Ordnungssortierte feature-Logik und Dependenzgrammatiken in der Computerlinguistik

Diplomarbeit

Friedrich Steimann 1991

betreut durch

Prof. Dr. P. Deussen

Christoph Brzoska

Institut für Logik, Komplexität und Deduktionssysteme

Universität Karlsruhe

Die vorliegende Arbeit habe ich selbst verfaßt und dabei keine anderen Hilfsmittel als die angegebenen verwendet.

Karlsruhe, 24.6.1991

Kurzfassung

Zwei unabhängige Formalismen werden vorgestellt und auf ihre Eignung zur Verwendung in der Computerlinguistik hin untersucht: die ordnungssortierte feature-Logik und die Dependenz-Unifikationsgrammatik. Nach ihrer theoretischen Einführung wird der Versuch unternommen, sie zusammen in ein bestehendes Konzept des logischen Programmierens, den Prolog-Dialekt Lopster der Universität Karlsruhe, zu integrieren, und damit ein linguistisches Werkzeug zu schaffen, das es erlaubt, die Dependenzgrammatik einer natürlichen Sprache direkt in Horn-Klauseln der ordnungssortierten Logik zu transformieren.

Inhaltsverzeichnis

Ordnungssortierte feature-Logik und Dependenzgrammatiken in der Computerlinguistik

Die vorliegende Arbeit stellt einen Versuch dar, Konzepte der formalen Logik mit denen der modernen Computerlinguistik zu vereinen und daraus ein linguistisches Werkzeug zu schaffen, das eine adäquate Formulierung von Grammatiken natürlicher Sprachen einerseits und deren möglichst einfache Abarbeitung auf einer Rechenanalge andererseits erlaubt. Die Kombination beider Themenbereiche ist längst nicht so abwegig, wie sie auf Anhieb erscheint: Linguistik und Logik haben sich schon oft gegenseitig beeinflußt. Während in der Antike beide Gebiete kaum zu trennen waren und sie erst viel später als eigenständige Disziplinen betrachtet wurden, erhoffen sich gerade in jüngster Zeit wieder beide Seiten neue Erkenntnisse und Fortschritte aus enger Zusammenarbeit. So ist in vielen linguistischen Theorien, die es sich zur Aufgabe gemacht haben, natürliche Sprache formal in den Griff zu bekommen, die Unifikation längst zentraler Bestandteil geworden, und das Konzept der features, das in letzter Zeit in die Logik Einzug gehalten hat, scheint in erster Linie durch die Linguistik motiviert worden zu sein.

Trotz der starken gegenseitigen Beeinflussung sollen in dieser Arbeit beide Themenbereiche zunächst getrennt voneinander behandelt werden, um ihre spezifischen Eigenschaften besser herausarbeiten zu können - ihre gemeinsame Verwendbarkeit ist jedoch stets unterschwellig präsent und wird durch die Verwendung geeigneter Beispiele nahegelegt.

In diesem Sinne werden in den theoretischen Grundlagen in beiden Gebieten zunächst neue Konzepte vorgestellt: Der logische Teil beschäftigt sich mit der ordnungssortierten feature-Logik und stellt dazu eine Syntax, eine Semantik und einen geeigneten Unifikationsbegriff vor, die sich in einigen Punkten von denen der einsortigen Logik unterscheiden, während der linguistische Teil einen Weg von den bekannten kontextfreien Grammatiken zu einer attributierten Dependenzgrammatik aufzeigt und damit ein Werkzeug herleitet, das die Formulierung natürlichsprachlicher Grammatiken nach den Gesichtspunkten einer fundierten linguistischen Theorie erlaubt.

Im Anschluß daran wird aufgezeigt, wie sich beide Konzepte auf das logische Programmieren übertragen lassen, damit sie einer praktischen Verwendung zugänglich gemacht werden können.

Mit der Formulierung einer Beispielgrammatik werden die beiden Konzepte zusammen eingesetzt und dargestellt, wie sich komplexere grammatikalische Sachverhalte damit ausdrücken lassen.

Die Implementation geeigneter Vorübersetzungsverfahren wird im Anschluß daran beschrieben. Ausblick und Bewertung sollen die Erfahrungen mit dem praktischen Umgang und die Ergebnisse einer kritischen Betrachtung der Arbeit zusammenfassen.

Ich möchte mich an dieser Stelle für die intensive Betreuung durch Christoph Brzoska und die freundliche Unterstützung von Christoph Bläsi, Universität Heidelberg, bedanken, die mir immer mit Rat und Tat zur Seite gestanden und viele Fragen beantwortet haben.

Für sämtliche Fehler, die sich dennoch in dieser Arbeit befinden, bin ich allein verantwortlich.

1 Theoretische Grundlagen

Um den praktischen Teil meiner Arbeit theoretisch absichern und formal einordnen zu können, sollen in diesem ersten Teil die theoretischen Grundlagen erörtert werden. Ich werde dazu zunächst die ordnungssortierte feature-Logik und danach die Dependenzgrammtik getrennt voneinander behandeln.

1.1 Ordnungssortierte feature-Logik

Während in der einsortigen Logik keine Unterscheidung der Objekte, über die man Aussagen treffen kann, nach Zugehörigkeit zu verschiedenen Mengen vornimmt, wird das Universum in der mehrsortigen Logik in Teilmengen, sogenannte Sorten, aufgeteilt.

Ein vielzitiertes Beispiel, das den Nutzen dieser Sortierung[1] hervorhebt, ist die Ableitung der Tatsache, daß Sokrates sterblich ist, aus den Prämissen, daß Sokrates Mensch ist und alle Menschen sterblich sind. Während in der einsortigen Prädikatenlogik zum Beweis dieser Aussage noch ein Deduktionsschritt erforderlich ist, kann sie der mehrsortigen Logik direkt entnommen werden, wenn man voraussetzt, daß alle Objekte der Sorte Mensch sterblich sind und Sokrates nun einmal ein solches Objekt ist.

Durch Einführung der ordnungssortierten Prädikatenlogik läßt sich die Anzahl der einsparbaren Deduktionsschrittte je nach Problemstellung noch erhöhen, indem man die flache Einteilung des Universums in disjunkte Sorten um die Möglichkeit der Untersortenbildung ergänzt. In einer solchen Subsumptionsordnung werden Untersorten immer als Teilmenge ihrer Obersorten interpretiert. Für eine genauere Betrachtung des Prinzips der Ordnungssortiertheit sei der Leser auf [Walther 87], [Schmidt-Schauß 85], [Huber/Varšek 87] und [Weinstein 89] verwiesen.

Ein anderer Ansatz zur Erweiterung der Prädikatenlogik betrifft die Einführung von Objektmerkmalen, sogenannten *features*[2]. Verschiedene Auffassungen zur Integration von features in die Logik, speziell in das logische Programmieren, sind gebräuchlich ([Aït-Kaci 86], [Shieber 86], [Ludwig 88], [Gazdar 89]), und einige davon sind bereits implementiert

[1] Unter Sortierung wird hier nicht der Vorgang der Umordnung von Elementen in eine gewünschte Reihenfolge, sondern die Zuordnung zu Sorten verstanden.

[2] Obwohl der deutsche Begriff *Merkmal* die Bedeutung von feature in diesem Zusammenhang voll erfaßt, möchte ich doch die englische Bezeichnung vorziehen. Zur Begründung vergleiche man bitte Anhang: Begriffsklärung.

worden.

Die einfachste Variante darunter ist die Abkehr von den starren Termen der Prädikatenlogik erster Ordnung, die sich durch die vorgegebene Anzahl und Reihenfolge ihrer Argumente auszeichnen, hin zu sogenannten feature-Strukturen, in denen den Argumenten Namen zugeordnet werden und der Funktor selbst nur als ein besonderes Argument aufgefaßt wird. Diese Darstellungsweise , in der die Zahl und Reihenfolge der Argumente keine Rolle mehr spielt, zusammen mit einer geeigneten Unifikation machen sich auch einige linguistische Formalismen zunutze ([Shieber 86], FUG [Kay 82], LFG [Bresnan 82]).

Meine Arbeit möchte ich auf die Arbeit von Gert Smolka und Hassan Aït-Kaci abstützen, die einerseits ihren *feature types* eine fundierte Semantik zuordnen, und andererseits einige wichtige Beweise liefern, die ich mir zunutze machen kann [Smolka 89].

Während Smolka und Aït-Kaci ihre *feature-Terme* nur als abkürzende Schreibweise für Gleichungssysteme betrachten und das Problem der Unifikation zweier Terme auf das der Lösung des kombinierten Gleichungssystems zurückführen, möchte ich die Unifikation direkt auf der Termebene durchführen. Um dennoch Ergebnisse aus [Smolka 89] verwenden zu können, werde ich ein Verfahren vorstellen, wie sich zu jedem Unifikationsproblem ein äquivalentes Gleichungssystem finden läßt.

1.1.1 Grundlegende Vereinbarungen und Notationen

Die geschweiften Klammern {} schließen Mengen ein, $\cup$ bezeichne die Vereinigung, $\cap$ den Schnitt, $\in$ das Element in einer Menge und $\varnothing$ die leere Menge.

Ausgehend von einer Menge M sei M^* die Menge aller Wörter über M, M^+ die Menge aller nichtleeren Wörter über M und ϵ das leere Wort.

Eine zweistellige reflexive und transitive Relation $\leq$ über einer Menge M wird *Halbordnung* genannt. Durch $a_1 \ldots a_n \leq b_1 \ldots b_n \Leftrightarrow a_1 \leq b_1 \wedge \ldots \wedge a_n \leq b_n$ für $a_1, \ldots, a_n, b_1, \ldots, b_n \in M$ wird $\leq$ auf Wörter aus M fortgesetzt. Man notiert $a < b$, wenn $a \leq b$ und $a \neq b$ ist.

Weiterhin kennzeichne der Operator $\doteq$ die objektsprachliche gerichtete Gleichheit.

1.1.2 Syntax

Die Syntax konventioneller (einsortiger) und auch mehrsortiger Logik soll in dieser Arbeit unversehrt bleiben und lediglich um eine Notation zur Darstellung von feature-Termen ergänzt werden. Es wird dazu im folgenden zwischen konventionellen syntaktischen Objekten, die zur besseren Unterscheidung das Präfix *Konstruktor-* tragen, und solchen, die im Zusammenhang mit features eingeführt und durch eine entsprechendes Präfix gekennzeichnet werden, unterschieden.

Definition 1.1 (Sorten, Sortenverband, Funktionssymbole, Signatur)

Sei S_c eine Menge von *Konstruktor-Sorten*, S_f eine Menge von *feature-Sorten* mit $S_c \cap S_f = \varnothing$ und $S = S_c \cup S_f$ eine Menge von *Sorten*.

Mit der Halbordnung $\leq$ werde durch $(S, \leq)$ eine *Sortenstruktur* spezifiziert.

Sei weiterhin $\Sigma_c = \bigcup \Sigma_{w,s}$ mit $w \in S^*$ und $s \in S_c$ eine Menge von *Konstruktor-Symbolen*, $\Sigma_f = \bigcup \Sigma_{w,s}$ mit $w \in S_f$, $s \in S$ eine Menge von *feature-Symbolen* mit $\Sigma_c \cap \Sigma_f = \varnothing$ und $\Sigma = \Sigma_c \cup \Sigma_f$ eine Menge von *Funktionssymbolen*.

Dann ist das Tripel $(S, \leq, \Sigma)$ eine *Signatur*.
■

Bemerkung: Diese Definition für Signaturen enthält bereits einige Bedingungen, die bei Smolka erst für zulässige Signaturen gefordert werden.

Obwohl es sich bei den Konstruktor-Symbolen wie bei den feature-Symbolen gleichermaßen um Funktionssymbole handelt, kommt ihnen dennoch eine unterschiedliche Bedeutung zu: Während die Konstruktor-Symbole lediglich die Aufgabe der Repräsentation neuer Objekte durch die Konstruktion von Termen übernehmen, ohne als Funktion ausgewertet zu werden, werden die feature-Symbole ihrem Argument einen Term als feature-Wert fest zuordnen, wodurch das Argument genauer spezifiziert wird.

Man sagt, s_1 ist *Untersorte* von s_2 und s_2 ist *Obersorte* von s_1, wenn $s_1, s_2 \in S$ und $s_1 \leq s_2$ ist.

Der besseren Lesbarkeit wegen werde $f \in \Sigma_{w,s}$ mit $f{:}s_1 \ldots s_n \rightarrow s \in \Sigma$ bzw. nur mit $f{:}s_1 \ldots s_n \rightarrow s$ bezeichnet, wenn die Signatur aus dem Kontext klar ist. Mann nennt dann n auch die *Stelligkeit* des Funktionssymbols f.

Beispiel

$S_c =$ {kasus, numerus, genus, komparation, genusVerbi, passivisch}
$S_f =$ {nomen, substantiv, adjektiv, verb, deponens}

$(S, \leq) =$ {substantiv $\leq$ nomen, adjektiv $\leq$ nomen, deponens $\leq$ verb, passivisch $\leq$ genusVerbi}

$\Sigma_c =$ {nominativ: → kasus, genitiv: → kasus, dativ: → kasus, akkusativ: → kasus, ablativ: → kasus,
singular: → numerus, plural: → numerus,
masculinum: → genus, femininum: → genus, neutrum: → genus,
positiv: → komparation, komparativ: → komparation, superlativ: → komparation,
aktiv: → genusVerbi, passiv: → genusVerbi}

$\Sigma_f =$ {kas: nomen → kasus, num: nomen → numerus, gen: nomen → genus,
kmp: adjektiv → komparation, gnv: verb → genusVerbi, gnv: deponens → passivisch}

$(S_c \cup S_f, \leq, \Sigma_c \cup \Sigma_f)$ ist eine Signatur.
■

Um im folgenden einige gewünschte Ergebnisse erzielen zu können, müssen an die Signatur bestimmte Anforderungen gestellt werden.

Definition 1.2 (reguläre Signatur)
Eine Signatur ist *regulär*, wenn für jedes Funktionssymbol $f \in \Sigma$ und jedes $w \in S^*$ die Menge $\{s \mid f: w' \to s,\ w \leq w'\}$ entweder leer ist oder ein eindeutiges kleinstes Element enthält.[3]

Weitere Bedingungen sind:

- Aus $s' \leq s$ mit $s \in S_f$ folgt $s' \in S_f$, das heißt, alle Untersorten von feature-Sorten sind wieder feature-Sorten.

- Aus $f: w \to s$ und $f: w' \to s'$ mit $w, w' \in S^*$ und $w' \leq w$ folgt $s' \leq s$. Diese Bedingung wird aus der Semantik der Funktionen klar.

- Es darf keine unendliche Untersortenbeziehung der Form $\ldots s_3 \leq s_2 \leq s_1$ geben. Dies wird am einfachsten erreicht, indem man nur endliche Signaturen, das sind Signaturen mit endlichen Mengen S und Σ, zuläßt.

- Wenn zwei Sorten s_1 und $s_2 \in S$ gemeinsame Untersorten haben, dann existiert auch eine größte gemeinsame Untersorte $gcs^4(s_1, s_2) \in S$. (Abwärtsvollständigkeit)[5]

- Für alle $s \in S$, $w \in S^*$, $f: w' \to s'$ mit $s \leq s'$ und $w \leq w'$ ist die Menge
  $\{\omega \mid$ es existieren $\omega' \in S^*$, $\hat{s} \in S$ mit $f: \omega' \to \hat{s}$, $\hat{s} \leq s$, $\omega \leq w$, $\omega \leq \omega'\}$
  entweder leer oder besitzt ein größtes Element. (Zusammen mit der Abwärtsvollständigkeit heißt diese Eigenschaft Abwärtseindeutigkeit, vgl. [Waldmann 89].)

- Für jede feature-Sorte $s \in S_f$ mit ihren features $l_1: s_1' \to s_1$, ..., $l_n: s_n' \to s_n$, $s \leq s_1'$, ..., $s \leq s_n'$ und für jedes minimale $w \leq s_1 \ldots s_n$ gibt es eine minimale feature-Sorte $r \leq s$ mit den features $l_1: r_1' \to r_1$, ..., $l_n: r_n' \to r_n$ mit $r \leq r_1'$, ..., $r \leq r_n'$, so daß $w \leq r_1 \ldots r_n$ ist. (Die Notwendigkeit dieser Bedingung ist nicht unmittelbar einsichtig, sie wird jedoch im Zusammenhang mit Unifikation und gelösten Gleichungssystemen in Abschnitt 1.1.4 benötigt und dort anhand eines Beispiels erläutert.)

---

[3] Diese Forderung ist gleichbedeutend mit der Definition 7.2 für reguläre Signaturen in [Waldmann 89], die die Existenz einer kleinsten Sorte jedes Terms fordert und daher Terme benötigt. Da die folgende Definition der Terme bereits eine reguläre Signatur voraussetzt, ist Waldmann's Definition hier nicht verwendbar.

[4] gcs: englisch, Abkürzug für greatest common subsort

[5] Diese Bedingung stellt keine wirkliche Einschränkung dar, da solch eine fehlende Untersorte problemlos in den Sortenverband eingefügt werden kann. Semantisch werden dadurch keine neuen Objekte eingeführt, da die neue Sorte aus der Vereinigung ihrer Untersorten hervorgeht.

Definition 1.3 (zulässige Signatur)
Eine reguläre Signatur, die diesen Bedingungen genügt, heißt *zulässig*. Im folgenden werden nur noch zulässige Signaturen verwendet.

Durch die Definition der zulässigen Signaturen wird sichergestellt, daß

- der Sortenverband isolierte feature-Untersortenverbände enthält, deren feature-Sorten keine Konstruktor-Untersorten besitzen,
- features nur für feature-Sorten definiert sind,
- feature- und Konstruktor-Sorten als Argumentsorten für Konstruktor-Funktionen zugelassen sind, und daß
- Konstanten und Funktionen nur Konstruktor-Sorten als Ergebnissorte haben dürfen.

Außerdem ordnet die Signatur den feature-Sorten ihre features zu. Die *Menge der features* einer feature-Sorte s ist $\{l \mid l\colon s_0 \rightarrow s_1,\ s \leq s_0\}$.

Sowohl unter den Konstruktor-Deklarationen als auch unter den feature-Deklarationen sind Überladungen ausdrücklich zugelassen. Für feature-Deklarationen ist dies sogar notwendig, wenn für Untersorten die Ergebnissorte eines features eingeschränkt werden soll.

Man überzeuge sich, daß die Signatur aus dem vorigen Beispiel zulässig ist. Die Menge der features der Sorte 'adjektiv' ist zum Beispiel {kas, num, gen, kmp}.

Man beachte, daß die Menge der features einer feature-Sorte im allgemeinen auch features enthält, die nicht ausdrücklich für diese Sorte deklariert worden sind. Eine feature-Sorte erbt gewissermaßen die features ihrer Obersorten.

Definition 1.4 (Variablen)
Zu jeder Sorte $s \in S$ gibt es eine unendliche Menge V_s von Variablen der Sorte s. Die Menge V sei die Vereinigung aller Mengen V_s mit $s \in S$. Syntaktisch wird die Zugehörigkeit einer Variablen x zur Menge V_s durch x:s ausgedrückt.

Definition 1.5 (sortenrechte Terme)

Bemerkung: Anstatt erst die Syntax von Termen anzugeben und dann den Begriff der Wohlsortiertheit einzuführen, möchte ich gleich die Menge der sortenrechten Terme definieren.

Die Menge $T_{\Sigma,s}(V)$ sei die Menge der *sortenrechten Terme* der Sorte s über der Funktionssymbolmenge Σ mit Variablen aus V und wird wie folgt induktiv definiert:

$f \in T_{\Sigma,s}(V)$, wenn ein $f: \rightarrow s_i \in \Sigma_c$ existiert mit $s_i \leq s$

$x \in T_{\Sigma,s}(V)$, wenn $x \in V_{s'}$ mit $s' \leq s$

$f(t_1, \ldots, t_n) \in T_{\Sigma,s}(V)$, wenn
ein $(f:s_1 \ldots s_n \rightarrow s_0) \in \Sigma_c$ existiert mit $s_0 \leq s$ und $t_i \in T_{\Sigma,s_i}(V)$ für alle $1 \leq i \leq n$

$\{x \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\} \in T_{\Sigma,s}(V)$, wenn
$s \in S_f$ und $x \in V_{s'}$ mit $s' \leq s$ und alle l_i paarweise verschiedene features von s' sind und alle $t_i \in T_{\Sigma,s_i}(V)$ mit $s_i = \min(\{s_1 \mid (l_i:s_0 \rightarrow s_1) \in \Sigma_f,\ s' \leq s_0\})$ und $1 \leq i \leq n$ sind[6] und x in keinem der t_i vorkommt

Ein Term der Form $f(t_1, \ldots, t_n)$ heißt *Konstruktor-Term.*

Ein Term der Form $\{x \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\}$ heißt *feature-Term.* Die $l_i(x) \doteq t_i$ werden *feature-Gleichungen* oder kurz *features*, die Variable x die *feature-Variable* des Terms und $l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n$ *feature-Gleichungssystem* genannt. Ein feature-Term wird durch seine feature-Variable eindeutig benannt.

$T_\Sigma(V)$ sei die Vereinigung aller $T_{\Sigma,s}(V)$ mit $s \in S$ und damit die Menge aller sortenrechten Terme.
■

Beispiel

{x:nomen | kas(x) ≐ nominativ, num(x) ≐ singular, gen(x) ≐ masculinum}

ist unter der Signatur des vorigen Beispiels Term von $T_\Sigma(V)$.

* {x:nomen | kmp(x) ≐ superlativ} oder
* {x:nomen | kas(x) ≐ neutrum}

sind dagegen keine Terme von $T_\Sigma(V)$.

Die Termdefinition unterscheidet sich lediglich durch die zusätzliche Einführung von feature-Termen von der der ordnungssortierten Logik. Da die Menge der features für jede feature-Sorte durch die Signatur eindeutig festgelegt ist, könnte man, eine feste Zuordnung von feature-Symbolen zu Argumentstellen vorausgesetzt, jeden feature-Term auch durch einen entsprechenden Konstruktor-Term darstellen, wobei nicht spezifizierte features durch eine Variable ersetzt werden. Dieser scheinbar einfache Zusammenhang verliert jedoch spätestens bei der Unifikation nicht gleichsortiger feature-Terme seine Praktikabilität, da dann in der Regel neue features hinzukommen, für die keine Argumentstelle vorgesehen ist.

[6] Durch die Definition einer zulässigen Signatur wird garantiert, daß ein solches minimales s_i existiert, wenn l_i feature von s' ist. Andernfalls ist $\{x \mid l_1(x)=t_1, \ldots, l_n(x)=t_n\}$ auch kein sortenrechter Term.

Definition 1.6 (Gleichheit von Termen)
Zwei Terme sind genau dann gleich, wenn sie bis auf die Reihenfolge ihrer feature-Gleichungen syntaktisch gleich sind.

Innerhalb eines Terms kann ein feature-Term mehrfach auftreten: Daß derselbe Term gemeint ist, wird durch die Verwendung der gleichen feature-Variable ausgedrückt. Ein solcher Verweis wird *Koreferenz* genannt. Ein Term ist in *Normalform*, wenn alle darin enthaltenen koreferenzierten feature-Terme gleich sind.

Definition 1.7 (Σ-Gleichungen)
Eine Gleichung der Form $s \doteq t$ mit $s, t \in T_\Sigma(V)$, wobei weder s noch t feature-Terme enthalten, ist eine *Σ-Gleichung*. Darüberhinaus sind alle feature-Gleichungen der Form $l(X) \doteq t$, wobei t keine feature-Terme enthält, *Σ-Gleichungen*.

Bemerkung: Der Zusatz mit den feature-Gleichungen ist notwendig, da die linke Seite einer feature-Gleichung $l(x) \doteq t$ kein Term nach obiger Definition ist.

Wo die Signatur aus dem Kontext deutlich wird, werden Σ-Gleichungen schlicht Gleichungen genannt.

Definition 1.8 (Σ-Gleichungssysteme)
Ein *Σ-Gleichungssystem* ist eine Menge aus Σ-Gleichungen. Es wird auch manchmal *Σ-Gleichungsmenge* genannt.

Auch hier kann das Σ in Zukunft entfallen.

Definition 1.9 (gelöste Form)
Ein Gleichungssystem T der Form

$$\{x_1 \doteq s_1, \ldots, x_m \doteq s_m, l_1(y_1) \doteq t_1, \ldots, l_n(y_n) \doteq t_n\}$$

mit

- $x, y \in V$ und $x_i \neq y_j$ für alle $1 \leq i \leq m$, $1 \leq j \leq n$,
- $x_1, \ldots, x_n$ sind paarweise verschieden,
- $l_1(y_1), \ldots, l_n(y_n)$ sind paarweise verschieden,
- wenn s_i eine Variable ist, dann taucht sie nur einmal in T auf,
- wenn $x_i \in V_s$, dann ist $s_i \in T_{\Sigma,s}(V)$ und enthält keine feature-Terme,
- wenn $y_j \in V_s$, dann ist $t_j \in T_{\Sigma,s'}(V)$ mit $s' = \min(\{s_1 \mid (l_j : s_0 \rightarrow s_1) \in \Sigma_f,\ s \leq s_0\})$ und enthält keine feature-Terme,

und ohne zyklische Abhängigkeiten unter den Variablen aus T ist in gelöster Form.

Eine zyklische Abhängigkeit unter Variablen aus T liegt genau dann vor, wenn die Abhängigkeitsrelation $\rightarrow_T$ zwischen den Variablen aus T azyklisch ist. Dabei wird die Abhängigkeitsrelation wie folgt definiert: $x \rightarrow_T y$ genau dann, wenn T eine Gleichung $s \doteq t$ enthält, bei der x in s und y in t vorkommt.
■

Bemerkung: Die Voraussetzung, daß T getrimmt sein muß, kann entfallen, da alle

Σ-Gleichungen und Σ-Gleichungssysteme bereits in getrimmter Form sind. (Vergleiche bitte [Smolka 89])

Satz 1.10 (Zusammenhang zwischen Term und Gleichungssystem)
Zu jedem Term $t \in T_\Sigma(V)$ in Normalform existiert ein äquivalentes Gleichungssystem in gelöster Form.

Beweis (durch Konstruktion des Gleichungssystems induktiv über den Aufbau des Terms)

Zunächst wird eine Hilfsfunktion h definiert, die aus einem Term alle features entfernt und damit sein Konstruktor-Termgerüst liefert.

$$h(x) := x$$
$$h(f(t_1, \ldots, t_n) := f(h(t_1), \ldots, h(t_n))$$
$$h(\{x \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\}) := x$$

Mit Hilfe dieser Funktion wird eine Funktion g rekursiv definiert, die eine Menge von feature-Gleichungen liefert.

$$g(x) := \varnothing$$
$$g(f(t_1, \ldots, t_n)) := g(t_1) \cup \ldots \cup g(t_n)$$
$$g(\{x \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\}) := \{l_1(x) \doteq h_1\} \cup \ldots \cup \{l_n(x) \doteq h_n\} \cup g(t_1) \cup \ldots \cup g(t_n)$$
$$\text{mit } h_1 := h(t_1), \ldots, h_n := h(t_n)$$

Die Funktion g verflacht die geschachtelte Termstruktur, man spricht in diesem Zusammenhang auch von Entfaltung. Jedes von g erzeugte Gleichungssystem ist in gelöster Form, da es nur Gleichungen der Form $l(x) \doteq t$ enthält, x nicht in t vorkommt und jedes l(x) nur einmal darin auftritt.

Sinn und Zweck dieser Konstruktion ist es, zu einem feature-Term ein äquivalentes Gleichungssystem zu erhalten, das an der Bildung des Ausgangsgleichungssystems zur Lösung des Unifikationsproblems beteiligt wird. Da Konstruktor-Terme für sich allein keine Gleichung darstellen, entfällt das äußere Konstruktor-Termgerüst bei der Konversion in ein Gleichungssystem zunächst, so daß ein reiner Konstruktor-Term immer das leere Gleichungssystem repräsentiert. Bei der Reduktion des Unifikationsproblems zweier Terme s und t auf die Lösung eines Gleichungssystems stellt die Gleichung aus den Konstruktor-Termgerüsten h(s) und h(t) jedoch den Aufhänger der Lösung dar.

Bei Kenntnis des (Konstruktor-) Termgerüsts h(t) kann aus g(t) der ursprüngliche Term rekonstruiert werden, indem alle eine Variable x spezifizierenden feature-Gleichungen $l_i(x) \doteq t_i$ zu einem feature-Term zusammengefaßt werden und die Vorkommen von x in allen Konstruktortermen durch den entsprechenden feature-Term ersetzt werden. Die Entfaltung ist also umkehrbar.
■

Man sagt, der Term t repräsentiert das Gleichungssystem T, wenn T = g(t) ist. Er kann als abkürzende Schreibweise aufgefaßt werden.

Beispiel

Werde durch

$$S'_c = S_c$$
$$S'_f = S_f \cup \{satz\}$$
$$\Sigma'_c = \Sigma_c$$
$$\Sigma'_f = \Sigma_f \cup \{\text{subjekt: satz} \rightarrow \text{nomen, prädikat: satz} \rightarrow \text{verb, objekt: satz} \rightarrow \text{nomen}\}$$

die Signatur aus dem vorigen Beispiel erweitert. Dann ist

$$t = \{x{:}satz \mid subjekt(x) \doteq \{x_1{:}substantiv \mid kas(x_1) \doteq nominativ\},$$
$$prädikat(x) \doteq \{x_2{:}verb \mid gnv(x_2) \doteq aktiv\},$$
$$objekt(x) \doteq \{x_3{:}adjektiv \mid gen(x_3) \doteq neutrum\}\} \in T_\Sigma(V)$$

und

$$h(t) = x{:}satz \text{ und}$$
$$g(t) = \{subjekt(x) \doteq x_1{:}substantiv,\ prädikat(x) \doteq x_2{:}verb,\ objekt(x) \doteq x_3{:}adjektiv,$$
$$kas(x_1) \doteq nominativ,\ gnv(x_2) \doteq aktiv,\ gen(x_3) \doteq neutrum\}.$$

■

1.1.3 Semantik

Ich werde an dieser Stelle die Semantik nur skizzieren, der interessierte Leser sei auf [Smolka 89] verwiesen.

Sei $(S, \leq, \Sigma)$ eine Signatur. Eine Σ-Algebra A umfaßt dann

- für jede Sorte $s \in S$ eine Menge s^A, und
- für jedes Funktionssymbol $f: s_1 \ldots s_n \rightarrow s \in \Sigma$ eine Abbildung f^A mit $f^A: s_1{}^A \ldots s_n{}^A \rightarrow s^A$,

die folgenden Bedingungen genügen:

- wenn $s_1 \leq s_2$, dann ist $s_1{}^A \subseteq s_2{}^A$, und
- wenn $f: s_1 \ldots s_n \rightarrow s \in \Sigma$ und $a_i \in s_i{}^A$ ist für alle $1 \leq i \leq n$, dann ist $f^A(a_1, \ldots, a_n) \in s^A$.

In Worten ausgedrückt bedeutet das das folgende: Jede Sorte s bezeichnet eine Menge, und Untersorten bezeichnen Teilmengen davon, unabhängig, ob es sich um Konstruktor- oder feature-Sorten handelt. Ein Konstruktor-Term $f(t_1, \ldots, t_n)$ der Sorte s bezeichnet genau ein Element der dazugehörigen Menge s^A. Was jedoch durch einen feature-Term bezeichnet wird, ist aus dieser Definition der Semantik nicht direkt abzulesen.

Es gilt jedoch für jedes feature-Symbol l mit $l: s \rightarrow s' \in \Sigma_f$, daß mit $x \in s^A$ $l^A(x) \doteq y \in s'^A$ sein muß. Nun wird in einem feature-Term der (syntaktischen) Form $\{x{:}s \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\}$ für jedes l_i der Term t_i angegeben, der dieses Element $y \in s^A$, im folgenden $t_i{}^A$ genannt, bezeichnet. Semantisch spezifiziert ein feature-Term also genau die Teilmenge von s^A, die das Gleichungssystem $l_1{}^A(x) \doteq t_1{}^A, \ldots, l_n{}^A(x) \doteq t_n{}^A$ löst.

Um die Semantik der feature-Terme im Rahmen der ordnungssortierten Logik zu behandeln, definieren Smolka und Aït-Kaci für jede feature-Sorte $s \in S_f$ einen impliziten Konstruktor f_s: $s_1 \ldots s_n \rightarrow s$, der für jedes feature $l_i: s \rightarrow s_i$ von s genau eine Argumentstelle besitzt. Die gesuchte Lösung x des Gleichungssystems wird dann durch den impliziten Konstruktor-Term $f_s(x_1, \ldots, x_n)$ repräsentiert und jedes feature l_i als ein Selektor aufgefaßt, der aus dem impliziten Konstruktor-Term das i-te Argument auswählt. [Smolka 89]

Um das Gleichungssystem auch tatsächlich in eine gelöste Form bringen zu können, defineren sie eine entsprechende Ersetzungsregel

$$l_i(f_s(x_1, \ldots, x_i, \ldots, x_n)) \rightarrow x_i,$$

mit der sich eine feature-Gleichung der Form $l_i(x:s) \doteq t_i = l_i(f_s(x_1, \ldots, x_i, \ldots, x_n)) \doteq t_i$ in die Form $x_i \doteq t_i$ bringen läßt. Das aus der Anwendung der Ersetzungsregel auf alle Gleichungen resultierende Gleichungssystem $\{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$ ist dann in gelöster Form.

Zusammenfassend kann man sagen, daß durch feature-Terme einer Sorte s Teilmengen der durch s benannten Menge s^A spezifiziert werden, während Konstruktor-Terme einzelne Elemente repräsentieren.

1.1.4 Substitution, Konkretisierung und Unifikation

Aufgabe der Unifikation ist es, zwei Terme gleich zu machen. Die bekannte Termunifikation bewerkstelligt dies mit Hilfe von geeigneten Substitutionen, mit denen Variablen durch Terme ersetzt werden. Diese Methode ist jedoch für die Unifikation von feature-Termen nicht immer ausreichend: Ein feature-Term kann andere features enthalten als der mit ihm zu unifizierende, so daß eine Unifikation durch Substitution allein nicht möglich ist.

Bei Smolka und Aït-Kaci wird die Unifikation als Berechnung einer Lösung des kombinierten feature-Gleichungssystems betrachtet, aus der sich der als Ergebnis entstandene feature-Term rekonstruieren läßt. Man beobachtet, daß sich jeder Ausgangsterm durch geeignete Substitutionen und Erweiterung um die fehlenden features des jeweils anderen Terms in den Ergebnisterm überführen läßt. Um diese Ersetzung systematisch durchführen zu können, wird parallel zur Substitution ein neuer Operator, die Konkretisierung, eingeführt werden.

Definition 1.11 (Substitution)

Eine Abbildung $\sigma: V \rightarrow T_\Sigma(V)$ heißt *Substitution*, wenn
- für alle $x \in V_s$ mit $s \in S_c$ gilt, daß $\sigma(x) \in T_{\Sigma,s}(V)$ ist und
- für alle $x \in V_s$ mit $s \in S_f$ gilt, daß $\sigma(x) \in V_{s'}$ ist mit $s' \leq s$ und
- $\{x \in V \mid \sigma(x) \neq x\}$ endlich ist.

Bemerkungen: Ist $x \in V_s$ und $\sigma(x) \in V_s$, so nennt man $\sigma(x)$ eine *Umbennenung* von x. Ist $x \in V_s$ und $\sigma(x) \in V_{s'}$ mit $s' \leq s$, so nennt man $\sigma(x)$ eine *Einschränkung*[7] von x.

σ wird durch folgende Definition kanonisch zu σ^* auf Terme fortgesetzt:

[7] Den häufig verwendeten Begriff 'Abschwächung' verwende ich absichtlich nicht. (für eine Begründung siehe Anhang: Begriffsklärung)

$\sigma^*(x) := \sigma(x)$
$\sigma^*(f(t_1, \ldots, t_n)) := f(\sigma^*(t_1), \ldots, \sigma^*(t_n))$
$\sigma^*(\{x \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\}) := \{\sigma(x) \mid l_1(\sigma(x)) = \sigma^*(t_1), \ldots, l_n(\sigma(x)) = \sigma^*(t_n)$

Die Fortsetzung auf Gleichungssysteme erfolgt entsprechend:

$\sigma^*(\{s_1 \doteq t_1, \ldots, s_n \doteq t_n, l_1(x_1) \doteq r_1, \ldots, l_m(x_m) \doteq r_m\}) :=$
$\{\sigma^*(s_1) \doteq \sigma^*(t_1), \ldots, \sigma^*(s_n) \doteq \sigma^*(t_n), l_1(\sigma(x_1)) \doteq \sigma^*(r_1), \ldots, l_m(\sigma(x_m)) \doteq \sigma^*(r_m)\}$

■

Im folgenden werde σ^* wieder σ genannt.

Definition 1.12 (sortenrecht angewandte Substitution)
Die Anwendung einer Substitution heißt *sortenrecht*, wenn sie einen Term aus $T_\Sigma(V)$ auf einen Term aus $T_\Sigma(V)$, das heißt einen sortenrechten Term auf einen sortenrechten Term abbildet. Im werden Substitutionen nur sortenrecht angewandt.

Bemerkung: Daß nicht jede Anwendung einer Substitution sortenrecht ist, soll an einem Beispiel demonstriert werden:
$\{x{:}verb \mid gnv(x) \doteq aktiv\}$
wird durch
$\sigma(x) = y{:}deponens$
nicht sortenrecht substituiert, da
$\{y{:}deponens \mid gnv(y) \doteq aktiv\}$
kein Element von $T_\Sigma(V)$ nach der Signatur der vorangegangenen Beispiele ist.

Die Komposition zweier Substitutionen σ und τ ist definiert durch

$\tau \circ \sigma(t) := \tau(\sigma(t))$.

Definition 1.13 (Konkretisierung)
Eine Abbildung $\kappa: V \to T_\Sigma(V)$ heißt Konkretisierung, wenn
- für alle $x \in V_s$ mit $s \in S_c$ $\kappa(x) = x$ ist und
- für alle $x \in V_s$ mit $s \in S_f$ gilt, daß $\kappa(x) \in T_{\Sigma,s}(V)$ ein feature-Term mit der feature-Variable x ist und
- $\{x \in V \mid \kappa(x) \neq x\}$ endlich ist.

Bemerkung: Durch eine Konkretisierung wird die Wertemenge einer feature-Variablen mit feature-Gleichungen genauer spezifiziert, eben 'konkreter'.

κ werde auf Terme zu κ^* wie folgt fortgesetzt:

$\kappa^*(x) := \kappa(x)$
$\kappa^*(f(t_1, \ldots, t_n)) := f(\kappa^*(t_1), \ldots, \kappa^*(t_n))$
$\kappa^*(\{x \mid l_1(x) \doteq t_1, \ldots, l_n(x) \doteq t_n\}) := \{x \mid l_1(x) = \kappa^*(t_1), \ldots, l_n(x) = \kappa^*(t_n)\} \uplus \kappa(x)$.

Der nicht kommutative Hilfsoperator $\uplus$ vereinigt die Gleichungen von feature-Termen wie

folgt:

$$\{x|l_1(x)\doteq t_1,\ldots,l_n(x)\doteq t_n\} \uplus \{x\} := \{x|l_1(x)\doteq t_1,\ldots,l_n(x)\doteq t_n\}$$

$$\{x|l_1(x)\doteq t_1,\ldots,l_n(x)\doteq t_n\} \uplus \{x|k_1(x)=s_1,\ k_2(x)=s_2,\ \ldots,\ k_m(x)=s_m\} :=$$
$$\{x|l_1(x)\doteq t_1,\ldots,l_n(x)\doteq t_n\} \uplus \{x|k_2(x)=s_2,\ldots,k_m(x)=s_m\},\ \text{falls } k_1 = l_i \text{ für ein } 1\leq i\leq n,$$
$$\{x|l_1(x)\doteq t_1,\ldots,l_n(x)\doteq t_n,\ k_1(x)=s_1\} \uplus \{x|k_2(x)=s_2,\ldots,k_m(x)=s_m\} \text{ sonst}$$

Bemerkung: Die Konkretisierung fügt nur dann eine feature-Gleichung $l(x)\doteq t$ zu einem feature-Term hinzu, wenn deren linke Seite $l(x)$ nicht schon in der Gleichungsmenge des feature-Terms vorkommt. Deshalb kann durch die Konkretisierung weder eine doppelte feature-Gleichung in den Term aufgenommen werden, noch das Gleichungssystem durch einander widersprechende features inkonsistent werden.

Entsprechend wird κ^* auf Gleichungssysteme fortgesetzt:

$$\begin{aligned}\kappa^*(\{s_1\doteq t_1,\ \ldots,\ s_n\doteq t_n,\ l_1(x_1)\doteq r_1,\ \ldots,\ l_m(x_m)\doteq r_m\})\\ := \{s_1\doteq t_1,\ \ldots,\ s_n\doteq t_n\} \cup \kappa^*(\{l_1(x_1)\doteq r_1,\ \ldots,\ l_m(x_m)\doteq r_m\})\\ = \{s_1\doteq t_1,\ \ldots,\ s_n\doteq t_n\} \cup \{l_1(x_1)\doteq r_1,\ \ldots,\ l_m(x_m)\doteq r_m\} \cup\\ \{l(x)\doteq t \mid l(x)\doteq t \text{ ist feature-Gleichung von } \kappa(x) \text{ und}\\ x=x_i \text{ für mindestens ein } 1\leq i\leq m \text{ und}\\ l(x)\neq l_i(x_i) \text{ für alle } 1\leq i\leq m\}\end{aligned}$$

Im folgenden nenne ich κ^* wieder κ.
■

Die Komposition zweier Konkretisierungen κ und λ ist definiert durch

$$\lambda \circ \kappa(t) := \lambda(\kappa(t)).$$

Definition 1.14 (unifizierbar, Unifikator)
Zwei Terme s und $t \in T_\Sigma(V)$ heißen *unifizierbar*, wenn es ein Paar (σ, κ) bestehend aus einer Substitution σ und einer Konkretisierung κ gibt, so daß $\kappa(\sigma(s))$ und $\kappa(\sigma(t))$ gleich sind. $U := (\sigma, \kappa)$ heißt dann *Unifikator* von s und t.

Bemerkung: Gelegentlich wird auch das Ergebnis der Unifikation, $\kappa(\sigma(s))$ bzw. $\kappa(\sigma(t))$, Unifikator genannt.

Die Verknüpfung zweier Unifikatoren $U = (\sigma, \kappa)$ und $U' = (\sigma', \kappa')$ sei definiert durch

$$U' \circ U\ (t) = (\sigma', \kappa') \circ (\sigma, \kappa)\ (t) := \kappa'(\sigma'(\kappa(\sigma(t))))$$

Definition 1.15 (allgemeinster Unifikator)
Ein Unifikator zweier Terme s und $t \in T_\Sigma(V)$ heißt *allgemeinster Unifikator (englisch: most general unifier, kurz: mgu)*, wenn sich jeder Unifikator dieser Terme als Verknüpfung des

allgemeinsten Unifikators mit einem Paar (σ, κ) bestehend aus einer Substitution σ und einer Konkretisierung κ darstellen läßt.

Satz 1.16 (Existenz und Eindeutigkeit allgemeinster Unifikatoren)
Wenn zwei Terme s und t unifizierbar sind, dann existiert ein allgemeinster Unifikator und dieser ist bis auf Umbenennung eindeutig bestimmt.

Beweis (durch Angabe eines Verfahrens zur Konstruktion des allgemeinsten Unifikators)

Ich folge der Beweisführung in [Hofbauer 89] und verwende dabei Sätze aus [Smolka 89].

Der Beweis wird in mehreren Schritten durchgeführt. Zunächst wird das Problem der Unifikation zweier Terme auf die Lösung eines äquivalenten Gleichungssystems reduziert. Dann wird ein Verfahren zur Bestimmung der allgemeinsten Lösung des Gleichungssystems aufgeführt und dessen Korrektheit und Vollständigkeit gezeigt. Schließlich wird anhand der Lösung ein Unifikator konstruiert und nachgewiesen, daß dieser die Ausgangsterme unifiziert und daß es keinen allgemeineren gibt.

1. Schritt (Reduktion der Unifikationsaufgabe in ein Gleichungssystem)

Die herkömmliche Unifikation zweier Terme s und t läßt sich zurückführen auf die Lösung des Gleichungssystems $\{s \doteq t\}$ [Hofbauer 89]. Da die Terme der ordnungssortierten feature-Logik bereits selbst als abkürzende Schreibweise für Gleichungssysteme aufzufassen sind, müssen die durch die Terme repräsentierten Gleichungssysteme mit in das Startgleichungssystem aufgenommen werden. Von s und t werden dagegen nur die feature-freien Gerüste h(s) und h(t) benötigt. Als Startgleichungssystem T ergibt sich demnach $T := \{h(s) \doteq h(t)\} \cup g(s) \cup g(t)$. Es liegt zudem in getrimmter Form vor. (vgl. Definition 1.9 und Beweis zu Satz 1.10)

2. Schritt (Bestimmung der allgemeinsten Lösung des Gleichungssystems)

Zunächst wird eine Reihe von Regeln definiert, durch die ein Gleichungssystem in ein neues überführt wird. (Man vergleiche bitte mit [Smolka 89], Substitutionen werden in geschweiften Klammern vor den Gleichungssystemen, auf die sie angewandt werden, angegeben)

Decomposition

(D) $\{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\} \cup T \rightarrow \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \cup T$
wenn f ein Konstruktor-Symbol ist

Merging

(M1) $\{x \doteq s, x \doteq t\} \cup T \rightarrow \{x \doteq s, s \doteq t\} \cup T$
wenn s und t keine Variablen sind

(M2) $\{l(x)\doteq s, l(x)\doteq t\} \cup T \rightarrow \{l(x)\doteq s, s\doteq t\} \cup T$

Weakening

(W1) $\{x\doteq y\} \cup T \rightarrow \{x\doteq z, y\doteq z\} \cup \{x\leftarrow z, y\leftarrow z\}T$
wenn y keine Variable einer Untersorte der Sorte von x ist und z eine neue Variable der größten gemeinsamen Untersorte der Sorten von x und y ist

(W2) $\{l(x)\doteq y\} \cup T \rightarrow \{y\doteq z\} \cup \{y\leftarrow z\}(\{l(x)\doteq y\} \cup T)$
wenn y keine Variable einer Untersorte der Sorte von l(x) ist und z eine neue Variable der größten gemeinsamen Untersorte der Sorten von l(x) und y ist

Isolation

(I) $\{x\doteq y\} \cup T \rightarrow \{x\doteq y\} \cup \{x\leftarrow y\}T$
wenn $x \not\equiv y$ ist, x in T vorkommt die Sorte von y Untersorte der Sorte von x ist

Orientation

(O) $\{s\doteq x\} \cup T \rightarrow \{x\doteq s\} \cup T$
wenn s keine Variable und nicht von der Form l(y) ist

Elimination

(E) $\{x\doteq x\} \cup T \rightarrow T$

Mit Hilfe dieser Regeln soll sich nun eine Lösung[8] wie folgt finden lassen:

> Beginne mit dem Startgleichungssystem T und wende darauf so lange fortgesetzt eine der Regeln (D) - (E) an, bis keine mehr anwendbar ist. Nenne das Ergebnisgleichungssystem T'.

Daß dieses Verfahren eine Lösung liefert, wird durch die folgenden Lemmata gezeigt.

Lemma 1.17 (Unifikationsinvarianz der Regeln (D) - (E))
Wenn T_i ein getrimmtes Gleichungssystem und T_{i+1} aus T_i durch Anwendung einer der Regeln (D) - (E) hervorgegangen ist, dann ist T_{i+1} wieder ein getrimmtes Gleichungssystem und

(i) die Mengen der Unifikatoren von T_i und T_{i+1} sind gleich und
(ii) U ist genau dann allgemeinster Unifikator von T_i, wenn U allgemeinster Unifikator von T_{i+1} ist.

[8] In der Literatur wir die Lösung des Gleichungssystems gelegentlich auch Unifikator und der Vorgang der Lösung Unifikation genannt. Zur besseren Unterscheidung von der Termunifikation möchte ich hier jedoch auf diese Überladung verzichten.

Beweis
(i) siehe Beweis zu Theorem 6.4.1 in [Smolka 89]
(ii) folgt unmittelbar aus (i) (vgl. [Hofbauer 89])

Lemma 1.18 (Terminierung des Lösungsverfahrens)
Ausgehend von einem Startgleichungssystem T ist nach einer endlichen Zahl von Regelanwendungen keine Regel mehr anwendbar.

Beweis
siehe Beweis zu Theorem 6.4.2 in [Smolka 89]

Lemma 1.19 (Vollständigkeit und Korrektheit des Lösungsverfahrens)
(i) Das Gleichungssystem T ist genau dann lösbar, wenn es ein Gleichungssystem T' in gelöster Form gibt, das aus T durch Anwendung der Regeln (D) - (E) hervorgegangen ist.
(ii) Die allgemeinste Lösung von T' läßt sich dann aus T' eindeutig bestimmen.

Beweis

(i) Der Beweis des ersten Teils kann in [Smolka 89], Beweis zu Theorem 6.4.3, nachgelesen werden.

(ii) a) (Konstruktion einer Lösung)
Da T' in gelöster Form ist, muß es von der Form $\{x_1 \doteq s_1, \ldots, x_m \doteq s_m, l_1(y_1) \doteq t_1, \ldots, l_n(y_n) \doteq t_n\}$ sein. Nun ist $\sigma := \{x_i \leftarrow t_i \mid x_i \doteq t_i \in T'\}$ Lösung von T', da dann $\sigma(x_i) = t_i$ und $\sigma(t_i) = t_i$ ist und die feature-Gleichungen $l_1(y_1) \doteq t_1, \ldots, l_n(y_n) \doteq t_n$ aufgrund ihrer Semantik als gelöst anzusehen sind (vgl. Abschnitt 1.1.3 und [Smolka 89]).

b) (Lösung ist die allgemeinste)
Sei τ eine beliebige Lösung von T'. Dann ist zu beweisen, daß σ allgemeiner ist als τ, also zum Beispiel $\tau \circ \sigma(x) = \tau(x)$ für alle Variablen x. Doch genau dies ist der Fall, denn $\tau(\sigma(x_i)) = \tau(x_i)$ für $\sigma(x_i) = x_i$ und $\tau(\sigma(x_i)) = \tau(t_i) = \tau(x_i)$ für $\sigma(x_i) = t_i$, da τ Lösung sein soll. Also ist σ allgemeinste Lösung.

Bemerkung: Wenn die Signatur die letzte Zulässigkeitsbedingung nicht erfüllt, kann es gelöste Gleichungssysteme geben, aus denen sich keine Lösung bestimmen läßt. Ein solches Beispiel ist durch die Signatur

$$(S, \leq) = \{s_2 \leq s_1,\ s_4 \leq s_3,\ s_5 \leq s_3\}$$
$$\Sigma = \{l{:}s_1 \rightarrow s_3,\ l{:}s_2 \rightarrow s_4,\ d{:} \rightarrow s_4,\ e{:} \rightarrow s_5\}$$

gegeben, in der das System $\{l(x{:}s_1) \doteq e\}$ zwar in gelöster Form ist, jedoch keine Lösung (kein impliziter Konstruktor für x mit e als Argument) existiert.

Damit ist gezeigt: Wenn das Gleichungssystem lösbar ist, existiert eine allgemeinste Lösung. Es bleibt zu zeigen, daß dann auch ein allgemeinster Unifikator der Ausgangsterme s und t existiert.

3. Schritt (Konstruktion des allgemeinsten Unifikators)

Ein Paar (σ, κ) wir aus der Lösung T' wie folgt gewonnen:

$\sigma(x_i) := t_i$ für alle $x_i \doteq t_i \in T'$
$\sigma(x_i) := x_i$ sonst

$$\kappa(x_i) := \biguplus \{l_j(x_i) \doteq t_k\} \in (\sigma(g(s)) \cup \sigma(g(t))) \setminus (\sigma(g(s)) \cap \sigma(g(t)))$$
$$(= \sigma(g(s)) \setminus \sigma(g(t)) \cup \sigma(g(t)) \setminus \sigma(g(s)))$$

$\kappa(x_i) := x_i$ sonst

Lemma 1.20
$U = (\sigma, \kappa)$ ist allgemeinster Unifikator der beiden Terme s und t.

Beweis

a) (U ist Unifikator)

Im 2. Schritt wurde bewiesen, daß σ allgemeinste Lösung des Gleichungssystems T' und damit auch von T, insbesondere also auch Lösung von $h(s) \doteq h(t)$ ist, so daß $\sigma(h(s)) = \sigma(h(t))$ gilt. Da κ eine Funktion ist, folgt $\kappa(\sigma(h(s))) = \kappa(\sigma(h(t)))$ daraus.

Des weiteren sind wegen der Definition der Konkretisierung für Gleichungssysteme

$$\begin{aligned}\kappa(\sigma(g(s))) &= \sigma(g(s)) \cup (\sigma(g(s)) \cup \sigma(g(t))) \setminus (\sigma(g(s)) \cap \sigma(g(t))) \\ &= \sigma(g(s)) \cup \sigma(g(s)) \setminus \sigma(g(t)) \cup \sigma(g(t)) \setminus \sigma(g(s)) \\ &= \sigma(g(s)) \cup \sigma(g(t)) \setminus \sigma(g(s)) \\ &= \sigma(g(s)) \cup \sigma(g(t))\end{aligned}$$

und analog

$$\begin{aligned}\kappa(\sigma(g(t))) &= \sigma(g(t)) \cup (\sigma(g(s)) \cup \sigma(g(t))) \setminus (\sigma(g(s)) \cap \sigma(g(t))) \\ &= \sigma(g(t)) \cup \sigma(g(s)),\end{aligned}$$

also $\kappa(\sigma(g(s))) = \kappa(\sigma(g(t)))$.

Daraus folgt wegen der Umkehrbarkeit der von g und h durchgeführten Entfaltung, daß $\kappa(\sigma(s)) = \kappa(\sigma(t))$.

b) (U ist allgemeinster Unifikator)

Sei $U' = (\sigma', \kappa')$ ein beliebiger Unifikator von s und t, dann ist zu zeigen, daß $U = (\sigma, \kappa)$ allgemeiner ist. Dies ist zum Beispiel der Fall, wenn sich zeigen läßt, daß

$$U' \circ U\,(t) = (\sigma', \kappa') \circ (\sigma, \kappa)\,(t) = \kappa'(\sigma'(\kappa(\sigma(t)))) = \kappa'(\sigma'(t)) = (\sigma', \kappa')\,(t) = U'\,(t)$$

ist. Daß $\sigma' \circ \sigma\,(t) = \sigma'\,(t)$ ist, wurde bereits im Beweis zu Lemma 1.19 gezeigt.

Für alle x, für die $\kappa(\sigma(x)) = \sigma(x)$ ist, gilt offensichtlich:

$$\kappa'(\sigma'(\kappa(\sigma(x)))) = \kappa'(\sigma'(\sigma(x))) = \kappa'(\sigma'(x))$$

Für alle x, für die $\kappa(\sigma(x)) = \{\sigma(x) \mid l_1(\sigma(x)) \doteq \sigma(t_1), \ldots, l_n(\sigma(x)) \doteq \sigma(t_n)\}$ ist, gilt nach der Definition für Konkretisierungen von feature-Termen:

$$\begin{aligned}
&\kappa'(\sigma'(\kappa(\sigma(x))))\\
&= \kappa'(\sigma'(\{\sigma(x) \mid l_1(\sigma(x)) \doteq \sigma(t_1), \ldots, l_n(\sigma(x)) \doteq \sigma(t_n)\})\\
&= \kappa'(\{\sigma'(x) \mid l_1(\sigma'(x)) \doteq \sigma'(t_1), \ldots, l_n(\sigma'(x)) \doteq \sigma'(t_n)\})\\
&= \{\sigma'(x) \mid l_1(\sigma'(x)) \doteq \sigma'(t_1), \ldots, l_n(\sigma'(x)) \doteq \sigma'(t_n)\} \uplus\\
&\quad \{\sigma'(x) \mid l_1(\sigma'(x)) \doteq \sigma'(t_1), \ldots, l_n(\sigma'(x)) \doteq \sigma'(t_n), l_{n+1}(\sigma'(x)) \doteq \sigma'(t_{i+1}), \ldots, l_m(\sigma'(x)) \doteq \sigma'(t_m)\}\\
&= \{\sigma'(x) \mid l_1(\sigma'(x)) \doteq \sigma'(t_1), \ldots, l_n(\sigma'(x)) \doteq \sigma'(t_n), l_{n+1}(\sigma'(x)) \doteq \sigma'(t_{i+1}), \ldots, l_m(\sigma'(x)) \doteq \sigma'(t_m)\}\\
&= \kappa'(\sigma'(x))
\end{aligned}$$

da $\kappa'(\sigma'(x))$ für alle x mindestens die feature-Gleichungen von $\sigma'(\kappa(x))$ enthalten muß, um Lösung zu sein (vgl. dazu Teil a) dieses Beweises).

1.2 Einführung einer Dependenzgrammatik

Spätestens seit Chomsky gibt es unübersehbar Bestrebungen, natürliche Sprache mit Hilfe einer künstlichen Sprache zu beschreiben, eben zu formalisieren. "Damit steht die moderne Linguistik im scharfen Gegensatz zur traditionellen Sprachwissenschaft, bei der die natürliche Sprache in Doppelfunktion auftritt, einmal als Objektsprache, einmal als Beschreibungs- oder Metasprache." [Engel 77] Martin Kay schrieb 1982 in seinem Papier 'Parsing in Functional Unification Grammar' etwas polemisch, die Linguistik sei, wie die meisten anderen wissenschaftlichen Unternehmungen, ohne Formalismus impotent [Kay 82].

Verständlicher Weise hat sich diese Erkenntnis bei den meisten Linguisten noch nicht durchsetzen können, so daß die Zahl der Arbeiten, die sich mit der streng formalen Definition von Grammatiken natürlicher Sprache befassen, bislang in Grenzen hielt.

Dennoch sind bereits einige fomale Ansätze verfaßt worden. Shieber, der die wichtigsten Vertreter vergleichend gegenübergestellt hat, teilt die Formalismen ein in linguistische Theorien und linguistische Werkzeuge, und bewertet sie nach drei Kriterien [Shieber 86]:

- Linguistic felicity: The degree to which descriptions of linguistic phenomena can be stated directly (or indirectly) as linguists would wish to state them.

- Expressiveness: Which class of analyses can be stated at all.

- Computational effectiveness: Whether there exist computational devices for interpreting the grammars expressed in the formalism and, if they do exist, what computational limitations inhere them.

An diesen Maßstäben wird sich wohl jeder Ansatz zur formalen Beschreibung natürlicher Sprachen messen lassen müssen.

Im folgenden soll der Weg von der kontextfreien Grammatik zu einer attributierten Dependenzgrammatik anhand eines durchgängigen Beispiels aufgezeigt werden. Dazu wird zunächst eine kleine kontextfreie Grammatik und ihre Implementation in Form einer definiten Klauselgrammatik vorgestellt und im Anschluß daran gezeigt, wie sich einige Unzulänglichkeiten der Grammatik durch eine Attributierung beheben lassen. Eine Beseitigung der verbleibenden Mängel läßt sich vom Übergang zur Dependenzgrammatik und damit dem Wechsel der linguistischen Theorie erhoffen. Diese wird zunächst informal eingeführt, um im folgenden zwei Formalisierungsansätze, darunter einen eigenen vorstellen zu können. Diese werden den Einsatz der Dependenzgrammatik als einfaches linguistisches Werkzeug in Analogie zu den kontextfreien Grammatiken ermöglichen.

Diese Vorgehensweise stellt eine stark begradigte Schilderung der wirklichen 'Entdeckungsgeschichte' dar. Tatsächlich bin ich von der pragmatischen Definition einer Dependenz-Unifikationsgrammatik [Hellwig 86], wie sie im nächsten Kapitel vorgestellt werden wird, ausgegangen, und habe versucht, eine formale Definition dieses Grammatiktyps zu finden, die einen direkten Vergleich mit den gut verstandenen

kontextfreien Grammatiken ermöglicht. Dies geschah in der Hoffnung, dann Aussagen über diesen Grammatiktyp treffen und seine Stellung im Vergleich zu seinen Konkurrenten aufzeigen zu können. Nach einigen mehr oder weniger geeigneten Ansätzen stellte sich heraus, daß im Gegensatz zum linguistischen Dogma auf formaler Ebene ein enge Verwandtschaft zwischen den beiden besteht, die eine alternative Darstellung der Dependenzgrammatik überflüssig macht und dadurch eine Einordnung in die bekannte Chomsky-Grammatikhierarchie automatisch mit sich bringt.

1.2.1 Kontextfreie Grammatiken und definite Klauselgrammatiken

Eine der einfachsten linguistischen Theorien, die zugleich streng formal gefaßt ist, ist die der kontextfreien Grammatik. Während mit dem Attribut 'kontextfrei' zum einen die Ausdrucksstärke der Grammatik bezeichnet wird, wird in der Informatik vor allem das besondere Regelformat des zugrundegelegten Ersetzungssystems[1] darunter verstanden. Gerade dieses Regelformat ist es, das die kontextfreie Grammatik zur formalen Grundlage der Phrasenstruktur- oder Konstituentengrammatik macht.

Ein auf dieser Theorie aufbauendes linguistisches Werkzeug stellen die definiten Klauselgrammatiken (kurz: DCG[2]) von Pereira dar [Pereira 80]. Durch eine einfache syntaktische Transformation ist es möglich, Regeln einer kontextfreien Grammatik direkt in Hornklauseln zu überführen und so die Sprache einer beliebigen kontextfreien Grammatik mit Hilfe von Resolution und Unifikation sowohl zu generieren als auch zu akzeptieren. Während die Form der Umsetzung hier nicht interessieren soll, möchte ich die Eignung bzw. Unzulänglichkeiten des Formalismus' anhand eines kleinen Beispiels darstellen.

Doch zunächst noch eine Vereinbarung zur Form der kontextfreien Grammatik: Aus Gründen der Praktikabilität möchte ich die Menge der Nichtterminale einteilen in echte Nichtterminale, deren Symbole die Satzkonstituenten bezeichnen, und sogenannte Präterminale, die eine ganze Klasse von Wörtern bezeichnen und damit als Platzhalter für eine Menge von Terminalen fungieren. Entsprechend teile ich die Menge der Regeln ein in Nichtterminalproduktionen, an denen nur Nichtterminale beteiligt sind, und Terminalproduktionen, in denen den Präterminalsymbolen ihre Terminale zugeordnet werden. Dies hat den Zweck, die Grammatik aufzuteilen in einen Phrasenstrukturteil, der die Satzpläne beschreibt, und einen lexikalischen Teil, der das Vokabular bildet.

Beispiel:

Hans schenkt Peter ein Buch.

Ein solcher Satz kann leicht von einer kontextfreien Grammatik

---

[1] Semi-Thue-System

[2] englisch, Abkürzung für definite clause grammar

G = (T, N, P, S) mit

T = {Hans, Peter, Buch, ein, schläft, schenkt}
N = {s, np, vp, subst, art, verb}
P = {s → np vp, np → subst, np → art subst, vp → verb, vp → verb np np,
subst → Hans, subst → Peter, subst → Buch, art → ein,
verb → schenkt, verb → schläft}
S = s

abgeleitet werden. Echte Nichtterminale sind s, np und vp, und Präterminale sind subst, art und verb. Entsprechend teilt man die Produktionen ein.

Das dazugehörige DCG-Programm in Prolog sieht so aus:

```
s --> np, vp.
np --> subst.
np --> art, subst.
vp --> verb.
vp --> verb, np, np.
subst --> ['Hans'].
subst --> ['Peter'].
subst --> ['Buch'].
art --> [ein].
verb --> [schenkt].
verb --> [schläft].
```

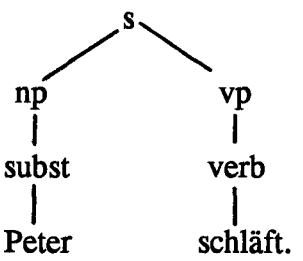
Ein einfacher Satz wie

Peter schläft.

läßt sich durch diese Grammatik wie folgt akzeptieren (ich habe eine trace-ähnliche Notation verwendet, links sind die Regelaufrufe und rechts der verbleibende Eingabesatz aufgeführt):

call s	['Peter', schläft]
call np	['Peter', schläft]
call subst	['Peter', schläft]
accept 'Hans'	['Peter', schläft]
fail	
back to subst	['Peter', schläft]
accept 'Peter'	[schläft]
call vp	[schläft]
call verb	[schläft]
accept schenkt	[schläft]
fail	
back to verb	[schläft]
accept schläft	[]

Der dazugehörige Ableitungsbaum sieht so aus:[3]



Es fällt sofort auf, daß das Verfahren recht 'blind' ist. Das Rücksetzen wird beide Male dadurch ausgelöst, daß der Versuch unternommen wird, ein Terminalsymbol zu akzeptieren, das gar nicht im Satz vorkommt. Je mehr Terminale die Grammatik umfaßt (das heißt je größer das Vokabular der Sprache ist), desto mehr vergebliche Akzeptionsversuche müssen im allgemeinen durchgeführt werden. In der Praxis würde man jedoch die Aufteilung der Regelmenge in Terminal- und Nichtterminalproduktionen dazu ausnutzen, um einen Satz wie

Hans schenkt Peter ein Buch.

durch einen Vorverarbeitungsschritt mit Hilfe der Terminalproduktionen in eine Folge von Präterminalen, in diesem Fall zum Beispiel

[subst, verb, subst, art, subst],

zu überführen und darauf die Analyse durchzuführen. Dabei würden dann die Präterminale die Rolle der Terminale übernehmen, so daß die Akzeption wie folgt verliefe:

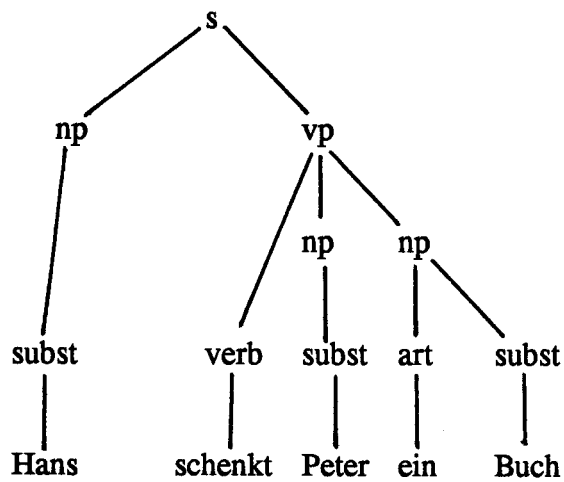
call s	[subst, verb, subst, art, subst]
call np	[subst, verb, subst, art, subst]
accept subst	[verb, subst, art, subst]
call vp	[verb, subst, art, subst]
accept verb	[subst, art, subst]
fail[4]	
back to vp	[verb, subst, art, subst]
accept verb	[subst, art, subst]
call np	[subst, art, subst]
accept subst	[art, subst]
call np	[art, subst]
accept subst	[art, subst]
fail	
back to np	[art, subst]

[3]Wie sich ein solcher Ableitungsbaum gleichzeitig mit der Akzeption durch Prolog erzeugen läßt, kann in beinahe jedem Prolog-Buch nachgelesen werden und soll hier nicht Gegenstand der Betrachtungen sein.

[4]Der Satz ist nicht vollständig akzeptiert!

accept art [subst]
accept subst []

mit dem dazugehörigen Ableitungsbaum



Man beachte, daß in diesem Fall das Rücksetzen andere Ursachen als im ersten Beispiel hat. Eine Vergrößerung des Lexikons führt nicht mehr zu einem Anstieg der vergeblichen Akzeptionsversuche von Terminalen, solange die Menge der Präterminale gleich bleibt. Nichtsdestotrotz hat sich die Auswahlstrategie, die durch die SLD-Resolutuion von Prolog vorgegeben wird, zweimal für die falsche Regel entschieden. Der Grund hierfür ist diesmal in der Angabe der Reihenfolge der Nichtterminalproduktionen, die Prolog bei der Regelauswahl berücksichtigt, zu suchen: Hätte man die Regeln zu 'vp' anders angeordnet, hätte Rücksetzen in diesem Fall vermieden werden können, jedoch hätte man sich dadurch Sackgassen in anderen Fällen eingehandelt.

Doch selbst wenn man diesen Umstand in Kauf nimmt, wird eine solch einfache Grammatik den Anforderungen einer natürlichen Sprache nicht gerecht, läßt sich doch ein falscher Satz wie

* Hans schenkt ein Peter Buch.

genauso akzeptieren. Eine brauchbare Grammatik müßte zum Beispiel verlangen, daß der Artikel 'ein' im gegebenen Fall im Dativ steht. Die meisten natürlichen Sprachen besitzen eine Vielzahl solcher Konsistenzbedingungen, die für den Erhalt eines korrekten Satzes eingehalten werden müssen.

1.2.2 Attributierte Grammatiken

Um solchen Konsistenzbedingungen Rechnung zu tragen, sehen DCG's die Möglichkeit vor, Symbole mit einer Reihe von Attributen zu versehen, die eine genauere Beschreibung der gewünschten Phrase (das heißt des gewünschten Teilbaums) ermöglichen. Mit Hilfe von

Variablen lassen sich dann Regeln formulieren, die das Einhalten der Konsistenzbedingungen automatisch überwachen:

```
s --> np(nominativ), vp.
np(Kasus) --> subst(Kasus).
np(Kasus) --> art(Kasus), subst(Kasus).
vp --> verb.
vp --> verb, np(dativ), np(akkusativ).
subst(nominativ) --> ['Hans'].
subst(dativ) --> ['Peter'].
subst(akkusativ) --> ['Buch'].
art(akkusativ) --> [ein].
verb --> [schenkt].
verb --> [schläft].
```

Natürlich sind viele Terminale wie 'Peter' mehrdeutig in dem Sinne, daß ihr Kasus nicht eindeutig bestimmbar ist. Dies stellt jedoch kein grundsätzliches Problem dar, da daß Lexikon leicht um alternative Einträge (Terminalproduktionen) mit anderen Kasus für dasselbe Terminal ergänzt werden kann, und soll daher an dieser Stelle ignoriert werden.

Der kritische Leser wird vielleicht bemerken, daß die um Attribute ergänzte DCG auch als kontextfreie Grammatik formuliert werden kann, indem man für jede Nichtterminal-Attribut-Kombination ein neues Nichtterminalsymbol einführt und das ursprünglich Symbol dadurch ersetzt.

Beispiel:

$\{ s \rightarrow np_{nominativ}\ vp,$
$np_{nominativ} \rightarrow subst_{nominativ},$
$np_{dativ} \rightarrow subst_{dativ},$
$np_{akkusativ} \rightarrow subst_{akkusativ}.$
$np_{nominativ} \rightarrow art_{nominativ}\ subst_{nominativ},$
$np_{dativ} \rightarrow art_{dativ}\ subst_{dativ},$
$np_{akkusativ} \rightarrow art_{akkusativ}\ subst_{akkusativ},$
$vp \rightarrow verb,$
$vp \rightarrow verb\ np_{dativ}\ np_{akkusativ},$
$subst_{nominativ} \rightarrow$ ['Hans'],
$subst_{dativ} \rightarrow$ ['Peter'],
$subst_{akkusativ} \rightarrow$ ['Buch'],
$art_{akkusativ} \rightarrow$ [ein],
verb → [schenkt],
verb → [schläft] $\}$

Abgesehen davon, daß eine so erzeugte kontextfreie Grammatik umfangreicher ist als die entsprechende DCG, sind die Sprachklassen auch nicht äquivalent:

$L = \{a^n b^n c^n \mid n \geq 0\}$ ist nicht kontextfrei, kann jedoch durch eine DCG akzeptiert werden

Daß L nicht kontextfrei ist, ist allgemein bekannt, zum Beispiel in [Aho 77]. L läßt sich aber durch eine geeignete DCG wie zum Beispiel

```
s --> ab(X), c(X).
ab(0).
ab(s(X)) --> [a], ab(X), [b].
c(0).
c(s(X)) --> [c], c(X).
```

erzeugen. Man beachte, daß es in diesem Fall unendlich viele Nichtterminal-Attribut-Kombinationen gibt.

Definite Klauselgrammatiken sind also in Bezug auf ihre Ausdrucksstärke mächtiger als kontextfreie Grammatiken.

Natürlich führen die in diesem Beispiel aufgeführten Terminalproduktionen wegen der notwendigen Erfassung aller von einem Wort bildbaren Formen zu einer Aufblähung des Lexikons. Dem kann jedoch durch eine Vorverarbeitung des Satzes Abhilfe geschaffen werden, durch die der Satz in eine Form gebracht wird, in der die einzelnen Wörter nur noch aus einer eindeutigen Grundform und ihren morphosyntaktischen Attributen bestehen, wie zum Beispiel [schlafen: 1.Person Singular Indikativ Präsens Aktiv] statt [schläft].[5] Es handelt sich dabei um eine informationserhaltende Lemmatisierung, die im Prinzip kein Problem darstellt und bereits für einige Sprachen realisiert worden ist.

Nicht so leicht läßt sich ein anderes Problem lösen: Der Akzeption des Satzes

* Hans schläft Peter ein Buch.

steht nach der obigen DCG nichts im Wege: Der Satzbau wird in kontextfreien Grammatiken und verwandten Formalismen durch die Nichtterminalproduktionen bestimmt und nicht durch seine Terminale, die einzelnen Wörter. Dabei gehört es zu dem sprachlichen Wissen eines menschlichen Sprechers, daß 'schlafen' keine Ergänzungen außer der des Schläfers verlangt. Zwar ließe sich dieses Problem noch relativ einfach durch die Einführung verschiedener Verbkategorien, von denen jede immer mit einer ihr eigenen Zahl und Art von Ergänzungen zusammen auftritt, beheben, doch wird eine hinreichend allgemeine Auflistung bereits recht umfangreich, ohne daß besondere sprachliche Figuren wie Idiome dadurch abgedeckt werden könnten.

Eine andere Möglichkeit bestünde darin, entsprechende Informationen in den Attributen einer DCG unterzubringen, jedoch erntfernt man sich mit dieser Maßnahmen recht weit von der zugrundegelegten linguistischen Theorie der kontextfreien Grammatik.

[5] Einen enstprechenden Vorverabeitungsschritt für die lateinische Sprache habe ich in meiner Studienarbeit vorgestellt und implementiert.

1.2.3 Der Begriff der Dependenzgrammatik

Die Dependenzgrammatik wird in der Literatur häufig als Antagonist der Konstituenten- oder Phrasenstrukturgrammatik[6] bezeichnet, wobei man sich nicht einig darüber ist, ob die beiden Grammatiktypen als komplementär oder als alternativ anzusehen sind [Tarvainen 81]. Der krasse Gegensatz wird in jedem Fall damit begründet, daß sich die Phrasenstrukturgrammatik die Teil-Ganzes-Beziehung als elementare Relation zur Beschreibung des Verhältnisses der Satzbausteine (Konstituenten) zueinander auserwählt hat, während die Dependenzgrammatik von der Abhängigkeit der einzelnen Wörter untereinander ausgeht.

Der eigentliche Kern der Dependenztheorie ist der Begriff der Valenz. Sie wurde von dem Wiener Philosophen und Sprachtheoretiker Karl Bühler im Jahr 1934 so beschrieben: "Es bestehen in jeder Sprache Wahlverwandtschaften; das Adverb sucht sein Verbum und ähnlich die anderen. Das läßt sich so ausdrücken, daß die Wörter einer bestimmten Wortklasse eine oder mehrere Leerstellen um sich eröffnen, die durch Wörter bestimmter anderer Wortklassen ausgefüllt werden müssen." [Bühler 65]

Es erweist sich darüberhinaus als sinnvoll, die Valenz zu unterscheiden in eine logisch-semantische Valenz und eine syntaktische Valenz. Während man unter der logisch-semantischen Valenz eines Wortes die Eigenschaft seiner Bedeutung versteht, daß es aufgrund seines begrifflichen Inhalts über Leerstellen zu seiner logischen Ergänzung verfügt[7], ist mit der syntaktischen Valenz die Ergänzung um Wörter vorgeschriebener Wortklassen mit genau vorgebenen syntaktischen Merkmalen gemeint. Obwohl diese beiden Ebenen der Valenz eng zusammengehören, entsprechen sie einander doch nicht isomorphisch. So haben die beiden deutschen Verben 'helfen' und 'unterstützen' zwar dieselbe logisch-semantische Valenz, da sie beide einen Helfer und einen, dem geholfen wird, als Ergänzung fordern, auf syntaktischer Ebene muß das Objekt zu 'helfen' im Dativ, das zu 'unterstützen' jedoch im Akkusativ stehen. [Tarvainen 81]

Während die Valenztheorie nur das geregelte Miteinandervorkommen der Wörter im Satz zu erklären versucht, ordnet die Dependenzgrammatik den Wörtern Rollen zu, die sie zu Regentien und Dependentien[8] machen, und etabliert so ein gerichtetes Abhängigkeitsverhältnis, daß dem ganzen Satz eine hierarchische Struktur verleiht. Die Beziehung zwischen einem Regens und seinen Dependentien ist die, daß die Dependentien von ihrem Regens und nur von ihm direkt abhängen. Dabei darf jedes Dependens selbst wieder Regens über andere Wörter sein, so daß die Struktur, in der alle Wörter miteinander verbunden sind, einen Baum darstellt, wenn es im Satz nur ein oberstes Regens gibt.

---

[6] Konstituentengrammatik und Phrasenstrukturgrammatik verwende ich im folgenden synonym.

[7] vergleichbar mit den Prädikaten der Prädikatenlogik erster Ordnung, daher auch die Bezeichnung logisch-semantische Valenz!

[8] Regens, Plural Regentien von lateinisch *regere* = regieren; Dependens, Plural Dependentien von lateinisch *dependere* = abhängen

Obwohl die Dependenztheorie in Europa einige Verbreitung gefunden hat und auch schon zur Grundlage einiger Grammatiklehrbücher gemacht worden ist (z.B. in [Engel 77]), hat sie, besonders im anglistischen Sprachraum, nicht dieselbe Aufmerksamkeit erfahren wie die Konstituentengrammatik, ist insbesondere nicht so oft Gegenstand linguistischer Formalisierungsversuche geworden.

1.2.4 'Dependency Systems' von Gaifman und Hays

Der bekannteste Ansatz zur Formalisierung von Dependenzgrammatiken stammt von Gaifman [Gaifman 65] und Hays [Hays 64]. Gaifman, der die Formalisierung stärker in den Vordergrund stellt, spricht von 'dependency systems' und vergleicht sie mit dem 'phrase-structure systems' Chomsky's. Im folgenden möchte ich kurz die formale Definition der 'dependency systems' wiedergegeben.

Ausgehend von zwei disjunkten Mengen von Symbolen, den Wörtern und den Kategorien, werden drei Mengen von Regeln definiert:

L_{II} Regeln, die zu jeder Kategorie die Liste aller dazugehörigen Wörter angeben. Zu jeder Kategorie gehört mindestens ein Wort, und jedes Wort gehört zu mindestens einer Kategorie, obwohl einzelne Wörter auch mehreren Kategorien angehören können.

L_{I} Regeln, die für jede Kategorie ihre direkten Dependentien zusammen mit deren relativen Positionen untereinander angeben. Eine solche Regel hat die allgemeine Form $X(Y_1, \ldots, Y_i, *, Y_{i+1}, \ldots, Y_n)$, die zum Ausdruck bringt, daß Y_1 bist Y_n alle von X direkt abhängen und in der angegebenen Reihenfolge im Satz auftreten, wobei X die Position des Sterns einnimmt. Eine Regel der Form X(*) bedeutet, daß X ohne Dependentien auftritt.

L_{III} Eine Regel, die die Liste aller Kategorien angibt, von denen der ganze Satz abhängen kann.

Bemerkung: Ich habe die Reihenfolge der Regelmengen absichtlich vertauscht, weil die Definition so intuitiv klarer wird.

Beispiel:

Wörter = {schenkt, schläft, Hans, Peter, Buch, ein}
Kategorien = {$verb_1$, $verb_2$, $subst_1$, $subst_2$, art}

L_{II} = {$verb_1$: (schläft),
$verb_2$: (schenkt),
$subst_1$: (Hans, Peter),
$subst_2$: (Buch),

art : (ein)}

L_I = {$verb_1(subst_1, *)$,
$verb_2(subst_1, *, subst_1, subst_2)$,
$subst_1(*)$,
$subst_2(art, *)$,
$art(*)$}

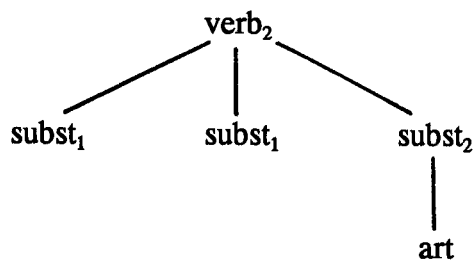
L_{III} = {$verb_1$, $verb_2$}

Im weiteren Verlauf definiert Gaifman eine zweistellige Abhängigkeitsrelation d unter den Kategorien der Wörter eines Satzes und gibt Bedingungen an, denen sie genügen muß, um die Dependenzanalyse des Satzes zu repräsentieren.

Die Relation kann als Baum dargestellt werden, der für einen Satz wie

Hans schenkt Peter ein Buch.

im angegebenen Beispiel die Form



hat.

Ich möchte an dieser Stelle den Ausführungen Gaifman's, der im folgenden eine schwache Äquivalenz zwischen den 'dependency systems' und den 'phrase-structure systems' beweist, nicht weiter folgen, sondern stattdessen andeuten, wie sich dieser Formalismus in Bekanntes einordnen läßt.

Die Interpretation der Menge der Wörter ist klar, hierbei handelt es sich um die Terminalsymbole. Dagegen sind die Kategorien Sammelbezeichnungen für Mengen von Wörtern gleicher oder ähnlicher Valenz und wären von daher am ehesten mit den Präterminalen zu vergleichen. Wie sich jedoch zeigen wird, tragen sie eine doppelte Funktion und werden wohl deswegen auch von Hays nicht Nichtterminale, sondern Hilfssymbole (auxiliaries) genannt.

Durch diesen Vergleich lassen sich zumindest die Terminalproduktionen, die das Vokabular bilden, in den Regeln von L_{II} wiedererkennen.

Dagegen handelt es sich bei den Regeln aus L_I offensichtlich um solche, die mögliche

Abhängigkeitsrelationen zum Ausdruck bringen: Einer Wortkategorie werden hier ihre direkt abhängigen als Valenzen zugeordnet. An dieser Stelle wird die doppelte Funktion der Kategoriesymbole deutlich: Sie treten einmal im Regelkopf als echtes Nichtterminal auf, um dann jedoch im Regelrumpf, vertreten durch den Stern, wieder die Rolle des Präterminals zu spielen.

Die Menge L_{III} schließlich übernimmt die Rolle solcher Produktionen einer kontextfreien Grammatik, die das Startsymbol als Regelkopf besitzen.

1.2.5 Die Dependenzgrammatik als kontextfreie Grammatik

Zwar werden Dependenzgrammatiken oft als Antagonisten der Phrasenstrukturgrammatiken bezeichnet, doch scheint sich dieser Gegensatz in erster Linie auf die die Zusammenhänge der Satzbestandteile beschreibende Relation zu beschränken. Wie sich schon bei Gaifman und Hays andeutete, läßt sich bei genauerer Betrachtung eine Verwandtschaft mit den kontextfreien Grammatiken nicht leugnen. Ich möchte daher auf eine unabhängige formale Definition der Dependenzgrammatik und anschließende Gegenüberstellung verzichten und stattdessen die Dependenzgrammatik als Spezialfall der kontextfreien Grammatik darstellen.

Definition 1.21 (Dependenzgrammatik[9])
Eine Dependenzgrammatik ist eine kontextfreie Grammatik G = (T, N, P, S) mit folgenden Einschränkungen:

- Es existiert eine bijektive Funktion f, die die Menge der Terminale T in die Menge der Nichtterminale N\{S} vollständig abbildet.

- Alle Produktionen aus P sind entweder von der Form
  $$\eta \rightarrow \eta_1 \ldots f^{-1}(\eta) \ldots \eta_n ,$$
  oder von der Form
  $$s \rightarrow \eta_1 \ldots \eta_n$$
  mit $\eta, \eta_1, \ldots, \eta_n \in N\backslash\{S\}$ und $s=S$.

■

In Worten: Zu jedem Terminalsymbol gehört genau ein Nichtterminalsymbol, und einziges zusätzliches Nichterminalsymbol ist das Startsymbol. Alle Regeln außer den Startregeln enthalten im Regelrumpf genau ein Terminal, und zwar jenes, zu welchem das Nichtterminal des Regelkopfes gehört. Die Startproduktionen dürfen kein Terminal enthalten.

Bemerkung: In gewisser Weise spiegelt sich die Definition der Regelmengen von Gaifman in dieser Definition wider. Ich weiche jedoch darin inhaltlich deutlich von seiner Auffassung einer Dependenzgrammatik ab, daß in meiner Definition keine Kategorien zugelassen sind,

[9] Absofort ist mit Dependenzgrammatik immer meine persönliche Auffassung dieses Grammatiktyps gemeint - alle anderen Auffassungen sollen davon unberührt bleiben.

die als Sammelbezeichnung für mehrere ähnliche Wörter dienen könnten.[10] Nach meiner Auffassung muß jedes Wort der Sprache die Funktion seiner Kategorie, nämlich die Angabe seiner Dependentien, mittragen. Dies kommt in der eineindeutigen Zuordnung von Terminalen (Wörtern) und Nichtterminalen (Kategorien) zum Ausdruck.

Beispiel:

Eine kontextfreie Grammatik G = (T, N, P, S) mit

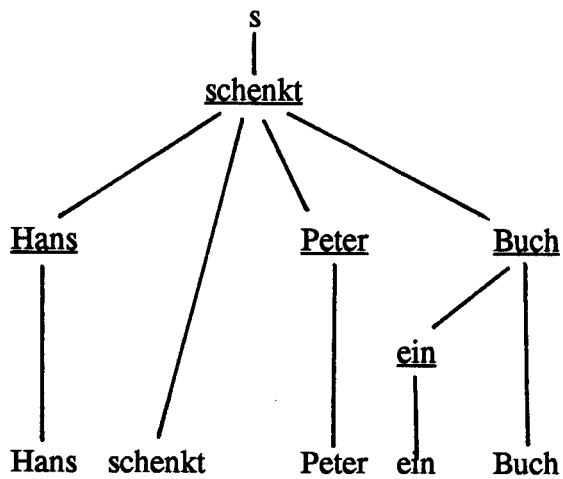
T = {schenkt, schläft, Hans, Peter, Buch, ein},
N = {schenkt, schläft, Hans, Peter, Buch, ein, s},
P = { s -> schenkt,
s -> schläft,
schenkt -> Hans schenkt Peter Buch,
schläft -> Hans schläft,
Hans -> Hans,
Peter -> Peter,
Buch -> ein Buch,
ein -> ein},
S = s,

f werde durch folgende Tabelle definiert:

T	schenkt	schläft	Hans	Peter	Buch	ein
N	schenkt	schläft	Hans	Peter	Buch	ein

ist eine Dependenzgrammatik. Der Satz 'Hans schenkt Peter ein Buch' hat folgenden Ableitungsbaum:

[10]Tatsächlich halte ich das auch nicht für sinnvoll, da es zum Beispiel im Deutschen nur wenige Wörter gibt, die wirklich vollständig synonym, das heißt in diesem Fall mit identischer syntaktischer und semantischer Valenz ausgestattet sind.



■

In der Praxis lassen sich die Regelmäßigkeiten einer in Form einer kontextfreien Grammatik kodierten Dependenzgrammatik ausnutzen, um effiziente Parsing-Algorithmen zu entwerfen. Die Tatsache, daß bei Kenntnis der Zuordnungsvorschrift f bereits vor der Ausführung einer Regel $\eta \rightarrow \eta_1 \ldots f^1(\eta) \ldots \eta_n$ deren Eignung zur Akzeption überprüft werden kann, kann zur Verbesserung der select-Strategie zur Auswahl von Regeln herangezogen werden. Wie dies möglich ist, möchte ich im nächsten Kapitel zeigen.

Dem Leser wird dennoch die mangelnde praktische Eignung einer so definierten Dependenzgrammatik nicht entgangen sein: Durch die eineindeutige Zuordnung von Nichtterminalen zu Terminalen, den Wörtern der Sprache, wächst die Regelmenge polynomial mit der Größe des Wortschatzes.

Die Komplexität läßt sich wie folgt abschätzen: Die Zahl der Valenzen eines Wortes ist begrenzt, sagen wir mit c. Sei n die Größe des Wortschatzes, dann gibt es zu jedem Wort höchstens n^c verschiedene Regeln, also insgesamt maximal $n \cdot n^c = n^{c+1}$ Einträge.

Dieses Problem stellt sich bei der Formulierung einer Phrasenstrukturgrammatik als kontextfreie Grammatik nicht, da die Präterminale der Grammatik als Platzhalter für Mengen von Terminalen, den verschiedenen Wörtern, betrachtet werden können, und so die Grammatik nur linear mit der Größe des Wortschatzes wächst, ein konstanter Umfang der Nichtterminalproduktionen vorausgesetzt.

Erfreulicherweise läßt sich die Komplexität einer solchen Dependenzgrammatik durch eine geringfügige Modifikation auf eine lineare Größenordnung reduzieren, wie sich im nächsten Abschnitt zeigen wird.

1.2.6 Dependenz-Unifikationsgrammatik

In einer Dependenz-Unifikationsgrammatik, kurz DUG genannt[11], wird auf die eindeutige Benennung von Terminalen und Nichtterminalen durch Konstanten zugunsten einer partiellen Spezifikation in Form von Termen verzichtet. Eine Entscheidung über Gleichheit von Termen, wie sie bei der Akzeption von Terminalen und dem Vergleich von Nichtterminalen zur Regelauswahl benötigt wird, wird von einer geeigneten Unifikation übernommen. Dadurch können viele Alternativen in einer Regel zusammengefaßt werden, so daß die Anzahl der Regeln pro Symbol (Wort der Sprache) im allgemeinen durch eine Konstante begrenzt sein wird.

Genau wie beim Übergang von kontextfreien Grammatiken zu DCG's wird beim Übergang von der Dependenzgrammatik zur DUG das Gebiet der kontextfreien Sprachen verlassen. Ich werde daher auch nicht versuchen, eine DUG formal zu fassen, sondern lediglich eine Möglichkeit der Implementation in Form einer DCG anhand eines Beispiels vorführen.

Als Konstruktor für den zur partiellen Spezifikation benötigten Term verwende ich v/3[12]. Das erste Argument bildet das Wort selbst, das zweite stellt die Wortart dar und das dritte ist hier nur beispielhaft für zusätzliche grammatische Information angeführt.

Beispiel:

```
s -->
     v(_, verb, 'Prädikat').

v(schenkt, verb, Rolle) -->
     v(_, subst, 'Subjekt'),
     [v(schenkt, verb, Rolle)],
     v(_, subst, 'Empfänger'),
     v(_, subst, 'Objekt').

v(schläft, verb, Rolle) -->
     v(_, subst, 'Subjekt'),
     [v(schläft, verb, Rolle)].

v('Hans', subst, Rolle) -->
     [v('Hans', subst, Rolle)].

v('Peter', subst, Rolle) -->
     [v('Peter', subst, Rolle)].

v('Buch', subst, Rolle) -->
     v(_, art, 'Artikel'),
     [v('Hans', subst, Rolle)].
```

[11]Die Bezeichnung Dependenz-Unifikationsgrammatik sowie die Abkürzung DUG habe ich von Hellwig aus [Hellwig 86] übernommen.

[12]wegen lateinisch *verbum* (Wort)

```
v(ein, art, Rolle) -->
     [v(ein, art, Rolle)].
```

■

Zwar läßt sich eine Dependenzgrammatik auf diese Art implementieren, es werden jedoch die praktischen Vorteile, die sich aus dieser Form der Grammatik ergeben, insbesondere eine Effizienzsteigerung, nicht ausgenutzt. Einen etwas abgewandelten Formalismus, der diese Möglichkeiten unterstützt, werde ich im nächsten Kapitel vorstellen.

Man beachte, daß der Übergang zu Termen gewissermaßen eine Attributierung der Regeln darstellt. In diesem Fall wurden die Attribute jedoch nicht dazu verwendet, morpho-syntaktische Eigenschaften abzudecken und damit die Einhaltung von Konsistenzbedingungen zu erzwingen, obwohl diese Funktion dem Grammatikentwerfer auch hier zur Verfügung steht.

Zum Abschluß möchte ich noch anhand eines Beispiels vorführen, wie die Akzeption eines Satzes mit der angegebenen Grammatik vonstatten geht.

Beispiel:

Der Satz

Peter schläft.

liegt nach einer Vorverarbeitung in folgender Form vor:

[v('Peter', subst, _), v('schläft', verb, _)]

Die Akzeption verläuft dann wie folgt (ich verwende wieder die trace-ähnliche Notation, links sind die Regelaufrufe und rechts der verbleibende Eingabesatz aufgeführt):

call s	[v('Peter', subst, _), v('schläft', verb, _)]
call v(schenkt, verb, 'Prädikat')	[v('Peter', subst, _), v('schläft', verb, _)]
call v('Hans', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
accept v('Hans', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
fail	
back to v('Peter', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
accept v('Peter', subst, 'Subjekt')	[v('schläft', verb, _)]
accept v(schenkt, verb, 'Prädikat')	[v('schläft', verb, _)]
fail	
back to v('Buch', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
accept v('Buch', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
fail	
back to v(schläft, verb, 'Prädikat')	[v('Peter', subst, _), v('schläft', verb, _)]
call v('Hans', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
accept v('Hans', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
fail	
back to v('Peter', subst, 'Subjekt')	[v('Peter', subst, _), v('schläft', verb, _)]
accept v('Peter', subst, 'Subjekt')	[v('schläft', verb, _)]
accept v(schläft, verb, 'Prädikat')	[]

■

Wie auch dieses Beispiel deutlich zeigt, verläuft der Akzeptionsvorgang wenig zielgerichtet. Insbesondere führt der Aufruf der Regel zu 'schenkt' bereits im zweiten Schritt in eine tiefe Sackgasse, die vermieden werden können hätte, wenn man die Tatsache berücksichtigt hätte, daß der Satz 'schenkt' gar nicht enthält, diese Regel also zum Scheitern verurteilt ist.

Man halte sich jedoch den prinzipiellen Unterschied zur Grammatik aus Abschnitt 1.2.1 vor Augen, der darin besteht, daß die Symbole, die Wörter des Satzes, an der Regelauswahl selbst unmittelbar beteiligt sind. Daß sich das Akzeptionsverhalten nicht wesentlich verbessert hat, liegt weniger an der Auswahlstrategie, sondern vielmehr an der Reihenfolge zwischen Regelaufruf und Akzeption, wie sich am Beispiel zu Abschnitt 2.2.1 zeigen wird.

2 Integration in bestehende Konzepte des logischen Programmierens

In den theoretischen Grundlagen habe ich, zunächst völlig unabhängig voneinander, zwei Formalismen vorgestellt, die zur Verarbeitung natürlicher Sprache beitragen können. Das Einsatzgebiet insbesondere der ordnungssortierten feature-Logik ist jedoch nicht auf natürlichsprachliche Anwendungen begrenzt: Leser, die sich mit den Verwendungsmöglichkeiten formaler Logik in verschiedenen Bereichen der Informatik befaßt haben, werden vielleicht auch andere Einsatzbereiche nennen können.

Um die Eignung der Formalismen in der Praxis überprüfen zu können, soll es möglich sein, unter ihrer Verwendung Programme zu schreiben. Wegen der engen Verwandtschaft mit der Logik bietet sich die Integration dieser Konzepte in das Prinzip des logischen Progammierens an.

Die Erkenntnis, daß sich die Prädikatenlogik erster Ordnung auch zum Formulieren von Programmen eignet, ist nicht neu und geht wohl auf [Kowalski 74] zurück. Einer der bekanntesten Vertreter des logischen Programmierens ist die Programmiersprache Prolog, deren Programme aus Horn-Klauseln, bestimmten Formeln der Prädikatenlogik, bestehen. Nach anfänglicher Begeisterung über diese sogenannte Programmiersprache der fünften Generation, die sich in erster Linie auf die Verwirklichung des deklarativen Programmierstils gründete, wurden doch bald Mängel augenscheinlich, die das Erstellen umfangreicher Programme erschwerten.

Als nachteilig erwies sich zum einen die vollständige Typfreiheit der Sprache, die das Auffinden bestimmter Fehler wie das Vertauschen von Argumenten dem Programmierer überlies, zum anderen das Fehlen des Modulkonzepts, das eine saubere Strukturierung des Programms sowie die Verwirklichung des Prinzips des *information hiding* erschwerte. Während andere Sprachen mit ähnlichen Mängeln entweder erweitert (z.B. Lisp zu Common-Lisp) oder durch neue abgelöst wurden (z.B. Pascal durch Modula), fehlt bei Prolog bislang ein Standard, der diesen Mängeln Abhilfe schafft.

Das Modulkonzept als eine der zentralen Forderungen der Software-Entwicklung wurde an der Universität Karlsruhe bereits bei der Entwicklung des Prolog-Compilers *KA-Prolog*

berücksichtig. [Lindenberg et al. 87]

In einer zweiten Phase wurde das KA-Prolog-System um das Prinzip der Ordnungssortiertheit ergänzt und damit zu *Lopster* ausgebaut [Weinstein 89]. Ein Vorteil des hierarchischen Typkonzepts, wie es durch die Ordnungssortiertheit verwirklicht wird, ist die Möglichkeit der Kombination typfreier mit typisierten Programmteilen, die durch die Typsubsumption ermöglicht wird, und die die Vorteile typfreier mit denen typisierter Sprachen verbindet. Des weiteren verspricht man sich von der Einschränkung des Suchraums durch Sortierung das Einsparen von Deduktionsschritten bei der Lösungsfindung.

Mit meiner Arbeit möchte ich Prolog um ein zusätzliches Konzept erweitern: das der features. Feature-Typen können vom programmiertechnischen Standpunkt aus als record-ähnliche Datenstrukturen aufgefaßt werden, denen zusätzlich die Möglichkeit einer Vererbung gegeben ist. Nachdem ich Syntax, Semantik und Vererbung der feature-Typen bereits im vorangegangenen Kapitel geklärt habe, ist hier die Integration in das Lopster-System Gegenstand der Betrachtungen.

Der resultierende Prolog-Dialekt erhält den Arbeitsnamen *osf-Prolog*. Wie in dieser Bezeichnung, so soll auch im folgenden die Vorsilbe *osf-* für *ordnungssortierte feature-* stehen (vgl. Begriffsklärung im Anhang).

In einem zweiten Abschnitt wird die Abbildung des Formalismus der Dependenz-Unifikationsgrammatik auf Hornklauseln in Analogie zu den definiten Klauselgrammatiken von Pereira und Warren vorgestellt [Pereira 80]. Dadurch wird es möglich, solche Grammatiken direkt in Prolog zu implementieren.

2.1 Einbettung von osf-Prolog in Lopster

Um Programme in osf-Prolog nicht nur schreiben, sondern auch ablaufen lassen zu können, soll eine Methode vorgestellt werden, mit deren Hilfe sie auf Lopster-Programme abgebildet werden können.

Im wesentlichen sind dazu drei Probleme zu lösen:

- die Abbildung der osf-Signatur auf eine os-Signatur
- die Darstellung von feature-Termen durch Konstruktor-Terme
- die Unifikation von feature-Termen

Doch zunächst soll eine Erweiterung der Syntax definiert werden, die eine Integration des feature-Konzepts in Lopster gestattet.

2.1.1 Syntax von osf-Prolog

Ein osf-Prolog-Programm besteht, genau wie ein Lopster-Programm, aus einem oder

mehreren Modulen. Im Gegensatz zu Lopster-Programmen werden die Sortendeklarationen jedoch nicht in einem seperaten Sortenmodul hinterlegt, sondern sind Bestandteil des Hauptmoduls. Hier und nur hier können sämtliche Konstruktor- und feature-Sorten bekanntgegeben werden. Das Hauptmodul enthält dazu einige neue Schlüsselwörter, die den Beginn des Sortendeklarationsteils, des feature-Deklarationsteils und des Konstruktor-Deklarationsteils einleiten.

2.1.1.1 Deklaration der Konstruktor-Sorten S_c

Die Konstruktor-Sorten einer Signatur werden in einem Abschnitt des Vereinbarungsteils des Hauptmoduls bekannt gegeben, der durch das Schlüsselwort `sorts` eingeleitet wird. Die Deklaration erfolgt, genau wie in Lopster, innerhalb einer Untersortenrelation, ist also impliziter Bestandteil der Sortenverbandspezifikation.

Beispiel:

```
sorts.
kasus < term.
numerus < term.
genus < term.
komparation < term.
genusVerbi < term.
passivisch < genusVerbi.
```

Das Ende der Konstruktor-Sortendeklaration wird durch eins der Schlüsselwörter `features`, `constructors` und `body` bekanntgegeben.

2.1.1.2 Deklaration der feature-Sorten und -Symbole S_f und Σ_f.

In dem durch das Schlüsselwort `features` eingeleiteten Teil werden in einem Zug feature-Sorten, feature-Symbole und feature-Untersortenrelationen deklariert, indem den feature-Sorten ihre feature-Symbole und Obersorten zugeordnet werden. Eine feature-Sorte s mit den features l_1: s → s_1, ..., l_n: s → s_n wird durch

```
type s & [l1=>s1, ..., ln=>sn].
```

vereinbart. Soll s gleichzeitig Untersorte von s' sein, lautet die Deklaration

```
type s & [l1=>s1, ..., ln=>sn] < s'.
```

Dabei ist zu beachten, daß zwar s' auch Konstruktor-Sorte sein darf, s jedoch in jedem Fall feature-Sorte ist. s darf nur einmal deklariert werden, insbesondere also zuvor nicht schon bei den Konstruktor-Sorten deklariert worden sein.

Bemerkung: Auf diese Weise wird sichergestellt, daß jede Untersorte einer feature-Sorte wieder feature-Sorte ist.

Beispiel:

```
features.
type nomen & [kas=>kasus,num=>numerus,gen=>genus]<wortart.
type substantiv < nomen.
type adjektiv & [kmp=>komparation] < nomen.
type verb & [nmv=>numerus, gnv=>genusVerbi]
type deponens & [gnv=>passivisch] < verb.
```

Bemerkung: Die features einer Sorte werden auf alle ihre Untersorten vererbt. Im Beispiel ist `kas` daher auch feature von `substantiv` und `adjektiv`.

Das Ende dieses Deklarationsabschnitts wird durch eins der Schlüsselwörter `constructors` und `body` angegeben.

2.1.1.3 Deklaration der Konstruktor-Symbole Σ_c

Die Deklaration der Konstruktor-Symbole beginnt mit dem Schlüsselwort `constructors`. Ein Symbol f: $s_1 \ldots s_n \rightarrow s \in \Sigma_c$ wird durch eine Klausel

```
sort f(s1, ..., sn) : s.
```

bekannt gemacht.

Beispiel:

```
constructors.
sort nominativ : kasus.
sort genitiv : kasus.
sort v(term, kategorie, wortart, rolle).
```

Bemerkung: Prädikate werden zusammen mit den Konstruktoren deklariert. Die Ergebnissorte von Funktionen (und Konstanten) muß immer eine Konstruktor-Sorte sein.

Das Ende der Vereinbarung wird durch `body` angemerkt.

2.1.1.4 Spezifikation des Sortenverbands (S, $\leq$)

Der Sortenverband wird zusammen mit der Deklaration der Konstruktor- und feature-Sorten spezifiziert.

2.1.1.5 Syntax der Terme $T_\Sigma(V)$

Da sich Lopster-Terme bis auf die fehlenden features nicht von den osf-Termen unterscheiden, muß ich hier nur auf die Darstellung der feature-Terme eingehen.

Ein feature-Term der Form $\{X:s|l_1(X)\doteq t_1, \ldots, l_n(X)\doteq t_n\}$ wird in osf-Prolog durch

```
X:s & [l1=>t1, ..., ln=>tn]
```

dargestellt. '&' ist als zweistelliger Infix-Operator deklariert, der die Verbindung der feature-Variablen mit ihren features ausdrücken soll, und '=>', ebenfalls zweistelliger Infix-Operator, verbindet das feature label mit dem feature value. Auf die Wiederholung der feature-Variablen als Argument des labels wird der besseren Lesbarkeit wegen verzichtet.

2.1.1.6 Zusammenfassung der Syntax

Ich möchte die Syntax nicht vollständig formal definieren, da sie insgesamt doch recht umfangreich ist. An dieser Stelle soll nur noch einmal übersichtlich dargestellt werden, wie ein osf-Prolog-Programm aufgebaut ist.

Ein osf-Prolog-Hauptmodul hat demnach die allgemeine Form:

```
module <id>.
export <id-list>.
...
from <id> import <id-list>.
...
private <id-list>.
...
sorts.
<sort> < <sort>.
...
features.
type <sort> & [<id> => <sort>,...,<id> => <sort>] < <sort>.
...
constructors.
sort <id> : <sort>.
...
body.
<clause>.
<clause>.
...
end_module.
```

2.1.2 Einbettung von osf-Prolog in Lopster

Die Semantik der neu eingeführten Programmelemente wird durch die Beschreibung ihrer Korrelate in Lopster definiert. Die Definition dient gleichzeitig als Spezifikation für einen noch zu beschreibenden Übersetzer, der diese Zuordnung automatisch vornimmt.

2.1.2.1 Sortendeklarationen

Die Sortendeklarationen des osf-Prolog-Hauptmoduls können direkt in das Lopster-Sortenmodul kopiert werden. Sie werden gegebenenfalls durch feature-Sortendeklarationen aus dem feature-Vereinbarungsteil ergänzt.

2.1.2.2 Abbildung der feature-Deklarationen

Kombinierte feature-Sorten und -Symboldeklarationen der Form

```
type nomen & [kas=>kasus,num=>numerus,gen=>genus]<wortart.
type substantiv < nomen.
type adjektiv & [kmp=>komparation] < nomen.
type verb & [nmv=>numerus, gnv=>genusVerbi]
type deponens & [gnv=>passivisch] < verb.
```

werden aufgeteilt in Untersortendeklarationen, die den feature-Sortenverband spezifizieren, und in feature-Deklarationen. Entsprechend ihrer Bedeutung werden features als einstellige Funktionen dargestellt, die eine Argumentsorte auf eine Ergebnissorte abbilden. Dabei entspricht das feature label dem Funktor, der Typ, für den das feature deklariert wird, der Argumentsorte, und das feature range der Ergebnissorte.

Die Sortenverbandspezifikation wird mit dem obigen Beispiel um folgende Einträge ergänzt:

```
nomen < wortart.
substantiv < nomen.
adjektiv < nomen.
deponens < verb.
```

Die dazugehörigen Funktionsdeklarationen sehen in Lopster so aus:

```
sort kas(nomen) : kasus.
sort num(nomen) : numerus.
sort gen(nomen) : genus.
sort kmp(adjektiv) : komparation.
sort nmv(verb) : numerus.
sort gnv(verb) : genusVerbi.
sort gnv(deponens) : passivisch.
```

Sehr angenehm fällt auf, daß man sich nicht weiter um die Vererbung von features auf Untertypen kümmern muß: Zwar ist das feature `kas` nur für den Typ `nomen` deklariert, da aber `substantiv` und `adjektiv` als Untertypen zu `nomen` angegeben sind, läßt sich das feature (die Funktion) auch auf sie anwenden, ohne daß die Wohlsortiertheit verletzt würde, und die Ergebnissorte ist, wie von der Vererbung erwartet, dieselbe. Das gilt jedoch nicht für das feature `gnv` des Typs `deponens`, denn dort wurde die Ergebnissorte absichtlich auf `passivisch` eingeschränkt.

2.1.2.3 Umsetzung der Konstruktor-Deklarationen

Die Konstruktor-Deklarationen entsprechen genau denen in Lopster und können daher direkt übernommen werden. Es bleibt wiederholt zu bemerken, daß auch Prädikatsdeklarationen an dieser Stelle vorgenommen werden.

2.1.2.4 Repräsentation von feature-Termen

Ein feature-Term $\{X{:}s \mid l_1(X) \doteq t_1, \ldots, l_n(X) \doteq t_n\}$ hat in osf-Prolog die Form

$$X{:}s \;\&\; [l_1 => t_1, \ldots, l_n => t_n],$$

wobei $t_1, \ldots, t_n$ Terme sind, also wiederum feature-Terme seien können.

Damit die sortenrechte Einbettung von feature-Termen in Prädikate und Funktionen von Lopster ermöglicht werden kann, muß ein Konstruktor-Symbol mit dem Funktor '&' zur Repräsentation eines feature-Terms mittels einer (überladenen) Funktionsdeklaration

sort '&'(s, list) : s

für alle feature-Sorten s deklariert werden. Das erste Argument eines mit diesem Symbol erzeugten Konstruktor-Terms wird durch die feature-Variable des repräsentierten feature-Terms besetzt, das zweite durch eine offene Liste, die die Menge der in dem Term angegebenen features enthält.

Leider konnte ich in Prolog keine bessere Möglichkeit zur Repräsentation der features als die sehr schwerfällige der offenen Liste finden. Sie hat die Aufgabe, einerseits alle angesammelten feature-Informationen zu führen und andererseits genügend Platz für den Fall vorzusehen, daß durch Unifikation weitere Information in Form von features dazukommt. Durch ihre Verwendung ist es jedoch nicht möglich, die Überprüfung auf Wohlsortiertheit Lopster zu überlassen, da an den Typ der Listenelemente keine Anforderungen gestellt werden können.[1] Weil darüberhinaus eine Deklaration der Argumentsorten von '=>' wegen der vielfachen Überladung nicht für eine Überprüfung auf sortenrechte Einbettung der Unterterme durch Lopster ausnutzbar ist, entfällt eine entsprechende Funktionsdeklaration ebenfalls.

Die Aufgabe einer feature-Variable X eines feature-Terms $X{:}s \;\&\; [l_1 => t_1, \ldots, l_n => t_n]$ ist zweigeteilt: Zum einen trägt sie die Sorteninformation des Terms, zum anderen benennt sie sie ihn eindeutig. Mit der Sorteninformation ist sie Kriterium sowohl für die sortenrechte Einbettung des Terms in andere als auch für die Wohlsortiertheit des Terms selbst, da alle seine Unterterme t_i Untersorte der Sorte von $l_i(X)$ sein müssen. Wegen der eindeutigen Benennung ist sie das Koreferenzsymbol, das die Gleichheit aller koreferenzierten feature-Terme innerhalb seines Geltungsbereichs erzwingen soll. Da X als feature-Variable

[1] Lopster erlaubt im Gegensatz zu z.B. PROTOS-L keine polymorphen Sortendeklarationen. Interessierte Leser seien auf [Beierle 89] verwiesen.

Platzhalter für eine Menge ist, sollte es von seiner Bedeutung her keine Werte annehmen, also an keinen Term gebunden werden können. Technisch kann dieser Umstand ausgenutzt werden, um eine Gleichheit aller koreferenzierten Terme herzustellen, indem diese an ihren Auftreten durch ihre feature-Variable ersetzt werden und dafür Sorge getragen wird, daß diese Variable an den normalisierten feature-Term gebunden ist. Damit das möglich ist, darf X nicht mehr in dem Term vorkommen (occur check). Dies ist jedoch kein Problem, da die feature-Variable nun der Koreferenzfunktion enthoben und nur noch Träger der Sorteninformation ist, also durch jede beliebige neue Variable ersetzt werden kann.

Beispiel:

$X{:}s_1$ & $[l_1{=>}Y{:}s_1$ & $[k_1{=>}t_1]$, $l_2{=>}Y{:}s_2$ & $[k_2{=>}t_2]$, $l_3{=>}Y]$

wird demnach als

X

dargestellt mit

$X = V_1{:}s_1$ & $[l_1{=>}Y$, $l_2{=>}Y$, $l_3{=>}Y|R_1]$ und
$Y = V_2{:}s_3$ & $[k_1{=>}t_1$, $k_2{=>}t_2|R_2]$,

wenn s_3 größte gemeinsame Untersorte von s_1 und s_2 ist.

2.1.2.5 Unifikation in osf-Prolog

Nachdem nun die Repräsentation der osf-Terme in Lopster geklärt ist, kann das Problem der Unifikation angegangen werden.

Für die osf-Unifikation wird zunächst ein neuer Unifikationsoperator eingeführt: '=+=' ergänzt das übliche '=', das hier weiterhin für die ordnungssortierte Unifikation in Lopster steht. Durch das Pluszeichen soll angedeutet werden, daß die osf-Unifikation Informationen durch Einschränkung des Typs und Ergänzung von zusätzlichen features ansammelt.

Eine Modifikation der Unifikation ist notwendig, weil die Anwendung der gewöhnlichen Unifikation auf feature-Terme zu falschen Ergebnissen führen kann. Unter der Signatur $(S, \leq, \Sigma)$ mit

$(S, \leq) = \{s_3 \leq s_1, s_3 \leq s_2, s_5 \leq s_4\}$ und
$\Sigma = \{f{:}\ s_1 \rightarrow s_4,\ f{:}\ s_2 \rightarrow s_4,\ f{:}\ s_3 \rightarrow s_5,\ a{:} \rightarrow s_4\}$

würden

$\{x{:}s_1 \mid f(x) \doteq a\}$ und
$\{y{:}s_2 \mid f(y) \doteq z{:}s_4\}$

alias

$V_1:s_1$ & [f=>a|R_1] und
$V_2:s_2$ & [f=>Z:s_4|R_2]

zwar von Lopster zu

$V_3:s_3$ & [f=>a|R_3]

unifiziert, weder die Antwortsubstitution $\sigma(Z) = a$ noch der Term selbst sind jedoch sortenrecht, da Z und a dann von der Sorte s_5 sein müßten. Umgekehrt sieht man leicht ein, daß feature-Terme, deren features sich nur in ihrer Reihenfolge unterscheiden, sich durch Lopster nicht unifizieren lassen, obwohl einer osf-Unifikation nichts im Wege steht.

Gefahrlos kann die gewöhnliche Unifikation jedoch zu Unifizierung von reinen Konstruktor-Termen sowie zur Instanziierung von Variablen auch mit feature-Termen herangezogen werden. Diese Tatsache nutze ich aus, um die automatisch ausgeführte gewöhnliche Unifikation von Literalen der Regelrümpfe mit denen der Regelköpfe beibehalten zu können, indem ich alle feature-Terme in den Regelköpfen durch Variablen ersetzte und die Unifikation mit den ersetzten Terme in den Rumpf verlagere. Der Fall, daß ein feature-Term mit einem Konstruktor-Term herkömmlich unifizierbar ist, wird zwar durch den Sortenverband und die Verwendung des feature-Konstruktors '&'/2 bereits weitgehend ausgeschlossen[2], kann aber durch den Übersetzer vollständig verhindert werden.

Term	Term	Op	Unifikation
Variable	Variable	=	gelingt
Variable	Konstruktor-Term	=	gelingt
Variable	feature-Term	=	gelingt
Konstruktor-Term	Konstruktor-Term	=	möglich
Konstruktor-Term	feature-Term	=	scheitert
feature-Term	feature-Term	=+=	möglich

Neu betrachtet werden muß demnach nur der Fall, daß zwei feature-Terme miteinander unifiziert werden sollen. Eine Isolierung dieser Fälle auf der Ebene der Klauseln bringt den Vorteil, daß der modifizierte Unifikationsalgorithmus nur angewandt wird, wo nötig, und wird durch die explizite Instanziierung der Variablen automatisch erreicht.

Um zwei Terme zu unifizieren, muß eine Sequenz von geeigneten Substitutionen und Konkretisierungen gefunden und angewandt werden, die diese gleich machen. Eine systematische Vorgehensweise wird durch folgenden Algorithmus beschrieben:

[2]Wenn es der Benutzer jedoch darauf anlegt, kann er durch Deklaration von '&'(term, list) : term eine solche Unifikation herbeiführen.

unifiziere(s, t)
betrachte die äußeren Terme von s und t
handelt es sich um zwei feature-Terme, dann
seien X und Y ihre feature-Variablen
(1:) wenn X = Y gelingt
/* s und t sind auf ggU(s, t) eingeschränkt */
(2:) *unifiziere(u, v)* für alle l mit (l=>u) in *features(s)* und (l=>v) in *features(t)*
(3:) *ergänze(s, l=>v)* für alle (l=>v) in *features(t)\features(s)*
(4:) *ergänze(t, l=>u)* für alle (l=>u) in *features(s)\features(t)*
/* die features von s und t sind gleich */
seien Q und R die die offenen feature-Listen abschließenden Variablen
(5:) Q = R
/* s und t sind auch in Zukunft gleich */
(6:) für alle (l=>u) in *features(s) restrict(u, sort(l(X)))*
(7:) für alle (l=>v) in *features(t) restrict(v, sort(l(Y)))*
/* s und t sind sortenrecht */
(4:) sonst scheitert die Unifikation
handelt es sich um zwei Konstruktor-Terme, dann
sind Funktor und Stelligkeit gleich, dann
(8:) *unifiziere(s_i, t_i)* für alle Argumentpaare (s_i, t_i) mit $1 \leq i \leq n$
sonst ist die Unifikation gescheitert
ist einer der beiden Terme eine Variable
(9:) s = t
/* Variable ist sortenrecht substituiert */
sonst scheitert die Unifikation
■

sort(t)
/* liefert die Sorte eines Terms */

restrict(t, s)
/* schränkt den Term t auf die Sorte s ein
Nach dem Aufruf ist sort(t) ≤ s, oder aber die Einschränkung schlägt fehl. Mit dem Fehlschlagen wird auch die aufrufende Unifikation beendet. */

features(t)
/* liefert die Menge der features eines feature-Terms */

ergänze(t, f)
/* ergänzt die Liste der features eines Terms t um das feature f */

Wenn die beiden Terme unifizierbar sind, werden sie durch diesen Algorithmus Schritt für

Schritt gleich gemacht. Dazu werden rekursiv von außen nach innen die potentiellen Unterschiedspaare isoliert und geprüft, ob eine Angleichung notwendig ist. Konkretisierungen eines Terms t werden durch den Aufruf der Prozedur *ergänze (t, f)* mit all denen features f durchgeführt, die in dem mit ihm zu unifizierenden Term s zusätzlich enthalten sind.[3] Durch diese zielgerichtete Vorgehensweise ist die Anzahl der Konkretisierungen minimal - eine notwendige Voraussetzung für den Erhalt eines allgemeinsten Unifikators. Zwar wird jede Konkretisierung nur auf den betrachteten feature-Term und nicht auf beide vollständigen Terme angewendet, jedoch ist zu bedenken, daß jeder Term durch Verwendung gleichnamiger Variablen vollständig referenziert ist, die Änderung eines feature-Unterterms sich also gleichzeitig auf alle seinen weiteren Auftreten auswirkt, gewöhnliche Konstruktor-Terme und Variablen von der Konkretisierung unberührt bleiben und der jeweils ander Term das feature sowieso schon enthält. Insbesondere bleiben daher normalisierte Terme durch *ergänze* in Normalform.

Daß die Konkretisierung sortenrecht ist, wird durch die Prozedur *ergänze* nicht sichergestellt, da die ursprünglichen features eines feature-Terms durch die Einschränkung seines Typs unter Umständen selbst schon nicht mehr sortenrecht sind. Zur Verdeutlichung versuche man, die Terme des vorigen Beispiels mit dem Algorithmus zu unifizieren.

In dem vorgeschlagenen Algorithmus wird die sortenrechte Einschränkung gesammelt für jedes feature der beiden Terme explizit durch die Prozedur *restrict* vorgenommen, die die Unifikation abbricht, wenn sich kein sortenrechte Einschränkung der feature values vornehmen läßt.

Die sortenrechten Substitutionen können von Lopster mit dem Unifikationsoperator '=' durchgeführt werden und bedürfen keiner besonderen Erläuterung.

Beispiel:

Es sei die Signatur $(S, \leq, \Sigma)$ mit

$$(S, \leq) = \{s_3 \leq s_1,\ s_3 \leq s_2,\ s_5 \leq s_4,\ s_6 \leq s_5\} \quad \text{und}$$
$$\Sigma = \{\ f\colon s_4 \to s_4,\ f\colon s_5 \to s_5,\ f\colon s_6 \to s_6,$$
$$l_1\colon s_1 \to s_4,\ l_1\colon s_2 \to s_5,\ l_1\colon s_3 \to s_6,$$
$$l_2\colon s_1 \to s_5,\ l_3\colon s_2 \to s_6,\ b\colon \to s_5,\ c\colon \to s_6\}$$

vorausgesetzt.

[3] Die Ergänzung der Liste erfolgt durch Instanziierung des offenen Restes mit einer neuen Liste bestehend aus dem zusätzlichen feature und einem offenen Rest. Die Konkretisierung ist damit auf die Substitution zurückgeführt worden.

Dann werden zwei Terme

$s = g(V_1{:}s_1 \;\&\; [l_1 => f(X{:}s_4),\ l_2 => b \mid R_1])$
$t = g(V_2{:}s_2 \;\&\; [l_1 => Y{:}s_5,\ l_3 => c \mid R_2])$

durch Aufruf von *unifiziere(s, t)* wie folgt unifiziert: (Die Ziffer vor dem Label gibt die Rekursionsebene an.)

1(8:) *unifiziere*$(V_1{:}s_1 \& [l_1 => f(X{:}s_4), l_2 => b \mid R_1],\ V_2{:}s_2 \& [l_1 => Y{:}s_5, l_3 => c \mid R_2])$

2(1:) $V_1{:}s_1 = V_2{:}s_2$ (V_1 und V_2 werden durch $V_3{:}s_3$ substituiert. s und t sind damit wegen $l_1 => f(X{:}s_4)$ und $l_1 => Y{:}s_5$ nicht mehr sortenrecht

2(2:) *unifiziere*$(f(X{:}s_4),\ Y{:}s_5)$

3(9:) $f(X{:}s_1) = Y{:}s_2$ (X wird nach s_5 eingeschränkt und Y mit f(X) substituiert)

2(3:) *ergänze*$(V_3{:}s_3 \;\&\; [l_1 => f(X{:}s_5),\ l_2 => b \mid R_1],\ l_3 => c)$
(ergibt $V_3{:}s_3 \;\&\; [l_1 => f(X{:}s_5),\ l_2 => b,\ l_3 => c \mid R_3]$)

2(4:) *ergänze*$(V_3{:}s_3 \;\&\; [l_1 => f(X{:}s_5),\ l_3 => c \mid R_2],\ l_2 => b)$
(ergibt $V_3{:}s_3 \;\&\; [l_1 => f(X{:}s_5),\ l_3 => c,\ l_2 => b \mid R_4]$)

2(5:) $R_3 = R_4$ (R_3 und R_4 werden durch R_5 substituiert)

2(6:) *restrict*$(f(X{:}s_5),\ s_6)$ (X wird nach s_6 eingeschränkt, die anderen features sind bereits sortenrecht und werden hier übersprungen)

Der Algorithmus terminiert und die beiden Ausgangsterme sind gleich:

$s = g(V_3{:}s_3 \;\&\; [l_1 => f(X{:}s_6),\ l_2 => b,\ l_3 => c \mid R_5])$
$t = g(V_3{:}s_3 \;\&\; [l_1 => f(X{:}s_6),\ l_3 => c,\ l_2 => b \mid R_5])$

■

Wegen der endlichen Schachtelungstiefe und der rekursiven Abarbeitung der Terme durch den Unifikationsalgorithmus ist die Terminierung desselben offensichtlich.

Es bleibt noch zu zeigen, daß der angegebene Algorithmus vollständig und korrekt ist in dem Sinne, daß er genau dann einen allgemeinsten Unifikator (in Lopster bestehend aus einer Antwortsubstitution) findet, wenn die beiden Ausgangsterme unifizierbar sind.

Auf eine exakte Beweisführung möchte ich hier jedoch verzichten und stattdessen den Beweis der Vollständigkeit und Korrektheit skizzieren.

Zunächst halte man sich vor Augen, daß der Algortihmus nur die Gleichungen des Startgleichungssystems und die, die durch Anwendung Smolka's Lösungsregeln (D) - (E) auf

das Startgleichungssystem hinzukommen, rekursiv aufzählt.

Jeder Aufruf von *unifiziere* repräsentiert eine Gleichung. Ausgehend von der Startgleichung s≐t werden die anderen Gleichungen an den mit Lables markierten Stellen gemäß den Regeln von Smolka erzeugt. Dabei entspricht (1:) (W1), (2:) (M2), (3:) zählt zusammen mit (4:) und (2:) die feature-Gleichungen des Startgleichungssystems auf, (5:) hat nur technische Bedeutung, (6:) und (7:) entsprechen (W2), (8:) entspricht (D) und (9:) (O) und (M1).

Beweisskizze (Vollständigkeit)

Für die Vollständigkeit ist zu beweisen, daß der Algorithmus immer dann eine Lösung liefert, wenn die beiden Terme unifizierbar sind. Ich tue dies, indem ich zeige: Wenn der Algorithmus abbricht, ohne einen Unifikator gefunden zu haben, dann sind die Terme auch nicht unfizierbar bzw. das Gleichungssystem nicht lösbar. (Bei Fragen der Lösbarkeit verweise ich auf die Lösungsregeln aus Abschnitt 1.1.4 bzw. [Smolka 89].)

Der Algorithmus bricht ohne Lösung ab,

- wenn ein Unterschiedspaar aus Konstruktor-Termen mit unterschiedlichem Funktor oder Stelligkeit besteht (8:),
  dann verbleibt eine Gleichung aus zwei Konstruktor-Termen im Gleichungssystem, die sich weder durch die Regel (D) noch durch andere Regeln eliminieren läßt, so daß das Gleichungssystem nicht in gelöste Form gebracht werden kann

- wenn für ein feature der Aufruf von *restrict* scheitert ((6:) oder (7:)),
  dann ist eine falsch sortierte feature-Gleichung im Gleichungssystem enthalten und die Regel (W2) darauf nicht anwendbar

- wenn die Unifikation zweier feature-Variablen scheitert (1:),
  dann ließ sich Regel (W1) nicht anwenden

- wenn einer der Terme eine Variable ist und die Unifikation scheitert (9:),
  dann ist entweder die dazugehörige Gleichung nicht sortenrecht und damit das Gleichungssystem nicht lösbar, oder, wenn beide Terme Variablen sind, die Regel (W1) nicht anwendbar

■

Beweisskizze (Korrektheit)

Zu zeigen ist: Immer wenn der Algorithmus ein Ergebnis liefert, dann sind die Terme unifizierbar und das dazugehörige Gleichungssystem ist lösbar und das Ergebnis repräsentiert das Gleichungssystem in gelöster Form.

Da der Algorithmus nur Operationen ausführt, für die es Entsprechungen in Smolka's Regelmengen (D) - (E) gibt, kann man für jeden Unifikationsschritt diese Regeln auf das Start-Gleichungssystem parallel anwenden. Bei Terminierung des Algorithmus erhält man dann ein Gleichungssystem, das durch den Ergebnisterm repräsentiert wird. Nach Satz 1.10 ist dieses Gleichungssystem dann aber in gelöster Form.

2.1.2.6 Darstellung von Klauseln mit feature-Termen

Aufgabe der Klauseln ist es, für den korrekten Ablauf eines in Lopster transformierten osf-Prolog-Programms zu sorgen. Wie bereits im vorangegangenen Abschnitt angedeutet wurde, treten Probleme bei der Programmabarbeitung mit feature-Termen immer dann auf, wenn an der Lopster-inhärenten Unifikation feature-Terme beteiligt sind.

Abgesehen von den innerhalb einer Klausel explizit angegebenen Unifikationen findet im Zuge der SLD-Resolution eine Unifikation immer dann statt, wenn zu einem Literal eines Regelrumpfes ein passendes Literal eines Regelkopfes gesucht wird. Die hierfür verwendete Unifikation ist im Lopster-System fest verankert und soll nicht verändert werden. Aus dem vorigen Abschnitt ist zu entnehmen, daß sich daraus Probleme ergeben (nämlich wenn zwei feature-Terme unifiziert werden müssen), die am einfachsten dadurch gelöst werden können, daß man den einen Unifikanden durch eine uninstanziierte Variable ersetzt. Diese Variable kann zunächst normal unifiziert (das heißt in diesem Fall immer instanziiert) werden und muß sodann explizit durch den modifizierten Unifikationsalgorithmus mit dem ursprünglichen Term unifiziert werden. Meine Wahl für den zu ersetzenden Unifikanden fiel auf den des Regelkopfes, weil dies die explizite Unifikation zum frühest möglichen Zeitpunkt in der Abarbeitung des Programms erlaubt.

Dabei ist jedoch zu beachten, daß auch Variablen im Regelkopf bereits mit feature-Termen instanziiert sein können, nämlich dann, wenn dieselbe Variable darin mehrfach auftritt und bei einer in dem Ablauf der Unifikation zurückliegenden Substitution bereits instanziiert worden ist. Dem kann entgegnet werden, indem alle weiteren Auftreten einer Variablen eindeutig umbenannt werden und hernach die Unifikation im Rumpf explizit durchgeführt wird.

Die Normalisierung aller Terme im Rumpf findet, wie bereits in Abschnitt 2.1.2.4 angedeutet, durch Unifikation aller koreferenzierten feature-Terme und anschließende Ersetzung durch das Ergebnis statt. Auf diese Weise können mit Versagen der Unifikation auch inkonsistente feature-Terme zur Übersetzungszeit entdeckt werden.

Beispiel:

Unter Vorraussetzung der Signatur

```
features.
type nomen & [kas=>kasus,num=>numerus,gen=>genus]<wortart.
type substantiv < nomen.
type adjektiv & [kmp=>komparation] < nomen.
type verb & [nmv=>numerus, gnv=>genusVerbi]
type deponens & [gnv=>passivisch] < verb.
type satz & [subjekt=>nomen, prädikat=>verb].
```

wird die osf-Klausel

```
v(X:verb & [nmv=>N]) :-
     v(Y:nomen & [num=>N]),
     v(Z:adjektiv & [num=>N]).
```

durch

```
v(Vn) :- Vn =+= V1:verb & [nmv=>N|R1],
     v(V2:nomen & [num=>N|R2]),
     v(V3:adjektiv & [num=>N|R3]).
```

in Lopster realisiert. Ein etwas komplexeres Beispiel stellt das folgende dar:

```
v(X:verb & [nmv=>N],
  Y:satz & [subjekt=>Z & [num=>N], prädikat=>X]):-
     v(X:deponens & [nmv=>singular]).
```

wird in Lopster als

```
v(Vn, Vm) :-
     Vn =+= V1:deponens & [nmv=>singular|R1],
     Vm =+= V2:satz&[subjekt=>V3:nomen&[num=>singular|R3],
          prädikat=>V1:deponens & [nmv=>singular|R1]|R2],
     v(V1:deponens & [nmv=>singular|R1]).
```

dargestellt. Man beachte, daß bei der Transformation eine Normalisierung der durch X koreferenzierten Terme stattgefunden hat.

■

2.2 DUG als linguistisches Werkzeug

Zwar läßt sich, wie ich in den theoretischen Grundlagen bereits gezeigt habe, eine DUG als DCG implementieren, jedoch werden damit die Möglichkeiten zur Effizienzsteigerung nicht ausgenutzt. Ich möchte daher einen anderen Weg beschreiten und DUG's als alternatives Werkzeug zu DCG's anbieten.

Zunächsteinmal soll dazu eine etwas andere Sichtweise der Dependenzgrammatik vorgestellt werden. Die Gültigkeit der alten Definition bleibt davon unberührt, aus praktischen Gründen ist es jedoch sinnvoll, einige Vereinfachungen zu treffen.

2.2.1 Pragmatische Neudefinition der DUG

Die systematische Zuordnung von Terminalsymbolen zu Nichtterminalsymbolen legt die Frage nahe, wozu man überhaupt noch zwischen den beiden unterscheidet. Tatsächlich schreibt Hellwig, ein Merkmal der Dependenzgrammatik sei, daß es diesen Unterschied nicht gäbe [Hellwig 86].

Sicher ist zumindest, daß eine solche Unterscheidung in der Praxis wegen des regelmäßigen Regelaufbaus nicht benötigt wird. (vgl. Definition 1.21)

Man gibt dazu die Bedingung, daß die Menge der Terminalsymbole und die der Nichtterminalsymbole disjunkt sein müssen, auf, und wählt die Identität als Zuordnungsvorschrift f. Da die Repräsentation von Terminalen und Nichtterminalen durch eindeutige Symbole zugunsten partieller Spezifikationen durch Terme aufgegeben worden ist, wird die Identität eines Terms in der Rolle eines Terminalsymbols und eines solchen in der Rolle eines Nichtterminalsymbols durch die Unifizierbarkeit entschieden.

Da in jeder Regel außer den Startregeln genau das Symbol des Regelkopfes akzeptiert wird, reicht es zunächst aus, die Stelle, an der das Symbol akzeptiert werden soll, durch ein spezielles Symbol 'self' zu markieren.[1]

Beispiel:

Um DUG's von DCG's unterscheiden zu können, soll ein neues Produktionssymbol ':>' eingeführt werden.

```
s :> v(_, verb, 'Prädikat').

v(schenkt, verb, Rolle) :>
     v(_, subst, 'Subjekt'),
     self,
     v(_, subst, 'Empfänger'),
     v(_, subst, 'Objekt').
```

[1]Eine ähnliche Notation wählten bereits Gaifman und Hays bei der Formulierung ihrer Dependenzregelmenge L_I. (vgl. Abschnitt 1.2.4)

```
v(schläft, verb, Rolle) :>
     v(_, subst, 'Subjekt'),
     self.

v('Hans', subst, Rolle) :> self.

v('Peter', subst, Rolle) :> self.

v('Buch', subst, Rolle) :>
     v(_, art, 'Artikel'),
     self.

v(ein, art, Rolle) :> self.[2]
```

Diese notationelle Vereinfachung allein führt jedoch noch zu keiner Verbesserung der Strategie. Voraussetzung für die Anwendbarkeit einer Regel zu Akzeption eines Satzes ist das Enthaltensein des Symbols des Regelkopfes in dem zu akzeptierenden Satz. Diese Tatsache kann ausgenutzt werden, um vor Abarbeitung der Regel bis zum Symbol 'self' und anschließenden Akzeptionsversuch zu entscheiden, ob die Regel zum Ziel führt, indem die Akzeption bereits vor Aufruf, also aus dem Rumpf der aufrufenden Regel heraus durchgeführt wird.

Dazu ist ein leicht modifizierter Akzeptionsbegriff notwendig, der das bis dahin nur teilweise spezifizierte zu akzeptierende Symbol an jeder beliebigen Stelle des Satzes per Unifikation entdecken und entfernen kann, gegebenenfalls fehlschlägt, wenn kein passendes Element existiert, und so ein Ausführen der dazugehörigen Regel frühzeitig verhindert.

Im Zusammenwirken mit den in DUG möglichen partiellen Spezifikationen der Symbole durch Terme wird der Akzeptionsvorgang sogar durch den Eingabesatz zielgerichtet gesteuert, da von vornherein nur Regeln zur Anwendung kommen, deren Symbole auch im Satz enthalten sind.

Die Abarbeitung der Klauseln verläuft dann wie folgt:

> Ausgehend von dem Startsymbol wird für jedes Literal im Regelrumpf zuerst das entsprechende Symbol akzeptiert und dann, falls die Akzeption gelungen ist, die (durch die Akzeption genauer spezifizierte) dazugehörige Regel abgearbeitet.

Beim Fehlschlagen einer Akzeption wird durch Rücksetzen die Eignung der nächsten Alternative überprüft.

---

[2] Produktionen dieser Art stellen kein Ersetzungssystem im eigentlichen Sinne mehr dar. Man überzeuge sich, daß bei der Auffasung der Produktionen als Ersetzungsregeln die durch die Beispielgrammatik erzeugte Sprache nur noch aus einer Reihe von 'self'-Symbolen besteht.

Beispiel:

Der Satz

Peter schläft.

liegt nach der üblichen Vorverarbeitung in folgender Form vor:

[v('Peter', subst, _), v('schläft', verb, _)]

Die Akzeption verläuft dann wie folgt:

call s	[v('Peter', subst, _), v('schläft', verb, _)]
accept v(_, verb, 'Prädikat')	[v('Peter', subst, _)]
call v(schläft, verb, 'Prädikat')	[v('Peter', subst, _)]
accept v(_, subst, 'Subjekt')	[]
call v('Peter', subst, 'Subjekt')	[]
call self	
call self	

■

Man vergleiche dieses Beispiel mit dem von Abschnitt 1.2.6, wo die Akzeption eines Wortes erst im Rumpf seiner Regel durchgeführt wurde. Es fällt auf, daß das Verfahren in diesem Fall mehr 'Weitblick' beweist (es werden auf Anhieb die richtigen Produktionen für Verb und Substantiv herangezogen), obwohl die Strategie der Auswahl von Regeln immer noch von der SLD-Resolution bestimmt wird. Der Leser halte sich ferner vor Augen, daß eine Hinzunahme von Regeln zu anderen Wörtern an dem oben gezeigten Akzeptionsverlauf nichts ändert: Lediglich die alternative Angabe von Regeln zu den gleichen Wörtern kann den Suchraum vergrößern und damit in Sackgassen führen. Im allgemeinen wird die Anzahl unterschiedlicher Regeln zu einem Wort zumindest im Vergleich zur Gesamtwortzahl jedoch recht klein sein, so daß nicht mit einer wesentlichen Verschlechterung des Verhaltens bei Vergrößerung des Lexikons zu rechnen ist.

Bemerkung: Links-Rekursionen, soweit überhaupt ausdrückbar, stellen in DUG's kein Problem dar, da vor jedem Regelaufruf ein Zeichen akzeptiert wird, unendliche Ableitungen bei endlicher Satzlänge also unmöglich sind.

Wenn man bereit ist, auf die Wortstellungsinformation einer Sprache zu verzichten, hat das 'self'-Symbol hier keine besondere Bedeutung mehr und kann daher entfallen. In Hellwig's Beschreibung der DUG werden die Phänomene der Wortstellung durch spezielle features abgedeckt, eine Verfahrensweise, die er ausführlich begründet [Hellwig 86].[3] Es ist jedoch möglich, trotz der vorgezogenen Akzeption die in der DUG codierte Reihenfolge der

[3]Tatsächlich übernimmt in manchen Sprachen, z.B. im Englischen, die Wortstellung ein Teil der Funktion, die in anderen Sprachen, zum Beispiel im Lateinischen oder im Deutschen, von bestimmten morpho-syntaktischen Merkmalen, z.B. dem Kasus, übernommen werden.

Symbole zu berücksichtigen, ohne die Effizienz zu beeinträchtigen. Auf das Verfahren möchte ich jedoch hier nur kurz eingehen.

Durch die vorgezogene Akzeption wird der Eingabesatz in eine linke und eine rechte Hälfte zerteilt und die beiden Hälften getrennt voneinander an die zum akzeptierten Symbol gehörige Regel weitergereicht. Diese muß bis zu ihrem Symbol 'self' die linke Hälfte vollständig akzeptieren, ab demselben wird die Akzeption in der rechten Hälfte des Eingabesatzes fortgesetzt.

Beispiel:

Mit diesem erweiterten Verfahren verläuft die Akzeption des Satzes

Hans schenkt Peter ein Buch.

wie folgt: (Die beiden Hälften des verbleibenden Eingabesatzes werden in diesem Beispiel durch '|' getrennt, der kompakteren Dartsellung wegen wird das dritte Argument zu v/3 weggelassen.)

call s	[v('Hans',subst),v(schenkt,verb),v('Peter',subst),v(ein,art),v('Buch',subst)]
accept v(_, verb)	[v('Hans',subst)] | [v('Peter',subst),v(ein,art),v('Buch',subst)]
call v(schenkt, verb)	[v('Hans',subst)]
accept v(_, subst)	[] | []
call v('Hans', subst)	[]
self	[]
self	[v('Peter',subst),v(ein,art),v('Buch',subst)]
accept v(_, subst)	[] | [v(ein,art),v('Buch',subst)]
call v(Peter, subst)	[]
self	[v(ein,art),v('Buch',subst)]
accept v(_, subst)	[v(ein,art)] | []
call v('Buch', subst)	[v(ein,art)]
accept v(_, art)	[] | []
call v(ein, art)	[]
self	[]
self	[]

Die Akzeption des Satzes

* Hans ein Buch schenkt Peter.

scheitert jedoch aufgrund der ungrammatischen Wortstellung.

call s	[v('Hans',subst),v(ein,art),v('Buch',subst),v(schenkt,verb),v('Peter',subst)]
accept v(_, verb)	[v('Hans',subst),v(ein,art),v('Buch',subst)] | [v('Peter',subst)]
call v(schenkt, verb)	[v('Hans',subst),v(ein,art),v('Buch',subst)]
accept v(_, subst)	[] | [v(ein,art),v('Buch',subst)]
call v('Hans', subst)	[]
self	[v(ein,art),v('Buch',subst)]
self	[v(ein,art),v('Buch',subst)]|[v('Peter',subst')]
fail	(der linke Teilsatz vor 'v(schenkt,verb)' ist noch nicht akzeptiert!
back to accept v(_,subst)	[v('Hans',subst),v(ein,art)]|[]
call v('Buch', subst)	[v('Hans',subst),v(ein,art)]
accept v(ein, art)	[v('Hans',subst]|[]
self	
fail	

■

2.2.2 Syntax

Die Syntax der DUG's ist sehr einfach. Abgesehen von dem Produktionsoperator ':>' sollen noch zwei zusätzliche Operatoren vorgestellt werden, die das Formulieren von Grammatiken eleganter machen können. Da ich mich in meiner Arbeit mit der lateinischen Sprache, in der die Wortstellung keine so große Rolle spielt, befassen werde, kann das Symbol 'self' sowie seine Behandlung entfallen.

In der Dependenzgrammatik einer natürlichen Sprache treten häufig fakultative Valenzen auf. Um deswegen nicht für jede Kombination von Valenzen eine eigene Regel angeben zu müssen, wird ein Präfix-Operator zur Kennzeichnung solcher optionaler Ergänzungen eingeführt: das Fragezeichen.

Beispiel:

```
v(lesen, verb, _) :>
     v(_, subst, 'Subjekt'),
     ? v(_, subst, 'Objekt').
```

Sollte die Dependenzregel eines Wortes genau der eines anderen entsprechen, so soll es möglich sein, mittels des Präfix-Operators '==>' einen Verweis auf die andere Regel anzugeben. Das Symbol wird wie 'geht wie' oder 'siehe' gelesen. Zwar stellt diese Möglichkeit eine Abkehr von der reinen Lehre dar, weil dadurch der Einteilung der Wörter in Kategorien Vorschub geleistet wird, doch ist eine solche Vorgehensweise vom praktischen Standpunkt aus sinnvoll, da man sich so beim Erstellen einer Grammatik eine Menge Schreibarbeit ersparen kann.

Beispiel:

```
v(geben, verb, Rolle) :> ==> v(schenken, verb, Rolle).
```

Das Fehlen von Dependentien eines Wortes wird durch die leere Liste [] im Regelrumpf ausgedrückt.

Beispiel:

```
v(ein, artikel, _) :> [].
```

Der Leser mag sich fragen, wodurch denn in der Grammatik jetzt noch angegeben wird, was akzeptiert werden soll. Es ist zu bedenken, daß die Akzeption immer Bedingung für den Aufruf einer Regel ist, also unmittelbar davor erfolgen muß.

Die formale Definition der Syntax der DUG's lautet schließlich wie folgt:

DUG-Regel ::= Kopf ':>' Rumpf.
Kopf ::= <Literal>.
Rumpf ::= <Literal>.
Rumpf ::= [].

Rumpf ::= '?' <Literal>.
Rumpf ::= '==>' <Literal>.
Rumpf ::= Rumpf ',' Rumpf.

■

2.2.3 Abbildung auf Horn-Klauseln

Die wesentlichen Aufgaben der Transformation einer DUG-Regel in eine Horn-Klausel sind die Erweiterungen der Literale um zusätzliche Argumente zur Aufnahme des (Rest-)Satzes sowie die Integration der Akzeptionen an den richtigen Stellen der Regelrümpfe. Bei den Umformungen sind darüberhinaus die Bedeutungen der zusätzlichen Symbole zu berücksichtigen.

Um an dieser Stelle nicht bereits auf das Algorithmische eingehen zu müssen, möchte ich die Umsetzung hier nur schematisch darstellen. Die algorithmische Transformation wird in Abschnitt 4.3.1 ausführlich beschrieben.

Wie bereits in Abschnitt 2.2.1 angedeutet, soll unmittelbar vor Aufruf einer Regel aus dem Rumpf einer DUG-Produktion heraus der Akzeptionsversuch erfolgen. Da ein Symbol eines Satzes grundsätzlich an jeder beliebigen Position im Satz akzeptiert werden kann, kann die Akzeption nicht wie bei der Umsetzung von DCG's durch die Verwendung von Differenzlisten erfolgen. Es wird daher ein spezielles Prädikat `accept/3` eingeführt, dem bei Aufruf das zu akzeptierende Symbol in Form eines Terms sowie der Ein- und der Ausgabesatz übergeben werden. Das Prädikat versucht, das Symbol im Eingabesatz zu finden und zu entfernen, so daß der um das Symbol verkürzte Eingabesatz als Ausgabesatz zurückgeliefert werden kann. Sollte das gesuchte Symbol nicht im Satz enthalten sein, scheitert das Prädikat. Beim Rücksetzen ist es in der Lage, andere Vorkommen eines gesuchten Symbols im Satz aufzufinden und erlaubt so, auch alternative Interpretationen des Satzes zu analysieren.

Da der Eingabesatz des Akzeptionsvorgangs an jede Regel, die am Akzeptionsvorgang beteilig ist, weitergereicht werden können muß, wird jedes Literal einer DUG-Regel um zwei Argumente ergänzt, und zwar nach folgendem Schema:

L :> L_1, L_2, ..., L_n.

wird transformiert in

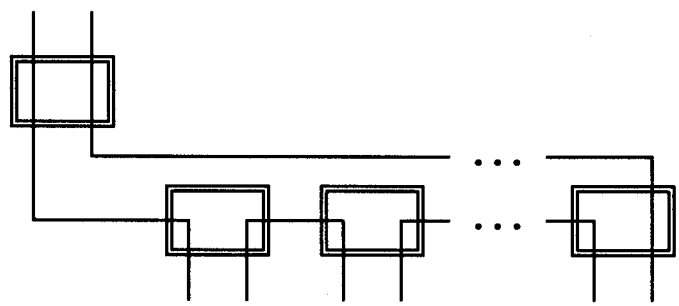
$L(S_1, S_{n+1})$:-
    accept(L_1, S_1, I_1), $L(I_1, S_2)$,
    accept(L_2, S_2, I_2), $L(I_2, S_3)$,
    ...
    accept(L_n, S_n, I_n), $L(I_n, S_{n+1})$.

Man beachte, wie in dieser Transformation die Doppelfunktion jedes Symbols der Grammatik zum Ausdruck kommt, einmal als Literal in der Funktion eines Nichtterminals

und einmal als Term (Argument von `accept/3`) in der Funktion eines Terminals. Dies ist ein Relikt der formalen Definition 1.21 einer Dependenzgrammatik!

Der Eingabesatz 'wandert' dabei wie folgt durch jede Regel:

$L(S_1, S_{n+1})$:-



$L_1(I_1, S_2), L_2(I_2, S_3), \ldots, L_n(I_n, S_{n+1})$.

In dasselbe Schema werden auch die mit einem zusätzlichen Operator versehenen Literale eingeordnet, wobei

==> L_i

entsprechend seiner Bedeutung einfach in

$L_i(S_i, S_{i+1})$

transformiert wird, so daß beim Verweis keine Akzeption des Symbols erfolgt, auf das verwiesen wird, und

? L_i

zum Ausdruck einer fakultativen Ergänzung in

(accept(L_i, S_i, I_i), $L_i(I_i, S_{i+1})$; $S_i = S_{i+1}$)

überführt wird, das ein Überspringen des Literals mit Weitergabe des Eingabesatzes als Alternative zuläßt. Der leere Regelrumpf im Fall

L :> [].

schließlich wird zu

$L(S_1, S_2)$:- $S_1 = S_2$.

übersetzt, oder, kürzer, zu

L(S, S).

Beispiel:

```
v(lesen, verb, _) :>
     v(_, subst, 'Subjekt'),
     ? v(_, subst, 'Objekt').
```

wird zu

```
v(lesen, verb, _, S1, S3) :-
     accept(v(V1, subst, 'Subjekt'), S1, I1),
          v(V1, subst, 'Subjekt', I1, S2),
     (accept(v(V2, subst, 'Objekt'), S2, I2),
          v(_, subst, 'Objekt', I2, S3); S2 = S3).


v(geben, verb, Rolle) :>
     ==> v(schenken, verb, Rolle).
```

wird zu

```
v(geben, verb, Rolle, S1, S2) :-
     v(schenken, verb, Rolle, S1, S2).
```

■

2.2.4 Semantik der Dependenz-Unifikationsgrammatik

Durch die eindeutige Abbildung von Grammatikregeln einer DUG auf Klauseln der Prädikatenlogik erster Ordnung kann den Regeln eine Bedeutung beigemessen werden, die in einer alternativen Lesart zum Ausdruck kommt.

Betrachtet man nämlich die einer DUG-Regel

$L :> L_1, L_2, \ldots, L_n.$

zugeordnete Horn-Klausel

```
L(S1, Sn+1) :-
     accept(L1, S1, I1), L(I1, S2),
     accept(L2, S2, I2), L(I2, S3),
     ...
     accept(Ln, Sn, In), L(In, Sn+1).   ,
```

so kann diese wie folgt gelesen werden:

> L ist Wort eines Satzes der Sprache L(G), wenn in dem Satz die Wörter L_1 bis L_n vorkommen, die selbst wiederum Wörter desselben Satzes sind.

3 Formulierung einer Beispielgrammatik

Die beiden unabhängig voneinader eingeführten Erweiterungen des logischen Programmierens um feature-Typen und um eine Möglichkeit zur Formulierung von Dependenz-Unifikationsgrammatiken sollen nun zusammengeführt und als linguistisches Werkzeug dazu verwendet werden, einen etwas größeren Auszug der Grammatik einer natürlichen Sprache zu formulieren.

Während die DUG's gerade im Hinblick auf ihre Verwendung in der Computerlinguistik vorgestellt worden sind, wurde die Definition der ordnungssortierten feature-Logik bislang nicht weiter motiviert. Dieses Kapitel wird nun zeigen, daß Ordnungssortiertheit und die Verfügbarkeit von features die Idee der Dependenzgrammatik um wirkungsvolle Mechanismen ergänzen, ohne die eine kompakte und elegante Formulierung einer natürlichsprachlichen Grammatik in der Form nicht möglich wäre.

Als Beispielsprache habe ich Latein gewählt. Vorteil dieser Sprache bei der automatischen Sprachverabeitung ist der hohe Informationsgehalt jedes einzelnen Wortes, der in der vergleichsweise eindeutigen Bestimmbarkeit der Wortformen zum Ausdruck kommt. Zusammen mit den strengen Kongruenzbedingungen, die der Sprache innewohnen, eignet sie sich deswegen außergewöhnlich gut zum 'Konstruieren', ein Hinweis, der wohl so manchem ehemaligen Lateinschüler noch in den Ohren klingen wird.

Darüberhinaus ist es so möglich, die morphologische Komponenete in Form einer lexikalische Analyse, die ich in meiner Studienarbeit vorgestellt und implementiert habe, einzubinden, um die automatische Vorverarbeitung der Sätze zu ermöglichen.

Die hier vorgestellte Grammatik soll nur als Beispiel dienen und erhebt keinerlei Anspruch auf Korrektheit vom linguistischen Standpunkt aus.

3.1 Einteilung der Wörter in Wortarten

In dem von mir betrachteten Teil einer Grammatik stellen die Wörter die kleinsten syntaktischen und semantischen Einheiten eines Satzes dar. Tatsächlich aber setzt sich ein Wort meistens noch aus mehreren Untereinheiten zusammen, den *Lexemen* und den *Flexemen*. Während das Lexem in etwa dem Wortstamm entspricht und damit der in erster Linie bedeutungstragende Teil eines Wortes ist, ist das Flexem Ausprägung einer bestimmten Erscheinungsform des Wortes, die seine Funktion innerhalb des Satzes zum Ausdruck bringt. Flexeme werden *Flexemkategorien* zugeordnet und Lexeme in *Lexemklassen* aufgeteilt, die nur in genau festgelegten Kombinationen miteinader auftreten können [Engel 77]. Eine solche Kombination wird von mir Wortart genannt werden.

3.1.1 Flexemkategorien

Im Lateinischen lassen sich acht Flexemkategorien unterscheiden: Kasus, Person, Numerus, Genus, Komparation, Tempus, Modus und Genus Verbi (vgl. [Engel 77]). Manche dieser Kategorien kommen bei mehreren Wortarten vor, andere wiederum nur bei einer. Jede Flexemkategorie läßt sich als (morpho-syntaktisches) Merkmal (feature!) einer Lexemklasse interpretieren, das dieser ein Paradigma als Merkmalswert zuordnet.

Es folgt eine Aufstellung der Flexemkatgorien zusammen mit ihren Paradigmen:

Kasus sind Nominativ, Genitiv, Dativ, Akkusativ und Ablativ.
Person sind erste, zweite und dritte Person.
Numerus sind Singular und Plural.
Genus sind Maskulinum, Femininum und Neutrum.
Komparation sind Positiv, Komparativ und Superlativ.
Tempus sind Präsens, Perfekt, Imperfekt, Plusquamperfekt, Futur I und Futur II.
Modus sind Indikativ und Konjunktiv.
Genus Verbi sind Aktiv und Passiv.

3.1.2 Lexemklassen

Um der Schwierigkeit einer allgemeinen Definition der Lexemklassen zu entgehen, sollen die Lexeme hier aufgrund der Kombinierbarkeit mit Flexemkategorien in Klassen eingeteilt werden. Obwohl das nicht ganz korrekt ist, soll die Verbindung einer Lexemklasse mit ihren Flexemkategorien Wortart genannt werden.

Die Lexemklassen können zunächst in zwei große Mengen aufgeteilt werden: die der flektierbaren und die der nicht flektierbaren. Jede dieser Klassen ist wieder aufteilbar in Unterklassen.

Eine große Lexemklasse ist die der Substantive. Ihr werden eigentlich nur die Flexemkategorien Kasus und Numerus zugordnet, ihr Genus dagegen ist fest. Aus Gründen der Praktikabilität soll jedoch auch das Genus als eine ihrer Flexemkategorien aufgefaßt werden, so daß sie zusammen mit der Lexemklasse der Adjektive, der die dieselben Flexemkategorien und zusätzlich die der Komparation zugeordnet werden kann, unter der Wortart *Nomen* zusammengefaßt werden können. Die Wortarten *Substantiv* und *Adjektiv* werden dann als Unterwortarten zur Wortart Nomen aufgefaßt, wobei das Genus der Substantive immer auf ein Paradigma festgelegt ist. Eine weitere Unterwortklasse ist die der *Pronomen*, die selbst wieder in verschiedene Klassen zerfällt, zum Beispiel Demonstrativpronomen oder Personalpronomen, zu denen zusätzlich die Flexemkategorie Person gehört.

Die andere große flektierbare Lexemklasse ist die der Verben. Verben können recht unterschiedliche Flexemkategorien zugeordnet werden, woraus auch entsprechend

unterschiedliche Wortarten resultieren. Zusammen mit den Flexemkategorien Person, Numerus, Modus, Tempus und Genus Verbi wird die Wortart der *finiten Verben* gebildet. Infinite Formen werden mit *Partizip*, *Gerundium*, *Gerundivum* und *Infinitiv* bezeichnet. Auf deren zugehörige Flexemkategorien möchte ich hier nicht weiter eingehen, sie können der Formulierung der Grammatik entnommen werden.

Zu den nicht flektierbaren Lexemklassen, den sogeannten Partikeln, zählen *Präpositionen*, *Konjunktionen* und *Interjektionen*. Da ihnen keine Flexemkategorien zugeordnet werden können, sind sie mit ihren Wortarten gleichzusetzen.

Diese Zusammenhänge legen die Repräsentation der Verhältnisse in Form einer Typsubsumption, wie sie in osf-Prolog durch feature-Typen realisiert wird, nahe. Sie können durch folgende Angaben recht direkt deklariert werden:

Flexemkategorien:

```
kasus < flexkat.
person < flexkat.
numerus < flexkat.
genus < flexkat.
komparation < flexkat.
modus < flexkat.
tempus < flexkat.
genusVerbi < flexkat.
```

Paradigmen:

```
sort [nom, gen, dat, akk, abl] : kasus.
sort [1, 2, 3] : person.
sort [sing, plur] : numerus.
sort [pos, komp, sup] : komparation.
sort [ind, konj] : modus.
sort [praes, perf, imp, plusq, fut1, fut2] : tempus.
sort [akt, pas] : genusVerbi.
```

Wortarten:

```
type nomen & [kas=>kasus,num=>numerus,gen=>genus]<wortart.
type substantiv < nomen.
type adjektiv & [kmp=>komparation] < nomen.
type pronomen [prs=>person] < nomen.
type verb & [tmp=>tempus, gnv=>genusVerbi] < wortart.
type finit & [prs=>person, nmv=>numerus, mod=>modus]<verb.
type infinit < verb.
type partizip < [verb, nomen].
type gerundivum < [verb, nomen].
type gerundium < [verb, substantiv].
type infinitiv < verb.
type praeposition < wortart.
type konjunktion < wortart.
type interjektion < wortart.
```

Diese Auffassung von der Einteilung der Wortarten mag aus der Sicht eines Linguisten nicht korrekt sein. Meine Absicht ist es jedoch nicht, einen Auszug der lateinischen Grammatik unter Berücksichtigung aller linguistischen Aspekte korrekt darzustellen, sondern mir geht es vielmehr darum, aufzuzeigen, daß sich das Prinzip der Ordnungssortiertheit sowie die Einführung von Merkmalsfunktionen in Form von features dazu eignen, die Grammatik einer natürlichen Sprache strukturiert darzustellen.

3.2 Einteilung der Wörter in Bedeutungskategorien

Um die bedeutungstragende Rolle des Wortes und deren Ausnutzbarkeit nicht vollständig zu vernachlässigen, möchte ich den Versuch unternehmen, die Wörter einer Sprache in Bedeutungskategorien einzuteilen. Dies soll es dem Entwickler einer Grammatik ermöglichen, nicht nur die syntaktische Valenz, sondern auch die logisch-semantische Valenz zur Analyse eines Satzes heranzuziehen.

Die Bedeutung des Wortstammes ist nur schwer faßbar. Als erster Ansatz soll hier der Versuch unternommen werden, Wörter zu klassifizieren, indem sie einer Anzahl Bedeutungskategorien zugeordnet werden. Die Wahl der Bedeutungskategorien ist so zu treffen, daß jedes Wort mindestens einer Kategorie zugeordnet werden kann. Auf der anderen Seite sollte die Zuordnung jedoch eindeutig sein, damit sie sich in osf-Prolog ausdrücken läßt. Sollte ein Wort so unterschiedliche Bedeutungen haben, daß es verschiedenen Kategorien zugeordnet werden müßte, so muß es entweder eine gemeinsame Unterkategorie dafür geben, oder die verschiedenen Lesarten müssen auch als unterschiedlich Wörter aufgefaßt und dementsprechend durch verschiedene Regeln abgedeckt werden.

Die folgende Aufzählung von Kategorien ist nicht zwingend: Sie ist recht naiv und soll daher nur als Vorschlag verstanden werden.

Kategorie	Beispiele
Name	marcus, gaius, lucius
Beruf	nauta, imperator, aber auch filius, avus
Tier	equus, avis
Pflanze	abor
Gegenstand	librum, domus
Ort	roma, lacum, domus
Zeit	hodie, vesper
Eigenschaft	dignitas
Handlung	dare, currere
Ereignis	bellum, pax

Es fällt auf, daß manche Kategorien inhaltlich miteinander verwandt sind, so daß sie sich unter gemeinsamen Oberbegriffen zusammenfassen lassen:

Name und Beruf bezeichnen Personen

Ort und Zeit bezeichnen Ausdehnungen (Dimensionen)
Personen, Tiere, Pflanzen und Gegenstände sind materielle Objekte
Orte, Zeit, Eigenschaften, Handlungen und Ereignisse sind immaterielle Objekte
materielle Objekte und immaterielle Objekte sind Objekte

Die Taxonomie der Bedeutungskategorien sieht in osf-Prolog formuliert demnach so aus:

```
[materiell, immateriell] < objekt.
[person, tier, pflanze, gegenstand] < materiell.
[ausdehnung, eigenschaft, handlung, ereignis] <immateriell.
[ort, zeit] < ausdehnung.
[name, beruf] < person.
```

Eine praktische Bedeutung erlangt die Zuordnung von Wörtern zu diesen Kategorien in Verbindung mit den logisch-semantischen Valenzen. Nun kann die Menge der Wörter, die die Rolle einer Valenz annehmen können, eingeschränkt werden, indem man die Zugehörigkeit zu einer Bedeutungskategorie fordert. Dieses Verfahren ist wiederum aus der Logik bekannt, dort ist es unter der sortierten Prädikatenlogik erster Ordnung geläufig. In der Linguistik ist es meist mit dem Begriff Selektionsbeschränkung[1] geführt.

Beispiel:

Die drei Valenzen des Verbs 'dare' (geben) können auf die Kategorien 'person', 'person' und 'gegenstand' für die Rolle des Gebers, Empfängers und Objekts eingeschränkt werden. In osf-Prolog formuliert gäbe das

```
v(dare, _:handlung, _:verb & [], prädikat) :>
     v(_, _:person, _:nomen & [], subjekt),
     v(_, _:person, _:nomen & [], empfänger),
     v(_, _:objekt, _:nomen & [], objekt).
```

Die obige Taxonomie vorausgesetzt wäre eine möglicher Satz

```
[v(marcus, _:name, _:nomen&[], subjekt),
 v(lucius, _:name, _:nomen&[], empfänger),
 v(librum, _:gegenstand, _:nomen&[], objekt)
 v(dare, _:handlung, _:verb&[], prädikat)],
```

der wenig sinnvolle Satz

```
[v(roma, _:ort, _:nomen&[], subjekt),
 v(bellum, _:ereignis, _:nomen&[], empfänger),
 v(abor, _:pflanze, _:nomen&[], objekt)
 v(dare, _:handlung, _:verb&[], prädikat)],
```

dagegen ist gleichzeitig ausgeschlossen.

[1] englisch: selectional restriction

Eine weiter Rolle kommt der Klassifizierung der Wörter nach Bedeutungskategorien speziell im Lateinischen zu. Viele Ablative wie der *ablativus temporalis* lassen sich nur aufgrund der Bedeutung seiner ihn bildenden Wörter von anderen abgrenzen. Unter Verwendung von Bedeutungskategorien sollte eine solche Zuordnung jedoch kein Problem mehr darstellen, wie sich im nächsten Kapitel zeigen wird.

3.3 Formulierung der Dependenzregeln

Zunächst müssen die Wörter des Lateinischen in eine bestimmte Form gebracht werden, die sich zur Bildung von Regeln im Formalismus der Dependenz-Unifikationsgrammtik eignen. Die Sammlung dieser Regeln bildet dann das Lexikon, in dem damit nicht nur die Wörter der Sprache selbst, sondern auch das gesamte Wissen zur Satzbildung enthalten ist. Es ist also neben der Definition der Wortarten einziger Bestandteil der Grammatik, wenn man einmal von den Regeln der Formenbildung zur Vorverarbeitung des Satzes, die hier nicht behandelt werden, absieht.

Für den allgemeinen Worteintrag wähle ich den Konstruktor v/4, in dem neben der Grundform auch Bedeutungskategorie, Wortart mit Bestimmung sowie die grammatikalische Rolle untergebracht werden. Als Startsymbol werde s/0 verwendet, und andere Konstruktoren werden im Verlauf der Vervollständigung der Grammatik bei Bedarf eingeführt.

Einfache Hauptsätze

In einfacheren Sätzen steht immer ein Verb als Prädikat an der Spitze des Dependenzbaums. Diese Spitzenstellung wir durch die sehr allgemeine Startregel

```
s :> v(_, _, _:verb, praedikat).
```

ausgedrückt. Ein einfacher Hauptsatz wie

marcus avum visitat.
(Marcus besucht den Großvater.)

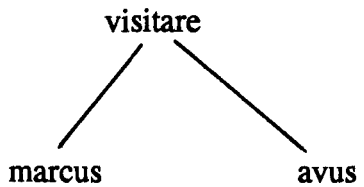
kann dann mit Hilfe der Regeln

```
v(visitare, _:handlung, _:verb & [nmv=>N], _) :>
      v(_, _:person, _:nomen & [kas=>nom, num=>N], subjekt),
      v(_, _:objekt, _:nomen & [kas=>akk], objekt).

v(avus, _:person, _:subst & [kas=>K, num=>N, gen=>G], _) :>
      ? v(_, _:person, _:adjektiv&[kas=>K, num=>N, gen=>G], attribut).

v(marcus, Kategorie:name, Wortart:subst, Rolle) :>
      ==> v(avus, Kategorie, Wortart, Rolle).
```

akzeptiert werden.



Die Rolle des Subjekts muß nicht notwendig von einem Substantiv ausgefüllt werden: Auch Adjektive und adjektivisch gebrauchte Wörter wie zum Beispiel ein Partizip können dessen Platz einnehmen. Dies braucht jedoch nicht durch jeweils andere Regeln ausgedrückt zu werden, die Tatsache, daß Substantive wie Adjektive und Partizipien Untersorten der Sorte Nomen sind, reicht allein aus, um mit einer Regel alle Fälle abzudecken. Mit

```
v(beatus, _:person, _:adjektiv & [], _) :> [].
```

läßt sich

beatus avum visitat.
(Der Glückliche besucht den Großvater.)

mit der gleichen Regel für 'visitare' akzeptieren.

Adjektivische Attribute

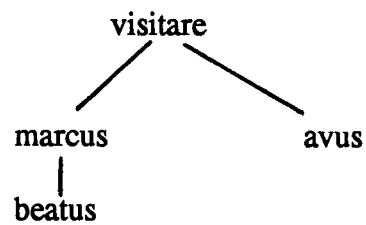
Nomen können durch Attribute näher beschrieben werden. Das adjektivische Attribut stimmt im Lateinischen in Kasus, Numerus und Genus mit seinem Bezugswort überein. Es ist deshalb vom syntaktischen Standpunkt aus sinnvoll, das Adjektiv als Dependens des Nomens zu betrachten. Dies kann durch folgende Regel ausgedrückt werden:

```
v(marcus, _:person, _:substantiv & [kas=>K, num=>N, gen=>G], _) :>
      ? v(_, _:person, _:adjektiv & [kas=>K, num=>N, gen=>G], adjAttr).
```

Daß das adjektivische Attribut fakultativ ist, wird durch das Fragezeichen ausgedrückt. Weil die Selektionsbeschränkung eines Adjektivs gerade die Menge von Objekten spezifiziert, die es beschreiben kann, wurde die gleiche wie die des Bezugswortes gewählt. Auf diese Weise läßt sich dann der Satz

marcus beatus avum vistitat.
(Der glückliche Marcus besucht den Großvater.)

akzeptieren.



Apposition

Ein Substantiv kann auch durch ein Substantiv im gleichen Kasus näher beschrieben werden: Diese Konstruktion wird Apposition genannt und läßt sich durch eine Regel wie

```
v(lucius, _:person, _:substantiv & [kas=>K], _) :>
      v(_, _:person, _:substantiv & [kas=>K], _).
```

mit

```
v(amicus, _:person, _:substantiv & [], _).
```

darstellen, wodurch die Akzeption von Sätzen wie

marcus lucium amicum visitat.
(Marcus besucht seinen Freund Lucius.)

möglich wird.

Genitiv-Attribut

Das Genitiv-Attribut als besitzanzeigendes Attribut wird im allgemeinen durch ein Nomen im Genitiv dargestellt, an das sonst keine weiteren Anforderungen gestellt werden. Mit

```
v(clamor, _:immateriell, _:substantiv & [], _) :>
      v(_, _, _:nomen & [kas=>gen], genAttr).
v(puer, _:person, _:substantiv & [], _) :> [].
```

läßt sich dann eine Phrase wie

clamor puerorum
(das Geschrei der Knaben)

akzeptieren.

In den eben gebrachten Beispielen werden in den Regeln zu der Wortart Substantiv mögliche Dependentien oft ausgelassen, wenn sie nicht zur Darstellung der beabsichtigten Aussage beitragen. Tatsächlich ist die Zahl der möglichen Dependentien eines Substantivs recht hoch und unterscheidet sich von Substantiv zu Substantiv in der Regel kaum. In der Praxis kann daher die Kodierung eines größeren Ausschnitts einer natürlichen Sprache zu einer recht beschwerlichen Angelegenheit werden, da für jedes Substantiv eine eigene Regel angegeben werden muß, die sich von anderen kaum unterscheidet. Aus diesem Grund ist es vielleicht sinnvoll, eine allgemeinste Dependenzregel anzugeben, die für alle Substantive gilt, für die nichts anderes spezifiziert ist.

```
v(_, K:objekt, _:substantiv & [kas=>K, num=>N, gen=>G], _) :>
      ? v(_, K, _:adjektiv & [kas=>K, num=>N, gen=>G], adjAttr),
      ? v(_, K, _:substantiv & [kas=>K], apposition),
      ? v(_, _, _:nomen & [kas=>gen], genAttr).
```

Das Passiv

Lange Zeit ist das Phänomen des Passivs Ursache für Kopfzerbrechen gewesen. Es wurde sogar als eines der beliebtesten Argumente für die Notwendigkeit eines Transformationsteils in der Grammatik einer natürlichen Sprache verwendet. Ich möchte hier kurz darstellen, wie sich die Eigenschaft des Passivs im Lateinischen mit der Dependenz-Unifikationsgrammatik ausdrücken läßt.

Steht ein Verb im Satz im Passiv, so wird seine Akkusativ-Objekt-Valenz zum Subjekt und das ursprüngliche Subjekt entfällt oder wird zu einer präpositionalen Ergänzung mit 'ab' im Ablativ.

marcus avum visitat. (aktiv)
(Marcus besucht den Großvater.)

avus a marco visitatur. (passiv)
(Marcus wird vom Großvater besucht.)

Der Eintrag für visitare in das Lexikon wird nun einfach in zwei Regeln aufgeteilt: eine für das Aktiv und eine für das Passiv.

```
v(visitare, _:handlung, _:verb & [gnv=>akt, nmv=>N], _) :>
      v(_, _:person, _:nomen & [kas=>nom, num=>N], subjekt),
      v(_, _:person, _:nomen & [kas=>akk], objekt).

v(visitare, _:handlung, _:verb & [gnv=>pas, nmv=>N], _) :>
      v(_, _:person, _:nomen & [kas=>nom, num=>N], subjekt),
      ? v(ab, _:person, _:praeposition & [], praepObj).

v(ab, BedKat, _:praeposition & [], _) :>
      v(_, BedKat, _:substantiv & [kas=>abl], bezug).
```

Um nicht für jedes Verb zwei Regeln angeben zu müssen, kann man eine allgemeine Verbregel formulieren, auf die alle Verben verweisen, die ein Passiv bilden können.

```
verb(_:verb & [gnv=>akt, nmv=>N]) :>
      v(_, _, _:nomen & [kas=>nom, num=>N], subjekt),
      v(_, _, _:nomen & [kas=>akk], objekt).

v(_:verb & [gnv=>pas, nmv=>N]) :>
      v(_, _, _:nomen & [kas=>nom, num=>N], subjekt),
      ? v(ab, _, _:praeposition & [], praepObj).

v(visitare, _:handlung, X:verb & [], _) :>
      ==> verb(X).
```

Freie Angaben

In einem Satz finden sich häufig freie Angaben, die nicht in dem Sinne zwingend zum Satz gehören, daß er ohne sie ungrammatisch würde, die aber dennoch zur Aussage des Satzes beitragen. Das Lateinische bietet dazu eine Fülle von Konstruktionen, von denen ich hier einige herausgreifen möchte.

Ablativus temporalis

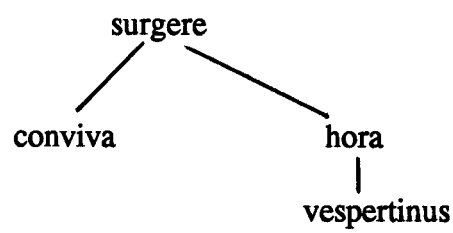
Dieser Ablativ enthält Angaben über den Zeitpunkt des Geschehens. Er kann von anderen freien Angaben im Ablativ nur aufgrund der Bedeutung seines Nomens abgegrenzt werden, das offensichtlich unter die Bedeutungskategorie 'Zeit' fallen muß.

hora vespertina convivae surgunt.
(Zur Abendstunde erheben sich die Gäste.)

Um diesen Satz akzeptieren zu können, kann entweder die Startregel für den Satz um eine Alternative ergänzt werden, die die freie Angabe als dem Prädikat gleichgestellte Teilaussage zuläßt, oder sie wird als Dependens des Verbs eingeordnet. An dieser Stelle wird die zweite Variante verwendet.

```
v(surgere, _:handlung, _:verb & [nmv=>N], _) :>
      v(_, _:materiell, _:nomen & [kas=>nom, num=>N], subjekt),
      ? v(_, _:zeit, _:nomen & [kas=>abl], ablTemp).

v(vespertinus, _:zeit, _:adjektiv & [], _).
```



Man beachte, daß die Regeln für die beiden Substantive 'conviva' und 'hora' durch die oben genannte allgemeine Substantivregel abgedeckt werden.

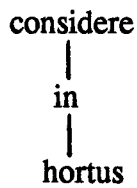
Ablativus locativus

Die Einbettung des ablativus locativus (Ablativ des Orts) verläuft analog. Er steht auf die Frage 'wo' (nicht 'wohin') und meist mit einer Präposition. Da aus der Sicht der Dependenz das Bezugswort der Präposition ihr Dependens ist, hängt das Substantiv im Ablativ nicht direkt vom Prädikat ab, sondern die Präposition steht dazwischen. Da die Präposition 'in' auf die Frage 'wo' aber den Ablativ fordert, bereitet dies keine Probleme.

in horto considunt.
(Im Garten setzen sie sich hin.)

```
v(considere, _:handlung, _:verb & [nmv=>N], _) :>
      ? v(_, _:person, _:nomen & [kas=>nom, num=>N], subjekt),
      ? v(in, _:ort, _:praeposition & [], ablTemp).

v(in, _:ort, _:praeposition & [], _) :>
      v(_, _:ort, _:substantiv & [kas=>abl], bezug).
```



Ablativus absolutus

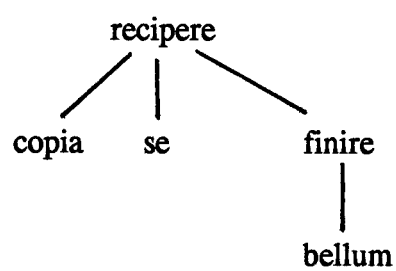
Ohne hier zu tief in die linguistische Materie einsteigen zu wollen, soll an dieser Stelle auch der ablativus absolutus besprochen werden. Ein Satz wie

> bello finito copiae se recipiunt.
> (Wenn der Krieg beendet ist, ziehen sich die Truppen zurück.)

besitzt als freie Angabe das Partizip Perfekt Passiv 'finito' im Ablativ, dessen Subjekt das Substantiv 'bello' ist. Nun ist das Partizip eine besondere Verbform und sollte daher durch eine einfache Regel mit seiner Grundform abgedeckt werden. Dies ist jedoch nicht so leicht möglich, an dieser Stelle soll daher mit einer extra Regel für das Partizip vorlieb genommen werden.

```
v(recipere, _:handlung, _:verb & [nmv=>N], _) :>
      ? v(_, _:person, _:nomen & [kas=>nom, num=>N], subjekt),
      v(se, _:person, _:pronomen & [], _),
      ? v(_, _, _:partizip & [kas=>abl], ablAbs).

v(finire, _, _:partizip & [gnv=>pas, kas=>K, num=>N, gen=>G], _) :>
      v(_,_:immateriell,_:substantiv&[kas=>K,num=>N,gen=>G],subjekt).
```



3.4 Zusammenfassung

Auch wenn hier nur die Formulierung eines kleinen Ausschnitts der Grammatik der lateinsichen Sprache vorgestellt wurde, so, denke ich, ist dieses Beispiel doch geeignet zu zeigen, daß man die Gesetzmäßigkeiten der Sprache recht direkt in einer Dependenz-Unifikationsgrammatik niederschreiben kann. Man halte sich dabei immer vor Augen, daß für jedes einzelne Wort individuelle Regeln angegeben werden können, so daß eher die Gefahr besteht, daß die Grammatik zu umfangreich wird, als daß man ein bestimmtes

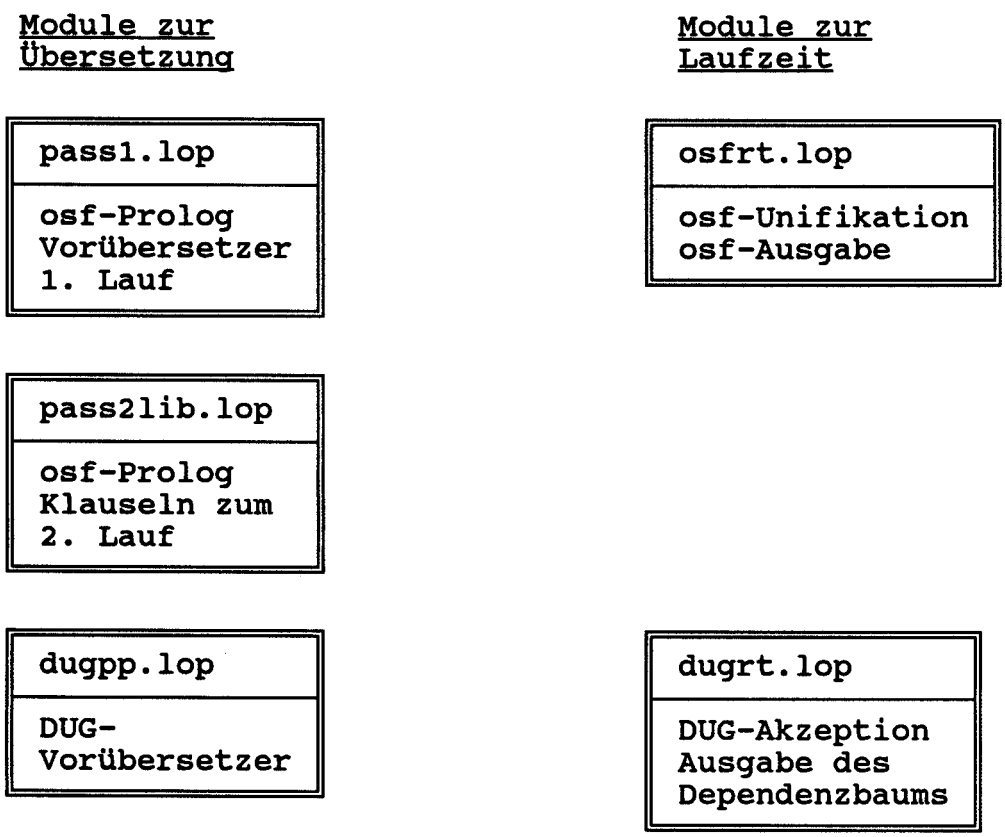
Phänomen überhaupt nicht damit ausdrücken kann.

Besonders betont werden soll der zweifache Nutzen der Ordnungssortiertheit: Zum einen kann sie im Dienste der Semantik dazu verwendet werden, Wörter nach ihrer Bedeutung hierarchisch zu klassifizieren und stellt damit ein Werkzeug dar, mit dessen Hilfe man Selektionsbeschränkungen effektiv und effizient realisieren kann. Zum anderen ermöglicht sie auf syntaktischer Ebene die Formulierung allgemeinerer Regeln (vgl. dazu die Ausführungen zu einfachen Hauptsätzen) und kann so als Mittel der Abstraktion eingesetzt werden.

4 Implementation

In diesem Kapitel sollen die Implementationen vorgestellt werden. Die zur Einbettung von osf-Prolog und DUG's in Lopster notwendigen Programme teilen sich jeweils auf in Prozeduren zur Übersetzung und solche, die zur Laufzeit der übersetzten Programme benötigt werden.

Übersicht über die implementierten Module



4.1 Implementation der osf-Unifikation

Die ordnungssortierte feature-Unifikation ist die einzige Prozedur, die zur Laufzeit eines osf-Prolog-Programmes für einen korrekten Ablauf benötigt wird[1]. Sie wird daher auch automatisch bei der Übersetzung eines solchen Programmes eingebunden, ohne daß der Benutzer sie explizit importieren muß. Die Implementation hält sich recht genau an den

[1]Vergleiche hierzu bitte die Abschlußbemerkung zu der Implementation des Übersetzters in Abschnitt 4.2.1.2

Unifikationsalgorithmus aus Kapitel 2.1 und soll hier nur auf ihre Besonderheiten hin angesprochen werden.

Wenn einer der beiden zu unifizierenden Terme eine Variable ist, ist die Unifikation mit der Instanziierung durch die normale Unifikation abgeschlossen. Selbst wenn beide Terme Variablen sind, werden diese dadurch sortenrecht eingeschränkt und aneinander gebunden, so daß die Unifikation korrekt durchgeführt wird.

Sind die äußeren Terme beider Unifikanden feature-Terme (dies ist an der Verwendung des Funktors '&' im führenden Term zu erkennen), dann kann die Unifikation, die Existenz einer größten gemeinsamen Untersorte vorausgesetzt, auf die Unifikation der offenen feature-Listen zurückgeführt werden. Die Einschränkung der features wird erst im Anschluß durchgeführt, so daß nur eine feature-Liste behandelt werden muß, da beide Listen über gemeinsame Variablen vollständig aneinader gebunden sind. Die geschickte Unifikation der feature-Listen ist ganz wesentlich für die effiziente Durchführung verantwortlich. Ich habe daher den Algorithmus von Boyer aus [Boyer 88] übernommen.

Der dritte Fall, in dem eine Unifikation möglich ist, nämlich wenn beide äußeren Terme Konstruktor-Terme sind, ist trivial und soll daher nicht weiter erläutert werden.

Für alle anderen Fälle, in denen die beiden Terme nicht unifizierbar sind, sind keine Klauseln vorgesehen, so daß mit dem 'fail' das Versagen der Unifikation quittiert wird.

Eine weitere Klausel, die beim Arbeiten mit osf-Prolog Verwendung finden kann, realisiert ein spezielles Ausgabeprädikat für feature-Terme. Da es jedoch nur bei Bedarf vom Benutzer in das Programm eingebunden werden soll, ist es im Quellprogramm mittels

```
import from osfrt write_osf/1.
```

zu importieren.

4.2 Vorübersetzer für osf-Prolog

In dieser ersten Entwicklungsstufe wird der Übersichtlikeit des Vorübersetzers der Vorzug gegeben, um ihn für Erweiterungen und Adaptation an die Zielsprache zugänglich zu machen. Letzteres wird nötig sein, da die Weiterentwicklung von Lopster zu Lopster II und damit eine Änderung der Lopster-Syntax bevorsteht. Soweit möglich, wird außerdem die Aufgabe der Prüfung der Signatur und der Wohlsortiertheit dem Zielsystem überlassen, so daß bestimmte Fehlermeldungen erst sehr spät in Erscheinung treten. Der Übersetzer zeichnet sich deswegen nicht unbedingt durch große Benutzerfreundlichkeit aus.

4.2.1 Implementation des Vorübersetzers

Der Übersetzungsvorgang eines osf-Prolog-Programms gliedert sich in zwei Phasen. In der

ersten Phase wird aus der Deklaration der osf-Signatur eine Lopster-Signatur erzeugt. Da für die Konvertierung der Programm-Klauseln die Signatur dem System bekannt sein muß und in Lopster weder eine dynamische Sortenverbandsspezifikation noch dynamische Sortierungsdeklarationen möglich sind, muß die Transformation des Programmrumpfes in einem seperaten zweiten Schritt erfolgen.

Die Ausgabe des ersten Durchlaufs ist daher ein Programm, daß neben der vollständigen Deklaration der Signatur und den ursprünglichen Programmklauseln noch ein Prädikat zum Aufruf des zweiten Übersetzerlaufs beinhaltet, der dadurch als Untermodul eingebunden wird. Dieser zweite Durchgang ist dann in der Lage, die notwendigen Transformastionen an den Klauseln vorzunehmen. Das Ergebnis kann dann von Lopster eingelesen werden.

Für eine übersichtliche Darstellung des Ablaufs der Übersetzung sei auf das Diagramm im Abschnitt zur Benutzung des Übersetzers verwiesen.

4.2.1.1 Erster Übersetzerlauf

Aufgabe des ersten Laufs ist es, die Korrektheit der Syntax der Signatur-Deklaration sowie die Einhaltung bestimmter Bedingungen an die Signatur zu überprüfen und daraus Module zu erzeugen, die von Lopster eingelesen werden können und den zweiten Lauf ermöglichen.

Die drei zusätzlichen Bedingungen für die Zulässigkeit der Signatur, deren Prüfung nicht von Lopster übernommen wird, sind die, daß jede Untersorte einer feature-Sorte wieder eine feature-Sorte sein muß, daß die Ergebnissorten von Konstruktor-Termen Konstruktor-Sorten sein müssen und die letzte Zulässigkeitsbedingung von Smolka. Da im Sortendeklarationsteil des Programmes nur Konstruktor-Sorten deklariert werden und jede im feature-Deklarationsteil neu deklarierte Sorte immer eine feature-Sorte ist, ergibt sich die erste Bedingung automatisch, wenn man wiederholte Deklarationen einer Sorte im feature-Teil verbietet. Ein Verstoß gegen dieses Verbot wird vom Übersetzer mit einem `type redeclaration error` quittiert.

Die Einhaltung der zweiten Bedingung läßt sich durch einfaches Kontrollieren der Ergebnissorte sicherstellen. Dazu müssen während des Durchlaufs durch die Signatur alle deklarierten feature- und Konstruktor-Sorten getrennt voneinander gesammelt werden.

Zur Überprüfung der dritten Bedingung schließlich ist es von Vorteil, wenn die Signatur dem Prüfverfahren vollständig zur Verfügung steht. Um nicht alle dazu notwendigen Operationen auf dem Sortenverband nachvollführen zu müssen, soll diese Prüfung dem zweiten Lauf überlassen werden.

Da das Gerüst der osf-Prolog-Syntax regulär ist, kann der Akzeption ein deterministischer endlicher Automat (DEA) zugrundelget werden:

$A = (T, Q, R, q_0, F)$

T = {sorts, types, declarations, body, end_module,
<module>, <clause>, <ifdecl>, <sortdecl>, <typedecl>, <decl>}
Q = {Start, Interface, Sorten, Typen, Deklarationen, Rumpf, Ende, Fehler}
R = {Start <module> → Interface,
Start <clause> → Fehler,
Interface sorts → Sorten,
Interface types → Typen,
Interface declarations → Deklarationen,
Interface body → Rumpf,
Interface <ifdecl> → Interface,
Sorten types → Typen,
Sorten declarations → Deklarationen,
Sorten body → Rumpf,
Sorten <sortdecl> → Sorten,
Sorten <clause> → Fehler,
Typen declarations → Deklarationen,
Typen body → Rumpf,
Typen <typedecl> → Typen,
Typen <clause> → Fehler,
Deklarationen body → Rumpf,
Deklarationen <decl> → Deklarationen,
Deklarationen <clause> → Fehler,
Rumpf <clause> → Rumpf,
Rumpf end_module → Ende}
q_0 = Start
F = {Fehler, Ende}

Die Terminale in spitzen Klammern stehen für Terme, die vom DEA nicht weiter zerteilt werden müssen. Diese Aufgabe wird durch die mustererkennende Eigenschaft (pattern matching) der Unifikation in der Prolog-Implementierung des Parsers übernommen.

Dabei stehen die Terminale für folgende Prolog-Terme:

<module> = module X,
<ifdecl> = X,
<sortdecl> = X < Y,
<typedecl> = type Type & Features < Subsorts,
<decl> = sort X,
<clause> = X,

wobei die Operatoren 'module', 'sort', 'type', '&' und '<' entsprechend deklariert sind.

Der durch <clause> entstehende Indeterminismus wird in der Prolog-Implementation durch Cuts (!) aufgehoben.

Dieser DEA wird um geeignete Strukturen erweitert, die in der Lage sind, die feature- und Konstruktor-Sorten sowie die feature label während ihrer Deklaration zu sammeln, um damit eine Überprüfung der bestehenden sowie die Erzeugung der zusätzlich benötigten Vereinbarungen zu steuern.

Die Ausgabe des ersten Übersetzungslaufs wird auf zwei Dateien verteilt: Die Spezifikation des Sortenverbands wird in die Datei `sorts` geschrieben, während die Funktionsdeklarationen sowie die restlichen Programmklauseln um Klauseln zum Import des Prädikats `pass2/1` aus dem Übersetzermodul `pass2lib.lop` ergänzt in die Datei `pass2.lop` geschrieben werden. Auch die notwendigen Prädikate zur Benutzung der osf-Unifikation werden ergänzt, so daß die erweiterte Unifikation automatisch zur Verfügung steht.

4.2.1.2 Zweiter Übersetzerlauf

Aufgabe des zweiten Laufs ist es, die Programmklauseln auf Wohlsortiertheit hin zu überprüfen, die feature-Terme einer jeden Klausel zu normalisieren sowie Auftreten von feature-Termen in Regelköpfen durch Variablen zu ersetzen und eine entsprechende explizite Unifikation in den Rumpf einzubinden. Darüberhinaus ist die letzte Zulässigkeitsbedingung aus [Smolka 89] zu überprüfen. Er kann dazu auf die vollständig deklarierte Signatur zurückgreifen, da das zu übersetzende Programm selbst Bestandteil des Übersetzers ist.

Der Deklarationsteil des Quellprogramms kann daher nahezu unverändert übernommen werden; lediglich der Import von `pass2/1` aus `pass2lib.lop` und die Deklaration von `pass2/0` werden wieder entfernt. (Dasselbe gilt natürlich auch für die Programmklausel, die den zweiten Lauf startet.) Dadurch wird das zu übersetzende Programm wieder vom Übersetzungsprogramm abgekoppelt.

Die Überprüfung der Zulässigkeit der Signatur erfolgt als erstes, da sie Voraussetzung für die korrekte Durchführung der im weiteren Ablauf benötigten Unifikation ist (vgl. Abschnitt 1.1.2 und Abschnitt 1.1.4). Der in der Formulierung der Bedingung auftretende Sortenvektor wird in seine Komponenten zerlegt und die Einhaltung komponentenweise überprüft. Die Testprozedur `check/3` wird für jedes Tripel bestehend aus einer feature-Sorte, ihren feature labels und den dazugehörigen minimalen Untersorten der feature ranges aufgerufen und muß für alle diese Tripel zu 'wahr' ausgewertet werden.

```
check(Sort, Labels, Mins) :-
     min(Sort, MinSort),
     checkf(MinSort, Labels, Mins).

checkf(MinSort, [], []).
checkf(MinSort, [Label|Labels], [Min|Mins]) :-
     frsort(Label, MinSort, Range),
     subsort(Min, Range), checkf(MinSort, Labels, Mins).
```

wobei

```
min(Sort, MinSort) :-
     subsort(X, Sort), X =/= Sort, !, min(X, MinSort).
min(Sort, Sort).
```

die minimale Untersorte `MinSort` einer angegebenen Sorte `Sort` liefert und

```
frsort(Label, Value, Range) :-
     declared_func(Label, 1, Range, [Value]).
```

die Sorte eines features liefert. Die Prädikate `subsort` und `declared_func` sind Lopster-Standardprädikate [Weinstein 89].

Die Klauseln des Rumpfs des zu übersetzenden Programms können der Reihe nach und unabhängig voneinander überprüft und gegebenenfalls transformiert werden, da jede einzelne Klausel für sich die kleinste zu betrachtende Einheit eines Prolog-Programms darstellt.

Zwar wird die Aufgabe der Überprüfung der sortenrechten Einbettung von feature-Termen in Konstruktor-Terme und Prädikate wegen der übeladenen Deklaration von '&'/2 vollständig von Lopster übernommen, jedoch ist eine automatische Überprüfung von feature-Termen auf Wohlsortiertheit nicht möglich. Deshalb wird in einem ersten Schritt für jeden feature-Term einer Klausel getestet, ob die Sorte des Werts jedes features auch tatsächlich Untersorte der durch das feature geforderten Sorte ist. Sollte dies nicht der Fall sein, wird die Klausel zusammen mit der Meldung `ill-sorted feature-term` ausgegeben und mit der Übersetzung der nächsten fortgefahren.

Im zweiten Schritt wird eine Normalisierung der feature-Terme innerhalb einer Klausel durchgeführt. Dazu werden zunächst alle Auftreten von feature-Termen durch ihre feature-Variablen ersetzt und das Paar bestehend aus feature-Variable und ersetztem feature-Term gespeichert, wobei die feature-Variable im Term gegen eine neue ausgetauscht wird, damit bei der nachfolgenden Unifikation von Variable und Term der occur-check nicht fehlschlägt. Um bei geschachtelten feature-Termen auch die Unterterme zu erfassen, erfolgt die Ersetzung rekursiv von innen nach außen. Da ein feature-Term und seine feature-Variable immer von derselben Sorte sind, ändert sich an der Wohlsortiertheit durch diese Maßnahme nichts.

Durch diesen Schritt wird jedoch gewährleistet, daß an allen koreferenzierten Auftreten eines feature-Terms tatsächlich derselbe Term steht, nämlich genau der, mit dem die nunmehr zum Platzhalter umfunktionierte feature-Variable instanziiert wird. Diese Instanziierung wird erreicht, indem jede feature-Variable mit den in der Liste der Ersetzungspaare gespeicherten dazugehörigen feature-Termen unifiziert wird. Kommt eine feature-Variable in der Liste mehrfach vor, dann wurde auch der dazugehörige feature-Term in der Klausel mehrfach (partiell) spezifiziert, so daß die wiederholte Unifikation mit allen Spezifikationen automatisch zu einer Normalisierung führt. Sollte die Unifikation fehlschlagen, dann ist der betreffende Term auch nicht normalisierbar und die Klausel fehlerhaft. Dies wird durch den Hinweis `clause cannot be normalized` vom Übersetzer bekanntgegeben.

Schließlich sind noch die feature-Terme aus den Regelköpfen zu entfernen. Dazu wird der Kopf jeder Regel rekursiv von außen nach innen durchsucht und jedes Auftreten eines feature-Terms durch eine neue Variable derselben Sorte ersetzt. Die dazugehörige explizite Unifikation wird an den Anfang des Regelrumpfes angefügt. Die genaue Durchführung ist nicht weiter schwierig und kann zusammen mit den anderen Übersetzungsregeln dem Anhang entnommen werden.

Die Ausgabe des zweiten Übersetzungslaufs wird normalerweise in die Datei `main.lop` geschrieben. Durch Aufruf eines alternativen Übersetzungsprädikats `pass2/1` mit dem Dateinamen als Argument kann jedoch auch ein anderer Name gewählt werden. Die erzeugte Datei kann von Lopster als Hauptmodul eingelesen werden. Das Hauptmodul kann, genau wie normale Lopster-Module, auch andere Module einbinden; da diese jedoch nicht automatisch mit übersetzt wurden, dürfen sie keine feature-Terme enthalten.

Noch eine Anmerkung zum Schluß: Nach erfolgreicher Beendigung dieses Laufs liegt ein korrektes osf-Prolog-Programm in dem Sinne vor, daß alle darin enthaltenen Terme sortenrecht und in Normalform sind. Solange neue Terme während des Programmablaufs nur durch Unifikation (das ist in diesem Fall die osf-Unifikation) entstehen, bleiben auch alle Terme sortenrecht. Für metalogische Konstruktionen von Termen zum Beispiel durch `univ (=..)` gilt das allerdings nicht.

4.2.2 Benutzung des Vorübersetzers

In der Benutzung des Übersetzers spiegelt sich sein zweiphasiger Aufbau wieder. Nachdem osf-Prolog-Programm erstellt und abgelegt worden ist, wird das Lopster-System wie gewohnt gestartet. Es sei an dieser Stelle noch einmal darauf hingewiesen, daß die Quelldatei die Erweiterung `.osf` tragen muß.

```
lopster
```

Sodann muß der erste Übersetzerlauf mittels

```
set_current_module(pass1).
```

geladen werden. Da der erste Lauf keine Sorten verwendet, kann die Angabe eines Sortenmoduls entfallen. Im zweiten Schritt wird dann mit

```
pass1(<Name>).
```

mit `<Name>` als Dateiname ohne die Erweiterung `.osf` die erste Phase der Übersetzung gestartet. Das Ergebnis sind das Sortenmodul `sorts` und das Modul `pass2.lop`, das in der Lage ist, sich selbst zu übersetzen. Doch zunächst wird durch

```
set_sort_module(sorts).
```

das Sortenmodul dem System bekanntgegeben und dann mit

```
set_current_module(pass2).
```

die zweite Stufe geladen. Mit

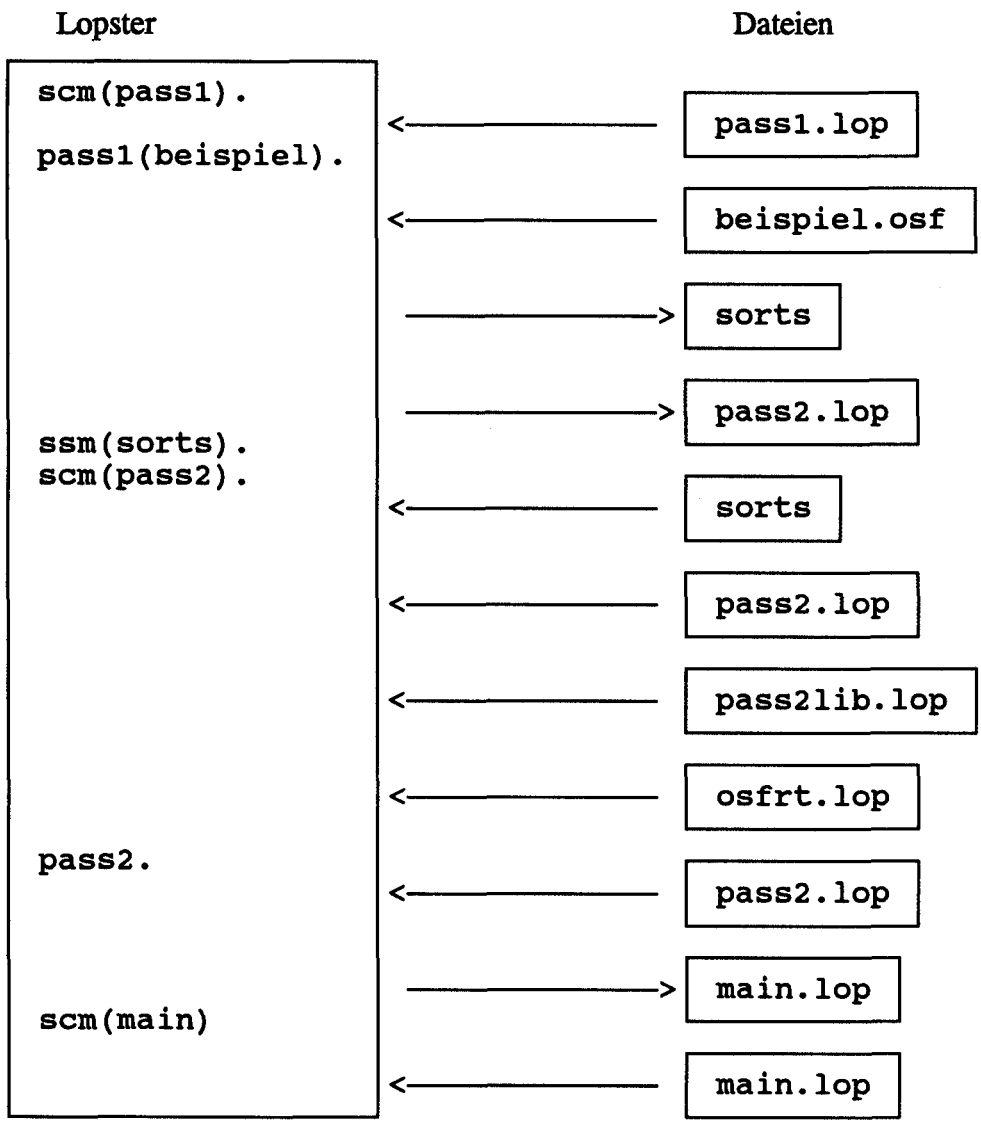
```
pass2.
```

schließlich wir die Vorübersetzung abgeschlossen. Das übersetzte Modul kann mittels

```
set_current_module(main).
```

geladen werden und ist damit zur Ausführung bereit.

Zur Verdeutlichung des Datenflusses folgt ein kleines Diagramm. Anstelle von set_sort_module und set_current_module werden die Lopster-üblichen Abkürzungen ssm und scm verwendet.



4.2.3 Übersetzung eines Beispielprogramms

Zur Verdeutlichung der Funktionsweise des Übersetzers soll ein kleines Beispielprogramm `beispiel.osf` vorgestellt und übersetzt werden.

```
module beispiel.

sorts.
land < element.
wasser < element.
luft < element.
ozean < wasser.
fluss < wasser.

features.
type objekt & [name=>list] < term.
type lebewesen & [lebensraum=>element] < objekt.
type pflanze < lebewesen.
type mensch & [lebensraum=>land] < lebewesen.
type fisch & [lebensraum=>wasser] < lebewesen.

constructors.
sort ueberall : element.
sort elbe : wasser.
sort lebt(lebewesen).
sort trifft(lebewesen, lebewesen).

body.
lebt(X:fisch & [name=>"Aal", lebensraum=>elbe]).
lebt(X:pflanze & [name=>"Alge", lebensraum=>Y:wasser]).
lebt(X:mensch & [lebensraum=>Z]) :-
     lebt(Y:lebewesen & [lebensraum=>Z]).
trifft(X, Y) :-
     lebt(X:lebewesen & [lebensraum=>Z]),
     lebt(Y:lebewesen & [lebensraum=>Z]).
end_module.
```

Durch den Start von Lopster und Eingabe von

```
set_current_module(pass1).
pass1(beispiel).
```

werden folgende zwei Dateien sorts und pass2.lop erzeugt:

sorts

```
sorts.
land < element.
wasser < element.
luft < element.
ozean < wasser.
fluss < wasser.
objekt < term.
```

```
lebewesen < objekt.
pflanze < lebewesen.
mensch < lebewesen.
fisch < lebewesen.
end_sorts.
```

pass2.lop

```
module pass2.
import from osfrt '=+='/2.
import from pass2lib pass2/1.
sort '&'(objekt, list) : objekt.
sort '&'(lebewesen, list) : lebewesen.
sort '&'(pflanze, list) : pflanze.
sort '&'(mensch, list) : mensch.
sort '&'(fisch, list) : fisch.
sort name(objekt) : list.
sort lebensraum(lebewesen) : element.
sort lebensraum(mensch) : land.
sort lebensraum(fisch) : wasser.
sort ueberall : land.
sort elbe : wasser.
sort lebt(lebewesen).
sort trifft(lebewesen, lebewesen).
body.
pass2 :- pass2('pass2.lop').
lebt(X:fisch & [name=>"Aal", lebensraum=>elbe]).
lebt(X:pflanze & [name=>"Alge", lebensraum=>Y:wasser]).
lebt(X:mensch & [lebensraum=>Z]) :-
     lebt(Y:lebewesen & [lebensraum=>Z]).
trifft(X, Y) :-
     lebt(X:lebewesen & [lebensraum=>Z]),
     lebt(Y:lebewesen & [lebensraum=>Z]).
end_module.
```

Durch Eingabe von

```
set_sort_module(sorts).
set_current_module(pass2).
pass2.
```

wird dann das Modul

main.lop

```
module main.
import from osfrt '=+='/2.
sort '&'(objekt, list) : objekt.
sort '&'(lebewesen, list) : lebewesen.
...
...
body.
lebt(V3:fisch) :-
```

```
        V3 =+= V1:fisch & [name=>"Aal", lebensraum=>elbe|V2]).
lebt(V3:pflanze) :-
        V3 =+= V1:pflanze & [name=>"Alge",
              lebensraum=>Y:wasser|V2]).
lebt(V3:mensch) :-
        V3 =+= V1:mensch & [lebensraum=>Z|V2],
        lebt(V4:lebewesen & [lebensraum=>Z|V5]).
trifft(V5:lebewesen, V6:lebewesen) :-
        V5 =+= V1:lebewesen & [lebensraum=>Z|V3],
        V6 =+= V2:lebewesen & [lebensraum=>Z|V4]
        lebt(V1:lebewesen & [lebensraum=>Z|V3]),
        lebt(V2:lebewesen & [lebensraum=>Z|V4]).
end_module.
```

erzeugt. Nach

```
set_current_module(main).
```

können nun Anfragen beantwortet werden. Wenn eine Anfrage selbst feature-Terme enthält, müssen diese zunächst auf Wohlsortiertheit überprüft und dann normalisiert werden. Dazu ist ein spezielles Anfrageprädikat `prove/1` vorgesehen, das die entsprechenden Operationen durchführt, bevor die Anfrage mit `call/2` gestellt wird.

Beispiel:

```
prove(trifft(X:lebewesen & [name=>"Aal"],
             Y:lebewesen & [name=>"Alge"])).
yes.
```

4.3 DUG-Vorübersetzer

Aufgabe des DUG-Vorübersetzers ist es, die Regeln einer Dependenz-Unifikationsgrammatik so in Horn-Klauseln zu überführen, daß deren Abarbeitung die Akzeption beziehungsweise Erzeugung von Sätzen bewirkt. Dazu müssen die Literale der Regeln um zusätzliche Argumente ergänzt werden und die Prädikate zur Akzeption von Symbolen an den richtigen Stellen eingefügt werden.

4.3.1 Implementation des DUG-Vorübersetzers

Die Übersetzung wird von einem Programm durchgeführt, das als Datei `dugpp.lop` abgelegt ist. Die Transformationsprozedur hält sich im wesentlichen an das Transformationsschema aus dem zweiten Kapitel dieser Arbeit und kann dem Ausdruck im Anhang entnommen werden.

Da der Vorübersetzer ein Programm vorgelegt bekommt, das DUG-Regeln enthalten kann, im allgemeinen aber auch normale Prolog-Klauseln enthält, müssen zunächst die DUG-

Regeln identifiziert werden. Diese sind leicht an dem Ableitungsoperator ':>' zu erkennen und werden allein der Transformation unterzogen.

Jede Regel bekommt von ihrer aufrufenden Regel einen Eingabesatz, dargestellt als Liste von Termen, vorgesetzt, aus dem sie ihre Wörter akzeptieren muß. Der nach der Akzeption verbleibende Rest stellt den Ausgabesatz der Regel dar, den sie an die aufrufende Regel zurückgibt. Variablen, die diese beiden Listen aufnehmen, werden als zusätzliche Argumente an den Regelkopf angehängt.

$$L(E, A) :- L_1(E, I_1), L_2(I_1, I_2), \ldots, L_n(I_n, A).$$

Gleiches gilt grundsätzlich auch für die Literale des Regelrumpfs. Da der Regelrumpf jedoch, unter anderem wegen der zusätzliche Operatoren ==> und ?, recht komplex aufgebaut sein kann, wird er zunächst durch einen Zerteiler rekursiv in seine einzelnen Literale zerlegt. Dabei ist zu berücksichtigen, daß der Eingabesatz des Regelkopfes zugleich die Liste ist, die vor der Abarbeitung des ersten Literals des Rumpfes vorliegt. Deren Ergebnisliste stellt die Eingabe für das zweite Literal im Rumpf dar, dessen Ergebnis wiederum die Eingabe für das dritte und so weiter bis zum letzten Literal des Rumpfes, dessen Ausgabe dem Ergebnis der gesamten Regel entspricht und damit mit der Ausgabe des Regelkopfes gleichzusetzen ist. Diesem Zusammenhang wird Rechnung getragen, indem bei der Zerteilung auch die Anschlußvariablen an die jeweiligen Bestandteile des Regelrumpfes weitergereicht werden.

Ein einzelnes Literal `LIn` wird, wenn es nicht durch ==> als Verweis gekennzeichnet wurde, durch den Aufruf des Akzeptionsprädikats `accept/3` gefolgt von dem um zwei Argumente ergänzten Literal `LOut` ersetzt. Die Argumente zu `accept/3` sind das zu akzeptierende Symbol `LIn` (eben das ursprüngliche Literal) zusammen mit seiner Eingabeliste `SIn`, und der Liste `SInter` als Zwischenspeicher für den Satz, der in dieser Form dem Aufruf der Regel zu `LOut` als erstes zusätzliches Argument dient. Das zweite zusätzliche Argument `SOut`, durch dessen Ergänzung `LOut` aus `LIn` erzeugt wurde, stellt die Ausgabe des urpsrünglichen Literals dar.

Die Behandlung des Verweises ist trivial, da hierbei lediglich die Ein- und Ausgabeliste an die aufgerufene Regel weitergereicht werden müssen.

Das leere Symbol `[]` hat auf den Eingabesatz keinen Einfluß und führt auch nicht zum Aufruf weiterer Regeln; es wird daher einfach durch `true` ersetzt.

Schließlich wird der Ableitungsoperator ':>' gegen ':-' ausgetauscht, so daß die entstandenen Regeln den üblichen Prolog-Regeln gleichen.

Eine recht nützliche Erweiterung der Transformationsvorschriften versieht jedes Literal mit einem zusätzlichen Argument, das für die Protokollierung des Akzeptionsvorgangs sorgt und so einen Ableitungsbaum erzeugt, der den Dependenzgraphen des akzeptierten Satzes darstellt. Zwar kann sich ein Benutzer der Grammatik genau wie bei DCG's den Baum selbst konstruieren, die automatische Erzeugung stellt jedoch besonders während der

Erprobungsphase der Grammatik ein brauchbares Hilfsmittels dar. Auf die Funktionsweise dieser Baumkonstruktion möchte ich nicht genauer eingehen, es sei nur soviel gesagt, daß sie eine geschachtelte Listenstruktur erzeugt, die mittels eines speziellen Prädikats pp (für pretty print) in übersichtlicher Form ausgegeben werden kann. Da der Akzeptionsvorgang an der Konstruktion des Baums beteiligt ist, wird das Prädikat `accept` um drei Argumente ergänzt, so daß sich die Stelligkeit von drei auf sechs erhöht.

4.3.2 Prädikate zur Laufzeit des transformierten Programmes

Das einzige zur Laufzeit des transformierten Programmes mit DUG-Regeln benötigte Prädikat ist das zur Akzeption eines Symbols aus dem Eingabesatz. Da die Akzeption der Symbole nicht in der Reihenfolge ihres Auftretens innerhalb des Satzes erfolgen muß, muß die Prozedur in der Lage sein, das Symbol an jeder beliebigen Stelle des vorliegenden Satzes zu entdecken und zu entfernen. Dies stellt jedoch kein Problem dar; die Implementation muß daher nicht weiter erläutert werden (und kann der Datei dugrt.lop im Anhang entnommen werden). Dort findet sich auch das Ausgabeprädikat pp für den erzeugten Dependenzgraph, das bei Bedarf importiert werden kann.

4.3.3 Benutzung des Vorübersetzers

Obwohl der Vorübersetzer für Dependenz-Unifikationsgrammatiken in den ersten Lauf des Vorübersetzers für osf-Prolog eingebunden werden kann, soll er auch seperat zu verwenden sein.

Bei der Implementation eines Programmes, das DUG-Regeln enthält, ist zu beachten, daß der Vorübersetzter die Literale dieser Klauseln um zwei beziehungsweise drei Argumente erweitert. Das muß der Benutzer berücksichtigen, indem er bei der Deklaration der Prädikate die Stelligkeit entsprechend erhöht.

Nach dem Start des Lopster-Systems und der Erstellung des zu übersetzenden Moduls verläuft die Übersetzung wie folgt:

```
set_current_module(dugpp).
dug(<source>, <target>).
```

Das übersetzte Programm kann nun eingelesen werden und steht zu Anfragen bereit. Wegen der Ergänzung der zusätzlichen Argumente ist eine Anfrage an eine DUG-Regel über das Startsymbol der Grammatik wiederum um den zu akzeptierenden Satz, den erwarteten Restsatz, der im allgemeinen die leere Liste sein wird, und eine Variable zur Aufnahme des Dependenzbaums zu ergänzen.

Beispiel:

Das Programm

```
module beispiel.
from dugrt import accept/6, pp/1.
private s/3, v/6.
body.
s :> v(_, verb, 'Prädikat').

v(schenkt, verb, Rolle) :>
     v(_, subst, 'Subjekt'),
     v(_, subst, 'Empfänger'),
     v(_, subst, 'Objekt').

v(schläft, verb, Rolle) :>
     v(_, subst, 'Subjekt').

v('Hans', subst, Rolle) :> [].
v('Peter', subst, Rolle) :> [].

v('Buch', subst, Rolle) :>
     v(_, art, 'Artikel').

v(ein, art, Rolle) :> [].
end_module.
```

wird durch den Vorübersetzer in

```
module beispiel.
from dugrt import accept/6, pp/1.
private s/3, v/6.
body.
s(S1, S2) :>
     accept(v(_, verb, 'Prädikat'), S1, I1),
     v(_, verb, 'Prädikat', I1, S2).

v(schenkt, verb, Rolle, S1, S4)) :-
     accept(v(_, subst, 'Subjekt'), S1, I1),
     v(_, subst, 'Subjekt', I1, S2),
     accept(v(_, subst, 'Empfänger'), S2, I2),
     v(_, subst, 'Empfänger', I2, S3),
     accept(v(_, subst, 'Objekt'), S3, I3),
     v(_, subst, 'Objekt', I3, S4).

v(schläft, verb, Rolle, S1, S2)) :-
     accept(v(_, subst, 'Subjekt'),S1,I1),
     v(_,subst,'Subjekt', I1, S2).

v('Hans', subst, Rolle, S, S).
v('Peter', subst, Rolle, S, S).

v('Buch', subst, Rolle, S1, S2)) :-
```

```
        accept(v(_, art, 'Artikel'), S1, I1),
        v(_, art, 'Artikel', I1, S2).

    v(ein, art, Rolle, S, S).
    end_module.
```

transformiert. Die Anfrage

```
    ?- s([  v('Hans', subst, _),
        v(schenkt, verb, _),
        v('Peter', subst, _),
        v(ein, art, _),
        v('Buch', subst, _)], [], Baum).
```

liefert dann

```
    Baum = [v(schenkt, subst, 'Prädikat'),
        [v('Hans', subst, 'Subjekt'), [],
        v('Peter', subst, 'Empfänger'), [],
        [v('Buch', subst, 'Objekt'),
        [v(ein, art, 'Artikel'), []]]]]
```

als Dependenzbaum, der mittels `pp(Baum)` in eingerückter Form ausgegeben werden kann.

```
    v(schenkt, subst, 'Prädikat')
        v('Hans', subst, 'Subjekt')
        v('Peter', subst, 'Empfänger')
        v('Buch', subst, 'Objekt')
            v(ein, art, 'Artikel')
```

■

5 Ausblick

Im Ausblick möchte ich zum einen Schwächen auf der Dependenzgrammatik hindeuten und Ideen präsentieren, wie diese beseitigt werden könnten, zum anderen aber auch dem Eindruck entgegentreten, osf-Prolog sei ein speziell auf die Computerlinguistik abgestimmtes Werkzeug, das in anderen Aufgabenbereichen keine adäquate Verwendung finden kann.

5.1 Überlegungen zur Dependenz

Bei der Formulierung der Beispielgrammatik sind einige linguistische Phänomene in Erscheinung getreten, die sich nur schwer mit Hilfe der Dependenz-Unifikationsgrammatik lösen ließen. Zwar lassen sich in den meisten Fällen Probleme durch die Aufnahme speziell darauf abgestimmter zusätzlicher Dependenzregeln beheben, doch ist ein zu starkes Anwachsen eines damit auch zunehmend redundanter werden Lexikons nicht wünschenswert. Auch muß man sich, je schwieriger und verschlungener die Darstellung einer Lösung wird, die Frage stellen, ob der Formalismus überhaupt noch als adäquat zu bezeichnen ist.

5.1.1 Schwächen der Dependenzgrammatik

Es stellt sich zunächst die Frage, ob man mit der strengen Formalisierung, insbesondere wenn sie so kurz und knapp ausfällt wie in diesem Fall, den Erfordernissen einer natürlichen Sprache überhaupt gerecht wird.

Ich möchte dazu vier Fälle herausgreifen, bei denen die Nachteile einer Dependenzgrammatik besonders augenscheinlich werden.

5.1.1.1 Das Problem der Adjektive

Vom syntaktische Standpunkt aus betrachtet hängt ein Adjektiv von seinem Substantiv ab. "schöne Blume" etwa würde durch zwei Regeln

```
v('Blume', substantiv) :> v(_, adjektiv).
v(schön, adjektiv) :> [].
```

analysiert, wobei v(_, adjektiv) die syntaktische Valenz des Substantivs Blume darstellt. Logisch-semantisch betrachtet jedoch ist "schön" als Prädikat aufzufassen, das über ein Argument, z.B. die Blume, eine Aussage trifft, nämlich daß es schön ist.

Sehr viel mehr noch kommt diese Verhältnisumkehrung darin zum Ausdruck, daß man eigentlich nicht dem Adjektiv selbst eine Bedeutungskategorie zuordnen kann, sondern vielmehr von seinem Bezugswort, das es beschreibt, die Zugehörigkeit in Form einer

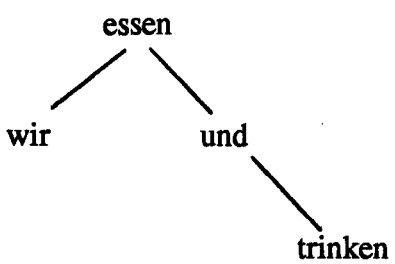
Selektionsbeschränkung fordert. So ist zum Beispiel das Adjektiv 'grün' nicht selbst ein materielles Objekt, sondern die Menge der Dinge, die grün sein können, ist die der materiellen Objekte. Die gerichtete, das heißt asymmetrische Relation der Dependenz ist also nicht unbedingt geeignet, syntaktische und semantische Zusammenhänge gleichzeitig adäquat auszudrücken.

5.1.1.2 Das Problem der Konjunktionen

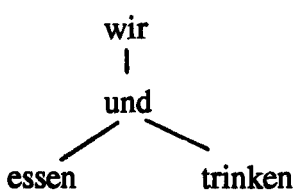
Konjunktionen werden in der Regel dazu benutzt, Häufungen gleichartiger Satzbestandteile aneinader zu binden. Im Deutschen etwa lassen sich mittels 'und' die verschiedensten Konstruktionen bilden:

Wir essen und trinken.
Sehr geehrte Damen und Herren!

Keine denkbare Dependenzstruktur gibt den Inhalt angemessen wider:

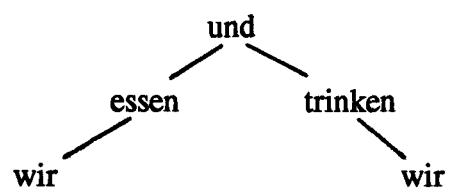


(wer trinkt?)



(warum hängt 'und' von 'wir' ab und warum regieren nicht 'essen' und 'trinken'?)

Selbst der Ansatz, den Satz als elliptisch zu betrachten und das fehlende 'wir' zu ergänzen vermag nicht zu befriedigen, da dann in



nicht zum Ausdruck kommt, daß die beiden 'wir' den identischen Personenkreis bezeichnen,

denn es könnte ja 'wir' einmal sie und ich und im anderen Fall er und ich bedeuten.

Andere Beispiele sind:

Oma schenkt Vater und Mutter ein Buch.
Oma schenkt Vater ein Buch und Mutter eine Vase.
Oma schenkt und kauft Mutter ein Buch und eine Vase.
Oma schenkt Vater ein Buch und kauft Mutter eine Vase.

Ähnliche Probleme wirft die Konjunktion 'als' im Zusammenhang mit Komparationen auf, wie folgende Beispiele zeigen:

Hans läuft schneller als Peter.
Peter kann schneller essen als trinken.

Auch wenn es sich bei diesen Beispielen nicht um bestes Deutsch handelt, sind solche Konstruktionen doch möglich und zeigen, wie schwer es ist, Konjunktionen adäquat in einer Dependenzgrammatik unterzubringen.

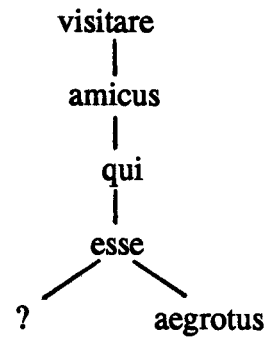
5.1.1.3 Das Problem der Relativsätze

Relativsätze sind Nebensätze und werden durch ein Relativpronomen eingeleitet. Das Relativpronomen bezieht sich als Stellvertreter auf sein Bezugswort im Hauptsatz, was sich im Lateinischen in der Kongruenz von Numerus und Genus zwischen den beiden niederschlägt. Der Kasus des Relativpronomens richtet sich jedoch nach seiner Rolle im Nebensatz, wird also durch die syntaktische Valenz des Prädikats des Relativsatzes bestimmt ([Holtermann 78]). Daraus ergibt sich nun folgendes Problem:

Der Relativsatz, der offensichtlich von seinem Bezugswort im Hauptsatz abhängig ist, braucht als Nebensatz in der Dependenzgrammatik ein oberstes Regens, das selbst Dependens des Bezugswortes ist. Wählt man das Relativpronomen als Regens, so läßt sich zwar die Kongruenz von Numerus und Genus bequem ausdrücken, jedoch ist der Kasus von dem Prädikat des Nebensatzes abhängig, das dann als Dependens des Relativpronomens eingeordnet werden müßte.

Beispiel: (zur Abwechslung mal wieder in Latein)

visito amicum, qui aegrotus est.
(Ich besuche einen Freund, der krank ist.)




```
v(amicus, _:substantiv & [num=>N, gen=>G]) :>
     v(qui, _:relativpronomen, [kas=>K, num=>N, gen=>G]).
v(qui, _:relativpronomen & []) :>
     v(_, _:verb & []).
v(esse, _:verb & [num=>N]) :>
     v(_, _:nomen & [kas=>nom, num=>N, gen=>G]),
     v(_, _:nomen & [kas=>nom, num=>N, gen=>G]).
```

Dies wird jedoch weder der Rolle des Prädikats gerecht, da es auf einmal von einer seiner Valenzen regiert wird, noch läßt sich der Kasus aus dem entsprechenden Dependens des Prädikats nach oben an sein Regens weiterleiten, weil nicht von vornherein feststeht, welches Dependens durch das Relativpronomen ersetzt wird.

Die Alternative stellt die Betrachtung des Verbs als Dependens des Bezugsworts dar, jedoch muß auch hier ausgedrückt werden, daß die Rolle einer Valenz des Verbs durch ein Relativpronomen eingenommen wird und dieses in Numerus und Genus mit dem Regens des Prädikats übereinstimmt. Diese Information muß also gewissermaßen durch den Knoten des Verbs hindurchgereicht werden, wodurch jedoch neue Regeln für Verben in der Rolle des Prädikats in Relativsätzen notwendig wären. Als Lösung bietet sich folgende Regel an:

```
v(amicus, _:substantiv & [num=>N, gen=>G]) :>
     (
     v(_, _:verb & []) :>
          v(qui, _:relativpronomen & [num=>N, gen=>G])
     ).
v(esse, _:verb & [num=>N]) :>
     v(_, _:nomen & [kas=>nom, num=>N, gen=>G]),
     v(_, _:nomen & [kas=>nom, num=>N, gen=>G]).
```

Es sei jedoch ausdrücklich darauf hingewiesen, daß dies keine gültige DUG-Regel ist.

5.1.1.4 Das Problem der phrasiologischen Ausdrücke

Zwar lassen sich in einer Dependenzgrammatik Idiome wie "es regnet Bindfäden" leicht ausdrücken, zum Beispiel in DUG:

```
v(regnen, verb, _) :>
     v(es, pronomen, 'Subjekt'),
     v('Bindfäden', substantiv, 'Objekt').
```

doch ist zu bedenken, daß in einer Dependenzregel immer nur die direkte Abhängigkeit über eine Stufe ausgedrückt werden kann. Abhängigkeiten, die sich über mehrere Stufen erstrecken und die, obwohl sie eigentlich ungrammatisch sind, fest in einer Sprache verwurzelt sind, wie zum Beispiel

nach Hause gehen

lassen sich nicht in Form einer Dependenzregel adäquat ausdrücken. Solche phrasiologischen Ausdrücke stellen in einem Dependenzgraphen größere, aber elementare, das heißt unauftrennbar zusammengesetzte Teilbäume dar und sollten daher auch als eine Regel im Lexikon abgelegt werden können.

Ein entsprechender Eintrag könnte in etwa so aussehen:

```
v(gehen, verb) :>
     (v(nach, praeposition) :>
          v('Hause',substantiv)).
```

Nach Prolog übersetzt entspräche das einem Konstrukt der Form

```
a :- (b :- c).
```

5.1.2 Konkomitanz statt Dependenz

Wie es scheint, resultieren alle genannten Probleme aus dem Zwang, den Zusammenhang der Wörter eines Satzes in Form eines Abhängigkeitsbaums darzustellen. Man kann sich des Eindrucks nicht erwehren, daß die Sätze einer natürlichen Sprache keineswegs immer eine Aussage repräsentieren, deren ausgedrückte Verhältnisse der Elemente untereinander sich in Form eines Baums darstellen lassen. Viele Beziehungen und Teilaussagen eines Satzes können zumindest inhaltlich als nebengeordnet betrachtet werden, so daß eine streng gerichtete Abhängigkeitsrelation nicht unbedingt zur semantisch adäquaten Darstellung geeignet scheint. Selbst wenn also die Syntaxanalyse durch eine Dependenzgrammatik durchgeführt wird, wäre zur inhaltlichen Darstellung immer noch ein zusätzlicher Verarbeitungsschritt notwendig, der den Baum in eine geeignete Struktur konvertiert. Es wäre aber wünschenswert, eine solche zweistufige Verabeitung nicht durchführen zu müssen, zumal da anzunehmen ist, daß sie nicht dem natürlichen Prozeß des Sprechens und Verstehens nahekommt, also die Regelmäßigkeiten, die in der Grammatik einer natürlichen

Sprache stecken und die sich der Mensch zunutze macht, nicht voll ausschöpft.

Der streng hierarchische Begriff der Dependenz läßt sich Abschwächen zu der freieren *Konkomitanz* [Engel 77], dem miteinander Vorkommen der Wörter innerhalb eines Satzes. Eine solche Konkomitanz läßt sich veranschaulichen, wenn man die Wörter eines (vorverarbeiteten) Satzes als gleichartige und prinzipiell gleichberechtigte Atome auffaßt, von denen jedes für sich individuelle Bindungsstellen besitzen. Jede solche Bindungsstelle verfügt über eine Spezifikation und soll hier Rezeptor genannt werden. Zwei Atome können nur dann eine Bindung eingehen, wenn jedes über einen (freien) Rezeptor verfügt, der zu dem jeweils anderen paßt. Was dieses 'Passen' genau heißt, ist zu klären, es liegt jedoch nahe, die Unifikation als Mechanismus einzusetzen, der einerseits über ein 'Passen' entscheidet und andererseits, falls die Rezeptoren nur teilweise spezifiziert sind, notwendige Angleichungen vornimmt. So eine Angleichung ist als genauere Spezifikation oder Einschränkung des Rezeptors und damit des Atoms aufzufassen und kann, bei einer Implementierung in Prolog zum Beispiel durch Variablen, an andere Rezeptoren desselben Atoms propagiert werden.

Beispiel:

```
X:relativpronomen&[name=>qui, kas=>K, num=>N, gen=>G,
          bezugswort=>Y:substantiv & [num=>N, gen=>G],
          praedikat=>Z:verb & [ergänzung=>X]
```

Die Aufgabe der Analyse eines Satzes wird dann zu dem Problem, aus seinen Atomen größere zusammenhängende Gebilde (Moleküle) zu konstruieren. Ob dazu alle Atome eines Satzes in irgendeiner Form zusammenhängen müssen oder ob auch isolierte Teilgruppen zur Repräsentation einer Nebenaussage oder vielleich auch übergeordnete Strukturen wie ganze Texte zugelassen werden sollen, bleibt der linguistischen Theorie überlassen und stellt kein technisches Problem dar.

Mögliche Darstellungen dieser Wörter als Atome mit Valenzen in osf-Prolog wären zum Beispiel:

```
X:substantiv & [name=>'Blume', attribut=>Y:adjektiv].
Y:adjektiv & [name=>schön, objekt=>X:substantiv].

X:verb & [name=>gehen,
     richtung=>Y:praeposition & [name=>nach,
          ort=>Z:substantiv & [name=>'Haus']]]
```

Allerdings muß berücksichtigt werden, daß in osf-Prolog zyklische Strukturen nicht erlaubt sind. Wie sich aus solchen Atomen systematisch ein Molekül konstruieren läßt, sei dahingestellt, vielleicht wird man aber im Gebiet des *constraint propagation* fündig.

Rein intuitiv muß eine solche netzartige Darstellungsart in jedem Fall als mächtiger angesehen werden, da die Baumstruktur nur einen Spezialfall davon realisiert.

5.2 osf-Prolog als Datenbanksprache

Um nicht den Eindruck zu erwecken, die feature-Logik sei ein speziell auf die Erfordernisse der Computerlinguistik abgestimmtes Werkzeug, möchte ich an dieser Stelle noch einen anderen Einsatzbereich vorstellen: den der Datenbanken.

Obwohl die Eignung von Prolog als Datenbankanfragesprache bereits diskutiert wurde (z.B. in [Lockemann 87]), ergeben sich aus der Verwendung von osf-Prolog deutliche Verbesserungsmöglichkeiten. Dazu soll anhand eines kleinen Beispiels vorgeführt werden, wie sich osf-Prolog sowohl als DDL (data description language) als auch DML (data manipulation language) anbietet.

5.2.1 Einleitung und Begriffsklärung

Die elementare Datenstruktur des Relationenmodells ist die *Relation*. Aus der Sicht der Datenbanktheorie ist eine solche Relation eine zweidimensionale Tabelle, deren Spalten *Attribute* und deren Zeilen *Tupel* genannt werden. Die Tupel machen die Elemente der Relation aus, während die Attribute den Komponenten der Tupel Namen geben. Den Attributen wird im allgemeinen eine *Domäne* zugeordnet, die den Wertebereich der Komponenten einschränkt.

Unter der *Intension* einer Relation versteht man ihre zeitunabhängigen Bestandteile, das sind die Attributnamen und Wertebereiche sowie damit zusammenhängende Integritätsbedingungen, während mit der *Extension* die aktuelle Menge von Tupeln gemeint ist, die mit der Zeit variieren kann.

Der Begriff Relation ist insofern aus der Sicht der Mathematik gerechtfertigt, als daß kein Tupel darin mehrfach enthalten sein darf und die Reihenfolge der Tupel keine Rolle spielt, die Relation also als eine Menge von geordneten Tupeln im Sinne der Mengentheorie aufgefaßt werden kann.

Die Beispiele zu den nachfolgenden Abschnitten sind durchweg [Date 81] entnommen. Da allen Beispielen dieselben drei Relationen zugrunde liegen, werden diese hier eingeführt und im folgenden als gegeben vorausgesetzt:

(Der besseren Vergleichbarkeit wegen habe ich die englischen Bezeichnungen beibehalten.)

S (Lieferanten, englisch suppliers)

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris

P (Teile, englisch parts)

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London

SP (Lieferungen, englisch shipments)

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S2	P1	300
S2	P2	400
S3	P2	200

5.2.2 Prädikatenlogik erster Ordnung und das Relationenkalkül

Das Konzept des Relationenkalküls als spezielle Form des Prädikatenkalküls wurde zuerst von [Codd 71] vorgeschlagen. Die Aufgabe des Kalküls ist es, eine formal logische Grundlage zu einer Anfragesprache für relationale Datenbanken zu bilden.

Die genaue Definition des Relationenkalküls soll hier nicht weiter diskutiert werden: Der interessierte Leser sei für einen einfachen Überblick auf [Date 81] und für eine grundlegende Einführung auf [Codd 71] verwiesen.

Den für meine Zwecke relevanten Zusammenhang zwischen der einsortigen Prädikatenlogik erster Ordnung und dem Relationenkalkül möchte ich im folgenden kurz skizzieren: (für eine ausführliche Diskussion vergleiche [Codd 77])

> Zu jedem Prädikat p einer Signatur der einsortigen Prädikatenlogik erster Ordnung existiert eine Relation R im Relationenkalkül derart, daß jeder Argumentstelle des Prädikats genau ein Attribut der Relation zugeordnet wird und die atomare Formel p über seine Argumente formuliert genau dann wahr ist, wenn das entsprechende Tupel bestehend aus denselben Argumenten als Attribute Element der dazugehörigen Relation ist.

Die Zuordnung der Argumente eines Prädikats zu den Attributen der Relation erfolgt im allgemeinen über die Reihenfolge: Das erste Argument entspricht dem ersten Attribut, das zweite dem zweiten und so weiter.

Dabei ist zu berücksichtigen, daß Relationen in der ersten Normalform vorliegen, was soviel heißt, daß alle Attribute elementar sind. Bei der Umsetzung in die Prädikatenlogik resultiert

daraus eine flache Termstruktur, das heißt, die Argumente der Prädikate sind allesamt einstellig, bestehen also nur aus Konstanten und Variablen.

Beispiel: s(S1, Smith, 20, London) ⇔ (S1, Smith, 20, London) ∈ S

Die Anfrage, ob ein Tupel (S1, Smith, 20, London) Element der Relation S ist, kann so auf den Beweis von s(S1, Smith, 20, London) zurückgeführt werden. Mit Hilfe von Variablen und logische Verknüpfungen können auf diese Weise Formeln und damit komplexe Anfragen formuliert werden.

Dieser einfache Zusammenhang läßt sich bereits ausnutzen, um Datenbankanfragen in Prolog zu formuliern. Voraussetzung hierfür ist, daß die Elemente der Relation, die Tupel, für das Prologsystem in irgendeiner Form verfügbar sind, sei es, daß sie explizit als Fakten hinterlegt sind, oder daß sie mit Hilfe von Regeln und geeigneten Schnittstellenprädikaten bei Bedarf der Datenbank entnommen werden. Wie solche Schnittstellen realisiert werden können, soll hier nicht Gegenstand der Untersuchung sein - Literatur zu diesem Thema ist zur genüge vorhanden, z.B. in [Lockemann 87].

Beispiele:[2]

Nenne die Nummern aller Lieferanten aus Paris mit einem Status > 20!

```
?- s(SN, _, STATUS, 'Paris'), STATUS > 20.
```
[3]

Nenne die Nummern und Städte aller Lieferanten von Teil 2!

```
?- s(SN, _, _, CITY), sp(SN, 'P2', _).
```

Nenne die Namen aller Lieferanten, die mindestens ein rotes Teil liefern!

```
?-s(SN,SNAME,_,_), sp(SN,PN,_), p(PN,_,'Red',_,_).
```

Wer sich die Mühe gemacht hat, die Richtigkeit dieser Beispiele zu überprüfen, dem wird sicherlich ein Nachteil solcher Anfrageformulierungen aufgefallen sein: Man muß schon die Definition der Relationen, insbesondere die Reihenfolge ihrer Attribute, genau vor Augen haben, um die Bedeutung der Argumente in der Prolog-Anfrage erschließen zu können. Darüberhinaus wirken die anonymen Variablen zum Überspringen der nicht interessierenden Argumente recht störend.

Abgesehen von den Schwächen in der Anfrageformulierung wird auch der Mehrsortigkeit des Relationenmodells, die in der Möglichkeit, Wertebereiche für Attribute anzugeben, zum

---

[2]Die Formulierung in Prolog ist eng verwandt mit dem domänenorientierten Relationenkalkül [Lacroix 77], das im Gegensatz zum tupelorientierten Relationenkalkül Codd's Variablen für die einzelnen Attribute vorsieht. Sogenannte Tupelvariablen finden hierbei keine Verwednung.

[3]Prolog liefert zunächst nur das erste Tupel als Lösung. Alle übrigen müssen durch Eingabe von ';' erfragt werden.

Ausdruck kommt, keine Rechnung getragen. Es wird sich zeigen, daß sich diese Mängel durch Verwendung von osf-Prolog weitgehend beheben lassen.

5.2.3 Das Relationenmodell und osf-Prolog

Zunächst möchte ich noch einmal auf die Definition einer Relation im Sinne des Relationenmodells zu sprechen kommen.

Definition (Relation, nach [Date 81])
Gegeben sei eine Sammlung von Wertemengen $D_1, D_2, \ldots, D_n$. Dann ist R eine Relation über den Mengen $D_1, D_2, \ldots, D_n$, wenn R eine Teilmenge des kartesischen Produkts $D_1 \times D_2 \times \ldots \times D_n$ ist.

Anders formuliert ist eine Relation eine Menge von geordneten n-Tupeln $(d_1, \ldots, d_n)$ mit $d_1 \in D_1, \ldots, d_n \in D_n$.

Beispiel:

Als Wertemengen seien string und integer als gegeben vorausgesetzt. Dann sind S, P, und SP die Relationen aus dem Beispiel mit den Intensionen

$S \subseteq \text{string} \times \text{string} \times \text{integer} \times \text{string},$
$P \subseteq \text{string} \times \text{string} \times \text{string} \times \text{integer} \times \text{string}$
$SP \subseteq \text{string} \times \text{string} \times \text{integer}$

und den Extensionen

S := {(S1, Smith, 20, London),
(S2, Jones, 10, Paris),
(S3, Blake, 30, Paris)}

P := {(P1, Nut, Red, 12, London),
(P2, Bolt, Green, 17, Paris),
(P3, Screw, Blue, 17, Rome),
(P4, Screw, Red, 14, London)}

SP := {(S1, P1, 300),
(S1, P2, 200),
(S1, P3, 400),
(S2, P1, 300),
(S2, P2, 400),
(S3, P2, 200)}.

Nun kann ein feature-Typ von osf-Prolog als Intension einer Relation betrachtet werden, wenn man die feature label als Attributnamen, die Typen der feature values als

Wertebereiche und den Typ selbst als das kartesische Produkt der Wertebereiche auffaßt.

```
type supplier & [sn=>string, sname=>string,
     status=>integer, city=>string]
type part & [pn=>string, pname=>string, color=>string,
     weight=>integer, city=>string]
type shipment & [sn=>string, pn=>string, qty=>integer]
```

Die Sortendeklarationen von string und integer werden als gegeben vorausgesetzt.

Die Vereinbarung einer Relation selbst wird durch eine Prädikatsdeklaration dem System bekanntgegeben.

```
sort s(supplier).
sort p(part).
sort sp(shipment).
```

Ein offensichtlicher Vorteil dieser Methode ist, daß unterschiedliche Relationen mit identischen Intensionen deklariert werden können.

Die Tupel der Relationen als Extension können schließlich als Fakten eingegeben werden:

```
s(X    &    [sn=>"S1",    sname=>"Smith",    status=>20,
     city=>"London"]).
s(X    &    [sn=>"S2",    sname=>"Jones",    status=>10,
     city=>"Paris"]).
...
p(X & [pn=>"P1", pname=>"Nut", color=>"Red",
     weight=>12, city=>"London"]).
p(X & [pn=>"P2", pname=>"Bolt", color=>"Green",
     weight=>17, city=>"Paris"]).
...
sp(X & [sn=>"S1", pn=>"P1", qty=>300]).
sp(X & [sn=>"S1", pn=>"P2", qty=>200]).
...
```

Diese Bestandteile eines osf-Prolog-Programms entsprechen ziemlich genau den Operationen zur Relationsdefinition und der Datenerfassung einer relationalen Datenbank. Als DDL und DML habe ich hier stellvertretend SQL verwendet.

```
CREATE TABLE S (SN (STRING), SNAME (STRING),
     STATUS (INTEGER), CITY (STRING))
CREATE TABLE P (PN (STRING), PNAME (STRING),
     COLOR (STRING), WEIGHT (INTEGER), CITY (STRING))
CREATE TABLE SP (SN (STRING), PN (STRING), QTY (INTEGER))


INSERT INTO S (S#, SNAME, STATUS, CITY):
     <'S1', 'Smith', 20, 'London'>
INSERT INTO S (S#, SNAME, STATUS, CITY):
     <'S2', 'Jones', 10, 'Paris'>
...
```

```
INSERT INTO P (P#, PNAME, COLOR, WEIGHT, CITY):
     <'P1', 'Nut', 'Red', 12, 'London'>
INSERT INTO P (P#, PNAME, COLOR, WEIGHT, CITY):
     <'P2', 'Bolt', 'Green', 17, 'Paris'>
...

INSERT INTO SP (S#, P#, QTY):
     <'S1', 'P1', 300>
INSERT INTO SP (S#, P#, QTY):
     <'S1', 'P2', 200>
...
```

Anfragen können nun alternativ in Prolog und in SQL gestellt werden:

Nenne alle Paare von Lieferantennummern, deren Lieferanten aus derselben Stadt kommen!

SQL:

```
SELECT     FIRST.S#, SECOND.S#
FROM       S FIRST, S SECOND
WHERE      FIRST.CITY = SECOND.CITY
AND        FIRST.S# < SECOND.S#
```

osf-Prolog:

```
s(X & [s#=>N1, city=>C]),s(Y & [s#=>N2, city=>C]), N1<N2.
```

[4]

Nenne die Nummern der Lieferanten in derselben Stadt wie Lieferant 1!

SQL:

```
SELECT     S#
FROM       S
WHERE      CITY =
           (SELECT CITY
           FROM S
           WHERE S# = 'S1')
```

osf-Prolog

```
s(X & [s#=>"S1", city=>C]), s(X & [s=>N, city=>C]).
```

5.2.4 Abschließende Bemerkungen

Diese Beispiel haben gezeigt, daß osf-Prolg bei der Formulierung von Anfragen gegenüber herkömmlichem Prolog eine bessere Lesbarkeit bietet. Allerdings ist dazu das Prinzip der

[4] Die Variable X in dieser Anfrage entspricht der Tupelvariablen des Tupel-orientierten Kalküls.

Ordnungssortiertheit nicht ausgenutzt worden: Ähnliche Ergebnisse hätten sich auch mit einer wesentlich einfacheren Variante von Prolog, in der lediglich die Argumente benannt werden und daher in ihrer Reihenfolge und Anzahl nicht fixiert sind, erzielt werden können.

Der eigentliche Gewinn liegt in der Datendefinition. Nicht nur macht ordnungssortiertes Prolog diese überhaupt erst möglich, sondern es können auch, wenn auch nur sehr elementare, Konsistenzbedingungen für die einzelnen Attribute einer Relation angegeben werden, und zwar die der Festlegung einer Domäne durch eine Sorte.

Die Möglichkeiten der Deduktion, insbesondere die der Vererbung, die osf-Prolog zusätzlich bietet, werden doch durch solch einfache Datenbankanwendungen nicht ausgeschöpft.

6 Bewertung

In den vorangegangenen Kapiteln wurde ein linguistisches Werkzeug erarbeitet, das von den definiten Klauselgrammatiken aus Standard-Prolog in drei Punkten wesentlich abweicht:

- das Prinzip der Ordnungssortiertheit steht der einsortigen Logik gegenüber
- die Verfügbarkeit von feature-Termen anstelle von Termen mit fester Stelligkeit und Reihenfolge der Argumente
- die Dependenz als satzbeschreibende Relation anstelle der Konstituenz

Vor- und Nachteile dieser drei Neuerungen sollen im folgenden kurz diskutiert werden.

6.1 Nutzen der Ordnungssortiertheit

Die Ordnungssortiertheit wird bei der Formulierung der Beispielgrammatik für zwei Ziele eingesetzt:

- zur Strukturierung der Wortarten und
- zur Klassifikation der Wortbedeutungen.

Die Strukturierung der Wortarten gestattet eine gewisse Eleganz bei der Formulierung der syntaktischen Valenzen. Zwar ließe sich die Sortenhierarchieinformation prinzipiell auch durch Deduktionsregeln ausdrücken, doch, wie folgendes Beispiel zeigt, verliert die Grammatik damit nicht nur an Lesbarkeit, auch die Ausführung wird wegen der Notwendigkeit zusätzlicher Deduktionsschritte erheblich verschlechtert.

Beispiel:

Während die einfache Regel

> Die syntaktische Valenz eines Verbs ist ein Substantiv, ein Adjektiv, ein Partizip oder ein Pronomen.

unter der Sortenhierarchie

```
substantiv < nomen, adjektiv < nomen,
partizip < nomen, pronomen < nomen
```

durch das einfache Fakt

```
valenz(verb, X:nomen).
```

ausgedrückt werden kann, benötigt man in einsortigem Prolog dazu vier Regeln.

```
valenz(verb, X) :- substantiv(X).
valenz(verb, X) :- adjektiv(X).
valenz(verb, X) :- partizip(X).
valenz(verb, X) :- pronomen(X).
```

■

Das gleiche gilt prinzipiell auch für die Klassifikation der Wortbedeutungen. Selektionsbeschränkungen, die die logisch-semantische Valenz der Wörter spezifizieren, können so direkt in die Grammatik integriert werden. Hier kann sogar noch intensiverer Nutzen aus der Ordnungssortiertheit (im Gegensatz zur bloßen Sortiertheit) gezogen werden, da eine halbwegs realistische Einteilung des Universums in Teilmengen wohl nur durch eine recht tiefe Hierarchie und unter Ausnutzung der Mehrfachvererbung dargestellt werden kann.

Der Nutzen der Ordnungssortiertheit wird daher wohl kaum in Frage zu stellen sein. Da man nicht gezwungen ist, sie zu verwenden, und es in der unsortierten Logik keine Entsprechungen außer der Darstellung durch explizite Deduktion gibt, lassen sich gegenüber einsortigem Prolog keine Nachteile finden.

6.2 Bewertung der features

Auch andere Arbeiten, die sich mit der Integration von features in die Logik befaßt haben, wurden durch die Computerlinguistik inspiriert [Ludwig 88]. Obwohl grundsätzlich, wie in Abschnitt 5.2 gezeigt, auch für andere Anwendungen von Vorteil, scheinen sie gerade in der Computerlinguistik zu einer Art Standard geworden zu sein, den man nicht mehr vermissen möchte [Shieber 86].

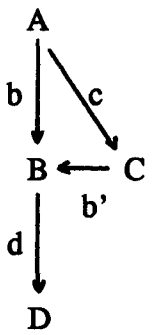
Bei genauerer Betrachtung unterscheiden sich die im Zusammenhang mit Sorten vorgestellten feature-Terme jedoch von den in der Computerlinguistik gebräuchlichen feature-Strukturen deutlich: Während eine feature-Struktur grundsätzlich über jedes feature verfügen darf, solange es nur einen innerhalb der Struktur eindeutigen Bezeichner trägt, und die feature values wieder beliebige feature-Strukturen sein können, ist die Zahl der möglichen features in feature-Termen durch die feature-Sorte fest vorbestimmt, und auch die Sorte ihrer feature values ist genau festgelegt. Wenn man allerdings auf eine strukturierte Sortenordnung verzichtet und nur eine feature-Sorte deklariert, die alle benötigten features umfaßt, macht sich der Unterschied kaum noch bemerkbar. Lediglich die Tatsache, daß in der Praxis während der Laufzeit eines osf-Prolog-Programms keine neuen features, deren Notwendigkeit nicht vorhersehbar war, eingeführt werden können, ist als Nachteil zu werten.

Auch wenn die Einführung von features die Ausdrucksstärke von Prolog nicht erhöht (wie in Abschnitt 2.1 gezeigt, werden feature-Terme auf gewöhnliche Terme abgebildet), so sind sie dennoch ein Beitrag zur adäquaten Repräsentation von Informationen und damit ein Schritt in Richtung auf die Konzepte der Wissensrepräsentation. Dies gilt insbesondere im

Zusammenhang mit der durch die Typsubsumption eingeführten Vererbung von Datenstrukturen, die z.B. in *KL-One*-artigen Sprachen oder auch in *Smalltalk*, einem Vertreter der objektorientierten Sprachen, wiederzufinden sind.

Auf eine schöne Eigenschaft sei noch hingewiesen: Durch die Verwendung von Koreferenzen ist es möglich, gerichtete azyklische Graphen (dags) mit beschrifteten Kanten direkt durch einen Term darzustellen.

Beispiel:



A & [b=>B & [d=>D], c=>C & [b'=>B]]

Ein äquivalente Darstellung ist in Prolog zwar prinzipiell möglich, jedoch bei weitem nicht so gut lesbar.

Gerade diese Möglichkeit von osf-Prolog könnte es auch zum geeigneten Werkzeug machen, die Satzanalyse mit anderen Mitteln als denen der Konstituenz- oder Dependenzrelation, die stets zu baumartigen Strukturbeschreibungen führen, in Angriff zu nehmen. (vgl. dazu Abschnitt 5.1)

6.3 Bewertung der Dependenz-Unifikationsgrammatik

Um Mißverständnisse zu vermeiden, möchte ich hier noch einmal darauf hinweisen, daß mit 'Dependenzgrammatik' und 'Dependenz-Unifikationsgrammatik (DUG)' meine persönliche Sichtweise und Definition gemeint ist. Andere Auffassungen, insbesondere die von [Hellwig 86], von der ich einige Konzepte und den Namen übernommen habe, bleiben davon unberührt.

Wie ich bereits an einigen Beispielen demonstrieren konnte, bietet die DUG gewisse Vorteile gegenüber mit Hilfe von DCG's kodierten Phrasenstrukturgrammatiken. Auf die linguistische Adäquatheit des einen oder anderen Formalismus möchte und kann ich hier gar nicht eingehen, sattdessen werde ich versuchen, unterschiedliche Eigenschaften zu explizieren und zu bewerten.

Das herausragende Merkmal einer DUG ist, daß sie für jedes Wort mindestens eine eigene

Regel besitzt. Sie ist damit gleichzeitig Lexikon und Grammatik einer Sprache, eine Eigenschaft, deren Tendenz sich auch in anderen linguistischen Formalismen abzeichnet (vgl. [Shieber 86]).

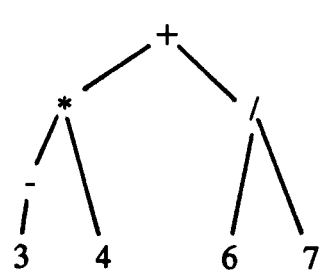
Der Akzeptionsvorgang verläuft immer 'mit einem Auge auf dem Satz', das heißt, durch Betrachtung des Satzes werden nur Regeln ausgeführt, die grundsätzlich zu einer Lösung führen können. Eine vorausschauende Regelauswahl, die viel 'unnützes Probieren' vermeidet, ist die automatische Folge. Der Preis, der für diese zielgerichtete Regelauswahl gezahlt wird, ist die wiederholte Durchsuchung des verbliebenen Restsatzes, ein Umstand, der die Eignung für extrem lange Sätze fragwürdig erscheinen läßt. Da in natürlichen Sprachen die Satzlänge im Vergleich zur Größe des Vokabulars (und damit der Regelmenge) immer sehr klein ist (die deutsche Sprache umfaßt ca. 400.000 Wörter), erscheint diese Vorgehensweise aber durchaus angebracht, zumal da sich der Suchprozeß mit algorithmischen Mitteln noch erheblich beschleunigen läßt. Nichtsdestotrotz wird die Satzanalyse nach dem Prinzip der SLD-Resolution, also durch eine links-rechts Tiefensuche betrieben wird, eine Eigenschaft, die sie mit den DCG's teilt.

Phrasenstrukturgrammatiken dagegen verfügen meist über eine im Vergleich zu Dependenzgrammatiken kleine Menge von Regeln. Während auf dem Gebiet der natürlichen Sprachen die Wahl zwischen den beiden anscheinend eine Streitfrage ist, wird bei den wohl am häufigsten analysierten Vertretern der künstlichen Sprachen, den Programmiersprachen, der Vorteil der Konstituenz, der Teil-Ganzes-Beziehung nicht in Frage gestellt. Ein Programm besteht nun einmal aus Vereinbarungs- und Programmteil und diese wiederum aus Typen- und Variablendeklarationen bzw. Prozeduraufrufen und Kontrollstrukturen etc.. Gerechter Weise muß man den Phrasenstrukturgrammatiken zugute halten, daß sich ihr Akzeptionsverhalten wesentlich verbessert, wenn man ihnen eine Vorausschau zugesteht, wie das bei der Akzeption von Programmiersprachen im allgemeinen der Fall ist.

Allerdings werden Operatorbäume, wie sie beim Analysieren von arithmetischen Ausdrücken entstehen, meist als Dependenzgraphen dargestellt:

Beispiel:

-3*4 + 6/7



Dennoch scheint die Zerteilung von Programmiersprachen durch eine DUG wenig sinnvoll,

da Programme im Vergleich zum Vokabular ihrer Sprache meist sehr lang sind, sich also genau die Inversion der Verhältnisse ergibt, die den DUG's zum Vorteil gereichten.

6.4 Abschließende Bemerkungen

Ohne zu übertreiben, kann man wohl zusammenfassend feststellen, daß sich unter Voraussetzung der theoretischen Grundlagen mit osf-Prolog ein Prolog-Dialekt implementieren läßt, dessen Eignung für die Computerlinguistik, aber auch für andere Anwendungen wie zum Beispiel Datenbanken, nicht von der Hand zu weisen ist.

In der Kombination mit dem vorgestellten Formalismus der Dependenz-Unifikationsgrammatik steht damit ein linguistisches Werkzeug zur Verfügung, dessen Verwendung zum einen die Formulierung von Grammatiken nach linguistischen Maßstäben erlaubt, zum anderen aber auch die direkte Ausführung auf einer Rechenanlage ermöglicht. Während für andere grammatikalische Formalismen zum Einsatz auf Computern spezielle Übersetzungsprogramme oder Interpretierer benötigt werden, ist die Abbildung von DUG's auf Horn-Klauseln so direkt, daß man sie auch, genau wie DCG's, als 'syntaktischen Zucker' für Prolog betrachten kann [Pereira 80].

Anhang A: Begriffsklärung

Abschwächung

siehe Einschränkung

DUG

Aus [Hellwig 86] übernommene Abkürzung für Dependenz-Unifikationsgrammatik (englisch: dependency unification grammar). Damit wird ein linguistische Werkzeug bezeichnet, das das linguistische Prinzip der Dependenz mit dem formal logischen Befgriff der Unifikation kombiniert und so die Erzeugung und Akzeption von Sätzen bestimmter Sprachen ermöglicht.

Einschränkung

Auf das oft synonym gebrauchte Wort 'Abschwächung' möchte ich in dieser Arbeit absichtlich verzichten, da es irreführend ist. Der allgemeinhin mit 'Abschwächung' bezeichnetete Vorgang der Überführung von einer Sorte in eine echte Untersorte stellt in Wirklichkeit einen Informationszuwachs dar, die bezeichnete Menge wird durch eine 'stärkere' Spezifikation präzisiert. 'Schwächer' sind dagegen die Bedingungenen, denen die Elemente einer Obersorte genügen müssen.

feature

Das englische Wort 'feature' bedeutet in diesem Zusammenhang Merkmal und könnte sehr gut durch seine deutsche Bezeichnung ersetzt werden. Da sich der englische Ausdruck jedoch eingebürget hat und überall verstanden wird, soll er auch in dieser Arbeit verwendet werden, ohne ihn jedoch 'einzudeutschen'. Er wird deshalb, abgesehen vom Satzanfang, klein geschrieben und englisch dekliniert.

Wie man im Deutschen, wenn man von Merkmalen spricht, auch zwischen der Art oder dem Namen des Merkmals, dem Merkmalsbezeichner und seiner Ausprägung, dem Merkmalswert unterscheidet, so unterscheidet man im englischen zwischen feature label und feature value.

Darüberhinaus ist der Begriff 'feature' stark überladen. Je nach Zusammenhang kann er zum Beispiel für feature-Gleichungen und für feature label stehen.

Konkomitatanz

Die Konkomitanz ist ein Begriff aus der Linguistik und bezeichnet das geregelte Miteinandervorkommen von Wörtern innerhalb eines Satzes. Es wird oft als Gegenstück der Konstituenz betrachtet.

Konstituenz

Die Konstituenz erhebt die Teil-Ganzes-Relation zur fundamentalen Beziehung der Satzbausteine untereinander. Oberster Satzbaustein ist der Satz selbst, der sich meistens aus einer Anzahl Phrasen (Teilsätze) 'konstituiert', die wiederum unterteilt werden können und so weiter bis hinab zu den einzelnen Wörtern.

osf-

Die Vorsilbe osf- ist Abkürzug für ordnungssortierte feature- und wird immer dann verwendet, wenn einem Begriff, der aus der unsortierten, mehrsortigen oder ordnungssortierten Logik bekannt ist, durch die Ergänzung von features eine neue Bedeutung zukommt und daher von den alten unterschieden werden soll. Beispiele sind osf-Unifikation und osf-Prolog .

Sorte

Mit Sorten und Typen sind in dieser Arbeit grundsätzlich dasselbe gemeint. Zur besseren Unterscheidung werden aber manchmal Konstruktor-Sorten feature-Typen gegenübergestellt. Wo es aus dem Kontext hervorgeht, wird Sorte abkürzend für constructor-Sorte und Typ abkürzend für feature-Typ verwendet. Sammelbegriff für Konstruktor-Sorten und feature-Sorten oder -Typen ist jedoch immer Sorten.

sortenrecht

Sortenrecht und wohlsortiert werden synonym gebraucht. Das dazugehörige Substantiv ist immer Wohlsortiertheit. Darunter wird die Einhaltung der Sortenbestimmungen für Funktionssymbole der Signatur bei der Konstruktion von Termen verstanden.

Typ, feature-Typ (siehe Sorte)

Unifikand

Mit Unifikand wird ein zu unifizierender Term bezeichnet.

Anhang B: Programmausdruck

```
module dugpp.
/*  dug preprocessor  */

% uebersetzt ein Programm mit DUG-Regeln in ein Prolog-Programm
% jede DUG-Regel wird in eine Horn-Klausel ueberfuehrt

export dug_trans/2. % Export der Uebersetzung einer Klausel
                    % kann von osf-Prolog-Voruebersetzer eingebunden werden

from list import append/3.

private dug/2.
private convert/0, convert/2, convert/4, convert/6.

runtime_op(1200, fy, module).
runtime_op(1200, fy, private).
runtime_op(1200, fy, from).
runtime_op(1100, xfy, import).
runtime_op(1200, xfx, ':-').

% DUG-spezifische Operatoren
program_op(1200, xfx, ':>').
runtime_op(1200, xfx, ':>').
program_op(600, fx, '?').
runtime_op(600, fx, '?').
program_op(500, fx, '==>').
runtime_op(500, fx, '==>').

% osf-Prolog-spezifische Operatoren
program_op(600, xfy, '=>').
runtime_op(600, xfy, '=>').
program_op(500, xfx, ':').
runtime_op(500, xfx, ':').
program_op(400, xfx, '&').
runtime_op(400, xfx, '&').

body.

dug_trans(In, Out) :-
        convert(In, Out).

% Uebersetzerrahmen
dug(Quelle, Ziel) :-
        see(Quelle),
        tell(Ziel),
        convert,
        seen,
        told.
```

```
% Uebersetzungsschleife
convert :-
        read(DUGClause),
                (DUGClause = end_of_file
                ;
                convert(DUGClause, PClause),
                displayq(PClause),
                write('.'), nl,
                convert).
```

```
% Konvertierung in Horn-Klauseln mit automatischer Erzeugung eines Baumes

convert((HeadIn :> BodyIn), (HeadOut :- BodyOut)) :-
        !,
        HeadIn =.. [Pred|Args],
        append(Args, [In, Out, Tree], Expanded),
        HeadOut =.. [Pred|Expanded],
        convert(BodyIn, BodyOut, In, Out, Tree, []).

convert(Clause, Clause).

convert((AIn, BIn), (AOut, BOut), SIn, SOut, TIn, TOut) :-
        convert(AIn, AOut, SIn, SIntermediate, TIn, TIntermediate),
        convert(BIn, BOut, SIntermediate, SOut, TIntermediate, TOut).

convert((AIn; BIn), (AOut; BOut), SIn, SOut, TIn, TOut) :-
        convert(AIn, AOut, SIn, SIntermediate, TIn, TIntermediate),
        convert(BIn, BOut, SIntermediate, SOut, TIntermediate, TOut).

convert([], true, S, S, T, T) :- !.

convert(? AIn, (AOut; (SIn = SOut, TIn = TOut)), SIn, SOut, TIn, TOut) :-
        convert(AIn, AOut, SIn, SOut, TIn, TOut).

convert(==> AIn, (AOut, append(D, TOut, TIn)), SIn, SOut, TIn, TOut) :-
        !, AIn =.. [Pred|Args],
        append(Args, [SIn, SOut, D], Expanded),
        AOut =.. [Pred|Expanded].

convert(AIn, (accept(AIn, SIn, SIntermediate, TIn, D, TOut), AOut),
                SIn, SOut, TIn, TOut) :-
        AIn =.. [Pred|Args],
        append(Args, [SIntermediate, SOut, D], Expanded),
        AOut =.. [Pred|Expanded].
```

```
% Konvertierung in Horn-Klauseln ohne  automatische Erzeugung eines Baumes

convert((HeadIn :> BodyIn), (HeadOut :- BodyOut), In, Out) :-
        HeadIn =.. [Pred|Args],
        append(Args, [In, Out], Expanded),
        HeadOut =.. [Pred|Expanded],
        convert(BodyIn, BodyOut, In, Out).

convert((AIn, BIn), (AOut, BOut), In, Out) :-
        convert(AIn, AOut, In, Intermediate),
        convert(BIn, BOut, Intermediate, Out).

convert(? AIn, (AOut; In = Out), In, Out) :-
        convert(AIn, AOut, In, Out).

convert(==> AIn, AOut, In, Out) :-
        !, AIn =.. [Pred|Args],
        append(Args, [In, Out], Expanded),
        AOut =.. [Pred|Expanded].

convert([], true, In, In) :- !.



convert(AIn, (accept(AIn, In, Intermediate), AOut), In, Out) :-
        AIn =.. [Pred|Args],
        append(Args, [Intermediate, Out], Expanded),
        AOut =.. [Pred|Expanded].
```

```
module dugrt.
/* DUG-runtime module */

% enthaelt das Praedikat zur Akzeption eines Symbols aus dem Eingabesatz
% und ein Praedikat zur strukturierten Ausgabe des Ableitungsbaumes

export accept/6, pp/1.

private pp/2, ppt/2.
% ppt/2 (pretty print term) muss an die zu akzeptierenden Terme
% angepasst werden.

body.

accept(Element, [Element|String], String, [Element, Dependence|Tree],
                Dependence, Tree).

accept(Element, [Other|StringIn], [Other|StringOut],
                TreeIn, Dependence, TreeOut) :-
        accept(Element, StringIn, StringOut, TreeIn, Dependence, TreeOut).


pp(Tree) :- pp(Tree, 0).

pp([], X).

pp([Node, Subtree|Siblings], X) :-
        ppt(Node, X),
        Y is X + 1,
        pp(Subtree, Y),
        pp(Siblings, X).

ppt(v(Lexem, _, Rolle), X) :-
        Pos is X * 4,
        tab(Pos),
        write(Rolle),
        write(' : '),
        write(Lexem),
        nl.

end_module.
```

```
module demo1. % Beispielprogramm zur Benutzung von DUG's

private s/0, s/1, s/3, v/6.

from dugrt import accept/6, pp/1.

body.

s :- s([v('Peter', subst, _), v(schlaeft, verb, _)]).

s :- s([v('Hans', subst, _), v(schenkt, verb, _),
        v('Peter', subst, _), v(ein, art, _), v('Buch', subst, _)]).

s(X) :-
        s(X, [], T),
        write(T), nl,
        pp(T).

s :> v(_, verb, praedikat).

v(schenkt, verb, _) :>
        v(_, subst, subjekt),
        v(_, subst, empfaenger),
        v(_, subst, objekt).

v(schlaeft, verb, _) :>
        v(_, subst, subjekt).

v('Hans', subst, _) :> [].

v('Peter', subst, _) :> [].

v('Buch', subst, _) :>
        v(_, art, det).

v(ein, art, _) :> [].
```

Anhang C: Literatur

Logik und allgemeine Informatik

[Aho 77] Alfred V. Aho and Jeffrey D. Ullman:'Principles of Compiler Design', Addison-Wesley 1977 26

[Aït-Kaci 86] Hassan Aït-Kaci and Roger Nasr:'Login: A Logic Programming Language with Built-in Inheritance', The Journal of Logic Programming 1986:3:185-215 3

[Beierle 89] Christoph Beierle:'Types, Modules and Databases in the Logic Programming Language PROTOS-L', IWBS Report 88, IBM Oktober 1989 42

[Hofbauer 89] Dieter Hofbauer, Ralf-Detlef Kutsche:'Grundlagen des maschinellen Beweisens', Vieweg 1989 15

[Huber/Varšek 87] M. Huber, I. Varšek: 'Extended Prolog with Order-Sorted Resolution', 4th Symposium of Logic Programming, San Francisco 1987 3

[Kowalski 74] R. Kowalski:'Predicate Logic as Programming Language', IFIP 74, Information Processing, North-Holland 1974 36

[Lindenberg et al. 87] N. Lindenberg, A. Bockmayr, R. Dietrich, P. Kursawe, B. Neidecker, C. Scharnhorst, I. Varšek:'KA-Prolog Sprachdefinition', Interner Bericht 5/87, Fakultät für Informatik, Universität Karlsruhe 1987 37

[Ludwig 88] Thomas Ludwig:'FLL: A First-Order Language for Deductive Retrieval of Feature-Terms', LILOG-Report 57, IBM Deutschland GmbH 1988 . . . . . 3, 101

[Pereira 80] F. C. N. Pereira and D. H. D. Warren:'Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks', Artificial Intelligence 1980, 13:231-278 . . . . . 21, 37

[Schmidt-Schauß 85] M. Schmidt-Schauß:'A Many-Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation', Memo Seki-85-II-KL, Fachbereich Informatik, Universität Kaiserslautern 1985 3

[Smolka 89] Gert Smolka and Hassan Aït-Kaci:'Inheritance Hierarchies: Semantics and Unification', Journal of Symbolic Computation, March/April 1989, Vol.7, Nr.3 . . . . . 4, 10, 11, 15-17, 48, 76

[Waldmann 89] Uwe Waldmann:'Unification in Order-Sorted Signatures', Universität Dortmund, Fachbereich Informatik, Forschungsbericht Nr. 298, 1989 6

[Walther 87] Christoph Walther: 'A Many-Sorted Calculus Based on Resolution and Paramodulation', Research Notes in Artificial Intelligence, Pitman, London 1987 3

[Weinstein 89] Thomas Weinstein, 'Lopster: Entwurf und Realisierung einer Laufzeitumgebung für mehrsortiges Prolog', Diplomarbeit, Institut für Informatik I, Universität Karlsruhe 1989 . . . . . 3, 37, 77

Linguistik

[Boyer 88] Michel Boyer:'Towards Functional Logic Grammar', in V. Dahl and P. Saint-Dizier (Ed.), 'Natural Language Understanding and Logic Programming, II', North-Holland 1988 73

[Bresnan 82] Bresnan, J. (Ed.):'The Mental Representation of Grammatical Relations', Cambridge, Mass., MIT Press 1982 4

[Bühler 65] Karl Bühler:'Sprachtheorie. Die Darstellungsfunktion der Sprache', 2. unveränderte Auflage, Stuttgart 1965 27

[Engel 77] Ullrich Engel:'Syntax der deutschen Gegenwartssprache', Erich Schmidt Verlag, Berlin 1977 . . . . . 20, 28, 59, 60, 92

[Gaifman 65] Haim Gaifman, 'Dependency-Systems and Phrase-Structure-Systems', Information and Control 1965 , Vol.8, 304-337 28

[Gazdar 89] Gazdar, Gerald:'Natural Language Processing in Prolog', Addison-Wesley 1989 3

[Hays 64] David G. Hays:'Dependency Theorie: A Formalism and Some Observations', Language, Vol.40, Nr.4, 1964 28

[Hellwig 86] Peter Hellwig: 'Dependency Unification Grammar', Germanistisches Seminar, Heidelberg, First draft, January 1986, und 'Dependency Unification Grammar', 11th International Conference on Computational Linguistics,
Proceedings of Coling '86, Bonn 20, 33, 51, 53, 102

[Holtermann 78] Horst Holtermann und Hans Baumgarten:'IANUA NOVA, Begleitgrammatik', 8. überarbeitete Auflage, Vandenhoeck & Ruprecht in Göttingen 1978

[Kay 82] Kay, Martin:'Parsing in Functional Unification Grammar', in 'Natural Language Parsing: Psychological, Computational and Theoretical Perspectives', chapter 7, 251-278, Cambridge University Press; auch in 'Readings in Natural Language Processing', Morgan Kaufmann Publishers, Inc. 1986 . . . . . 4, 20

[Shieber 86] Stuart M. Shieber: 'An Introduction to Unification-based Approaches to Grammar', CSLI Lecture Notes 4, Stanford University 1986 . . . . . 3, 4, 20

[Tarvainen 81] Kalevi Tarvainen:'Einführung in die Dependenzgrammatik', Max Niemeyer Verlag, Tübingen 1981 27

Datenbanken

[Codd 71] E. F. Codd:'Relational Completeness of Database Sublanguages', Courant Computer Science Symposium 6, Prentice-Hall 1971 94

[Codd 77] E. F. Codd:'Extending the Database Model to Capture More Meaning', ACM Transactions on Database Systems, Vol.4, Nr.4, 1977 94

[Date 81] C. J. Date:'An Introduction to Database Systems', Third Edition, Addison-Wesley 1981 . . . . . 93, 94, 96

[Lacroix 77] M. Lacroix and A. Pirotte:'Domain-Oriented Relational Languages', Proc. 3rd International Conference on Very Large Databases 1977 95

[Lockemann 87] P. C. Lockemann und J. W. Schmidt (Hrsg.):'Datenbank-Handbuch', Springer 1987 . . . . . .93, 95