mengen ist also eigentlich nicht auf das Matching von Mengen reduzierbar, es sei denn, man zögert die Unifikation solange hinaus, bis eine der beiden Mengen nur noch Grundterme als Elemente enthält. Das ist aber wohl nicht immer möglich.

In dem Artikel [ShmTsuZan 92] schlagen die Autoren vor, Mengen in – im allgemeinen mehrere – Terme zu compilieren. Die Kommutativität* und Idempotenz* wird dabei berücksichtigt, indem alle möglichen Permutationen und Zusammenfassungen von Elementen der Menge erzeugt werden [ShmTsuZan 92, 105-109]. Die Codegröße kann exponentiell mit der Anzahl der Elemente in der Menge ansteigen. Es sind jedoch eine Reihe von Otimierungen vorgesehen. Es gilt etwa:

$$\{x, c\} \rightarrow set_of(x, c), set_of(c, x), set_of(x \leftarrow c)$$

In [Jayaraman 92] wird ein weiteres Verfahren vorgestellt. Mengen sind in der Form $\{e|s\}$ repräsentiert, wobei e ein Element ist und s eine Menge. Ein Algorithmus zum Matching von Mengen in PROLOG wird angegeben [Jayaraman 92, 317]. Der Artikel beschäftigt sich mit mengenwertigen Funktionen und ihrer Implementation. Er soll deshalb hier nicht näher betrachtet werden.

3.4 Funktionen und Negation

Das Problem, Funktionen in UBS einzuführen, und die Implementation der Negation können in SEPIA durch den gleichen Mechanismus behandelt werden, nämlich durch die Fähigkeit von SEPIA, Prädikate verzögert aufzurufen. Darum werden beide Probleme in *einem* Abschnitt zusammen abgehandelt.

3.4.1 Funktionen

Durch Funktionen werden in Logikprogrammen Terme mit einer bestimmten Semantik zugelassen, die ausgewertet werden. In UBS werden Funktionen durch Voranstellung des Dollarzeichens `$` von gewöhnlichen Termen unterschieden. Zur Illustration dessen, was mit Funktionen gemeint ist, wollen wir noch einmal die Funktion `append/3` zur Konkatenation zweier Listen als Beispiel bemühen. Sie kann in UBS – unter anderem – auch wie folgt definiert werden:

```
append([],LIST,LIST).
append([ELEM|LIST1],LIST2,[ELEM|$append(LIST1,LIST2)]).
```

Falls der Funktionsaufruf von `append(LIST1,LIST2)` wie ein gewöhnlicher Term bearbeitet wird, so würde die Anfrage `append([a],[b],L)` zur Antwort

$$\theta = \{\texttt{L} \leftarrow \texttt{[a|append([],[b])]}\}$$

führen. Das ist jedoch nicht gewünscht. Der Ausdruck `append([],[b])` sollte stattdessen zu `[b]` ausgewertet werden. Im folgenden sollen zwei Ansätze vorgestellt werden, wie diese Auswertung erreicht werden kann.

Ein Vorschlag zur Behandlung von Funktionen findet sich in [Furbach 91]. Dabei wird die Definition 19 verändert und eine sogenannte *erweiterte Unifikation* eingeführt: Bei der Unifikation zweier Atome werden die Terme, dessen Funktionssymbol mit einer Funktionsdefinition verknüpft ist, ausgewertet, soweit das möglich ist. Manchmal werden bei diesem Verfahren Unifikationsgleichungen in die Zielklausel aufgenommen.

Die eigentliche Auswertung eines Terms, dessen Funktionssymbol mit einer Funktionsdefinition verknüpft ist, geschieht durch die Funktion *eval* wie folgt nach [Furbach 91, 41-42]:

1. $eval(s) = s$, falls s eine Variable oder Konstante ist;

2. $eval(f(t_1, \ldots, t_n)) = (\mathrm{DEF}f)(t_1, \ldots, t_n)$, falls die Funktion f mit der Definition $\mathrm{DEF}f$ an der Stelle $(t_1, \ldots, t_n)$ definiert ist;

3. $eval(f(t_1, \ldots, t_n)) = f(eval(t_1), \ldots, eval(t_n))$, falls die Funktion f mit der Definition $\mathrm{DEF}f$ an der Stelle $(t_1, \ldots, t_n)$ nicht definiert ist oder f ein bloßer Konstruktor ist.

Die Funktionsausdrücke werden nach und nach ausgewertet. Da das Verfahren bewiesenermaßen terminiert, liefert es, integriert in Logikprogramme, alle korrekten Antworten zu einer Anfrage, d.h. das Verfahren ist vollständig und korrekt.

Diese Integration von logischer und funktionaler Programmierung verbindet die Effizienz funktionaler Programmierung und die Flexibilität in der Ausdrucksweise von Logikprogrammen miteinander. Durch die Einführung inverser Funktionen [Furbach 91, 76-77] kann die Effizienz und Flexibilität eines solchen Systems sogar noch weiter gesteigert werden.

3.4.2 Relationen

In UBS wird die Einbettung von Funktionen in die Sprache anders vorgenommen: Es sind nicht nur Funktionen, sondern Relationen im allgemeinen zugelassen, die durch PROLOG-Prädikate zu definieren sind. Wie das funktioniert, wurde schon auf Seite 2.2.1 beschrieben.

Dabei wird die Möglichkeit von SEPIA ausgenutzt, Prädikate verzögert aufrufen zu können. Der Benutzer von UBS muß oder sollte zumindest die in Funktionsaufrufen verwendeten Prädikate mit einer `delay`-Klausel versehen und so selbst für die Termination des Programms sorgen.

Wir wollen nun die Funktion `append/3` weiter strapazieren. Dazu betrachte man noch einmal die Definition im letzten Abschnitt (Seite 68). – Funktionen in UBS werden durch Prädikate definiert, deren letztes Argument als Resultat der Funktion gilt. Bei der Programmtransformation von UBS nach SEPIA entsteht genau die übliche Version von `append/3`, wie sie in vielen Lehrbüchern zu finden ist (siehe auch Seite 22).

Ein Problem bei der Verwendung von `append/3` kann es sein, daß der Aufruf

```
append(LIST1,[],LIST2)
```

eine unendliche Berechnung in Gang setzt. Bei jedem Backtracking, das ausgelöst wird – etwa durch ein fehlschlagendes, nachfolgend berechnetes Prädikat –, wird eine weitere Lösung erzeugt. Dabei entsteht die folgende unendliche Menge von Lösungen:

```
LIST1 = LIST2 =
1.   []
2.   [X1]
3.   [X1,X2]
4.   [X1,X2,X3]
     ...
```

Diese unendliche Berechnung kann durch eine `delay`-Klausel vermieden werden. Eine `delay`-Klausel [SEPIA 91, (1) 98-99] spezifiziert, unter welchen Bedingungen ein Prädikat verzögert aufgerufen wird. Die Syntax ist

```
delay PROC if BODY.
```

Dabei ist `PROC` ein Prädikat und `BODY` eine Klausel, die aus einer Konjunktion von ein oder mehreren Aufrufen der Prädikate `var/1`, `\==/2`, `nonground/1` oder einfachen externen Prädikaten besteht. Disjunktionen werden durch mehrere Klauseln in Folge ausgedrückt.

Die Semantik einer `delay`-Klausel ist die folgende: Soll eine Prozedur (Prädikat) `PRED` aufgerufen werden, für die eine solche Klausel definiert ist, so wird der Aufruf verzögert, falls für mindestens eine passende `delay`-Klausel gilt:

1. Der Aufruf `PRED` matcht das Prädikat `PROC`.
2. Die Prädikate im `BODY` sind erfüllt.

Für unser Beispiel ist die Definition einer `delay`-Klausel sinnvoll. Nachfolgend findet sich die Klausel zusammen mit einem Beispielaufruf in UBS. Der verzögerte Aufruf wird zum Schluß angegeben.

```
[sepia 6]: compile(user).
 delay append(LIST1,_,LIST2) if var(LIST1), var(LIST2).

 append([],LIST,LIST).
 append([ELEM|LIST1],LIST2,[ELEM|LIST3]) :-
     append(LIST1,LIST2,LIST3).
```

```
 user       compiled traceable 360 bytes in 0.02 seconds

yes.
[sepia 7]: append(LIST1,[],LIST2).
LIST1=_d288
LIST2=_d280

append(_d288,
       [],
       _d280).

yes.
[sepia 8]:
```

Die Implementation von Funktionen bzw. Relationen könnte noch stark verbessert werden, indem alle Eigenschaften der Funktionen ausgenutzt werden. Für das Beispiel gilt etwa folgendes:

1. Das Axiom, daß die leere Liste `[]` das neutrale Element bezüglich der Funktion `append`/3 ist, kann ausgenutzt werden. Dann läßt sich sofort `LIST1 = LIST2` für die Anfrage `append(LIST1,[],LIST2)` folgern.

2. Die Eigenschaft, daß `append`/3 eine wirkliche Funktion ist, kann hilfreich sein, falls mehrere Aufrufe von `append`/3 gleichzeitig in der Zielklausel vorkommen. Falls die Zielklausel nämlich die Aufrufe `append(LIST1,LIST2,RES1)` und `append(LIST1,LIST2,RES2)` enthält, können beide Aufrufe zu einem gekürzt werden, nämlich `append(LIST1,LIST2,RES)`, wobei `RES` die Unifikation von `RES1` und `RES2` ist. Dieser Schritt ist zulässig, weil Funktionen rechtseindeutige Relationen sind, d.h. aus der Gleichheit der Argumente folgt die Gleichheit des Resultats.

3.4.3 Die Negation

Die Negation ist in UBS in ziemlich ähnlicher Weise wie in UBS* [Stolzenburg 92, 18-20] implementiert: Die Negation von logischen Operationen wird nach den Regeln von De Morgan vereinfacht. Die Negation einer typisierten Merkmalstruktur t wird in eine Disjunktion umgesetzt. Sie umfaßt alle die Ausdrücke, deren Typ nicht mit dem von t unifizierbar ist, und die Merkmalstrukturen, die sich in mindestens einem Attribut von t unterscheiden. Man betrachte hierzu noch einmal das Beispiel auf Seite 27.

So wird die Negation letztendlich auf die Negation von Variablen, Konstanten bzw. Termen reduziert. Diese Fälle werden wiederum mit Hilfe des folgenden Prädikats `neq`/2 aus UBS bewerkstelligt. Es besitzt eine `delay`-Klausel, die die Verwendung des Parameters `NEG` aus UBS* überflüssig macht. Der Ausdruck

```
(IF -> THEN ; ELSE)
```

steht für eine Wenn-Dann-Klausel [SEPIA 91, (2) 21]. Der Cut darf im Zusammenhang mit einer `delay`-Klausel nicht benutzt werden bzw. führt zu unerwünschten Resultaten [SEPIA 91, (1) 103].

```
delay neq(X,Y) if var(X), Y\=='?', X\==Y.

neq(X,Y) :-
    X\==Y,                  % identische Terme
    X\=='?', Y\=='?',       % beliebige Struktur erlaubt
    (var(Y)                 % noch keine Entscheidung moeglich
    ->  neq(Y,X)
    ;   (X=..[FUNC|ARG1],   % gleichartige, zusammengesetzte Terme
         Y=..[FUNC|ARG2],
         pair(ARG1,ARG2,LIST)
         ->  member((X0,Y0),LIST),
             neq(X0,Y0)
         ;   true)).        % erwiesenermassen ungleiche Terme
```

Das System SEPIA stellt die konstruktive Negation zur Verfügung. Sie ist aber im Handbuch [SEPIA 91, (1) 125-127] nur sehr knapp dokumentiert. Das Prädikat `~=/2` in SEPIA entspricht dabei im Prinzip dem Prädikat `neq/2` in UBS. Beide Prädikate sollen die Ungleichheit ihrer Argumente gewährleisten. Doch in SEPIA werden Negationen nicht in Disjunktionen umgesetzt, und es finden keine Vereinfachungen statt, etwa `f(X) ~= f(c)` zu `X ~= c`.

3.5 Das Problem der Disjunktion

Mehrdeutigkeit ist ein Phänomen, das in jeder natürlichen Sprache zu beobachten ist, auf den verschiedensten Ebenen linguistischer Information. Die adäquate Darstellung linguistischer Daten erfordert daher die Möglichkeit, Disjunktionen ausdrücken zu können. Die Folge davon jedoch ist, daß sich nun mit Hilfe der Disjunktion NP-vollständige Probleme formulieren lassen [KasRou 86, 262]. Im schlimmsten Falle ergibt sich also exponentielles Laufzeitverhalten. Das sollte natürlich nach Möglichkeit vermieden werden.

Im folgenden werden einige Ansätze vorgestellt, wie man die Disjunktion implementieren kann in einem Formalismus, der ansonsten noch (mindestens) Merkmalstrukturen und Unifikation vorsieht. Die ersten hier vorgestellten Ansätze sind einfache Verfahren, die im allgemeinen recht ineffizient sind; die weiteren Ansätze versuchen, durch geschickte Behandlung von Disjunktionen des Eintreten des schlimmsten Falls zu verhindern.

Die Darstellung der Verfahren erfolgt anhand von Beispielen. Wer detailliertere Auskünfte haben möchte, sei auf die jeweils angegebenen Originalaufsätze verwiesen. Soweit möglich, wird ein Vergleich mit UBS angestrebt. Obwohl die Behandlung der Disjunktion in UBS (noch) keiner besonderen Optimierung unterzogen wird, lassen sich doch einige der dargestellten Verfahren in UBS imitieren.

3.5.1 Disjunktive Normalform

In [Smolka 89] wird rigoros formal eine Sprache zur Beschreibung von Merkmalstrukturen entwickelt. Sie kann als Spezialfall der Prädikatenlogik erster Stufe mit Gleichheit angesehen werden. Daraus abgeleitet wird eine Notation für Merkmalsterme.

Die Übersetzung von Merkmalstermen in Formeln der Beschreibungssprache ist möglich, sogar in linearer Zeit. Auf diese Weise können die Operationen Unifikation und Disjunktion von Merkmalstrukturen zurückgeführt werden auf eine Normalform durch ein Reduktionsverfahren. Jeder Normalform entspricht eine Merkmalstruktur. Die in dem Aufsatz beschriebene Vorgehensweise ist sehr ineffizient. Es geht dem Autor auch mehr um formale Genauigkeit als um die Beschreibung effektiver Verfahren.

Ein Merkmalsterm wird mit Hilfe der folgenden Gleichungen in disjunktive Normalform (DNF) überführt. Dabei ist immer das Auftreten einer linken Seite durch die rechte Seite der Gleichung zu ersetzen.

$$\begin{aligned} S \sqcap (T \sqcup U) &= (S \sqcap T) \sqcup (S \sqcap U) \\ (S \sqcup T) \sqcap U &= (S \sqcap U) \sqcup (T \sqcap U) \\ l : (S \sqcup T) &= l : S \sqcup l : T \end{aligned}$$

Die Gleichungen sind aus [Smolka 89, 33] übernommen. Dabei wird schon vorausgesetzt, daß eventuelle Negationen nur vor Konstanten oder Variablen stehen.

Die durch die Umformungen entstehende DNF eines Merkmalterms S hat die Gestalt $S_1 \sqcup \ldots \sqcup S_n$, wobei die einzelnen S_i keine Disjunktionen mehr enthalten. Sie heißen (Disjunktions)Glieder. Die einzelnen S_i können nun in Formeln der Beschreibungssprache übersetzt werden, Merkmalsklauseln genannt, die in quadratischer Zeit gelöst, d.h. auf Normalform reduziert werden können [Smolka 89, 25].

Der große Nachteil des Verfahrens ist, daß ein Merkmalsterm bei der Umformung in DNF exponentiell aufgebläht werden kann. Sie enthält viel Redundanz, da Merkmale beim Umformen vervielfältigt werden. Sie werden bei jeder Reduktion eines Glieds S_i immer wieder von neuem bearbeitet.

Das folgende Beispiel zeigt einen Merkmalsterm und seine zugehörige DNF.

Beispiel 1:

$$\begin{bmatrix} \text{WORD: } \textit{go} \\ \text{AGR: } [\text{NUM: } \textit{pl}] \sqcup \begin{bmatrix} \text{NUM: } \textit{sng} \\ \text{PER: } \textit{first} \sqcup \textit{second} \end{bmatrix} \end{bmatrix} =$$

$$\begin{bmatrix} \text{WORD: } \textit{go} \\ \text{AGR: } [\text{NUM: } \textit{pl}] \end{bmatrix} \sqcup \begin{bmatrix} \text{WORD: } \textit{go} \\ \text{AGR: } \begin{bmatrix} \text{NUM: } \textit{sng} \\ \text{PER: } \textit{first} \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{WORD: } \textit{go} \\ \text{AGR: } \begin{bmatrix} \text{NUM: } \textit{sng} \\ \text{PER: } \textit{second} \end{bmatrix} \end{bmatrix}$$

3.5.2 Eine verbesserte Methode

Das im folgenden vorgestellte Verfahren nach [Kasper 87] bewirkt eine optimierte Behandlung von Disjunktionen zur Laufzeit. Der Hauptgedanke dabei ist, die Expansion zur DNF solange wie möglich hinauszuschieben. Es setzt explizite Kontrolle von Disjunktions- und Unifikationsgliedern zur Laufzeit voraus.

Daher läßt es sich schlecht für UBS anwenden, weil dort die Philosophie verfolgt worden ist, die Behandlung von Merkmalstrukturen soweit wie möglich dem PROLOG-System zu überlassen. PROLOG führt aber von sich aus keine Optimierung bei der Ausführung von Disjunktionen aus.

In die Überlegungen einbezogen werden nur Merkmalsterme S der Form $T \wedge D_1 \wedge \ldots \wedge D_n$,

die sich aus zwei Bestandteilen zusammensetzen [Kasper 87, 236]:

1. **definit:** Das ist der Merkmalsterm T, der keine Disjunktionen enthält.
2. **indefinit:** Hierbei handelt es sich um eine Menge von Disjunktionen $D_1 \wedge \ldots \wedge D_n$. Jede Disjunktion D hat die Gestalt $S_1 \vee \ldots \vee S_m$. Die Merkmalsterme S müssen wieder die oben angegebene Form haben.

Die Disjunktionen dürfen, wie aus obiger Disjunktion zu ersehen, nur auf oberster Ebene auftreten und nur hinter Unifikationen von Merkmalstermen. Jeder Merkmalsterm kann jedoch auf diese Form gebracht werden, wobei jedoch zum Teil Merkmale vervielfältigt werden. Die Umformung ist mit Hilfe der folgenden Umformungen zu bewerkstelligen, die als Termersetzungsregeln (von links nach rechts) gelesen werden können:

$$\begin{array}{rcl} l : (S \sqcup T) & = & l : S \sqcup l : T \\ (S \sqcup T) \sqcap U & = & U \sqcap (S \sqcup T) \\ S \sqcup (T \sqcap U) & = & (T \sqcap U) \sqcup S \end{array}$$

Das Verfahren soll nun anhand eines Beispiels erklärt werden:

Beispiel 2: Miteinander zu unifizieren seien die beiden in der Abbildung gegebenen Merkmalsterme S und T. Die einzelnen Merkmalstrukturen sind alle mit einer Variablennummer versehen, z.B. der definite Bestandteil des ersten Mermalterms mit $\boxed{1}$, um auf sie einfacher Bezug nehmen zu können. Falls zwei Merkmalstrukturen miteinander unifiziert werden, erhält das Ergebnis die Aneinanderfügung beider Nummern annotiert.

$$\begin{array}{rcl}
S & = & \boxed{1}\begin{bmatrix} \textsc{rank}: \textit{clause} \\ \textsc{subj:case}: \textit{nom} \end{bmatrix} \\[2ex]
& \sqcap & \left(\boxed{2}\begin{bmatrix} \textsc{voice}: \textit{passive} \\ \textsc{trans}: + \\ \textsc{subj}: \boxed{8} \\ \textsc{goal}: \boxed{8} \end{bmatrix} \sqcup \boxed{3}\begin{bmatrix} \textsc{voice}: \textit{passive} \\ \textsc{subj}: \boxed{9} \\ \textsc{agent}: \boxed{9} \end{bmatrix} \right) \\[2ex]
& \sqcap & \left(\boxed{4}\begin{bmatrix} \textsc{trans}: - \\ \textsc{agent:per}: \textit{third} \end{bmatrix} \sqcup \boxed{5}\begin{bmatrix} \textsc{trans}: + \\ \textsc{goal:per}: \textit{third} \end{bmatrix} \right) \\[2ex]
& \sqcap & \left(\boxed{6}\begin{bmatrix} \textsc{num}: \textit{sng} \\ \textsc{subj:num}: \textit{sng} \end{bmatrix} \sqcup \boxed{7}\begin{bmatrix} \textsc{num}: \textit{pl} \\ \textsc{subj:num}: \textit{pl} \end{bmatrix} \right)
\end{array}$$

$$T = \boxed{0}\begin{bmatrix} \textsc{subj}: \begin{bmatrix} \textsc{per}: \textit{second} \\ \textsc{num}: \textit{pl} \end{bmatrix} \end{bmatrix}$$

Das Verfahren umfaßt drei Schritte, die zunehmend aufwendiger werden. Nicht immer sind alle Schritte zu durchlaufen, dann nämlich, wenn schon frühzeitig Inkonsistenz von Merkmalstermen erkannt werden kann. Oft terminiert das Verfahren schon, bevor Schritt 3 erreicht wird, welcher allein exponentielle Laufzeit verursachen kann.

Schritt 1: Zuerst werden die beiden definiten Anteile der beiden Merkmalsterme miteinander unifiziert. Das kann mit einem effizienten Algorithmus zur Unifikation von

Graphen geschehen. Die beiden indefiniten Bestandteile werden einfach miteinander vereinigt.

Falls die beiden definiten Anteile nicht miteinander unifizierbar sind, d.h. die entstehende Merkmalstruktur ist inkonsistent, dann kann das Verfahren schon an dieser Stelle abgebrochen werden. Das ist auch der Fall, wenn im indefiniten Teil keinerlei Disjunktionen vorhanden sind.

Das Ergebnis der Unifikation von S und T nach Schritt 1 zeigt die folgende Abbildung. Die Merkmalstrukturen sind zum Teil nur durch ihre Variablennummer repräsentiert:

$$
\begin{aligned}
S \sqcap T \; = \; & \boxed{10} \begin{bmatrix} \text{RANK: } \textit{clause} \\ \text{SUBJ: } \begin{bmatrix} \text{CASE: } \textit{nom} \\ \text{NUM: } \textit{pl} \\ \text{PER: } \textit{second} \end{bmatrix} \end{bmatrix} \\
\sqcap \; & \left(\boxed{2} \sqcup \boxed{3} \right) \\
\sqcap \; & \left(\boxed{4} \sqcup \boxed{5} \right) \\
\sqcap \; & \left(\boxed{6} \sqcup \boxed{7} \right)
\end{aligned}
$$

Schritt 2: Nun wird die Konsistenz der Disjunktionen mit dem definiten Anteil $\boxed{10}$ des Merkmalsterms überprüft. In jeder Disjunktion muß es mindestens ein Glied geben, das mit dem definiten Bestandteil unifizierbar ist.

Zu dieser Überprüfung ist es notwendig, daß das System sogenannte *Testunifikationen* durchführen kann, d.h. die Unifizierbarkeit zweier Merkmalstrukturen testen kann, ohne sie wirklich miteinander zu unifizieren. Dadurch werden Unifikationen zum Teil doppelt ausgeführt: einmal zum Testen und später eventuell wirklich. Das verbraucht Rechenzeit und auch Speicherplatz.

Für die ersten beiden Disjunktionen gilt, daß jeweils beide Glieder mit der Merkmalstruktur $\boxed{10}$ konsistent sind. In der letzten Disjunktion trifft das nur für das zweite Glied $\boxed{7}$, nicht aber für $\boxed{6}$ zu. In einem solchen Fall, wo nur genau ein Glied in Frage kommt, wird dieses mit dem definiten Bestandteil unifiziert und die Disjunktion entfernt.

Das Ergebnis ist nachfolgend dargestellt. Zu beachten ist, daß nach obigem Schritt wieder von vorne die Konsistenz der Disjunktionsglieder mit dem definiten Anteil untersucht werden muß.

$$
\boxed{107} \begin{bmatrix} \text{NUM: } \textit{pl} \\ \text{RANK: } \textit{clause} \\ \text{SUBJ: } \begin{bmatrix} \text{CASE: } \textit{nom} \\ \text{NUM: } \textit{pl} \\ \text{PER: } \textit{second} \end{bmatrix} \end{bmatrix} \sqcap \left(\boxed{2} \sqcup \boxed{3} \right) \sqcap \left(\boxed{4} \sqcup \boxed{5} \right)
$$

Wieder kann das Verfahren nach diesem Schritt terminieren, und zwar dann, wenn keine Disjunktionen mehr übrig sind oder eine Disjunktion nur mit dem definiten Bestandteil inkonsistente Glieder enthält. Dann schlägt die Unifikation insgesamt fehl.

Schritt 3: Der letzte Schritt ist der aufwendigste. Er entspricht im Prinzip der Expansion zur DNF. Es ist jedoch möglich, ihn ganz auszulassen und zu einem späteren Zeitpunkt nachzuholen. Hier gilt es, gute Heuristiken für geeignete Zeitpunkte zu ent-

wickeln.

Aus dem Merkmalsterm wird jetzt eine (beliebige) Disjunktion entfernt. Der Rest des Terms muß mit (mindestens) einem Glied dieser Disjunktion unifiziert werden können. Es wird einfach hypothetisch eines ausgewählt. Bei der Unifikation dieses Gliedes mit dem Rest des Merkmalterms sind alle Schritte des Verfahrens durchzuführen.

Im Verlaufe dieser Rekursion gelangt man entweder zu einem Ergebnis, oder die Unifikation schlägt an irgendeiner Stelle fehl. Dann muß ein anderes Disjunktionsglied als Hypothese ausgewählt werden. Bei dem Vorgehen ist also Backtracking nötig.

Sei die erste Disjunktion und daraus das erste Glied $\boxed{2}$ ausgewählt. Durch Unifikation mit dem bis dahin berechneten definiten Anteil $\boxed{107}$ entsteht nun:

$$\boxed{1072} \begin{bmatrix} \textsc{num:}\ \mathit{pl} \\ \textsc{rank:}\ \mathit{clause} \\ \textsc{subj:}\ \boxed{8} \begin{bmatrix} \textsc{case:}\ \mathit{nom} \\ \textsc{num:}\ \mathit{pl} \\ \textsc{per:}\ \mathit{second} \end{bmatrix} \\ \textsc{goal:}\ \boxed{8} \\ \textsc{trans:}\ + \\ \textsc{voice:}\ \mathit{passive} \end{bmatrix} \sqcap \left(\boxed{4} \sqcup \boxed{5}\right)$$

Jetzt zeigt sich aber (Schritt 2), daß keines der Glieder $\boxed{4}$ und $\boxed{5}$ aus der einzigen verbleibenden Disjunktion mit dem definiten Bestandteil konsistent ist. Der Leser möge das nachvollziehen.

Daher ist das zweite Glied $\boxed{3}$ aus der ersten Disjunktion heranzuziehen. Mit ihm ist nur das Glied $\boxed{5}$ aus der zweiten Disjunktion konsistent. Das endgültige Ergebnis wird nun durch Unifikation mit diesen beiden Merkmalstrukturen erhalten. Es ist:

$$\boxed{10735} \begin{bmatrix} \textsc{num:}\ \mathit{pl} \\ \textsc{rank:}\ \mathit{clause} \\ \textsc{subj:}\ \boxed{9} \begin{bmatrix} \textsc{case:}\ \mathit{nom} \\ \textsc{num:}\ \mathit{pl} \\ \textsc{per:}\ \mathit{second} \end{bmatrix} \\ \textsc{agent:}\ \boxed{9}\ \textsc{goal:per:}\ \mathit{third} \\ \textsc{trans:}\ + \\ \textsc{voice:}\ \mathit{active} \end{bmatrix}$$

3.5.3 Die Vorgehensweise in UBS

Die Disjunktion in Merkmalstrukturen wird in UBS auf die Disjunktion in PROLOG (Operator ;/2) zurückgeführt [Stolzenburg 92, 11-12]. Das liefert zwar immer noch keine besonders effiziente Behandlung der Disjunktion, vermeidet aber manches unnötige mehrfache Berechnen von Merkmalen. Es folgt die Implementation von Beispiel 1 in UBS:

```
example1(word:go..agr:(num:pl#num:sng..per:(first#second))).
```

Im Gegensatz zur Expansion in DNF werden die Merkmale `word:go` und `num:sng` in UBS nur einmal berechnet. Für jede Disjunktion wird eine neue Variable eingeführt. Bei der Angabe des transformierten Codes (nachfolgend) sind die Merkmalstrukturen noch nicht in Wertlisten übersetzt aus Gründen der Übersichtlichkeit.

```
example1(word:go..agr:AGR) :-
    (AGR = num:pl ;
     AGR = num:sng..per:PER, (PER = first ;
                              PER = second)).
```

3.5.4 Verteilte oder benannte Disjunktionen

Eine Schwäche der bisher vorgestellten Ansätze besteht darin, daß Disjunktionen nicht so lokal wie möglich behandelt werden, sondern oft Teile von Merkmaltermen mehrfach berechnet werden. Man betrachte dazu die letzte Disjunktion in Beispiel 2:

$$\begin{bmatrix} \text{NUM: } sng \\ \text{SUBJ:NUM: } sng \end{bmatrix} \sqcup \begin{bmatrix} \text{NUM: } pl \\ \text{SUBJ:NUM: } pl \end{bmatrix}$$

Die Disjunktionsglieder sind völlig gleich aufgebaut, nur daß die Werte in den Merkmalen einmal *sng*, einmal *pl* sind. Darum scheint es sinnvoll, die Disjunktion auf die Ebene der Werte herunterzuziehen. – Man könnte nun versucht sein, die obige Disjunktion wie folgt kompakter zu formulieren:

$$\begin{bmatrix} \text{NUM: } sng \sqcup pl \\ \text{SUBJ:NUM: } sng \sqcup pl \end{bmatrix}$$

Das liefert jedoch nicht das gewünschte Ergebnis. Die ursprünglich *eine* Disjunktion tritt nun in der Neuformulierung verteilt auf *zwei* Disjunktionen im Merkmalsterm auf. Dabei ist jedoch die Information verloren gegangen, daß beide Disjunktionen synchron behandelt werden müssen. Entweder ist in beiden Disjunktionen die linke oder in beiden die rechte Alternative auszuwählen.

Um diese Beziehung der beiden Disjunktionen ausdrücken zu können, werden die Disjunktionen *benannt*. Die Notation für Merkmalsterme ist deshalb zu erweitern nach [DörEis 89], [EisDör 90]: Jede Disjunktion wird mit einem Namen indiziert. Die beiden obigen Disjunktionen sollen nun mit d benannt werden, so daß das Beispiel korrekt so ausgedrückt werden kann:

$$\begin{bmatrix} \text{NUM: } sng \sqcup_d pl \\ \text{SUBJ:NUM: } sng \sqcup_d pl \end{bmatrix}$$

Voneinander unabhängige Disjunktionen erhalten verschiedene Namen. Welche Alternative in jeder Disjunktion ausgewählt ist, wird durch den sogenannten *Kontext* festgelegt. Das ist eine Funktion, die jedem Disjunktionsnamen einen der Werte l (links) und r (rechts) zuordnet. Eine automatische Optimierung, die selbsttätig benannte Disjunktionen erzeugt, scheint nicht vorgesehen zu sein.

Es ist möglich, den gleichen Effekt wie mit benannten Disjunktionen zu erzielen, ohne eine neue Notation einführen zu müssen, und zwar mit Hilfe von Relationen. In UBS läßt sich z.B. die Disjunktion aus dem Beispiel unter Zuhilfenahme der Relation `disjunct/1` folgendermaßen ausdrücken (wobei beide Werte von `num` zu einem zusammengefaßt

sind):

```
example2(num:NUM..subj:num:NUM) :-
    disjunct(NUM).

disjunct(sng).
disjunct(pl).
```

3.5.5 Vermeidung struktureller Disjunktion

Da die Berechnung von Disjunktionen aufwendig, die von Unifikation und Negation jedoch effizient implementierbar ist, ist es eine gute Idee, Disjunktionen in Unifikationen und Negation umzuwandeln, wo das möglich ist. Vorschläge in dieser Richtung stammen von [DörSei 91, 5-8] und [Ramsay 90]. Beispiel 1 könnte wie folgt ohne Verwendung der Disjunktion ausgedrückt werden:

$$\begin{bmatrix} \text{WORD: } \textit{go} \\ \text{AGR: } \neg \begin{bmatrix} \text{NUM: } \textit{pl} \\ \text{PER: } \textit{third} \end{bmatrix} \end{bmatrix}$$

Diese Umformung ist aber nur korrekt unter der Voraussetzung, daß

1. das Attribut NUM genau einen der beiden (voneinander verschiedenen) Werte *sng* und *pl* und das Attribut PER einen der drei Werte *first*, *second* oder *third* annimmt;

2. der Wertebereich von AGR auf Merkmalstrukturen mit (mindestens) den Attributen NUM und PER festgelegt ist.

Hier kommen also sehr stark Typinformationen zum Tragen, die in allen vorhergehenden Ansätzen praktisch gar nicht berücksichtigt worden sind.

Diese Optimierung kann auch automatisch durchgeführt werden. Das geht aber nur, wenn die Wertebereiche der auftretenden Merkmale endlich sind. Weil das nicht immer leicht zu erkennen ist, wird in STUF [DörSei 91, 8] eine spezielle Direktive (`finite`) eingeführt. Ein Compiler kann dann gezielt in Typen, die als endlich deklariert sind, zu optimieren versuchen. Der Wertebereich zum Attribut AGR kann wie folgt angegeben werden: (Das Semikolon bezeichnet hierbei eine Disjunktion von atomaren, unvereinbaren Typen.)

```
finite
   agr : num : (sg ; pl)
         per : (first ; second ; third)
         gen : (fem ; masc ; neut).
```

3.5.6 Weitere Ideen

Eine Fortführung des Ansatzes in [EisDör 90] wird in [Könyves-Tóth 91] skizziert. Die Darstellung in dem Aufsatz orientiert sich gleichzeitig an [Kasper 87]. Es wird im Prinzip eine automatische Behandlung von verteilten Disjunktionen vorgeschlagen. Jede Dis-

junktion soll so lokal wie möglich behandelt werden. Merkmale, die nichts miteinander zu tun haben, werden nicht überflüssigerweise gemeinsam betrachtet.

Eine interessante Idee dabei ist, das Problem der Disjunktion in Beziehung zu setzen mit dem Problem des Kopierens von Merkmalstrukturen (für Testunifikationen) und Disjunktionen, die durch disjunkte Typen im Typenverband enthalten sind [Könyves-Tóth 91, 4-5]. Ein Fortschritt auf allen diesen Gebieten zusammen scheint nötig, um Disjunktionen wirklich effizient behandeln zu können.

In [Emele 91] wird eine sehr effiziente Repräsentation durch Graphen für Merkmalstrukturen vorgestellt. Sie vermeidet redundantes Kopieren von Merkmalstrukturen. Durch die Unterscheidung verschiedener Generationen von Knoten im Graphen wird effektives Backtracking ermöglicht. Das kann als Grundlage für eine effiziente Implementierung der Disjunktion dienen.

4 Diskussion und Ausblick

4.1 Vergleich mit anderen Systemen

Parallel mit der Entwicklung neuer linguistischer Theorien und Formalismen entstanden zahlreiche Ansätze und Systeme zu ihrer Implementation auf einem Computer. Es sollen hier nur solche Systeme interessieren, die für Unifikationsgrammatiken geeignet sind.

Unter dem Begriff *Unifikationsgrammatik* können einen ganze Reihe von linguistischen Theorien zusammengefaßt werden. Sie benutzen alle einen Formalismus, der auf (einfachen) Merkmalstrukturen und der Operation Unifikation basiert, wenn auch zum Teil andere Schreibweisen oder Begriffe dafür verwendet werden.

Die im folgenden vorgestellten Systeme sind teilweise speziell auf eine linguistische Theorie ausgerichtet: LFG, GPSG[11] oder HPSG; andere verstehen sich allgemein als Programmiersprache für Unifikationsgrammatiken.

Der Schwerpunkt bei der Beschreibung der Systeme besteht in der Untersuchung, inwieweit sie für HPSG als die Unifikationsgrammatik mit dem ausdruckstärksten (und deshalb auch umfangreichsten) Formalismus geeignet sind. Außerdem sollen aber auch andere, darüber hinausgehende Besonderheiten der Systeme herausgestellt werden.

Die Reihenfolge der Beschreibungen entspricht in etwa der historischen Reihenfolge der Entwicklung der Systeme.

4.1.1 Das System AVAG

Das System AVAG [Sedogbo 86] wurde schon zu einem relativ frühen Zeitpunkt (1986) in Tübingen entwickelt. Bemerkenswerterweise kann es mit Disjunktion und Negation umgehen; trotzdem fand es wenig Beachtung.

Das System zielt auf die Implementation von GPSG ab. Es versteht sich als Übertragung von PATR II [Shieber 86], das in LISP geschrieben ist, nach PROLOG, genauer PROLOG II [Colmerauer 86]. Die Syntax des Systems ist im wesentlichen von PATR II übernommen, allerdings mit einigen Änderungen.

Die grundlegende Datenstruktur in AVAG sind DAGs, gerichtete azyklische Graphen, die im Prinzip aber nichts anderes als azyklische Merkmalstrukturen sind. Die (zunächst) einzige darauf vorgesehene Operation ist die Unifikation. Regeln in AVAG bestehen aus einer (kontextfreien oder ID-)Ersetzungsregel, gefolgt von dazugehörigen (Pfad-)Gleichungen und Ungleichungen. Beispiel:

```
s --> np vp
    <np agr> = <vp head agr>
    <s head> = <vp head>.
```

Wie zu bemerken ist, entspricht das Regelformat eher den Schreibweisen von LFG als dem Ansatz von HPSG, der ganz ohne CFG-Regeln auskommt.

Disjunktion (von Werten) und Negation (durch Ungleichungen beschrieben) kommen z.B. im folgenden, kompakten Lexikoneintrag des deutschen Substantivs *Bonbon* vor,

[11] Beide Theorien werden dargestellt in [Sells 85]).

das schwankenden Genus (Maskulinum oder Neutrum) und jeden Kasus außer Genitiv haben kann:

```
Bonbon:
    <cat> = noun
    <agr number> = sing
    <agr gender> = [masc,neut]
    <case> /=  gen.
```

Ähnlich wie in UBS wird eine in AVAG spezifizierte Grammatik nach PROLOG II compiliert. Die Negation wird mit Hilfe des eingebauten Prädikats `dif`/2, die Disjunktion als verzögerter Aufruf mit Hilfe von `freeze`/2 realisiert (siehe [Carlsson 87]).

Die Compilation erfolgt in drei Phasen. Das erlaubt es, obgleich keine Deklaration von Typen vorgesehen ist, recht kompakte interne Repräsentationen für Merkmalstrukturen zu erzeugen:

Zu jedem Attribut werden in der ersten Phase der Compilation Listen `L` erstellt, die nur genau die Attribute enthalten, die an irgendeiner Stelle der Grammatik im Wert des Attributs `A` vorkommen. Die Wertlisten des Attributs `A` brauchen dann nur für die Attribute aus der zugehörigen Liste `L` eine Position vorsehen. Das geschieht in den weiteren Phasen der Compilation.

Um dem GPSG-Formalismus noch näher zu kommen, erlaubt AVAG es, LP-Regeln zu formulieren, die allerdings nur lokal, d.h für eine einzige Regel, gelten. Es kann eine Menge von Attributen angegeben werden, für die das FFP gelten soll. Die Einhaltung der HFC ist durch den Schreiber einer Grammatik anscheinend selbst zu gewährleisten (siehe oben). Die Formulierung von Metatregeln und optionalen Konstituenten ist auch zugelassen. – Zum Teil kommt es bei diesen Möglichkeiten aber zu einer Vervielfachung der intern aufgestellten Regeln.

Weiterhin können Schablonen definiert werden; sie entsprechen im wesentlichen den Makros in UBS*. Schablonen wirken im Unterschied zu Makros jedoch nur wie ein Default: Explizit angegebene Merkmale überschreiben gegebenenfalls die Werte aus der Schablone.

Schließlich gibt es noch die Möglichkeit der sogenannten Verallgemeinerung von Merkmalen, die für koordinierte Phrasen benötigt wird. In der Phrase *Peter und Paul* z.B. ist der Numerus Singular der einzelnen Konstituenten zu verallgemeinern auf Plural für die ganze Phrase.

4.1.2 Die PROLOG-Erweiterung GULP

Aus den USA stammt das System GULP [Covington 89], entwickelt an der Universität Athens, Georgia. Es ist eine simple Erweiterung von PROLOG um eine Notation für einfache Merkmalstrukturen, die exakt so wie in UBS* implementiert sind.

Die Formulierung von Disjunktion, Negation, Mengen oder Typen ist direkt nicht vorgesehen. Da es sich bei GULP nur um einen Zusatz zu PROLOG handelt, nicht um eine wirklich eigenständige Programmiersprache, bleiben dem Benutzer alle Möglichkeiten vorbehalten, das System selbst zu ergänzen. Denkbar ist z.B. das Schreiben eines Parsers mit speziellen Eigenschaften oder Erweiterungen im Hinblick auf bestimmte

Grammatikformalismen.

Der Vorteil von GULP liegt in der Einfachheit des Systems: Ein Programm, das GULP benötigt, kann in reguläres PROLOG übersetzt werden, ohne daß die Programmstruktur verändert werden muß (keine Einfügung von Code in den Klauseln).

Für die Übersetzung werden Vorwärtsübersetzungsschemata für jedes Attribut und (insgesamt) genau ein Schema zur Rückwärtsübersetzung benötigt, da weder Deklaration von Typen wie in UBS noch das mehrere Phasen der Compilation erfordernde Verfahren von AVAG vorgesehen ist. Der Nachteil dieser Vorgehensweise ist jedoch Speichervergeudung bei der internen Repräsentation von Merkmalstrukturen.

Trotz der Beschränkungen von GULP erbrachte die Analyse dieses Systems viele Anregungen für die Entwicklung von UBS. Im weitesten Sinne stellt UBS einfach eine Fortführung der Konzepte und Ideen aus GULP dar.

4.1.3 Der Formalismus STUF

Der bei IBM Deutschland in Stuttgart entwickelte Formalismus STUF [BoKöUs 88], [STUF 91], [DörSei 91] hat schon eine längere Entwicklung hinter sich, die noch immer nicht abgeschlossen ist. STUF entstand im Rahmen des LILOG-Projektes in Deutschland. Es ist keiner speziellen linguistischen Theorie verpflichtet, sondern allgemein für Unifikationsgrammatiken gedacht.

Im Laufe der Entwicklung von STUF ist die theoretische Fundierung, auf die viel Wert gelegt wird, gleich geblieben – zu nennen sind hier insbesondere die beiden Arbeiten [HöhSmo 88] und [Smolka 89] –, während Schreibweisen und einzelne Komponenten des Systems immer wieder erweitert worden sind. Zur Zeit wird es gerade neu geschrieben in PROLOG, wobei die Integration von Typen und Relationen eine wichtige Rolle spielt.

Der für das System grundlegende Datentyp ist der gerichtete Graph, wobei die Unifikation von Graphen recht effizient mit dem sogenannten Union-Find-Algorithmus [Ebert 81, 87-89] implementiert ist. Das System erlaubt neben der Unifikation und Disjunktion von Graphen auch die Definition von Graphabbildungen, die selbst als Graphen definiert werden können. Damit lassen sich so verschiedene Konstrukte wie die funktionale Applikation in CUG, die Metaregeln in GPSG oder die Lexikonnregeln in HPSG formulieren.

Es ist möglich, eine CFG oder auch eine ID/LP-Grammatik in STUF zu schreiben. Jede Regel bekommt ebenso wie jede Schablone (mit oder ohne Parameter) einen Namen. Jeder Kategorie in einer Regel wird eine (einfache) Merkmalstruktur zugeordnet.

Das folgende Beispiel zeigt eine kontextfreie Regel, eine Schablone und einen Lexikoneintrag für den Namen *John* (in dieser Reihenfolge) nach [STUF 91, (3) 48-49]:

```
np_pn :=
   np -> name --
   np :   (cat: np
           sem: _P
           pred: _P
           ref: _Name)
   name : (cat: name
```

```
            sem: _Name).

name(_X) :=  (cat: name
              sem: _X).

john :=  name(john).
```

Im STUF-System ist ein WFST-Chart-Parser zum Analysieren von Sätzen oder Texten aufgrund der mit einem Editor eingegebenen Grammatik implementiert. Außerdem angeschlossen ist eine sogenannte Workbench [STUF 91, (4)]; das ist ein komfortables Werkzug zum Testen und Entwickeln von Grammatiken.

Damit ist es möglich, die vom Parser erzeugte Chart oder auch nur Teile davon, wenn gewünscht, anzuschauen. Auch die einzelnen Elemente der Chart können näher betrachtet werden: ihre zugehörige Merkmalstruktur oder die Namen der Schablonen und Regeln, die zu ihrer Erzeugung geführt haben, werden als Baum dargestellt. – Das alles erleichtert die Fehlerbehebung bei der Entwicklung einer Grammatik.

Die Disjunktion ist in STUF recht effizient implementiert: Es ist eine explizite Schreibweise für verteilte oder benannte Disjunktionen vorgesehen. Das ermöglicht auch die kompakte Darstellung von mehreren, alternativen Lösungen auf einmal, auch auf dem Bildschirm.

Neuerdings wird in STUF ein stärker typorientierter und damit für HPSG noch passenderer Ansatz verfolgt. Er erlaubt auch die automatische Optimierung struktureller Disjunktionen durch Überführung in Negation und Unifikation.

Relationen können als parametrisierte Typen verstanden werden. Sie haben gewisse Ähnlichkeit mit Funktionen bzw. (parametrisierten) Makros in UBS. Eine Relation mit dem Namen `R` und den Parametern `T1,...,Tn` und `T0`, die beliebige Merkmalstrukturen sein dürfen, wird in einer Funktionsschreibweise dargestellt, wobei `T0` als das Ergebnis anzusehen ist. Sie wird durch ein oder mehrere Klauseln der folgenden Form definiert:

```
R(T1,...,Tn) ==> T0.
```

In [DörSei 91, 14-16] wird vorgeschlagen, eine HPSG-Grammatik mit Hilfe von Relationen zu formulieren, und zwar die Prinzipien und Grammatikregeln von HPSG als Relationen mit den Parametern `HEAD-DAUGHTER` und `COMPLEMENT-DAUGHTER-LIST`. Diese Grammatik diente als Vorlage für die Beispielgrammatik in Kapitel 2.

4.1.4 Ein Parser für HPSG

Ein Parser, speziell für HPSG bestimmt, wurde an der Carnegie Mellon University, Pittsburgh, USA entwickelt [Franz 90]. Er ist in LISP implementiert.

Es wird ein stark typorientierter Ansatz vertreten: Der HPSG-Formalismus wird gesehen als eine Menge von Gleichungen (Constraints) über einem Typenschema von sortierten Merkmalstrukturen; die Aufgabe eines Interpreters ist es dann, diese Gleichungen zu lösen [Franz 90, 3]. Kontextfreie Regeln – und damit auch die darauf anwendbaren Techniken – finden hier keine Verwendung mehr.

Die Definition einer Grammatik ist in mehrere Bestandteile zerlegt: Zunächst ist eine Hierarchie von Typen zu spezifizieren. Wie in UBS ist auch hier nur einfache Vererbung

vorgesehen. Eine subs-Klausel definiert die direkten Subtypen eines Typs. Mit einer approp-Klausel wird festgelgt, welche Attribute bei einem Typ (neu) zugelassen werden. Gleichzeitig wird ihr Wertebereich definiert. Beispiel:

```
sign subs
   (word phrase)

sign approp
   ((PHON nelist-phon)
    (SYNSEM synsem))

phrase approp
   (DTRS constituent-structure)
```

Darüberhinaus werden Sortendefinitionen (sortdef-Klauseln) benötigt, aus denen sich die zu lösenden Gleichungen ableiten. Sie besteht aus einem konjunktiven (definite) und einem disjunktiven (disjunctive) Anteil, wobei nur der konjunktive Anteil an die Subtypen vererbt wird. Diese Aufteilung ermöglicht die optimierte Behandlung der Disjunktion nach [Kasper 87].

Die in HPSG vorgesehenen typgebundenen Implikationen, z.B. das Kopfprinzip (HFP), das Subkategorisierungsprinzip (SUBCATP) oder das semantische Prinzip (SEMP), werden auch hier wie in UBS durch Konjunktion (entspricht Unifikation) ausgedrückt.

Man beachte nun die folgende Sortendefinition nach [Franz 90, 58-59] für phrase, wobei eine Disjunktion von ID-Regeln (IDS) und Regeln zur Anordnung der Konstituenten (COP) benötigt werden:

```
phrase sortdef
   definite
      and ((@template HFP) (@template SUBCATP) (@template SEMP))
   disjunctive
      or ( and ((@template IDS1) (@template COP1))
                 ...
           and ((@template IDS4) (@template COP4)) )
```

Wie zu sehen ist, sind auch in diesem System Schablonen (template) eingeführt, die im Gegensatz zu UBS jedoch nicht parametrisiert werden können. Ebenfalls möglich ist die Vereinbarung von Abkürzungen für längere Pfade in Merkmalstrukturen.

Negation und Mengen sind (noch) nicht in diesem System implementiert. Listen können auch nur mit Hilfe der Attribute FIRST, REST und der Konstanten END ausgedrückt werden, wobei letztere für die leere Liste (Typ elist) steht.

Allgemeine Funktionen, sogar Relationen können aber definiert werden. Die Definition einer Relation (reldef-Klausel) hat enge Verwandtschaft mit einer Sortendefinition. Hier dürfen Parameter verwendet werden, deren Typen zu spezifizieren sind. Eine Einschränkung ist, daß im Rumpf einer Klausel keine freien Variablen neu eingeführt werden dürfen. Es folgt die Definition der append-Funktion zur Konkatenation von Listen nach [Franz 90, 59]:

```
append(A B C) reldef
   or ( and ((at (A) elist)
```

```
          (B = C))
and ((at (A) nelist)
          (at (C) nelist)
          (B FIRST = C FIRST)
          (append((A REST) (B) (C REST))) )
```

Eine Grammatik wird intern in eine Graphdarstellung überführt. Die Verarbeitung geschieht nun so: Der Benutzer gibt eine unvollständige Lösung ein. Sie wird nach und nach vervollständigt, indem die Gleichungen, die an den Knoten der Graphen eingetragen sind, gelöst werden. Dabei wird im Prinzip eine Unifikation von Graphen durchgeführt.

Weil identische Graphenteile kopiert werden und nicht als ein Objekt im Speicher gehalten werden und weil die Überprüfung der Angemessenheit von Attributen für einen Typ einzeln für jedes Attribut durchgeführt wird, ist das System recht langsam. Interessant an dem System ist aber die Möglichkeit, die Lösungen in LATEX [Kopka 91] auszugeben. Das System unterstützt Fehlerbehandlung durch Tracing.

4.1.5 Typisierte Merkmalstrukturen in TFS

Die Schaffung eines Systems, das es erlaubt,

1. linguistische Information auf verschiedenen Ebenen (Phonologie, Morphologie, Syntax, Semantik, Pragmatik) separat beschreiben und
2. Abhängigkeiten der einzelnen Ebenen voneinander völlig deklarativ (ohne Bevorzugung einer Ebene) ausdrücken

zu können, waren die Hauptziele der Entwicklung von TFS [EmHeMoZa 90], [EmZa 90], [Zajac 92]. Das System wurde an der Universität Stuttgart in LISP geschrieben und hat im Laufe seiner Entwicklung schon einige Erweiterungen erfahren.

Die grundlegende Datenstruktur dieses Systems sind typisierte Merkmalstrukturen, die dem in HPSG verwendeten Ansatz sehr entgegenkommen. Das Formulieren einer Grammatik mit TFS bedeutet das Definieren einer Menge von Typen von Merkmalstrukturen.

Die Grammatik wird compiliert in einen Typenhierarchiegraphen (multiple Vererbung ist möglich), wobei jedem Typ Merkmalstrukturen zugeordnet sind, es sei denn, es handelt sich um einen atomaren Typ. Auf der Menge der Merkmalstrukturen ist die übliche Subsumptionsrelation definiert. Der Typenverband in TFS wird durch Typdefinitionen festgelegt. Eine Typdefiniton hat die Form

- `T = S1|...|Sn.` (einfache Vererbung; alle Subtypen sind disjunkt) oder
- `S < T1,...,Tn.` (Mehrfachvererbung; besonders bei Lexikoneinträgen).

Dabei stehen die `T` für Typnamen und die `S` für ihre direkten Subtypen. Zu jedem Merkmalstrukturtyp sind außerdem die (neu) zu ihm gehörigen Attribute samt ihrem Wertebereich anzugeben. Das geschieht mit einer Definition der Form

```
T[F1:C1,...,Fn:Cn].
```

Dadurch wird der Typ `T` mit den (zusätzlichen) Attributen `F1`,...,`Fn` mit Werten von den Typen `C1`,...,`Cn` ausgestattet. Es folgt ein Auschnitt aus dem Typenverband für eine HPSG-Grammatik zur Illustration:

```
sign = phrase | word.

sign[PHON:list-of-phonemes,SYNSEM:synsem,QSTORE:set-of-quantifiers].

phrase[DTRS:constituent-structure].

John < saturated, noun.
```

Zusätzliche Bedingungen an Typen, die die Gleichsetzung von Werten erfordern, werden wie in UBS einfach an die Typen heranunifiziert (mit dem Operator `&`). Das trifft auf das HFP zu, das für alle Merkmalstrukturen vom Typ `headed-phrase` gilt. Dazu betrachte man das folgende Beipiel: Das HFP wird hierbei als Makro definiert. Das Symbol `#` kennzeichnet Platzhalter für Merkmalstrukturen (Variablen).

```
headed-phrase & HFP.

HFP :=  [SYNSEM:[LOC:[CAT: #head]]],
         DTRS:[HEAD-DTR:[SYNSEM:[LOC:[CAT: #head]]]]]].
```

Die grundlegende Operation in TFS ist die Unifikation von typisierten Merkmalstrukturen: Zwei typisierte Merkmalstrukturen werden miteinander unifiziert, nachdem die Menge der allgemeinsten Subtypen der beiden Typen gebildet worden ist. Die Analyse eines Satzes wird durch eine Anfrage angestoßen, z.B.:

```
?- phrase[PHON: <Hans geht spazieren>].
```

Folgende zwei Schritte bewirken die Deduktion der fehlenden Angaben in der typisierten Merkmalstruktur nach [Zajac 92, 166] und [Henschel 92]:

1. **Typüberprüfung:** Alle Typen in der Anfrage (im Beispiel nur `phrase`) sind mit den Beschreibungen des Typs und aller seiner Supertypen in der Grammatik zu unifizieren.

2. **Spezialisierung:** Alle nichtminimalen Typen, d.h. solche, die noch Subtypen besitzen, werden mit einem der Subtypen unifiziert. Das reflektiert die Aussage, daß jeder Typ erschöpfend durch seine Subtypen abgedeckt wird (CWA).

Für die neu eingeführten Typbezeichner ist das Verfahren solange zu wiederholen, bis alle Typen überprüft sind und nur noch minimale Typen vorkommen. Da im zweiten Schritt gegebenenfalls unter mehreren Möglichkeiten ausgewählt werden kann, gibt es im allgemeinen mehr als eine Lösung, unter Umständen sehr viele, weil nur minimale Typen vorkommen dürfen. Das Verfahren ist nicht-deterministisch und kann als Termersetzungssystem mit Lazy Evaluation realisiert werden [Zajac 92, 167-170]. Die theoretische Grundlage für diesen Ansatz liefert (insbesondere) die Arbeit [Aït-Kaci 86]. Dort werden typisierte Merkmalstrukturen unter dem Namen ϵ-Typen eingeführt.

Typen dürfen in TFS zwar keine Parameter haben, funktionale Abhängigkeiten können aber durch Bedingungen, die hinter dem Operator `:-/2` anzugeben sind, ausgedrückt

werden. Dadurch lassen sich Funktionen und Relationen definieren. Disjunktion ist in TFS nur über den Typenverband durch Einführung disjunkter Subtypen möglich. Negation und Mengen sind in TFS (bis jetzt noch) nicht vorgesehen.

4.2 Alles auf einen Blick

4.2.1 Eine Tabelle ...

Die folgende Tabelle gibt einen knappen Überblick über die Eigenschaften der hier vorgestellten Systeme. Mit CMU ist der an der Carnegie Mellon Universität entwickelte Parser gemeint. Die Zeichen in der Tabelle besagen, ob die jeweilige Eigenschaft in dem System

- schwach (−),
- nur zum Teil (o) oder
- sehr gut (+)

realisiert ist. Die drei Kategorien können natürlich nur eine grobe Bewertung geben.

Zunächst werden einige technische Daten angegeben, und zwar in welchem Jahr die Entwicklung des System seinen Anfang nahm und in welcher Programmiersprache es implementiert ist. Dann ist aufgelistet, welche Datenstrukturen bzw. Schreibweisen und welche Operationen das System kennt. Zum Schluß folgen einige Angaben zur Benutzeroberfläche: Ist eine strukturierte Ausgabe vorgesehen? Ist (nachvollziehbares) Tracing von Programmen bzw. Grammatiken möglich? Werden Ausgaben als Bäume dargestellt?

		AVAG	GULP	STUF	CMU	TFS	UBS
Technika	Jahr	1986	1988	1988	1990	1990	1990
	Sprache	PROLOG II	PROLOG	PROLOG/C	LISP	LISP	SEPIA
Strukturen	Typen	−	−	o	o	+	o
	Listen	o	+	+	−	+	+
	Mengen	−	−	−	−	−	+
	Terme	o	+	−	−	−	+
Operationen	Disjunktion	o	−	o	+	o	+
	Relationen	−	o	+	+	o	+
	Negation	o	−	−	−	−	+
Oberfläche	Ausgabe	o	−	+	+	+	+
	Tracing	−	o	+	+	+	o
	Bäume	+	−	+	−	−	−

4.2.2 ... und ihre Interpretation

Der Tabelle nach schneidet das System UBS recht gut ab. Dieser Eindruck ist jedoch zu relativieren. Wirklich essentielle Bestandteile eines Systems zur Implementation von HPSG scheinen mir die Typen und Relationen zu sein. Beide Merkmale weisen jedoch alle neueren Systeme auf.

Aus der Tabelle geht hervor, daß TFS, was die Realisierung des Typkonzepts von HPSG anbelangt, am weitesten entwickelt ist; es erlaubt nämlich als einziges System multiple Vererbung. Insgesamt stellt TFS das zur Zeit wohl ausgereifteste und effizienteste System – zumindest in dieser Hinsicht – dar. Um wirklich Grammatiken von größerem Umfang erstellen zu können, ist eine komfortable Benutzeroberfläche nötig. In dieser Beziehung bitet das Sytem STUF am meisten.

UBS genießt den Vorzug, daß es einen erstaunlich kleinen Umfang hat. Das Programm UBS ist nur gut 32 kBytes lang und damit sogar wesentlich kürzer als der Vorläufer UBS*, trotz erweiterter Funktionalität. Das ist natürlich bedingt durch die Benutzung der sehr mächtigen, aber auch umfangreichen Programmiersprache SEPIA. Die Verwendung von SEPIA hat die Entwicklung von UBS meiner Meinung nach stark beschleunigt. Insbesondere die Möglichkeiten, Prädikate verzögert aufrufen zu können, hat vieles vereinfacht.

4.3 Ausblick

4.3.1 Wo noch etwas getan werden kann

Irgendwann muß jede Diplomarbeit ihr Ende haben. Perfektionisten finden sicherlich immer noch genügend Ansatzpunkte, wo etwas verbessert werden kann. Manchmal schien es auch mir, als ob durch diese Arbeit viel mehr Fragen aufgeworfen als gelöst wurden. Aber es ist einfach wichtig, eine Arbeit einmal abschließen und beiseite legen zu können. Nichtsdestotrotz gibt es viele Punkte, an denen weitergearbeitet werden kann. Es folgen nun einige Hinweise, wo das denkbar ist.

Theorie: Ein formaler Beweis der Verfahren, die die Begriffe Logikprogrammierung und typisierte Merkmalstrukturen miteinander verbinden, steht noch aus. Eine präzise und gleichzeitig knappe Aufarbeitung des Formalismus in diesem Sinne wird sicherlich weitere Impulse zur Implementation von HPSG liefern. Abzuwarten bleibt in diesem Zusammenhang die Herausgabe der endgültigen Version von [Carpenter 90] und des neuen STUF-Systems.

Ein weiteres wichtiges theoretisches Problem ist die Untersuchung, unter welchen Bedingungen ein Programm terminiert. Überlegungen dazu finden sich in [Weisweber 89], [Weisweber 92] und [Zajac 92, 175-176].

Praxis: Die Typdeklarationen können dahingehend verfeinert werden, daß zu den Attributen ein Wertebereich angegeben werden kann. Ferner sollte auch die Vererbung von Koreferenzen und Constraints ins Auge gefaßt werden. Weiter interessant scheint die Einführung parametrisierter Typen zu sein, besonders für Listen und Mengen. Außerdem mag die Einführung einer sogenannten Default-Unifikation [Bouma 92] hilfreich sein.

Was noch aussteht, ist die Einführung von multipler Vererbung und die Extensionalität von Typen, insbesondere der Mengen. In einer erweiterten Version von UBS könnten drei Typklassen unterschieden werden: intensionale und extensionale Typen und Konstanten.

Implementation: Wie in Kapitel 3 schon festgestellt, kann am System UBS noch einiges verbessert werden. Denn nicht immer wurde die effizienteste Implementation für die Datenstrukturen und Operationen gewählt: Multiple Typenhierarchien können nach dem in [AïBoLiNa 89] vorgestellten Verfahren effektiv implementiert werden. Zur effizienten Behandlung der Disjunktion scheinen insbesondere die Verwendung benannter Disjunktionen [EisDör 90] und die interne Repräsentation von Merkmalstrukturen nach [Emele 91] interessant zu sein.

Oberfläche: Ein deutlicher Schwachpunkt an UBS ist, daß es keine besonders komfortable Benutzeroberfläche besitzt. Ein erster Schritt dahin könnte sein, bei der Ausgabe von Strukturen Koreferenzen noch besser sichtbar zu machen. Es ist weiter interessant, in einer Struktur sondieren zu können, d.h. die Möglichkeit zu schaffen, sich bestimmte Teile der Ausgabe gezielt anzuschauen. Außerdem ist eine interaktive Editierung bzw. Manipulation von Merkmalstrukturen eventuell nützlich. SEPIA unterstützt die Entwicklung grafischer Benutzeroberflächen.

Zum Nachvollziehen von Programmen bzw. zur Fehlersuche ist es nötig, Tracing vorzusehen. Dabei sollten die interne Repräsentation und gewisse Abläufe für den Benutzer unsichtbar bleiben. Möglichkeiten zu einer Weiterentwicklung in diese Richtung bietet der mit SEPIA gelieferte komfortable Debugger *Opium* [SEPIA 91, (2)].

4.3.2 Schlußbemerkungen

Diese Aussichten auf Erweiterungsmöglichkeiten sollen an dieser Stelle genügen. Nun ist Raum für eigene Ideen gegeben. Wie jedenfalls zu sehen ist, kann noch einiges getan werden. Die Programmiersprache SEPIA bietet dafür auch einige bis jetzt noch gar nicht ganz ausgeschöpfte Möglichkeiten. Eine weitere Arbeit an der Sprache UBS sollte sich vielleicht noch mehr auf einzelne Probleme beschränken, z.B. der Behandlung der Disjunktion, und mehr in die Tiefe als in die Breite gehen.

Meiner Meinung nach ist die Sprache UBS nicht nur für Computerlinguisten (zur Implementation von HPSG) interessant, sondern sie stellt auch eine interessante Fortsetzung der Ideen aus der Logikprogrammierung dar und geht dabei in den Bereich der objektorientierten Programmierung hinein. Die Kombination dieser beiden Paradigmen kann ein Weg sein zu einer effektiven und sicheren Entwicklung von Software überhaupt.

Abkürzungen

ACI	associative commutative idempotent
AVAG	attribute / value grammar tool
AVM	attribute value matrix
CFG	context free grammar
CUG	categorial unification grammar
CWA	closed world assumption
DAG	directed acyclic graph
DNF	disjunctive normal form
FCR	feature cooccurrence restriction
FFP	foot feature principle
GB	government binding
GPSG	generalized phrase structure grammar
GULP	graph unification and logic programming
HFC	head feature convention
HFP	head feature principle
HPSG	head-driven phrase structure grammar
ID	immediate dominance
LATEX	Lamport TEX
LFG	lexical functional grammar
LILOG	linguistic and logic methods
LISP	list processing language
LP	linear precedence
MGU	most general unifier
PATR	parsing and translation
PROLOG	programming in logic
SEPIA	standard ECRC PROLOG integrating advanced features
SLD	linear resolution with selection function for definite clauses
STUF	Stuttgart type unification formalism
TFS	typed features structure system
UBS	unifikationsbasierte Sprache (unification-based language)
UDC	unbounded dependency construction
WFST	well-formed substring table

Danksagung

All denen, die meine Arbeit in irgendeiner Weise unterstützt haben, möchte ich an dieser Stelle meinen herzlichen Dank ausdrücken. Zuerst möchte ich mich bei Martin Volk und Prof. Ulrich Furbach bedanken, die sich zur Betreuung der Diplomarbeit bereit erklärt haben und mir in Gesprächen und Diskussionen viele Anregungen geliefert haben. Außerdem freue ich mich über das Interesse von Dr. Wilhelm Weisweber für meine Arbeit, der auch ein Gutachten über die Arbeit verfassen will.

Hilfreich waren auch die Vorlesungen, Seminare, Tips, der Zugang zu interessanter Literatur und sonstige Begleitung der Arbeit durch Prof. Istvan Bátori, Dr. Renate Henschel und Dr. Hans-Joachim Novak. Letzterer vermittelte mir einen Besuch im Mai dieses Jahres bei IBM in Stuttgart, wo ich meine bis dahin geleistete Arbeit vorstellen konnte. Danach ergaben sich sehr anregende Gespräche unter anderem mit Ingo Raasch, Martin Emele sowie Jochen Dörre.

Schließlich möchte ich mich noch bei Heiner Mittermaier bedanken für das Korrekturlesen dieser Arbeit. Meiner Frau Elke sage ich ganz besonderen Dank für ihre Geduld, die sie während der gesamten Arbeit für mich aufbrachte.

Erklärung

Hiermit erkläre ich, Frieder Stolzenburg, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Koblenz, im November 1992

Literatur

[Aït-Kaci 86] Aït-Kaci, Hassan: An Algebraic Semantics Approach to the Effective Resolution of Type Equations. In: Theoretical Computer Science 45 (1986), 293-351.

[AïBoLiNa 89] Aït-Kaci, Hassan; Boyer, Robert; Lincoln, Patrick; Nasr, Robert: Efficient Implementation of Lattice Operations. ACM Transactions on Programming Languages and Systems 11, 1 (1989), 115-146.

[BaaBüt 88] Baader, Franz; Büttner, Wolfram: Unification in Commutative Idempotent Monoids. Theoretical Computer Science 56 (1988), 345-353.

[BoKöUs 88] Bouma, Gosse; König, Esther; Uskoreit, Hans: A flexible graph-unification formalism and its applications to natural-language processing. IBM Journal of Research and Development 32, 2 (1988), 170-184.

[Bouma 92] Bouma, Gosse: Feature Structures and Nonmonoticity. Computational Linguistics 18, 2 (1992), 183-203.

[Büttner 86] Büttner, Wolfram: Unification in the Data Structure Sets. In: Siekmann, Jörg H. (Ed.): Proceedings of the 8th International Conference on Automated Deduction, Oxford, 1986, 489-495. (LNCS 230) Berlin; Heidelberg; New York: Springer, 1986.

[Carlsson 87] Carlsson, Mats: Freeze, Indexing and Other Implementation Issues in the WAM. In: Lassez, Jean-Louis (Ed.): Logic Programming. Volume 1. Proceedings of the 4th International Conference, University of Melbourne, Australia, 1987, 40-58. (MIT Press Series in Logic Programming) Massachusetts Institute of Technology, 1987.

[Carpenter 90] Carpenter, Bob: The Logic of Typed Feature Structures. (Draft Version) Pittsburgh: Carnegy Mellon University, Philosophy Department, December 1990.

[ClocMell 87] Clocksin, W.F.; Mellish, C.S.: Programming in Prolog. Berlin; Heidelberg; New York: Springer, [3]1987.

[Colmerauer 86] Colmerauer, Alain: Theoretical Model of Prolog II. In: van Caneghham, Michel; Warren, David H.D. (Eds.): Logic programming and its applications. Norwood, New Jersey: Ablex Publishing Corporation, 1986, 3-31.

[Cooper 90] Cooper, Richard: An Introduction to HPSG. (Course Material) Leuven: The Second European Summer School on Language, Logic and Information. 1990.

[Covington 89] Covington, Michael A.: GULP 2.0: An Extension of Prolog for Unification-Based Grammar. (Research Report AI-1989-01) Athens, Georgia: The University of Georgia. 1989.

[DawVij 89] Dawar, Anuj; Vijay-Shanker, K.: A Three-Valued Interpretation of Negation in Feature Structure Descriptions. In: Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, University of British Columbia, June 1989, 18-24. Vancouver, British Columbia, Canada: ACL, 1989.

[DörEis 89] Dörre, Jochen; Eisele, Andreas: Determining Consistency of Feature Structure Terms with Distributed Disjunctions. In: Metzing, D. (Ed.): Proceedings of the 13th German Workshop on Artificial Intelligence, Eringerfeld, September 1989, 270-279. (IFB 216) Berlin; Heidelberg; New York: Springer, 1989.

[DörSei 91] Dörre, Jochen; Seiffert, Roland: Sorted Terms and Relational Dependencies. (IWBS Report 153) Stuttgart: IBM Deutschland, February 1991.

[Ebert 81] Ebert, Jürgen: Effiziente Graphenalgorithmen. Wiesbaden: Akademische Verlagsgesellschaft, 1981.

[EisDör 90] Eisele, Andreas; Dörre, Jochen: Disjunctive Unification. (IWBS Report 124) Stuttgart: IBM Deutschland, May 1990.

[Emele 91] Emele, Martin C.: Unification with lazy non-redundant copying. In: Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics, University of California, Berkeley, June 1991, 323-330. Berkeley, California, USA: ACL, 1991.

[EmHeMoZa 90] Emele, Martin C.; Heid, Ulrich; Momma, Stefan; Zajac, Rémi: Organizing linguistic knowledge for multilingual generation. In: Karlgren, Hans (Ed.): Proceedings of the 13th International Conference on Computational Linguistics, Helsinki University, 1990, Volume 3, 102-107.

[EmZa 90] Emele, Martin C.; Zajac, Rémi: Typed Unification Gramars. In: Karlgren, Hans (Ed.): Proceedings of the 13th International Conference on Computational Linguistics, Helsinki University, 1990, Volume 3, 293-298.

[Franz 90] Franz, Alex: A Parser for HPSG. (CMU-LCL-90-3) Pittsburgh: Carnegie Mellon University, Laboratory for Computational Linguistics. July 1990.

[Furbach 91] Furbach, Ulrich: Logische und Funktionale Programmierung. Grundlagen einer Kombination. Braunschweig: Vieweg, 1991.

[Henschel 92] Henschel, Renate: Eine Einführung in HPSG. (Vorlesungsmitschrift) Universität Koblenz-Landau, Sommersemester 1992.

[HöhSmo 88] Höhfeld, Markus; Smolka, Gert: Definite Relations over Constraint Languages. (LILOG Report 53) Stuttgart: IBM Deutschland, October 1988.

[Jayaraman 92] Jayaraman, Bharat: Implementation of Subset-Equational Programs. Journal of Logic Programming 12, 4 (1992), 299-324.

[KapuNare 86] Kapur, Depak; Narendran, Paliath: NP-Completeness of the Set Unification and Matching Problems. In: Siekmann, Jörg H. (Ed.): Proceedings of the 8th International Conference on Automated Deduction, Oxford, 1986, 470-488. (LNCS 230) Berlin; Heidelberg; New York: Springer, 1986.

[Kasper 87] Kasper, Robert T.: A Unification Method for Disjunctive Feature descriptions. In: Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics, Stanford University, July 1987, 235-242. Stanford, California, USA: ACL, 1987.

[Kasper 88] Kasper, Robert T.: Conditional Descriptions in Functional Unification Grammar. In: Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics, University of New York at Buffalo, June 1988, 233-240. Buffalo, New York, USA: ACL, 1988.

[KasRou 86] Kasper, Robert T.; Rounds, William C.: A Logical Semantics for Feature Structures. In: Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, Columbia University, New York, June 1986, 257-266. New York, NY: ACL, 1986.

[Knight 89] Knight, Kevin: Unification: A multidisciplinary survey. ACM computing surveys 21, 1 (1989), 93-124.

[Könyves-Tóth 91] Könyves-Tóth, Michael: Incremental Evaluation of Disjunctive Feature Terms. (Arbeitspapiere der GMD 593) Sankt Augustin: GMD, November 1991.

[Kopka 91] Kopka, Helmut: LATEX. Eine Einführung. Bonn; München; Reading, Massachusetts: Addison-Wesley, [3]1991.

[LinChr 88] Lincoln, Patrick; Christian, Jim: Adventures in Associative-Commutative Unification (A Summary). In: Lusk, Ewing; Overbeek, Ross: Proceedings of the 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 1988, 358-367. (LNCS 310) Berlin; Heidelberg; New York: Springer, 1988.

[Lloyd 84] Lloyd, John Wylie: Foundations of Logic Programming. Berlin; Heidelberg; New York: Springer, 1984.

[MeiSchRos 90] Meier, Micha; Schimpf, Joachim; van Rossum, Emmanuel: A Guide to SEPIA Customization and Advanced Programming. (TR-LP-50) München: ECRC GmbH, 1990.

[Pollard 91] Pollard, Carl J.: Topics in Constraint-Based Syntactic Theory. Chapters (1) to (8). (Course Material) Saarbrücken: The Third European Summer School on Language, Logic and Information, 1991.

[PolMos 90] Pollard, Carl J.; Moshier, M. Drew: Unifying Partial Description of Sets. In: Hanson, Philip (Ed.): Information, Language, and Cognition. (Vancouver Studies in Cognitive Sciences) Vancouver, BC: University of British Columbia Press, 1990, 285-322.

[PolSag 87] Pollard, Carl J.; Sag, Ivan A.: Information-Based Syntax and Semantics. Volume 1: Fundamentals. (CSLI Lecture Notes 13) Leland Stanford Junior University: CSLI. 1987.

[Ramsay 90] Ramsay, Allan: Disjunction Without Tears. Computational Linguistics 16, 3 (1990), 171-174.

[ReaHep 89] (1) Reape, Mike; Hepple, Mark: Word Order and Constituency in West Continental Germanic. (2) Reape, Mike: A Theory of Word Order and Discontinuous Constituency in West Continental Germanic. (Manuscript) University of Edinburgh, 1989.

[Rounds 88] Rounds, William C.: Set Values for Unification-Based Grammar Formalisms and Logic Programming. (Technical Report CSLI-88-129) University of Michigan and CSLI. 1988.

[Sag 86] Sag, Ivan A.: Grammatical Hierarchy and Linear Precedence. In: Huck, Geoffrey; Ojeda, Almerindo E. (Eds.): Discontinuos Constituency. (Syntax and Semantics 20) London: Academic Press, 1987, 303-340.

[SchmWern 89] Schmitt, P.H.; Wernecke, W.: Tableau Calculus for Order-Sorted Logic. In: Bläsius, Karl Hans; Hedtstück, Ulrich; Rollinger, Claus-Rainer (Eds.): Sorts and Types in Artificial Intelligence, Workshop, Eringerfeld, April 1989, 49-60. (LNAI 418) Berlin; Heidelberg; New York: Springer, 1989.

[Schöning 87] Schöning, Uwe: Logik für Informatiker. Mannheim: BI-Wissenschaftsverlag, 1987.

[Sedogbo 86] Sedogbo, Celestin: AVAG: An Attribute / Value Grammar Tool. (FNS-Bericht 86-10) Universität Tübingen: Seminar für natürlichsprachliche Systeme. 1986.

[Sells 85] Sells, Peter: Lectures on Contemporary Syntactic Theories. An Introduction to Government-Binding Theory, Generalized Phrase Structure Grammar and Lexical Functional Grammar. (CSLI Lecture Notes 3) Leland Stanford Junior University: CSLI. 1985.

[SEPIA 91] (1) SEPIA 3.1 User Manual. (2) SEPIA BIP Book. (3) Opium 3.1 – User Manual. (SEP/UM/064; SEP/UM/065; TR-LP-60) München: ECRC GmbH, 1991.

[Shieber 86] Shieber, Stuart M.: An Introduction to Unification-Based Approaches to Grammar. (CSLI Lecture Notes 4) Leland Stanford Junior University: CSLI. 1986.

[ShmTsuZan 92] Shmueli, Oded; Tsur, Shalom; Zaniolo, Carlo: Compilation of Set Terms in the Logic Data Language (LDL). Journal of Logic Programming 12, 1&2 (1992), 89-119.

[ShPeKaKa 86] Shieber, Stuart M.; Pereira, Fernando, C. N.; Karttunen, Lauri; Kay, Martin: A Compilation of Papers on Unification-Based Grammar Formalisms. Parts I and II. (CSLI-86-48) Leland Stanford Junior University: CSLI. April 1986.

[Siekmann 84] Siekmann, Jörg H.: Universal Unification. In: Shostak, R.E. (Ed.): Proceedings of the 7th International Conference on Automated Deduction, Napa, California, USA, May 1984, 1-42. (LNCS 170) Berlin; Heidelberg; New York: Springer, 1984.

[Smolka 89] Smolka, Gert: Feature Constraint Logics for Unification Grammar. (IWBS Report 93) Stuttgart: IBM Deutschland, November 1989.

[Stolzenburg 91] Stolzenburg, Frieder: UBS – Eine unifikationsbasierte Sprache zur Implementation von HPSG. Studienarbeit. Universität Koblenz-Landau. September 1991.

[Stolzenburg 92] Stolzenburg, Frieder: UBS – A Unification-Based Language for the Implementation of HPSG. (Fachberichte Informatik 2/92) Koblenz: Universität Koblenz-Landau, February 1992.

[STUF 91] (1) Dörre, Jochen; Seiffert, Roland: A Formalism for Natural Language – STUF. (2) Dörre, Jochen: The Language of STUF. (3) Seiffert, Roland: Chart-Parsing of STUF-Grammars. (4) Dörre, Jochen; Raasch, Ingo: The STUF Workbench. In: Herzog, Otthein; Rollinger, Claus-Rainer: Text Understanding in LILOG. (LNAI 546) Berlin; Heidelberg; New York: Springer, 1991, 33-62.

[Walther 89] Walther, Christoph: Many-Sorted Inferences in Automated Theorem Proving. In: Bläsius, Karl Hans; Hedtstück, Ulrich; Rollinger, Claus-Rainer (Eds.): Sorts and Types in Artificial Intelligence, Workshop, Eringerfeld, April 1989, 18-48. (LNAI 418) Berlin; Heidelberg; New York: Springer, 1989.

[Weisweber 89] Weisweber, Wilhelm: Transfer in Machine Translation by Non-Confluent Term-Rewrite Systems. In: Metzing, D. (Ed.): Proceedings of the 13th German Workshop on Artificial Intelligence, Eringerfeld, September 1989, 264-269. (IFB 216) Berlin; Heidelberg; New York: Springer, 1989.

[Weisweber 92] Weisweber, Wilhelm: Term-Rewriting as a Basis for a Uniform Architecture in Machine Translation. (Submitted Paper) Nantes: COLING, 1992.

[Zajac 92] Zajac, Rémi: Inheritance and Constraint-Based Grammar Formalisms. Computational Linguistics 18, 2 (1992), 159-182.