Centrum für Informations-
und Sprachverarbeitung
Universität München

99-121

Anaphernresolution mit beschränkten Parametern

Stefan Wilke

CIS—Universität München
Oettingenstr. 67 80538 München

Centrum für Informations-
und Sprachverarbeitung
Universität München
Oettingenstr. 67
80538 München

Tel: 089 / 2178 2721, FAX: 089 / 2178 2701
EMAIL: sekr@cis.uni-muenchen.de

Universität München
Centrum für Informations- und Sprachverarbeitung

Anaphernresolution mit beschränkten Parametern

Stefan Wilke

CIS-Bericht-99-121

Verantwortlich:
Petra Maier, Centrum für Informations- und Sprachverarbeitung,
Universität München, Oettingenstr. 67, 80538 München
EMAIL: pmaier@cis.uni-muenchen.de

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Philosophie an der Ludwig-Maximilians-Universität
München

vorgelegt von

Stefan Wilke

Referent: Prof. Dr. Klaus Schulz
Koreferent: Prof. Dr. Dietmar Zaefferer
Tag der mündlichen Prüfung: 23. Februar 1998

CIS
Universität München
Oettingenstr. 67
80538 München

ISBN 3-930859-15-7
Februar 1999

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Idee 1

1.2 Testprädikate und Lösungsgraphen 1

1.3 Anaphernresolution 4

1.4 Forschungsbeitrag 5

2 Typisierte Merkmalsstrukturen **7**

2.1 Subsumption und Unifikation 8

2.2 Supportrelationen für Merkmalsstrukturen 11

2.3 Updaterelationen für Merkmalsstrukturen 13

3 Eine Update-Semantik für Prolog **14**

3.1 Der klassische Unfikationsalgorithmus 15

3.2 Updates auf TMSen 15

3.3 Unifikation von TMSen 16

3.4 Prolog-Programme und Infon-Programme 18

3.5 Ein Metainterpreter 19

3.6 Metaterme 22

3.7 Variablen einfrieren 24

3.8 Goal Delaying 27

3.9 Lösungsmengen als Terme 31

3.10 Testprädikate 37

4 Ein Anwendungsbeispiel **39**

4.1 Eine kleine Bindungstheorie 42

5 Anaphernresolution in der HPSG **50**

5.1 Die Prinzipien der HPSG 50

5.2 Semantik 60

5.2.1 Der Inhalt von Nomina und Adjektiven 60

5.2.2 Der Inhalt von Verben 64

5.2.3 Der Inhalt von Determinatoren 65
5.2.4 Das Semantikprinzip 66
5.3 Das $\bar{X}$-Prinzip 72
5.3.1 Namen 76
5.4 Anaphernresolution und Bindungstheorie 78
5.4.1 Dynamische Semantik 78
5.4.2 Anaphernresolution 85
5.4.3 Bindungstheorie 96
5.5 Beispielanalysen 97
5.6 Der vollständige Programmtext 100

1 Einleitung

1.1 Idee

Die Grundidee der vorliegenden Arbeit ist sehr einfach: Typisierte Merkmalsstrukturen (TMSen) wie sie ein HPSG-Parser aufbaut, werden als Modelle betrachtet. Die Knoten einer TMS sind dann die Individuen des Modells und die Relationen zwischen den Knoten werden durch ein Prolog-Programm festgelegt, das auf die Typen der Knoten und die Übergänge zwischen ihnen Bezug nimmt.

Verfolgt nun ein Parser eine Left-First/Top-Down-Strategie, so steht ihm zum Zeitpunkt einer terminalen Regelanwendung eine partielle typisierte Merkmalsstruktur zur Verfügung, die die grammatischen Eigenschaften des Satzes bis zum gerade erkannten Wort repräsentiert. Handelt es sich bei diesem Wort um ein anaphorisches Pronomen, so kann die TMS nun mit einem Prologprädikat nach geeigneten Antezedenten abgesucht werden.

Zu diesem Zweck wird Prolog um eine Testoperation erweitert, mit der ein beliebiges Prologprädikat in ein Testprädikat verwandelt werden kann. Dieses Testprädikat wird nicht im Herbrand-Modell des Programms ausgewertet, sondern in einem Modell, das durch den Zustand der Lösungsmenge zum Zeitpunkt des Aufrufs repräsentiert wird. Dadurch ergibt sich ein kompositionaler, deklarativer und modularer Ansatz zur Anaphernresolution in der HPSG.

1.2 Testprädikate und Lösungsgraphen

Mit Hilfe des Operators `?` kann jedes Prologprädikat ϕ in ein Testprädikat `?`ϕ verwandelt werden, das überprüft, ob die aktuelle Lösungsmenge schon genügend Information enthält, um ϕ zu unterstützen. Dabei unterstützt eine Lösungsmenge A ein Prädikat ϕ (bzgl. eines Programms P), falls ϕ mit P zu einer Lösungsmenge B reduziert werden kann, die weniger Information enthält als A. Um diese Definition zu präzisieren, müssen Lösungsmengen nach ihrem Informationsgehalt angeordnet werden. Wir wählen das Verfahren, die Lösungsmengen als spezielle Graphen zu repräsentieren - nämlich typisierte Merkmalsstrukturen -, für die in der einschlägigen Literatur bereits eine Informationsordnung vorliegt. Eine Termgleichung

```
S=s(np(ind,john),vp(v(loves),np(ind,mary)))
```

wird durch den Unifikationsalgorithmus von Prolog zu folgender Lösungsmenge reduziert:

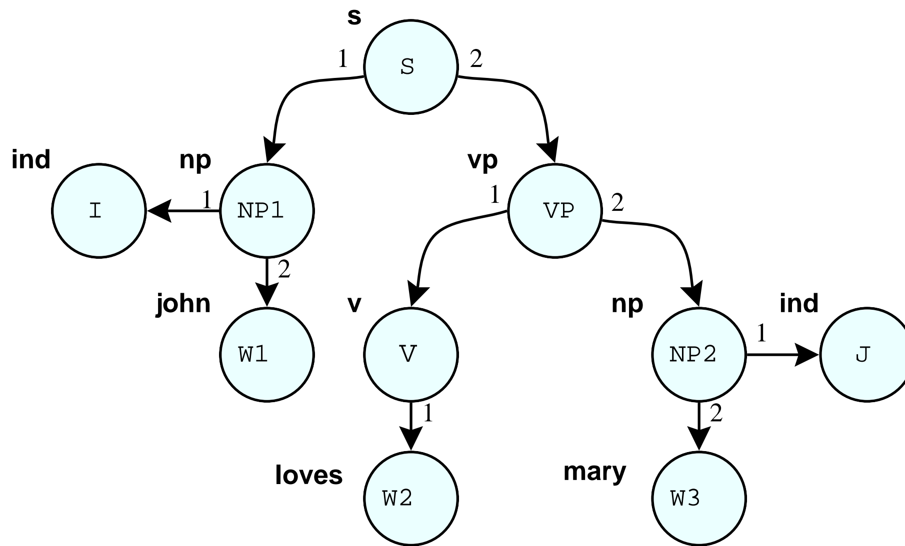

Abbildung 1: Der Lösungsgraph F

```
S   = s(NP1,VP)     V   = v(W2)
NP1 = np(I,W1)      W2  = loves
I   = ind           NP2 = np(J,W3)
W1  = john          J   = ind
VP  = vp(V,NP2)     W3  = mary
```

Diese Lösungsmenge wird durch einen Graphen F repräsentiert, der folgende Eigenschaften hat (s. Abb. 1.2 auf S. 2): Die Knoten des Graphen sind die Äquivalenzklassen unifizierter Variablen in der Lösungsmenge. Jeder Knoten ist mit dem Funktor des Terms markiert, an den seine Variablen gebunden sind. Ist ein Knoten frei, d.h. sind seine Variablen ungebunden, so wird er mit dem Symbol $\perp$ markiert. Die Funktoren sind in einem flachen Verband geordnet, in dem $\perp$ unter allen Funktoren und das Symbol $\top$ über allen Funktoren liegt. Die Funktoren selbst sind paarweise unvergleichbar. Die Übergange der Knoten sind die Argumentstellen der Funktoren, mit denen die Knoten markiert sind.

Zwei Lösungsmengen sind unifizierbar, wenn ihre Vereinigung zu einer Lösungsmenge reduziert werden kann. Sind A, B unifizierbar, so haben ihre repräsentierenden Graphen $g(A)$, $g(B)$ ein Supremum in der Graphenordnung, das keinen Knoten enthält, der mit $\top$ markiert ist. Wenn wir z.B. der oben gezeigten Lösungsmenge die Gleichung `I=J` hinzufügen und die entstehende Gleichungsmenge normalisieren, ergibt das den repräsentierenden Graphen in Abb. 1.2 auf S. 3.

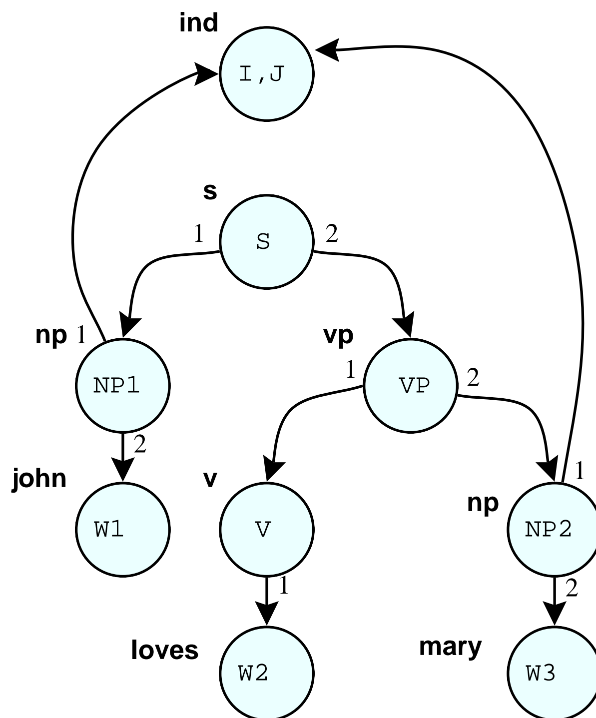

Abbildung 2: Das Supremum von F und *g(*`I=J`*)*

Ein solcher Lösungsgraph kann nun mit beliebigen Prologprädikaten abgefragt werden. Angenommen, wir haben folgende Prologdefinition von c-Kommando:

```
c_commands(NP,Comp):-
  node(s(NP,VP)),
  dominates(VP,Comp).

node(s(Subj,Pred)).
node(np(Index,Lex)).
node(vp(V,Comp)).
node(v(Lex)).

dominates(Node,Node).
dominates(Node,Dom):-
  daughter(Node,Dtr),
  dominates(Dtr,Dom).

daughter(s(X,Y),X).
daughter(s(X,Y),Y).
daughter(np(A,X,Y),X).
daughter(np(A,X,Y),Y).
```

```
daughter(vp(X,Y),X).
daughter(vp(X,Y),Y).
```

Dann unterstützt F das Prädikat c_commands(NP1,VP) und die folgende Abfrage gelingt:

```
?- S=s(np(ind,john),vp(v(loves),np(ind,mary))),?c_commands(NP1,VP)
true
?-
```

Dabei denotieren NP1 und VP diejenigen Knoten von F, in denen sie enthalten sind. Als Abfrageprädikat ist ?c_commands(NP1,VP) also ein geschlossener Term. Verwendet man hingegen Variablen, die nicht in F vorkommen, so erzeugt die Abfrage eine Bindung an geeignete Knoten:

```
?- S=s(np(ind,john),vp(v(loves),np(ind,mary))), ?c_commands(X,VP)}
X=NP1
no (more) solution
?-
```

Durch diese Abfrage ändert sich die Struktur von F nicht, jedoch enthält der Knoten NP1 nun die zusätzliche Variable X, d.h. X ist an NP1 gebunden worden.

1.3 Anaphernresolution

Ausgerüstet mit dieser konservativen Erweiterung der Standard-Semantik von Prolog, können wir nun ein Anaphernresolutionsverfahren vorschlagen, das von der zugrundeliegenden Bindungstheorie völlig unabhängig ist. Jede Bindungstheorie definiert eine Relation poss_binder(Ante,Ana) zwischen Anaphern Ana und ihren möglichen Bindern Ante. Bekanntlich unterscheiden sich die Bindungstheorien darin, ob sie syntaktische oder semantische Kriterien für die Definition dieser Relation verwenden. Da HPSG-Parser eine Analyse erzeugen, die die syntaktischen und semantischen Eigenschaften der gegebenen Phrase in einem einheitlichen Format repräsentieren, kann sich die Definition von poss_binder/2 auf jede beliebige dieser Bindungstheorien stützen. Angenommen, wir wollen eine syntaktische Bindungstheorie implementieren. Dann würden wir die Antezedensrelation über das c-Kommando definieren:

```
poss_binder(Ante,Phr):-
  c_commands(Ante,Phr),
  not((c_commands(Com,Phr), c_commands(Ante,Com))).
```

Die einzige Stelle, an der in der Grammatik die Relation `poss_binder/2` vorkommt, ist die PRO-Regel. Wenn der Parser mit einer Top-Down/Left-First-Strategie arbeitet, repräsentiert der Zustand der Lösungsmenge zum Zeitpunkt der PRO-Regel-Anwendung eine mögliche syntaktische Struktur des Vortexts bis zu `herself`:

```
pro(Pro=phr(cat(noun(ProIndex=fem,obj),[]),empty)) --> [herself],
  { ?poss_binder(Ante,Pro),
    index(Ante,AnteIndex),
    AnteIndex=ProIndex }.
```

`?poss_binder(Ante,Pro)` sucht in der aktuellen syntaktischen Struktur nach einer bindungsfähigen Phrase `Ante`, die `Pro` c-kommandiert und die näher an `Pro` liegt, als jede andere Phrase mit dieser Eigenschaft. Dann wird der Index von `Ante` bestimmt und mit dem Index von `Pro` unifiziert. Damit ist die Anapher `herself` aufgelöst. Sollte es mehrere geeignete Antezedenten für `herself` geben, sorgt der Backtracking-Mechanismus von Prolog für eine systematische Generierung aller Lesarten.

1.4 Forschungsbeitrag

Dass die typisierten Merkmalsstrukturen der HPSG eine Situationsstruktur nach Barwise/Perry/Etchemendy bilden, ist keine neue Beobachtung. Jedoch finden sich in der Literatur bis auf fußnotenartige Erwähnungen keine systematischen Überlegungen zu diesem Thema. Auch die Idee, den Modellcharakter der TMS, die von einem HPSG-Parser erzeugt wird, zur Anaphernresolution zu verwenden, ist schon vereinzelt geäußert worden. Es wurde jedoch nie versucht, diese Idee zu präzisieren oder gar ein lauffähiges System daraus zu entwickeln.

Diese Arbeit soll ein Beitrag zu drei Forschungsgebieten sein, die traditonell eng miteinander zusammenarbeiten: Situationssemantik, HPSG und Dynamische Semantik. Durch die Definition einer berechenbaren Supportrelation zwischen typisierten Merkmalsstrukturen und Prologprädikaten wird ein theorieunabhängiges Anaphernresolutionsverfahren in die HPSG integriert, das durch eine minimale Erweiterung von Standard-Prolog realisiert werden kann. Der Testoperator `?` hat eine klare und einfache Semantik, die die Standard-Semantik von Prolog auf eine natürliche Weise 'dynamisiert'. Dadurch gelingt es, eine theoretische Brücke zwischen Situationssemantik und Dynamischer Semantik zu schlagen.

In der vorliegenden Arbeit werden die beiden oben genannten Ideen nicht nur in theoretischer, sondern auch in praktischer Hinsicht entwickelt: Zunächst wird ein Metainterpreter für ein Prolog mit Testprädikaten implementiert. In dieser

Prologerweiterung wird dann eine umfangreiche und erweiterungsfähige HPSG-Grammatik für ein Fragment des Englischen formuliert, das alle syntaktischen Konstruktionen, die Kamp in seiner klassischen Arbeit behandelt, umfasst. Diese Grammatik verbessert Kamps Verfahren in drei Punkten:

Es ist *kompositional*: Die Bindung eines Pronomens an einen geeigneten Antezedenten im Vortext wird durch ein Testprädikat in der PRO-Regel des Parsers errechnet. Dazu ist es nicht notwendig, eine wie auch immer geartete Diskursrepräsentation im Kopf der Regel mitzuführen. Mitzuführen ist lediglich die Position des zu bindenden Index im bisher errechneten Inhalt (*content*) des Textes. Diese Position wird jedoch lokal und nach einem einheitlichen Prinzip vererbt, sodass die klassischen Prinzipien der HPSG nur um dieses Prinzip erweitert werden müssen, um Anaphernresolution zu ermöglichen.

Es ist *kumulativ*: Die Konstruktion der syntaktischen und semantischen Struktur eines Textes wird simultan und in einer einheitlichen Repräsentation ausgeführt, nämlich in Form einer typisierten Merkmalsstruktur. Vom informatischen Standpunkt aus gesehen, ist das ein deutlicher Fortschritt gegenüber der in der DRT verfolgten Strategie, zunächst eine syntaktische Struktur zu errechnen, die erst in einem zweiten Schritt in eine semantische Struktur übersetzt wird.

Es ist *deklarativ*, *modular* und *theorieunabhängig*. Das ist vielleicht der wichtigste Beitrag der Arbeit: So wie man eine Grammatik in einen Parser verwandeln kann, indem man sie in Prolog formuliert, so kann man mit dem hier vorgestellten Ansatz eine Bindungstheorie in ein Anaphernresolutionsverfahren verwandeln, indem man sie in Prolog formuliert. Dabei kann die Bindungstheorie der Grammatik als Modul definiert werden, dessen Schnittstelle das Prädikat `poss_binder/2` ist. Die Definition dieses Prädikates kann sich auf jede Eigenschaft stützen, die eine HPSG-Merkmalsstruktur zur Verfügung stellt, d.h. auf strukturelle, funktionale oder semantische Eigenschaften des Vortextes. So lassen sich verschiedene Bindungstheorien austesten, ohne dass irgendwelche Veränderungen am Rest der Grammatik vorgenommen werden müssen.

2 Typisierte Merkmalsstrukturen

In der HPSG (Head-Driven Phrase Structure Grammar) werden typisierte Merkmalsstrukturen verwendet, um syntaktische Strukturen zu repräsentieren. Üblicherweise werden sie als gerichtete, verwurzelte Graphen aufgefasst, deren Knoten mit einem Typ dekoriert sind. Die Kanten, die von einem gegebenen Knoten ausgehen, entsprechen den Merkmalen des Knotens und sind mit dem Namen des Merkmals beschriftet. Unsere typisierten Merkmalsstrukturen unterscheiden sich davon in zwei wichtigen Punkten:

- Die Knoten werden explizit als Mengen äquivalenter Variablen konstruiert.
- Die Knoten einer TMS brauchen nicht zusammenhängend zu sein. Insbesondere wird nicht die Existenz eines Wurzelknotens gefordert.

Diese Modifikationen sind notwendig, weil wir TMSen verwenden wollen, um Prolog-Lösungsmengen zu repräsentieren.

Typen sind Objekte aus einer als gegeben angenommenen Menge Type, zwischen denen eine Halbordnungsbeziehung besteht. Zur Erinnerung: $(\mathsf{Type}, \sqsubseteq)$ ist eine Halbordnung gdw $\sqsubseteq$ reflexiv, transitiv und antisymetrisch ist. Wir schreiben $X \sqsubseteq y$, falls $x \sqsubseteq y$ für alle $x \in X$, und nennen dann X beschränkt und y eine obere Schranke für X. $(\mathsf{Type}, \sqsubseteq)$ ist beschränkt vollständig gdw es für jedes beschränkte $X \subseteq \mathsf{Type}$ eine kleinste obere Schranke $\bigsqcup X$ gibt. Wir werden im folgenden nur endliche Typhierarchien betrachten, die ein kleinstes Element $\bot$ enthalten. Wir können diese Typhierarchien durch die Hinzufügung eines größten Elements $\top$ vervollständigen, indem wir definieren: $\bigsqcup X = \top$ falls X nicht beschränkt ist. Dadurch werden Typhierarchien zu oberen Halbverbänden. Definieren wir weiterhin: $\sqcap X = \bigsqcup\{x \mid x \sqsubseteq X\}$ für alle $X \subseteq \mathsf{Type}$ so erhalten wir Typhierarchien als endliche Verbände. $\bigsqcup X = \top$ hat die intuitive Interpretation, dass die Typinformation in X inkonsistent ist. Ist $\bigsqcup X \neq \top$ heißt X konsistent. $x \sqsubseteq y$ bedeutet, dass der Typ x allgemeiner, und damit informationsärmer, als y ist. $\bigsqcup X$ ist der Typ, der die gesamte Information der Typen aus X enthält, aber nicht mehr. Eine flache Typhierarchie ist ein Verband $(\mathsf{Type}, \sqsubseteq)$, in dem für je zwei Typen $x, y \in \mathsf{Type}$ gilt: $\bigsqcup\{x, y\} = \top$ und $\sqcap\{x, y\} = \bot$.

Im Grunde genommen sind wir in dieser Arbeit nur an flachen Typhierarchien interessiert. In [2] beschreibt Aït-Kaci sogenannte Ψ-Terme als eine Verallgemeinerung von Prologtermen. Zwei Prologterme unifizieren bekanntlich, wenn ihre Funktoren gleich sind und ihre Argumente unifizieren. Bei Ψ-Termen wird der Funktor des Terms als sein Typ aufgefasst. Zwei Ψ-Terme unifizieren, wenn ihre Funktoren konsistent sind und ihre Argumente unifizieren. Macht man die Anzahl der Argumente eines Terms von seinem Typ abhängig und lässt man darüberhinaus zu, dass Terme mit einer unterschiedlichen Anzahl von Argumenten unifizieren können, muss man den Argumenten Namen geben, um sie identifizieren

zu können. Solche getypten Terme mit benannte Argumenten nennt man Merkmalsterme. Auch für sie gibt es eine Unifikationsoperation.

2.1 Subsumption und Unifikation

Die durch eine Äquivalenzrelation $\sim$ bestimmten Äquivalenzklassen seien wie üblich als $[x]_\sim = \{y \mid x \sim y\}$ definiert. Sei M eine beliebige Menge und $D \subseteq 2^M$. Abweichend von der üblichen Praxis definieren wir $D/_\sim = \{\bigcup[x]_\sim \mid x \in D\}$ als die Quotientenmenge von D.

Im folgenden sei eine Menge Var von Variablen, eine Menge Feat von Merkmalen und eine Typhierarchie $(\mathsf{Type}, \sqsubseteq)$ gegeben.

Def. 1 – $F = (Q, \delta, \theta)$ ist eine TMS gdw

- $\bigcup Q \subseteq \mathsf{Var}$ und $\bigcap Q = \emptyset$ und $\emptyset \notin Q$
- $\delta : Q \times \mathsf{Feat} \to Q$ ist eine partielle Übergangsfunktion
- $\theta : Q \mapsto \mathsf{Type}$ ist eine totale Typisierungsfunktion

Die Elemente $p, q, r \ldots$ aus Q heißen die Knoten von F. $\operatorname{var} F = \bigcup Q$ sind die Variablen von F. Wir können TMSen nach ihrem Informationsgehalt anordnen. Anschaulich ausgedrückt, kann der Informationsgehalt einer TMS F auf drei Arten wachsen: Entweder wachsen die Knoten von F, d.h. neue Variablen kommen hinzu, ohne dass sich die Struktur des Graphen ändert; oder die Typinformation an den Knoten wird verschärft; oder die Struktur des Graphen von F wird verschärft. Letzteres soll bedeuten, dass neue Knoten hinzukommen oder alte Knoten vereinigt werden:

Def. 2 – $F \sqsubseteq F'$ (F subsumiert F') gdw es ein $h : Q_F \to Q_{F'}$ gibt, mit den folgenden Eigenschaften:

- $q \subseteq h(q)$ für alle $q \in Q_F$
- $\theta_F(q) \sqsubseteq \theta_{F'}(h(q))$ für alle $q \in Q_F$
- $h(\delta_F(q, a)) = \delta_{F'}(h(q), a)$ für alle $q \in Q_F$, $a \in \mathsf{Feat}$ sodass $\delta_F(q, a)$ definiert

Satz 1 – $\sqsubseteq$ ist eine Halbordnung auf TMSen.

Beweis - Für Reflexivität ist id_F der gesuchte Homomorphismus, für Transitivität fg, falls $F \sqsubseteq_f F'$ und $F' \sqsubseteq_g F''$. Für Antisymetrie ist zu beobachten, dass $fg = \mathrm{id}_F$ und $gf = \mathrm{id}_{F'}$. Denn angenommen $F \sqsubseteq_f F'$, $F' \sqsubseteq_g F$ und $g(f(q)) \neq q$. Da Q_F eine Zerlegung ist, müssen ungleiche Elemente disjunkt sein. Es gilt aber nach Annahme $q \subseteq g(f(q))$. Widerspruch. ($gf = \mathrm{id}_{F'}$ analog) Also sind f, g invers zueinander und bijektiv. Damit haben wir für alle $q \in Q_F$:

- $q \subseteq f(q)$ und $f(q) \subseteq g(f(q)) = q$. (Analog erhält man $q = g(q)$)
- $\theta_F(q) \sqsubseteq \theta_{F'}(f(q)) = \theta_{F'}(q)$ und $\theta_{F'}(q) \sqsubseteq \theta_F(g(q)) = \theta_F(q)$
- $\delta_F(q,a) = f(\delta_F(q,a)) = \delta_{F'}(f(q),a) = \delta_{F'}(q,a)$

Def. 3 – Seien F, F' TMSen. Die Unifikationsrelation auf F und F' ist die kleinste transitive Relation $\bowtie$ auf $Q_F \cup Q_{F'}$ für die gilt:

- Wenn $q \cap p \neq \emptyset$ dann $q \bowtie p$
- Wenn $q \bowtie p$ dann $\delta_F(q,a) \bowtie \delta_{F'}(p,a)$

Satz 2 – $\bowtie$ ist Äquivalenzrelation auf $Q_F \cup Q_{F'}$ und es gilt:

- $[\delta_F(q,a)]_\bowtie = \bigcup_{p \bowtie q}(\delta_F(p,a) \cup \delta_{F'}(p,a))$
- $[\delta_{F'}(q,a)]_\bowtie = \bigcup_{p \bowtie q}(\delta_F(p,a) \cup \delta_{F'}(p,a))$

Def. 4 – Seien F, F' TMSen und $\bowtie$ die Unifikationsrelation auf F und F'. Dann ist $F \sqcup F'$ (die Unifikation von F und F') definiert durch:

- $Q_{F \sqcup F'} = (Q_F \cup Q_{F'})/\bowtie$
- $\delta_{F \sqcup F'}(q,a) = \bigcup_{p \subseteq q}(\delta_F \cup \delta_{F'})(p,a)$
- $\theta_{F \sqcup F'}(q) = \bigsqcup_{p \subseteq q}(\theta_F \sqcup \theta_{F'})(p)$

Satz 3 – $F \sqcup F'$ ist das Supremum von F und F'

Beweis - Wir zeigen zunächst, dass $F \sqsubseteq_h F \sqcup F'$ falls $h(q) = \bigcup[q]_\bowtie$. Dabei sei $\bowtie$ die Unifikationsrelation auf $Q_F \cup Q_{F'}$. Wir verwenden folgendes Lemma:

(†) $[q]_\bowtie = \{p \subseteq \bigcup[q]_\bowtie \mid p \in Q_F \cup Q_{F'}\}$ für alle $q \in Q_F \cup Q_{F'}$

($\subseteq$) ist trivial. Für ($\supseteq$) nehmen wir an, dass $p \notin [q]_\bowtie$. Dann ist $p \cap p' = \emptyset$ für alle $p' \in [q]_\bowtie$ und damit $p \cap \bigcup[q]_\bowtie = \emptyset$. Widerspruch. Damit haben wir:

$$\begin{aligned}
q \subseteq \bigcup[q]_\bowtie &= h(q) \\
h(\delta_F(q,a)) &= \bigcup[\delta_F(q,a)]_\bowtie \\
&= \bigcup\{(\delta_F \cup \delta_{F'})(p,a) \mid p \bowtie q\} \\
&= \bigcup\{(\delta_F \cup \delta_{F'})(p,a) \mid p \subseteq \bigcup[q]_\bowtie\} \\
&= \delta_{F \sqcup F'}(h(q),a) \\
\theta_F(q) &\sqsubseteq \bigsqcup\{\theta_F(p) \mid p \in [q]_\bowtie\} \sqcup \bigsqcup\{\theta_{F'}(p) \mid p \in [q]_\bowtie\} \\
&= \bigsqcup\{(\theta_F \sqcup \theta_{F'})(p) \mid p \in [q]_\bowtie\} \\
&= \bigsqcup\{(\theta_F \sqcup \theta_{F'})(p) \mid p \subseteq \bigcup[q]_\bowtie\} \\
&= \theta_{F \sqcup F'}(h(q))
\end{aligned}$$

Der Beweis für $F' \sqsubseteq_h F \sqcup F'$ ist analog. Also ist $F \sqcup F'$ eine obere Schranke für F und F'.

Wir zeigen nun, dass $F \sqcup F' \sqsubseteq_{f \sqcup f'} G$ falls $F \sqsubseteq_f G$ und $F' \sqsubseteq_{f'} G$. Dafür definieren wir:

$$f \sqcup f'(q) = \bigcup\{(f \cup f')(p) \mid p \subseteq q\}$$

Wir müssen uns zunächst vergewissern, dass $f \cup f'$ tatsächlich eine Funktion ist. Sei also $q \in Q_F \cap Q_{F'}$. Dann $q \subseteq f(q)$ und $q \subseteq f'(q)$ und damit $f(q) \cap f'(q) \neq \emptyset$. Q_G ist eine Zerlegung, also gilt $f(q) = f'(q)$.

(*) Wenn $q \bowtie p$ dann $(f \cup f')(q) = (f \cup f')(p)$

Beweis mit Induktion: Sei $q \cap p \neq \emptyset$. Dann $q \cap p \subseteq (f \cup f')(q)$ und $q \cap p \subseteq (f \cup f')(p)$ und damit $(f \cup f')(q) = (f \cup f')(p)$. Sei $q = \delta_F(q', a)$ und $p = \delta_{F'}(p', a)$ und $q' \bowtie p'$. Dann haben wir:

$$\begin{aligned}
(f \cup f')(\delta_F(q', a)) &= \delta_G((f \cup f')(q'), a) \\
&= \delta_G((f \cup f')(p'), a) \\
&= (f \cup f')(\delta_{F'}(q', a))
\end{aligned}$$

Also gibt es für alle $q \in Q_{F \sqcup F'}$ ein $p \in Q_F \cup Q_{F'}$ sodass:

$$\begin{aligned}
f \sqcup f'(q) &= f \sqcup f'(\bigcup [p]_{\bowtie}) \\
&= \bigcup\{(f \cup f')(r) \mid r \subseteq \bigcup [p]_{\bowtie}\} \\
&= \bigcup\{(f \cup f')(r) \mid r \bowtie p\} \\
&= (f \cup f')(p)
\end{aligned}$$

und damit $f \sqcup f' : F \sqcup F' \mapsto G$.

Nach Voraussetzung gilt $p \subseteq (f \cup f')(p)$ für alle $p \in Q_F \cup Q_{F'}$ und damit:

$$\begin{aligned}
q &= \bigcup\{p \mid p \subseteq q\} \\
&\subseteq \bigcup\{(f \cup f')(p) \mid p \subseteq q\} \\
&= f \sqcup f'(q)
\end{aligned}$$

Nach Voraussetzung und mit (*) und (†) gilt für alle $p \in Q_F \cup Q_{F'}$ mit $p \subseteq q$:

(**) $\delta_G((f \cup f')(p), a) = (f \cup f')(\delta_F(p,a)) \cup (f \cup f')(\delta_{F'}(p,a))$

und damit für alle $q \in Q_{F \sqcup F'}$:

$$
\begin{aligned}
\delta_G(f \sqcup f'(q), a) &= \delta_G(\bigcup_{p \subseteq q} (f \cup f')(p), a) \\
&= \bigcup_{p \subseteq q} \delta_G((f \cup f')(p), a) \\
&= \bigcup_{p \subseteq q} ((f \cup f')(\delta_F(p,a)) \cup (f \cup f')(\delta_{F'}(p,a))) \\
&= \bigcup \{(f \cup f')(p) \mid p \subseteq \bigcup_{r \subseteq q} (\delta_F(r,a) \cup \delta_{F'}(r,a))\} \\
&= f \sqcup f'(\delta_F \sqcup \delta_{F'}(q,a))
\end{aligned}
$$

Schließlich bemerken wir noch, dass für alle $q \in Q_{F \sqcup F'}$:

$$
\begin{aligned}
\theta_{F \sqcup F'}(q) &= \bigsqcup_{p \subseteq q} (\theta_F \sqcup \theta_{F'})(p) \\
&\sqsubseteq \bigsqcup_{p \subseteq q} \theta_G((f \cup f')(p)) \\
&= \theta_G(\bigcup_{p \subseteq q} (f \cup f')(p)) \\
&= \theta_G(f \sqcup f'(q))
\end{aligned}
$$

2.2 Supportrelationen für Merkmalsstrukturen

Wenn wir TMSen als Situationen auffassen wollen, müssen wir $\models$ eine Bedeutung geben, d.h. wir müssen uns überlegen, welche Art von Information eine TMS unterstützt. Zwei Aussagetypen springen sofort ins Auge:

- $x : \tau$ (x ist vom Typ τ)
- xay (x hat das a-Attribut y)

Formeln dieser Art sind durch Var, Type und Feat gegeben und bilden die Menge Basic der *elementaren* Infons. Sei weiterhin eine Menge Rel von Relationszeichen $p, q, r \ldots$ gegeben. Formeln der Gestalt $p(x_1 \ldots x_n)$ heißen *atomare* Infons. Der Abschluss elementarer und atomarer Infons unter Konjunktion, Implikation und Existenzquantifikation bildet die Menge Infon der Infons.

Def. 5 – Sei F eine TMS und $x_F = q$, falls es ein $q \in Q_F$ gibt mit $x \in q$ und $x_F = \emptyset$ sonst. Eine Relation $\models \subseteq \mathsf{TMS} \times \mathsf{Infon}$ heißt *Supportrelation* gdw

- $F \models xay$ gdw $\delta_F(x_F, a) = y_F \neq \emptyset$
- $F \models x : \tau$ gdw $\tau \sqsubseteq \theta_F(x_F)$
- $F \models \alpha \wedge \beta$ gdw $F \models \alpha$ und $F \models \beta$
- $F \models \alpha \rightarrow \beta$ gdw $F \not\models \alpha$ oder $F \models \beta$
- $F \models \exists_x \alpha$ gdw es ein $y \in \operatorname{var} F$ gibt mit $F \models a[x/y]$

Für die elementaren Infons ist die Supportrelation durch diese Definition eindeutig gegeben. Wir nennen sie *Basissupport* **B**. Eine Supportrelation S *erfüllt* ein Infon α gdw $\alpha \in \operatorname{ran} S$. Ein Infon α ist erfüllbar, falls es eine erfüllende Supportrelation für α gibt. Schließt man die elementaren Infons unter Konjunktion ab, so sind die kleinsten erfüllenden TMSen für diese Teilsprache von Infon berechenbar:

- $F_{x:\tau} = (\{\{x\}\}, \emptyset, \{\langle\{x\}, \tau\rangle\})$
- $F_{xay} = (\{\{x\}, \{y\}\}, \{\langle\langle\{x\}, a\rangle, \{y\}\rangle\}, \{\langle\{x\}, \bot\rangle, \langle\{y\}, \bot\rangle\})$
- $F_{\alpha\wedge\beta} = F_\alpha \sqcup F_\beta$

Sei α eine Konjunktion von elementaren Infons, β eine Konjunktion von atomaren Infons. Formeln der Gestalt

$$p(\bar{x}) \leftarrow \exists_{\bar{y}}(\alpha \wedge \beta)$$

heißen Klauseln; $p(\bar{x})$ heißt Kopf, $\exists_{\bar{y}}(\alpha \wedge \beta)$ heißt Körper der Klausel. Ein Klausel ist *geschlossen*, falls die $\bar{x}$ genau die freien Variablen im Körper der Klausel sind. Eine endliche Menge von geschlossenen Klauseln ist ein *Infon-Programm*. Wir schreiben $A \sim B$, falls B durch Variablenumbenennung aus A entsteht, und definieren

$$P^{\sim} = \{A \mid \text{es gibt ein } B \in P \text{ sodass } A \sim B\}$$

Eine Supportrelation erfüllt ein Programm P gdw sie jede Klausel $A \in P^{\sim}$ erfüllt. Zu jedem Programm P gibt es eine kleinste Supportrelation S_P, die P erfüllt, und zwar die kleinste Menge S für die gilt:

- $\mathbf{B} \subseteq S$
- $\mathsf{TMS} \times P^{\sim} \subseteq S$
- Wenn $(F, \alpha) \in S$ und $(F, \beta) \in S$ dann $(F, \alpha \wedge \beta) \in S$
- Wenn $(F, \alpha \rightarrow \beta) \in S$ und $(F, \alpha) \in S$ dann $(F, \beta) \in S$
- Wenn $(F, \alpha[x]) \in S$ und $y \in \mathsf{Var}$ dann $(F, \exists\alpha[x/y]) \in S$

2.3 Updaterelationen für Merkmalsstrukturen

Wir wollen uns jetzt der Frage zuwenden, was wir unter dem Informationsgehalt $[\![\sigma]\!]$ eines Infons σ verstehen wollen. In der klassischen Logik wird der Informationsgehalt einer Formel ϕ oft als die Menge der erfüllenden Modelle Mod_ϕ repräsentiert. ψ ist dann informationsreicher als ϕ gdw $\mathrm{Mod}_\psi \subseteq \mathrm{Mod}_\phi$. In der Dynamischen Semantik ist der Informationsbegriff eliminativ: Der Nulldiskurs lässt alle möglichen Interpretationen (Welten, Modelle) zu. Je informationsreicher der Diskurs wird, desto kleiner wird die Menge der möglichen Interpretationen. Informationszuwachs bedeutet also Modellelimination. Das ist auch nicht anders zu erwarten, da in dieser Semantiktradition im allgemeinen mit totalen Modellen gearbeitet wird. Uns stehen aber partielle Modelle - nämlich TMSen - zur Verfügung, daher können wir den Informationszuwachs, den ein Infon liefert, auch als Konstruktionsprozess auffassen. Zunächst müssen wir uns aber überlegen, was wir unter einem Informationszustand verstehen wollen.

Ein Informationszustand S ist ein Menge von TMSen: $S \subset \mathsf{TMS}$. Er repräsentiert eine Menge von minimalen Möglichkeiten, eine gegebene Information σ zu unterstützen. Die Minimalitätsbedingung ist entscheidend. Ein Informationszustand S' ist informationsreicher als S, falls S' weniger Möglichkeiten als S enthält, und diese Möglichkeiten durch Verschärfung von Möglichkeiten in S entstehen. Formal ausgedrückt:

Def. 6 – Seien $S, S' \subseteq \mathsf{TMS}$ Informationszustände. $S \sqsubseteq S'$ gdw es für alle $F' \in S'$ ein $F \in S$ gibt mit $F \sqsubseteq F'$.

Der Informationsgehalt $[\![\sigma]\!]$ eines Infons σ soll es uns dann ermöglichen, aus einer gegebenen TMS F, die unseren aktuellen Wissensstand repräsentiert, Situationen $F' \sqsupseteq F$ zu konstruieren, sodass $F' \models \sigma$. Darüberhinaus sollten diese F' nicht *mehr* Information unterstützen als durch σ gerechtfertigt ist, d.h. sie sollten minimale $F' \sqsupseteq F$ sein mit $F' \models \sigma$.

Wir wollen jetzt für jedes Infon σ eine Update-Relation $[\![\sigma]\!]$ festlegen, für die gilt: Wenn $F[\![\sigma]\!]F'$ dann $F \sqsubseteq F'$ und $F' \models \sigma$. F' heißt dann ein *Update* von F. Ein Update F' von F soll mindestens so informationsreich sein wie F, und seine Information sollte ausreichen, um das Infon zu unterstützen, mit dem geupdatet wurde. Dabei müssen wir uns entscheiden, wie der Existenzquantor $\exists$ zu interpretieren ist. Angenommen $F \models p(x)$. Soll dann $F[\![\exists_x p(x)]\!]F$ gelten? Wenn wir diese Frage mit ja beantworten, haben wir das Problem, dass wir durch ein Update keine neuen Knoten erzeugen können. Deshalb interpretieren wir den Existenzquantor dynamisch: $F[\![\exists_x p(x)]\!]F'$ soll gelten, wenn F' einen Knoten $q = \{y\}$ enthält, der nicht in Q_F enthalten ist (also $y \notin \mathrm{var}\, F$), und $F' \models p(y)$ gilt. Darüberhinaus soll F' minimal informationsreicher als F sein, d.h. es soll kein $F'' \sqsupseteq F$ geben mit $F'' \models p(y)$ und $F'' \sqsubseteq F'$.

Def. 7 – Seien $U, U' \subseteq \mathsf{TMS} \times \mathsf{TMS}$ und $F, F' \in \mathsf{TMS}$. Wir schreiben $F[U]$ für $\{F' \mid FUF'\}$. Wenn $X, Y \subseteq \mathsf{TMS}$, bedeutet $F \sqsubseteq X$, dass $F \sqsubseteq F'$ für alle $F' \in X$. $X \sqsubseteq Y$ bedeutet, dass es für alle $F' \in Y$ ein $F \in X$ gibt mit $F \sqsubseteq F'$. $U \sqsubseteq U'$ bedeutet, dass $F[U] \sqsubseteq F[U']$ für alle $F \in \mathsf{TMS}$.

- U heißt steigend gdw $F \sqsubseteq F[U]$ für alle $F \in \mathsf{TMS}$.
- U heißt monoton gdw $F[U] \sqsubseteq F'[U]$ falls $F \sqsubseteq F'$ für alle $F, F' \in \mathsf{TMS}$.
- U ist eine Updaterelation auf TMS gdw U monoton und steigend ist.

Man sieht sofort, dass $U_\phi = \{\langle F, F \sqcup F_\phi \rangle \mid F \in \mathsf{TMS}\}$ für jedes $\phi \in \mathsf{Basic}$ eine Updaterelation ist. Vereinigung und Komposition von Updaterelationen sind ebenfalls Updaterelationen.

Def. 8 – Eine Funktion $[\![]\!] : \mathsf{Infon} \mapsto \mathsf{Update}$ heißt Informationszuordnung gdw für alle $F \in \mathsf{TMS}$ gilt:

- $F[\![\phi]\!] = \{F \sqcup F_\phi\}$ für alle $\phi \in \mathsf{Basic}$
- $F[\![\sigma \wedge \rho]\!] = F[\![\sigma]\!][\![\rho]\!]$
- $F[\![\exists x \sigma]\!] = \bigcup\{F[\![\sigma[x/y]]\!] \mid y \notin \mathrm{var}\, F\}$

Eine Informationszuordnung erfüllt ein Programm P falls für alle $\alpha \to \beta \in P^\sim$ gilt: $[\![\beta]\!] \sqsubseteq [\![\alpha]\!]$.

Def. 9 – Wir definieren induktiv eine Folge $[\![]\!]^P_n$ von Informationszuordnungen und setzen $[\![\sigma]\!]^P = \bigcup_n [\![\sigma]\!]^P_n$. Die in jedem Schritt n für die atomaren Infons definierte Zuordnung wird wie in Def. 8 auf die elementaren und komplexen Infons fortgesetzt:

- $F[\![\alpha]\!]^P_0 = \{F\}$ für alle $\alpha \in \mathsf{Atom}$
- $F[\![\beta]\!]^P_{n+1} = \bigcup\{F[\![\alpha]\!]^P_n \mid \alpha \to \beta \in P^\sim\}$

Satz 4 – $[\![]\!]^P$ ist die $\sqsubseteq$–kleinste Informationszuordnung, die P erfüllt.

3 Eine Update-Semantik für Prolog

Prolog hat eine natürliche Update-Semantik: Jeder Reduktionsschritt bedeutet ein nichtdeterministisches Update der Lösungsmenge S durch Erweiterung von S mit einer Gleichung $g \doteq h$ und Normalisierung der dadurch entstandenen Gleichungsmenge. Eine Gleichungsmenge ist eine Menge von Formeln der Gestalt

$t \doteq s$, wobei t und s Prologterme sind. Der Algorithmus nf, der eine Gleichungsmenge E normalisiert, heißt Unifikationsalgorithmus, da sich aus der Lösungsmenge $\mathrm{nf}\, E$ ein kleinster Unifikator für E konstruieren lässt, d.h. eine Substitution θ der Variablen in E durch Prologterme, sodass $t[\theta] = s[\theta]$ für alle Gleichungen in E. Existiert ein solcher Unifikator, nennt man E erfüllbar. Natürlich ist ein solcher Unifikator nicht für jede Gleichungsmenge gegeben. In einem solchen Fall scheitert nf oder liefert $\mathrm{nf}\, E = \bot$.

3.1 Der klassische Unfikationsalgorithmus

Wir formulieren den klassischen Unifikationsalgorithmus als nichtdeterministischen Übergang $\Rightarrow$ auf Gleichungsmengen. Der transitive Abschluss $\Rightarrow^*$ hat die Eigenschaft, dass es für jedes erfüllbare E eine eindeutige erfüllungsäquivalente Normalform $\mathrm{nf}\, E$ gibt mit $E \Rightarrow^* \mathrm{nf}\, E$, und dass $E \Rightarrow^* \bot$ falls E unerfüllbar ist. ($E[x/t]$ notiert die Substitution sämtlicher Vorkommen von x in E durch t):

- $\{f(t_1 \dots t_n) \doteq f(s_1 \dots s_n)\} \cup E \Rightarrow \{t_1 \doteq s_1 \dots t_n \doteq s_n\} \cup E$ (Reduktion)
- $\{f(\bar{t}) \doteq g(\bar{s})\} \cup E \Rightarrow \bot$ falls $f \neq g$ (Clash)
- $\{t \doteq x\} \cup E \Rightarrow \{x \doteq t\} \cup E$ falls $t \notin \mathsf{Var}$ (Ausrichtung)
- $\{x \doteq t\} \cup E \Rightarrow \{x \doteq t\} \cup E[x/t]$ (Substitution)

Iterierte Anwendung dieser Mengentransformationsregeln führt entweder zu $\bot$ oder zu einer Normalform, in der alle Gleichungen die Gestalt $x \doteq t$ haben. Daher können wir definieren: $\theta(x) = t$ gdw $x \doteq t \in \mathrm{nf}\, E$. Für alle $t \doteq s \in E$ gilt dann $t\theta = s\theta$, d.h. θ ist ein Unifikator für E und zwar der kleinste. Wir wollen weder thematisieren, was in diesem Zusammenhang kleinster Unifikator bedeutet, noch beweisen, dass die gerade aufgestellte Behauptung stimmt. Dies lässt sich z.B. in nachlesen.

3.2 Updates auf TMSen

Ein Reduktionsschritt in Prolog ist ein nichtdeterministischer Übergang

$$(E, g, g_1 \dots g_n) \Rightarrow (\mathrm{nf}(E \cup \{h \doteq g\}), b_1 \dots b_n, g_1 \dots g_n)$$

wobei $h : -b_1 \dots b_n$ eine Klauselvariante des Programms ist, die keine Variablen aus $(E, g, g_1 \dots g_n)$ enthält. Die Erweiterung von E zu $\mathrm{nf}(E \cup \{h \doteq g\})$ ist ein Update. Da wir Lösungsmengen im folgenden als TMSen repräsentieren wollen, ist ein Update also eine Relation auf TMSen, von der wir folgende abstrakte Eigenschaften fordern:

3.3 Unifikation von TMSen

Für die spezielle Art von TMSen, die wir im folgenden ausschließlich betrachten werden, nämlich solche, bei denen die Typhierchie flach ist und natürliche Zahlen als Attribute verwendet werden, führen wir neue elementare Infons ein, Infons der Gestalt $x = f(x_1 \dots x_n)$ und $x = y$. Für diese Infons definieren wir folgenden Basissupport:

- $F \models x = f(x_1 \dots x_n)$ gdw $f \sqsubseteq \theta(x_F)$ und $\delta(x_F, i) = x_i$ für alle $1 \leq i \leq n$
- $F \models x = y$ gdw $x_F = y_F$

Es ist klar, dass es zu jeder Konjunktion ϕ dieser elementaren Infons eine kleinste TMS F_ϕ gibt, sodass $F \models \phi$:

- $F_{x=f(x_1 \dots x_n)} = (Q, \delta, \tau)$ mit
  - $Q = \{\{x\}, \{x_1\} \dots \{x_n\}\}$
  - $\delta(\{x\}, i) = \{x_i\}$ für alle $1 \leq i \leq n$
  - $\tau(\{x\}) = f$ und $\tau(\{x_i\}) = \bot$ für alle $1 \leq i \leq n$
- $F_{x \doteq y} = (Q, \delta, \tau)$ mit
  - $Q = \{\{x, y\}\}$
  - $\delta = \emptyset$
  - $\tau(\{x, y\}) = \bot$
- $F_{\phi \wedge \psi} = F_\phi \sqcup F_\psi$

Diese kleinste TMS ist durch einen Normalformalgorithmus berechenbar. Der Einfachheit halber betrachten wir Mengen statt Konjunktionen von elementaren Infons und formulieren den Algorithmus wieder als eine nichtdeterministische Übergangsrelation, deren Iteration terminierend und konfluent ist. Sei $=_E$ der transitive und symetrische Abschluss von $\{(x, y) \mid x \doteq y \in E\} \cup \{(x, x) \mid x \in \operatorname{var} E\}$:

- $\{f(x_1 \dots x_n) \doteq f(y_1 \dots y_n)\} \cup E \Rightarrow \{x_1 \doteq y_1 \dots x_n \doteq y_n\} \cup E$ (Reduktion)
- $\{f(\bar{x}) \doteq g(\bar{y})\} \cup E \Rightarrow \bot$ falls $f \neq g$ (Clash)
- $\{t \doteq x\} \cup E \Rightarrow \{x \doteq t\} \cup E$ falls $t \notin \mathsf{Var}$ (Ausrichtung)
- $\{x \doteq t, y \doteq s\} \cup E \Rightarrow \{x \doteq t, t \doteq s\} \cup E$ falls $x =_E y$

Iteration dieser Regeln auf einer Menge von elementaren Infons führt entweder zu $\perp$ oder zu einer Menge S mit folgenden Eigenschaften:

- $S = B \cup E$
- $B = \{x_1 \doteq f_1(\bar{y}_1) \ldots x_n \doteq f_n(\bar{y}_n)\}$ und $E = \{u_1 \doteq v_1 \ldots u_m \doteq v_m\}$
- $x_i \neq_E x_j$ für alle $1 \leq i, j \leq n$ mit $i \neq j$

Eine Variable $x \in \operatorname{var} S$ heißt gebunden, falls $x =_E x_i$ für $1 \leq i \leq n$, ansonsten heißt sie frei. Die Funktoren f_i können auch nullstellig sein; dann heißen sie Konstanten, und der Variablenvektor $\bar{y}_i$ ist leer. Man sieht leicht, dass S als eine TMS $(Q, \mathsf{Type}, \mathsf{Att}, \tau, \delta)$ repräsentiert werden kann: Man wählt $Q = \{[x] \mid x \in \operatorname{var} S\}$, wobei $[x] = \{y \mid x =_E y\}$. Type ist der flache Verband aller f_i mit $\perp$ und $\top$ als Bottom- bzw. Topelement. Att ist die Menge der natürlichen Zahlen. Typisierungs- und Übergangsfunktion erhält man durch: $\tau([x_i]) = f_i$ für die gebundenen, $\tau([x]) = \perp$ für die freien Variablen. Ebenso $\delta([x_i], n) = [(\bar{y}_i)_n]$ für die gebundenen Variablen, und $\delta([x], n) = \perp$ sonst. Die Wohldefiniertheit erhält man aus der Tatsache, dass $=_E$ eine Äquivalenzrelation auf S ist.

Der klassische Unifikationsalgorithmus hat einen Nachteil, der ihn für unsere Zwecke unbrauchbar macht: Die Identität der Terme geht verloren. Betrachten wir ein Beispiel. Wir wenden nf auf eine Menge elementarer Infons an (und markieren dabei mit * den zu reduzierenden Term):

1. $\{x \doteq a, y \doteq a, x \doteq y^*\}$
2. $\{y \doteq a^*, y \doteq a, x \doteq y\}$
3. $\{y \doteq a, a \doteq a^*, x \doteq a\}$
4. $\{y \doteq a, x \doteq a\}$

Das liefert zwar den richtigen Unifikator, aber nicht die kleinste TMS F, für die gilt $F \models x = a \wedge y = a \wedge x = y$. Dieses F hätte nämlich nur einen Knoten: $\{x, y\}$. Die TMS F' aber, die die obige Lösungsmenge repräsentiert, hat zwei Knoten: $\{x\}$ und $\{y\}$. Mit dem neuen Algorithmus erhalten wir dagegen folgende Lösungsmenge:

1. $\{x \doteq a, y \doteq a, x \doteq y^*\}$
2. $\{y \doteq a, a \doteq a^*, x \doteq y\}$
3. $\{y \doteq a, x \doteq y\}$

In der Tat hat die TMS, die diese Lösungmenge repräsentiert, nur einen Knoten, nämlich $\{x, y\}$.

3.4 Prolog-Programme und Infon-Programme

Wir beobachten, dass sich jedes Prolog-Programm in eine Normalform bringen lässt, die unserer Definition von Infon-Programm entspricht, falls Basic als Gesamtheit dieser neuen elementaren Infons interpretiert wird. Man erhält diese Normalform, indem man jede Klausel K im Prolog-Programm mit folgendem Algorithmus umformt:

1. Sei $K \equiv h(t_1 \dots t_n) : -B$.
2. Seien $x_1 \dots x_n$ Variablen, die in K nicht vorkommen.
3. Setze $K = h(x_1 \dots x_n) : -x_1 = t_1 \wedge \dots \wedge x_n = t_n, B$.
4. Sei $K \equiv H : -\phi, b_1 \dots b_n$ mit $\phi \in \mathsf{Eq}$.
5. Wenn $n = 0$ ist K fertig, sonst:
6. Setze $i = 1$.
7. Sei $b_i \equiv b(t_1 \dots t_m)$.
8. Wenn $m = 0$ setze $i = i + 1$ und gehe zu 7.
9. Seien $x_1 \dots x_m$ Variablen, die in K nicht vorkommen.
10. Setze $\phi = \exists x_1 \dots x_m \phi \wedge x_1 = t_1 \wedge \dots \wedge x_m = t_m$.
11. Wenn $i < n$ setze $i = i + 1$ und gehe zu 7.
12. Sei $K \equiv H : -\phi, B$ mit $\phi \in \mathsf{Eq}$.
13. Setze $\phi = \phi^*$.

wobei

- $\emptyset^* = \emptyset$
- $(\{x \doteq t\} \cup E)^* = \{x \doteq t\} \cup E^*$ falls $t \in \mathsf{Var} \cup \mathsf{Konst}$
- $(\{x \doteq f(t_1 \dots t_n)\} \cup E)^* = \{x \doteq f(x_1 \dots x_n)\} \cup (\{x_1 \doteq t_1 \dots x_n \doteq t_n\} \cup E)^*$ wobei $x_1 \dots x_n \notin E \cup \{x\}$

3.5 Ein Metainterpreter

Wir wollen nun schrittweise einen Metainterpreter für die angebene Semantik programmieren. Angenommen, die vom Prologinterpreter intern gehandhabte Lösungsmenge sei wie in Abb. 2.1 strukturiert und ihr aktueller Zustand stehe in einer globalen Variablen `#Sol`[1] als Liste von Prologtermen zur Verfügung:

```
#Sol = [X1,...,Xn]
```

Die `X1,...,Xn` denotieren ihre Werte – die Liste ist also Subterm-abgeschlossen. Wir sagen `X` kommt in `#Sol` vor gdw `X` modulo `==` in `#Sol` vorkommt.

Nun können wir die Goalsyntax von Prolog um den Operator ? erweitern, dem wir folgende Semantik verleihen:

```
true:-!.
(Phi,Psi):-!
  Phi,Psi.
(Phi;Psi):-!
  Phi;Psi.
Goal:-
  clause(Goal,SubGoals),
  SubGoals.

?true:-!.
?(Phi,Psi):-!
  ?Phi,?Psi.
?(Phi;Psi):-!
  ?Phi;?Psi.
?Goal:-
  Goal=..[Pred|Args],
  freeze(#Sol),
  members(Args,#Sol),
  clause(Goal,SubGoals),
  ?SubGoals,
  melt(#Sol).
```

Diese Definition ist metasprachlich aufzufassen, d.h. die darin vorkommenden Variablen gehören selbst nicht zur Lösungsmenge. `clause(Goal,Body)` führt zu

[1] Das Kreuzzeichen `#` vor einer Variablen soll bedeuten, dass diese Variable global zur Verfügung steht.

einem Update von `#Sol`: Angenommen `Head:-Body` sei eine Klauselvariante, deren Variablen in `#Sol` nicht vorkommen. Falls `Goal=Head` gelingt, kommen nun die Argumente von `Head` und `Goal` in `#Sol` vor, die restlichen Variablen von `Body` jedoch nicht. `members/2` ist eine Fortsetzung von `member/2`. Es führt ebenfalls zu einem Update auf `#Sol`: Nach einem erfolgreichen Aufruf von `members(Args,#Sol)` kommen alle Variablen in `Args`, die vorher nicht in `#Sol` vorkamen, nun ebenfalls in `#Sol` vor.

```
members([],_).
members([X|R],Sol):-
  member(X,Sol),
  members(R,Sol).
```

Entscheidend ist das Zusammenspiel von `freeze/1` und `members/2`. Der Aufruf `freeze(Term)` friert alle freien Variablen in `Term` ein. Eine eingefrorene Variable `X` verhält sich wie eine Konstante, d.h. `X=Y` gelingt gdw `var(Y)` oder `X==Y` gelingen. Ein eingefrorener Term verhält sich also wie ein geschlossener Term; `melt/1` macht diesen Zustand rückgängig.

Durch das Einfrieren des Lösungsterms `#Sol` können die bis zu diesem Zeitpunkt eingeführten freien Variablen nicht weiter instantiiert werden. Betrachten wir nun die Argumente `Args` von `Goal`: Die Variablen in `Args`, die in `#Sol` vorkommen, nennen wir *Parameter*, die anderen *syntaktische Variablen*. Sie sind nicht eingefroren und werden daher durch das nichtdeterministische Prädikat `members(Args,#Sol)` an Objekte in `#Sol` gebunden. `Goal:-?SubGoals` ist also nach dem Aufruf von `members(Args,#Sol)` geschlossen, und das Antecedens `?SubGoals` kann getestet werden. Scheitert der Test, tritt solange Backtracking ein, bis entweder eine Substitution der syntaktischen Variablen in `Goal` durch Objekte in `#Sol` gefunden wurde, die vom Programm bewiesen werden kann, oder das Metaprädikat `?Goal` scheitert.

Der Operator `?` macht also aus einem Prädikat `Goal` ein Testprädikat. `?Goal` gelingt an einem beliebigen Punkt der Programmausführung, wenn es eine Bindung `GoalInst` der syntaktischen Variablen in `Goal` an Elemente von `#Sol` gibt, sodass `GoalInst` aus dem Programm folgt. Diese Bindung ist zwar auch ein Update der Lösungsmenge (es kommen ja neue Variablengleichungen hinzu), dennoch ändert sich `#Sol` durch ein gelungenes Testprädikat nicht, denn seine Elemente werden modulo `==` betrachtet.

Syntaktische Variablen sind genau diejenigen Variablen in einer Klausel, die nicht im Kopf vorkommen. Ein Testprädikat `?p(a,b)` gelingt, wenn es Konstanten `a`, `b` in `#Sol` gibt, für die `p(a,b)` aus dem Programm folgt. Es kann mehrere solche Konstanten geben und für jedes dieser Paare wird `?p(a,b)` gelingen. Ein Beispiel:

```
[0] listing.
p(a,b).
yes
[1] p(a,b), ?p(a,b).
yes
[2] p(a,b), p(a,b), ?p(a,b).
yes
[3] p(a,b), ?p(X,b).
X=a
[4] p(a,b), p(a,b), ?p(X,b).
X=a
More?
X=a
[5] p(a,b), p(a,b), ?p(X,Y).
X=a Y=b
More?
X=a Y=b
More?
X=a Y=b
More?
X=a Y=b
[6] retract(p(a,b)), assert(p(a,a)).
yes
[7] listing.
p(a,a).
[8] p(a,a), ?p(X,Y).
X=a Y=a
More?
X=a Y=a
More?
X=a Y=a
More?
X=a Y=a
[9]
```

Wenn wir die syntaktischen Variablen eines Goals in einer Merkmalsstruktur F statt im Herbranduniversum binden, dann sind syntaktisch gleiche Terme nicht identisch: Die syntaktischen Variablen rangieren über Q_F, der Menge der Knoten von F, und die Knoten einer Lösungsmenge sind die Äquivalenzklassen unifizierter Variablen. Es kann durchaus sein, dass zwei Variablen nicht unifiziert sind, aber syntaktisch gleiche Terme binden:

$$\{\ldots, x_i = a, \ldots, x_j = a, \ldots, x_i = \bar{y}_i, \ldots x_j = \bar{y}_j, \ldots\}$$

Dann können beide Terme als Werte einer syntaktischen Variablen auftauchen. Entscheidend sind also die Tokens, nicht die Types der Terme in einer Lösungsmenge.

Die oben gegebene Definition ist nur vorläufig. Im folgenden wird eine explizite und lauffähige Implementation von ? in Prolog angegeben. Leider stellt uns Standard-Prolog seine Lösungsmenge nicht als erststufiges Objekt zur Verfügung. Wollen wir das erreichen, müssen wir Prolog metainterpretieren. Glücklicherweise gibt es eine Prolog-Implementation, die tiefe Eingriffe in den Unifikationsalgorithmus erlaubt, ohne dass die Unifikationsoperation selbst als syntaktische Operation auf Lösungsmengen metainterpretiert werden muss: ECL^{i}PS^e.

3.6 Metaterme

ECL^{i}PS^{e}stellt ein elegantes und effizientes Werkzeug zur Modifikation des eingebauten Unifikationsalgorithmus von Prolog zur Verfügung: Metaterme. Metaterme spielen in der vorliegenden Implementation eine wichtige Rolle und sollen daher kurz erläutert werden. Ein Metaterm ist eine Variable, der eine Liste von Attributen zugeordnet ist, die ihr Unifikationsverhalten bestimmen. Attribute können beliebige Prologterme sein. Jedes Attribut eines Metaterms ist durch einen Namen gekennzeichnet, über den das benutzerdefinierte Prädikat identifiziert wird, das bei der Unifikation des Metaterms mit einem anderen Metaterm oder einer Nichtvariablen aufgerufen wird. Dieses Prädikat, der sogenannte (Unifikations-)Handler des Attributs, ist zweistellig: Beim Aufruf enthält das erste Argument den Unifikationspartner (ein Metaterm oder eine Nichtvariable) und das zweite Argument enthält das Attribut selbst.

Die Syntax eines Metaterms `Var` ist:

```
Var{Name0:Attr0,...,NameN:AttrN}
```

Für jedes Attribut müssen Unifikationshandler festgelegt werden. Das geschieht mit `meta_attribute/2`:

```
:- meta_attribute(Name0,HandlerList0).
...
:- meta_attribute(NameN,HandlerListN).
```

Der eingebaute Unifikationsalgorithmus unterstützt vordefinierte Prädikate, deren Semantik durch benutzerdefinierte Unifikationshandler modifiziert werden kann. Diese sind: `unify` (=), `test_unify` (==), `copy_term`, `compare_instances`, `delayed_goals`, `delayed_goals_number` und `print`. Für jedes davon kann ein Unifikationshandler angegeben werden. Die allgemeinste Form einer Handlerspezifikation ist also:

```
:- meta_attribute(Name0,[unify:UnifyHandler0,
                         test_unify:TestUnifyHandler0,...]).
...
:- meta_attribute(NameN,[unify:UnifyHandlerN,
                         test_unify:TestUnifyHandlerN,...]).
```

Metaterme werden mit `add_attribute(Var,Name,Attr)` erzeugt bzw. modifiziert. Ist `Var` eine freie Variable, so wird sie durch diesen Aufruf zum Metaterm

```
Var{Name0:X0,...,Name:Attr,...,NameN:Xn}
```

wobei die `X` freie Variablen sind. Ist `Var` der Metaterm

```
Var{Name0:Attr0,...,Name:AttrI,...,NameN:AttrN}
```

so wird er durch den Aufruf zu:

```
Var{Name0:Attr0,...,Name:Attr,...,NameN:AttrN}
```

sofern `Attr=AttrI` gelingt.

Die Abfrage der Attribute eines Metaterms ist etwas trickreich. Man benötigt dazu den Matchoperator `-?->`. Folgt dieser Operator unmittelbar auf den Kopf einer Klausel, wird dieser mit dem aufrufenden Goal gematcht statt unifiziert. Damit lässt sich ein Prädikat `get_attribute/3` folgendermaßen definieren:

```
get_attribute(Var{name0:Attr},name0,A):-
  -?-> A = Attr.
...
get_attribute(Var{nameN:Attr},nameN,A):-
  -?-> A = Attr.
```

Würden wir hier unifizieren, dann würde der Aufruf `get_attribute(X,name,A)` auch dann gelingen, wenn `X` eine freie Variable ist (sie würde durch den Aufruf zum Metaterm `X{name:A}`) oder eine Nichtvariable. Das ließe sich noch durch Testprädikate abfangen. Aber in jedem Fall würde der Unifikationshandler aktiviert werden, was zu unerwünschten Seiteneffekten führen könnte. Man beachte noch, dass die Unifikation `A=Attr` erst nach dem Match stattfinden darf. Sonst würde zwar der Aufruf `get_attribute(X{name:attr},name,attr)` gelingen, aber der Aufruf `get_attribute(X{name:attr},name,Attr)` scheitern.

In ECL^{i}PS^{e}kann ein Programm in (Prädikats-)Namensräume zerlegt werden, sogenannte *Module*. Ein Modul kann Prädikate exportieren und importieren. Es ist gute Programmierpraxis, für jedes Attribut ein eigenes Modul zu erzeugen und den Namen des Moduls als Attributsnamen zu vergeben. ECL^{i}PS^{e}ergänzt nämlich automatisch den Namen der Modulumgebung, wenn der Name eines Attributs nicht explizit angegeben wird. Ein im Modul `module` notierter Metaterm `Var{Attr}` hat also die explizite Form `Var{module:Attr}`. Auf diese Weise kann der Programmierer zweistelligen Versionen von `add_attribute` und `get_attribute` verwenden, was die Lesbarkeit des Programms erhöht.

Module, die ein Attribut besitzen und Unifikationshandler dafür definieren, nennt man *Extensionen*. Sie definieren auch die Prädikate, die dieses Attribut zuweisen oder modifizieren, und exportieren sie. Ein Metaterm `X{...}` in einem Programm hat immer so viele Attribute, wie das Programm Extensionen kompiliert hat. Man spricht dann auch von den Extensionen von `X`. Hier gilt es einen subtilen Punkt zu beachten: Vergibt eine Extension `ext` ein Attribut `Attr` an eine freie Variable `X`, so wird diese ja zum Metaterm `X{...,ext:Attr,...}`. Was ist nun mit den anderen Attributen von `X`? Die Regel lautet: Für alle anderen Extensionen von `X` werden freie Variablen als Attribute vergeben. Diese Tatsache muss bei den Fallunterscheidungen eines Unifikationshandlers im Auge behalten werden.

3.7 Variablen einfrieren

Wir wollen nun ein Beispiel für einen nützlichen Eingriff in den Unifikationsalgorithmus geben, der mit Metatermen realisiert werden kann: das zeitweilige Einfrieren von Termen. Eine eingefrorene Variable `X` verhält sich wie eine Konstante, d.h. `X=Y` gelingt gdw `Y` eine freie Variable ist oder `X==Y` gelingt. Ein eingefrorener offener Term verhält sich daher wie ein Grundterm. Wir benötigen zwei Prädikate, `freeze/1` und `melt/1`, sowie einen Unifikationshandler `unify_frozen` für die Operation `unify`:

```
:- module_interface(freezer).
:- export freeze/1, melt/1.
:- import setarg/3 from sepia_kernel.
```

`sepia_kernel` ist ein mitgeliefertes Modul von ECL^{i}PS^e, das `setarg/3` exportiert. `setarg(N,Term,Arg)` substituiert das `N`te Argument von `Term` durch `Arg`, falls dieses keine Variable ist, ansonsten unifiziert es `Arg` mit dieser.

Die Idee des Programms ist folgende: Wird eine Variable eingefroren, so erhält sie das Attribut `frozen(Id)`, wobei `Id` eine neue Variable ist, die als Identifikator

dient. Wird eine bereits eingefrorene Variable ein zweites Mal eingefroren, so sorgt die Semantik von `setarg/3` dafür, dass der neue Identifikator mit dem alten unifiziert wird. Beim Auftauen wird der Identifikator mit `setarg/3` auf `no` gesetzt. Ein Metaterm `X{frozen(no)}` muss von `unify_frozen/2` wie eine freie Variable behandelt werden.

Bei der Fallunterscheidung in `unify_frozen/2` muss - wie weiter oben schon ausgeführt, der Fall betrachtet werden, dass das `freezer`-Attribut undefiniert, d.h. eine freie Variable ist, weil eine andere Extension die betrachtete Metavariable erzeugt hat:

```
:- begin_module(freezer).

:- meta_attribute(freezer,[unify:unify_frozen/2]).
%============
% Meta + Meta
%============
unify_frozen(_{X},Y):- % Linke Seite ohne Attribut
  -?-> var(X),!.
unify_frozen(_{X},Y):- % Rechte Seite ohne Attribut
  -?-> var(Y).
unify_frozen(_{frozen(X)},frozen(Y)):- % Linke Seite aufgetaut
  -?-> X == no,!.
unify_frozen(_{frozen(X)},frozen(Y)):- % Rechte Seite aufgetaut
  -?-> Y == no.
unify_frozen(_{frozen(X)},frozen(Y)):- % Beide Seiten gefroren
  -?-> X == Y.
%==============
% Nonvar + Meta
%==============
unify_frozen(X,Y):- % Rechte Seite ohne Attribut
  nonvar(X), var(Y),!.
unify_frozen(X,frozen(Id)):- % Rechte Seite aufgetaut
  nonvar(X), Id == no,!.
unify_frozen(X,frozen(Id)):- % Rechte Seite gefroren
  nonvar(X), var(Id), fail.
```

Die Unifikation eines Metaterms `M` mit einer freien Variablen gelingt immer. Der Unifikationshandler wird nur bei der Unifikation eines Metaterms mit einem anderen Metaterm oder einer Nichtvariablen aufgerufen, und zwar unmittelbar nach stattgefundener Unifikation. Seine Aufgabe besteht darin, zu überprüfen, ob die

Unifikation von `M` rechtens war. Gelingt der Aufruf, so bleibt die Unifikation bestehen. Scheitert er, so wird sie rückgängig gemacht. Zur Überprüfung steht im ersten Argument der Unifikationspartner zur Verfügung und im zweiten Argument das `freezer`-Attribut von `M`. `M` selbst steht nicht zur Verfügung. Das ist der Grund, warum von `freeze/1` eine Identifikationsvariable generiert werden muss.

Die ganze Arbeit wird in `freeze_var/1` (bzw. `melt_var/1`) getan. Es sind zwei Fälle zu betrachten: Das Argument ist frei oder ein Metaterm. Ist es frei, so muss das Attribut `frozen(Id)` (`Id` neu) zugewiesen werden. Ist es ein Metaterm, so sind wieder zwei Fälle zu betrachten: Das Attribut `Attr` ist frei (undefiniert) oder matcht mit `frozen(Arg)`. Im ersten Fall wird `Attr` an `frozen(Id)` gebunden; ansonsten wird `Arg` durch `Id` ersetzt. War `Arg` eine Variable (`X` also bereits gefroren), so wird `Arg` mit `Id` unifiziert:

```
freeze_var(X):-
  free(X),
  add_attribute(X,frozen(Id)).

freeze_var(X{Attr}):-
  -?-> (var(Attr) -> Attr=frozen(Id); setarg(1,Attr,Id)).

freeze_vars([]).
freeze_vars([X|Vars]):-
  freeze_var(X),
  freeze_vars(Vars).

freeze(X):-
  term_variables(X,Vars), % Vars ist die Liste der Variablen in X
  freeze_vars(Vars).
```

`melt_var/1` funktioniert komplementär zu `freeze_var/1`, die Fallunterscheidungen sind dieselben. Eine freie Variable *ist* bereits aufgetaut. Eine gefrorene Variable wird aufgetaut, indem das `frozen`-Argument auf `no` gesetzt wird.

```
melt_var(X):-
  free(X).
melt_var(X{Attr}):-
  -?-> (var(Attr) -> Attr=frozen(no); setarg(1,Attr,no)).

melt_vars([]).
```

```
melt_vars([X|Vars]):-
  melt_var(X),
  melt_vars(Vars).

melt(X):-
  term_variables(X,Vars),
  melt_vars(Vars).

:- end_module(freezer).
```

3.8 Goal Delaying

Metaterme können in ECL^{i}PS^{e}nicht nur zur Modifikation des Unifikationsalgorithmus von Prolog verwendet werden, sondern auch zur Veränderung der Standardreihenfolge der Goalreduktion von links nach rechts. Solche Modifikationen von Prolog werden unter dem Begriff *Delay-Techniken* zusammengefasst. Die Datenstruktur, die ECL^{i}PS^{e}zu diesem Zweck zur Verfügung stellt, heißt *suspension*. Eine Suspension, im folgenden *suspendiertes Goal* genannt, ist ein Goal, dessen Aufruf nicht von seiner Position im Goal-Stapel, sondern vom Eintreten eines bestimmten Ereignisses abhängt. Üblicherweise ist das die Bindung oder Instantiierung einer oder mehrerer Variablen. ECL^{i}PS^{e}stellt sehr elementare Prädikate zur Verfügung, die suspendierte Goals erzeugen, als Attribute zuweisen, zur Ausführung übergeben und löschen können.

Soll die Bindung oder Instantiierung einer Variablen den Aufruf eines suspendierten Goals bewirken, so muss diese Variable ein Metaterm sein, der ein `suspend`-Attribut hat. Dieses Attribut ist ein dreistelliger Term, dessen Argumente Listen von suspendierten Goals enthalten. Verwaltet wird dieses Attribut von der Extension `suspend`, die suspendierte Goals erzeugt und in eine dieser Listen einhängt. Sie stellt auch die Unifikationshandler zur Verfügung, die entscheiden, was bei der Unifikation eines Metaterms mit seinen suspendierten Goals geschehen soll. Die drei Listen stehen für die drei Klassen von Unifikationsereignissen, die den Aufruf suspendierter Goals bewirken können: `inst` (Bindung an eine Nichtvariable), `bound` (`inst` oder Bindung an einen Metaterm mit definiertem `suspend`-Attribut) und `constrained` (`bound` oder Nachricht von einer anderen Extension, dass eine Einschränkung stattgefunden hat). Was eine Einschränkung ist, kann nur der Benutzer definieren. Hat eine benutzerdefinierte Extension einen Metaterm `X` irgendwie eingeschränkt, so kann sie `suspend` durch den Aufruf `notify_constrained(X)` davon benachrichtigen. (Dies ist natürlich nur dann nötig, wenn die Einschränkung von `X` durch die Unifikation mit einem Metaterm erfolgte, dessen `suspend`-Attribut undefiniert ist.)

Ein suspendiertes Goal ist kein Prologterm, sondern eine interne Datenstruktur, die vom ECL^{i}PS^e-Scheduler gehandhabt werden kann. Ein suspendiertes Goal wird durch den Aufruf `make_suspension(Goal,Priority,Susp)` erzeugt und durch den Aufruf `insert_suspension(Vars,Susp,Index)` in eine der drei Listen im `suspend`-Attribut der Variablen in `Vars` eingehängt. (`Index` kann also die Werte `inst`, `bound` oder `constrained` annehmen.) Tritt eines der drei Unifikationsereignisse ein, übergibt die `suspend`-Extension mit `schedule_woken(SuspList)` die entsprechende Liste dem Scheduler, ohne ihn dabei zu aktivieren. Das suspendierte Goal ist nun *erwacht* (davor war es *schlafend*), aber noch nicht ausgeführt. Wird der Scheduler aktiviert, was defaultmäßig nach der Beendigung des aktuellen Reduktionsschrittes geschieht, überprüft er die Priorität aller erwachten suspendierten Goals. Diejenigen unter ihnen, deren Priorität höher ist als die Priorität des aktuellen Goals, werden aufgerufen.

Wir wollen uns in das trickreiche und sehr tiefgreifende Suspension Handling von ECL^{i}PS^{e}nicht weiter vertiefen, da wir im folgenden lediglich ein bereits vordefiniertes Prädikat benötigen: `delay/2`. `delay(Term,Goal)` macht aus `Goal` ein suspendiertes Goal `Susp` und hängt es in die `bound`-Liste des `suspend`-Attributs aller Variablen in `Term`. Mit anderen Worten: Es stellt den Aufruf von `Goal` zurück, bis eine der Variablen in `Term` gebunden wird. Wir wollen damit ein Metaprädikat `constraint/1` definieren, das die Ausführung eines nichtdeterministischen Goals solange verschiebt, bis dessen Argumente ausreichend instantiiert sind, um eine deterministische Reduktion zu gewährleisten.

`constraint/1` liefert ein Reduktionsverfahren für Prologprogramme, deren Klauseln folgende Syntax haben:

```
Head :- {constraint(Constraint)} Goal.
```

`Head` steht für ein Prologprädikat, `Constraint` und `Goal` stehen für Prologgoals nach der Standardsyntax. `Goal:- true.` wird wie üblich als `Goal.` geschrieben. `true` und die logischen Konjunktoren `,` und `;` behalten ihre ihre übliche Bedeutung:

```
constraint(true):-!.
constraint((Phi,Psi)):-!,
  constraint(Phi),
  constraint(Psi).
constraint((Phi;Psi)):-!,
  constraint(Phi);
  constraint(Psi).
```

Soll ein Goal wie ein Constraint behandelt werden, sind drei Fälle zu unterscheiden: Das Goal ist

- vordefiniert, deterministisch und akzeptiert jedes Bindungsmuster.
- vordefiniert, deterministisch und akzeptiert nicht jedes Bindungsmuster.
- benutzerdefiniert.

Bei vordefinierten Goals kann das Programm nicht selbst entscheiden, ob es sich um eine deterministisches oder nichtdeterministisches Prädikat handelt. Deshalb muss diese Information explizit gegeben werden:

```
deterministic(=).       % Unifikation
deterministic(\=).      % nicht unifizierbar (Test)
deterministic(~=).      % nicht unifizierbar (Constraint)
deterministic(==).      % gleich (Test)
deterministic(\==).     % ungleich (Test)
deterministic(var).     % frei (Metapraedikat)
deterministic(nonvar). % instantiiert (Metapraedikat)
```

Ist ein Goal vordefiniert und deterministisch, kann es aufgerufen werden:

```
constraint(Goal):-
  Goal=..[Rel|_],
  deterministic(Rel),!,
  call(Goal).
```

Leider lässt ECL^{i}PS^{e}keine variablen Funktoren zu. Will man einen Term in Funktor und Argument zerlegen, muss man `Term =..[Func|Args]` aufrufen. Dieser Aufruf führt allerdings zu einer Fehlermeldung, wenn `Term` und `Func` ungebunden sind. In diesem Fall wäre es aber auch möglich, das Goal zu delayen. Mit `constraint/1` erzeugt man genau dieses Verhalten:

```
constraint(Goal):-
  Goal=..[=..,Term,[Rel|Args]],!,
  ( var(Term) ->
    ( atom(Rel) ->
        !,call(Goal);
        !,var(Rel),delay([Term,Rel],Goal)
    );
    call(Goal)
  ).
```

Jedes weitere vordefinierte Prädikat, das für bestimmte Bindungsmuster undefiniert ist, muss auf diese Weise abgefangen werden. `=..` soll hier als Beispiel genügen.

Mit `clause(Head,Body)` kann man sich alle dynamischen Klauseln ausgeben lassen, deren Kopf mit `Head` und deren Körper mit `Body` unifiziert. Dynamische Klauseln werden mit `dynamic/1` deklariert, beispielsweise:

```
:- dynamic member/2, append/3, subterm/2.
```

Um zu entscheiden, ob ein Goal aufgerufen oder zurückgestellt werden soll, berechnen wir mit `choice_points/2` die Anzahl der Wahlpunkte, den ein Aufruf des Goals erzeugen würde. `choice_points(Goal,Num)` gelingt, wenn mindestens ein Wahlpunkt gefunden wird. Ist er eindeutig, werden die Subgoals von `Goal` ermittelt und ebenfalls als Constraint ausgeführt. Gibt es mehrere Wahlpunkte, wird `constraint(Goal)` zurückgestellt, bis eine der freien Variablen in `Goal` gebunden wird. (`term_variables/2` ist vordefiniert und liefert eine Liste der freien Variablen eines Terms.)

```
constraint(Goal):-
  choice_points(Goal,Num),
  ( Num == 1 ->
      !,cclause(Goal,SubGoals),
      !,constraint(SubGoals)
    ;
      term_variables(Goal,Vars),
      delay(Vars,constraint(Goal))
  ).
```

Bei einem Reduktionsversuch müssen die Constraints im Körper einer Klausel mitausgeführt werden, sonst wird eine deterministische Reduktion fälschlicherweise als nichtdeterministisch behandelt:

```
:- dynamic member/2.

member(X,[X|_]).
member(X,[Y|R]):-
  constraint(X~=Y),
  member(X,R).
```

Hier würde der Aufruf `constraint(member(a,[a,b,c]))` zu einem Delay führen, obwohl die Reduktion in der intendierten Semantik eindeutig ist. Wir müssen daher eine modifizierte Version von `clause/2` verwenden: `cclause/2`:

```
cclause(Head,SubGoals):-
  clause(Head,Body),
  split(Body,Constraints,SubGoals),
  call(Constraints).

split(constraint(Cons),constraint(Cons),true):-!.
split((constraint(Cons),Goals),constraint(Cons),Goals):-!.
split(Goals,true,Goals).
```

Jetzt können wir `choice_points/2` definieren. Subtilitäten der `findall/3` Implementation von ECL^{i}PS^{e}zwingen uns dazu, `subcall/2` zu verwenden: Durch einen Aufruf von `Goal` mit `subcall(Goal,Delayed)` werden alle aufgrund dieses Aufrufs zu suspendierenden Goals in der Liste `Delayed` gesammelt anstatt wirklich suspendiert zu werden. Der Benutzer kann dann selbst entscheiden, was er damit machen will. Wir schmeißen sie einfach weg:

```
choice_points(Goal,Num):-
  findall(Goal,subcall(cclause(Goal,_),Delayed),ChoicePoints),
  length(ChoicePoints,Num),
  Num > 0.
```

3.9 Lösungsmengen als Terme

Nachdem wir uns jetzt an einem kleinen Beispiel mit dem Metaterm-Mechanismus von ECL^{i}PS^{e}vertraut gemacht haben, können wir unser Hauptziel angreifen: Die Definition einer Relation `update(Goal,Sol,Update)`, die die kleinste Herbrand-erfüllbare Lösungsmenge `Update` errechnet, zu der es eine Klausel `Head:-Body` in der dynamischen Datenbank gibt, sodass `[Goal=Head|Sol]` erfüllungsäquivalent zu `Update` ist und `subset(Sol,Update)` gelingt. Dazu müssen wir uns entscheiden, wie wir eine Lösungsmenge als Prologterm repräsentieren wollen. Eine naheliegende Antwort wäre: Als Liste von Termen. D.h. aus

$$\{x_1 = f_1(\bar{z}_1), \ldots, x_m = f_m(\bar{z}_m), \ldots, x_{m+1} = \bar{y}_{m+1}, \ldots, x_n = \bar{y}_n\}$$

wird

```
[f1(Z1),...,fm(ZM),XMplus1,...,XN]
```

Das Problem ist nur, dass die Identität der gebundenen Variablen dabei verloren geht. Angenommen, das Programm bestehe aus der einzigen Klausel `p(a,a)`. Die Frage ist: Gelingt nun `update((?p(X,Y),X=Y),[a,a],[a])`? Nach der intendierten Semantik sollte der Aufruf gelingen, denn `X` kann an das erste Vorkommen von `a` gebunden werden und `Y` an das zweite. Nach der Unifikation `X=Y` (die ein Update ist) sind die beiden Vorkommen identisch und `a` taucht nur noch einmal in der Lösung auf. Wir wollen also das folgende Verhalten für `update/3`:

```
[1] update(f(g(a),X)=f(Y,g(Z)),[],Sol).
Sol=[f(g(a),g(Z)),g(a),a,g(Z),Z]
yes
[2] update(f(g(a),X)=f(X,g(Z)),[],Sol).
Sol=[f(g(a),g(a)),g(a),a]
yes
[3]
```

Um das programmieren zu können, müssen wir die Vorkommen der Terme markieren, sodass wir, wenn wir zwei Vorkommen syntaktisch gleicher Terme unifizieren, ihre Markierungen ebenfalls identifizieren können. In einem zweiten Schritt können wir dann die überflüssig gewordenen Termvorkommen entfernen. Unter Vorkommen ist natürlich Vorkommen eines Terms auf der Liste gemeint, nicht Vorkommen als Teilterm eines Terms auf der Liste. Außerdem ist klar: Haben wir die Markierungen zweier Termvorkommen identifiziert, so müssen wir auch die Markierungen ihrer Teilterme paarweise identifizieren. Schließlich handelt es sich ja um eine Implementation des Unifikationsalgorithmus'.

Die Lösung unseres Problems bieten wieder einmal die Metaterme von ECL^{i}PS^e. Testprädikate sollen ja ihre syntaktischen Variablen an Elemente der Lösungsliste binden. Anstatt dort nun die Werte der Variablen aufzulisten, listen wir die Variablen selbst auf und beschränken bei jedem Update ihre Unifizierbarkeit in einem Metaterm-Attribut. Eine Lösungsmenge

$$\{x_1 = f_1(\bar{z}_1), \ldots, x_m = f_m(\bar{z}_m), x_{m+1} = \bar{y}_{m+1}, \ldots, x_n = \bar{y}_n\}$$

wird also durch die folgende Liste von Metatermen repräsentiert:

```
[X1{f1(Z1)},...,Xm{fm(Zm)},XMplus1{YMplus1},...,Xn{Yn}]
```

Die `YMplus1,...,Yn` sind `free`-Variable, d.h. metasprachlich freie Variable, für die `free/1` gelingt. Die `X1,...,Xn` sind `meta`-Variable, d.h. metasprachlich freie

Variable für die `meta/1` gelingt. Die `X1,...,Xm` sind objektsprachlich gebunden, d.h. objektsprachlich werden sie so behandelt, als ob sie an `f1(Z1),...,fm(Zm)` gebunden sind. Die restlichen `meta`-Variablen auf der Liste (`XMplus1,...,Xn`) sind objektsprachlich freie Variable.

Die Idee ist folgende: Jede objektsprachliche Variable taucht als Metaterm auf der Lösungsliste auf, der ihre Bindung als Attribut enthält. Zwei Metaterme dürfen unifizieren, wenn ihre Attribute unifizieren. Daraus folgt: Ist eine Variable objektsprachlich frei, so ist sie durch eine `free`-Variable in ihrem Attribut beschränkt und kann mit jeder anderen objektsprachlichen Variablen unifizieren. Ist sie objektsprachlich gebunden, so darf sie nur mit objektsprachlichen Variablen unifizieren, deren Attribut mit ihrem eigenen Attribut unifiziert. Bei diesem Unifkationsvorgang werden rekursiv alle Teilterme der Attribute unifiziert. Danach gilt es nur noch aufzuräumen, d.h. Mehrfachvorkommen desselben (modulo `==`) Metaterms zu beseitigen:

```
:- module_interface(meta_interpreter).
:- import freeze/1, melt/1 from freezer.
:- begin_module(meta_interpreter).
:- meta_attribute(meta_interpreter,[unify:meta_unify/2]).
:- dynamic unify/2, bye/0.

% ===========
% Meta + Meta
% ===========
meta_unify(_{Left},Right):-
  -?-> Left=Right.
% =============
% Nonvar + Meta
% =============
meta_unify(Left,Right):-
  nonvar(Left),
  Left=Right.

unify(X,X).
```

Nach dem oben Gesagten mag es erstaunen, dass wir eine Unifikation von Nichtvariablen und Metatermen zulassen. Dieser Fall sollte eigentlich nie eintreten, denn ein Update auf der Lösungsmenge kann nur auf zwei Weisen geschehen: Es kommen neue `meta`-Variablen hinzu oder es werden alte unifiziert. Allerdings wollen wir mit `update/3` einen Metainterpreter realisieren, und für die Ausgabe der Lösungsmenge bzw. des instantiierten Goals ist es notwendig (oder zumindest lesbarer) die gefundenen Bindungen zu realisieren. Das geschieht mit `bind/1`:

```
loop:-
  write(?-),
  read(Goal),
  update(Goal,[],Sol),
  bind(Sol),
  writeln(Goal),
  fail.
loop:-
  retract(bye),!.
loop:-
  loop.

bind([]).
bind([Var{Value}|Sol]):-
  -?->
  Var=Value,
  bind(Sol).
```

Indem wir `unify/2` als dynamisch deklarieren, können wir mit `clause/2` darauf zugreifen. Das ist notwendig, um die objektsprachliche Unifikation zu interpretieren. `update/3` funktioniert nun folgendermaßen: Bevor wir mit `clause/2` einen passenden `Head` für das zu reduzierende `Goal` suchen, müssen wir die Argumente von `Goal` durch frische Variablen ersetzen. Wir können nicht einfach `clause(Goal,SubGoals)` aufrufen, denn `Goal` könnte `meta`-Variablen enthalten, die durch diesen Aufruf gebunden würden. Die in `Goal` enthaltenen `meta`-Variablen befinden sich bereits in der Lösungsmenge, alle anderen sind neu. Sie müssen der Lösungsmenge als `meta`-Variablen hinzugefügt werden. Außerdem muss jeder komplexe Term in `Goal` in seine Teilterme zerlegt und mit frischen Variablen markiert werden, denn Lösungsmengen sind Teilterm-abgeschlossen. Diese Aufgabe erfüllt `extend_solution/4`. Im zweiten Argument liefert es die Aufrufvariablen von `Head` und `Goal` zurück, die nach der Erweiterung der Lösungsmenge unifiziert werden. War die Unifikation erfolgreich, können einige Variable in der Lösungsmenge mehrfach vorkommen. `compact_solution/2` beseitigt diese Mehrfachvorkommen und die Lösungsmenge wird mit den Subgoals der gefundenen Klausel geupdatet:

```
update(bye,Sol,Sol):-!,
  assert(bye),fail.
update(true,Sol,Sol):-!.
```

```
update((Phi,Psi),Sol0,Sol2):-!,
  update(Phi,Sol0,Sol1),
  update(Psi,Sol1,Sol2).
update(?Goal,Sol,Sol):-!, % Wird im folgenden Abschnitt
  test(Goal,Sol).         % besprochen
update(Goal,Sol0,Sol1):-
  Goal=..[=,LHS,RHS],!,
  update(unify(LHS,RHS),Sol0,Sol1).
update(Goal,Sol0,Sol4):-
  Goal=..[Pred|GoalArgs],
  length(GoalArgs,N),
  length(HeadArgs,N),
  Head=..[Pred|HeadArgs],
  clause(Head,Body),
  extend_solution(GoalArgs,GoalVars,Sol0,Sol1),
  extend_solution(HeadArgs,HeadVars,Sol1,Sol2),
  GoalVars=HeadVars,
  compact_solution(Sol2,Sol3),
  update(Body,Sol3,Sol4).
```

Betrachten wir nun `extend_solution/4` genauer. Das erste Argument enthält eine Liste mit Termen. Diese Terme zerfallen in drei Klassen: Nichtvariable, `free`-Variable und `meta`-Variable. Die letzteren kommen in der Lösungsmenge bereits vor und brauchen daher nur in die Aufrufliste übernommen zu werden. `free`-Variable sind neu. Da sie unbeschränkt sind, werden sie in einen Metaterm mit einem leeren Attribut (`Free`) umgewandelt und in die Aufrufliste übernommen. Nichtvariable, d.h. komplexe Terme und Konstanten, sind ebenfalls neu, aber anonym. Für sie und ihre Teilterme müssen rekursiv neue Variablen generiert werden (`New`), die sie markieren. Was ist nun, wenn wir einen komplexen Term oder eine Konstante innerhalb einer Klausel identifizieren wollen, z.B.:

```
s(VPHead=fin,List0,List2):-
  np(NPHead,List0,List1),
  vp(VPHead,List1,List2).
```

Das ist so zu lesen, dass die jeweils ersten Argumente des Kopfes und des `vp/3`-Goals nicht nur syntaktisch gleich, sondern identisch sein sollen. Diese Syntax lässt Prolog nicht zu. Angenommen, wir schreiben:

```
s(fin,List0,List2):-
  np(NPHead,List0,List1),
  vp(fin,List1,List2).
```

dann erhalten wir zwei verschiedene, wenn auch syntaktisch gleiche `fin`-Elemente in der Lösungsmenge. Wir müssen die Klausel also so formulieren:

```
s(VPHead,List0,List2):-
  VPHead=fin,
  np(NPHead,List0,List1),
  vp(VPHead,List1,List2).
```

Die Notation `s(VPHead=fin,List0,List2)` ist aber sehr praktisch. Deshalb, und nur deshalb, haben wir die zweite Klausel von `extend_solution/4` aufgenommen. Semantisch gesehen ist sie überfüssig.

```
extend_solution([],[],Sol,Sol).
extend_solution([Eq|Args],[Var|Vars],Sol0,[Var{Type}|Sol2]):-
  nonvar(Eq),
  Eq=..[=,Var,Term],
  var(Var),!,
  Term=..[Func|SubTerms],
  length(SubTerms,N),
  length(NewVars,N),
  Type=..[Func|NewVars],
  extend_solution(SubTerms,NewVars,Sol0,Sol1),
  extend_solution(Args,Vars,Sol1,Sol2).
extend_solution([Term|Args],[New|Vars],Sol0,[New{Type}|Sol2]):-
  nonvar(Term),
  Term=..[Func|SubTerms],
  length(SubTerms,N),
  length(NewVars,N),
  Type=..[Func|NewVars],
  extend_solution(SubTerms,NewVars,Sol0,Sol1),
  extend_solution(Args,Vars,Sol1,Sol2).
extend_solution([Var|Args],[Var|Vars],Sol0,[Var|Sol1]):-
  free(Var),
  add_attribute(Var,Free),
  extend_solution(Args,Vars,Sol0,Sol1).
extend_solution([Meta|Args],[Meta|Vars],Sol0,Sol1):-
  meta(Meta),
  extend_solution(Args,Vars,Sol0,Sol1).
```

Nach erfolgreicher Unifikation der Aufrufvariablen, müssen die Mehrfachvorkommen von `meta`-Variablen beseitigt werden:

```
compact_solution([],[]).
compact_solution([X|List],Set):-
  meta_member(X,List),!,
  compact_solution(List,Set).
compact_solution([X|List],[X|Set]):-
  compact_solution(List,Set).

meta_member(X,[Y|R]):-
  X == Y,!.
meta_member(X,[_|R]):-
  meta_member(X,R).
```

3.10 Testprädikate

Jedes Prädikat `Goal` kann durch den Operator `?` zum Testprädikat `?Goal` gemacht werden. Ein Update mit einem Testprädikat darf die Lösungsmenge nicht verändern. Daher sieht die `update/3`-Klausel für Testprädikate so aus:

```
update(?Goal,Sol,Sol):-!,
  test(Goal,Sol).
```

`test/2` beginnt zunächst wie `update/3`. Die Argumente des Testgoals und eines passenden `Heads` werden normalisiert und der Lösungsmenge hinzugefügt. Bevor wir nun aber die Aufrufvariablen unifizieren, frieren wir die Lösungsmenge ein, wodurch diejenigen Variablen in `GoalVars`, die ungefroren bleiben, zu syntaktischen Variablen werden. Nur diese können von `members/2` an Elemente der Lösungsmenge gebunden werden, und zwar nur an solche, deren Attribut eine Instanz ihres eigenen ist. Der Aufruf `GoalVars=HeadVars` gelingt ebenfalls nur, wenn die Attribute der Variablen in `GoalVars` Instanzen der Attribute ihrer Unifikationspartner sind.

```
test(true,Sol):-!.
test((Phi,Psi),Sol):-!,
  test(Phi,Sol),
  test(Psi,Sol).
test(Goal,Sol):-
  Goal=..[Pred|GoalArgs],
  length(GoalArgs,N),
  length(HeadArgs,N),
  Head=..[Pred|HeadArgs],
```

```
  clause(Head,Body),
  extend_solution(GoalArgs,GoalVars,Sol,Sol1),
  extend_solution(HeadArgs,HeadVars,Sol1,Sol2),
  freeze_solution(Sol),
  members(GoalVars,Sol),
  GoalVars=HeadVars,
  test(Body,Sol),
  melt_solution(Sol).

members([],_).
members([X|R],Sol):-
  member(X,Sol),
  members(R,Sol).
```

Das Einfrieren der Lösungsmenge hat den Zweck, eine weitere Instantiierung ihrer Variablen zu blockieren. Eine Instantiierung kann auf zwei Weisen geschehen: Entweder wird eine objektsprachlich freie Variablen gebunden oder zwei objektsprachlich gebundene Variablen unifiziert. Objektsprachlich freie Variablen sind diejenigen Variablen der Lösungsmenge, deren Attribut eine metasprachlich freie Variable ist. Daher müssen wir dieses Attribut einfrieren, um eine Instantiierung zu blockieren. Objektsprachlich gebundene Variablen sind diejenigen Variablen der Lösungsmenge, deren Attribut metasprachlich gebunden ist. Hier müssen wir die Metavariable selbst einfrieren, um eine Unifikation mit einer anderen gebundenen Variablen zu verhindern. Nach dem Einfrieren der Lösungsmenge kann eine syntaktische Variable `X{AttrX}` nur noch mit Elementen `Y{AttrY}` aus der Lösungsmenge unifizieren, für die `AttrY` eine Instanz von `AttrX` ist.

```
freeze_solution([]).
freeze_solution([Var{Value}|Vars]):-
  -?->
  var(Value),!,
  freeze(Value),
  freeze_solution(Vars).
freeze_solution([Var{Value}|Vars]):-
  -?->
  nonvar(Value),
  freeze(Var),
  freeze_solution(Vars).

melt_solution([]).
melt_solution([Var{Value}|Vars]):-
  -?->
```

```
  var(Value),!,
  melt(Value),
  melt_solution(Vars).
melt_solution([Var{Value}|Vars]):-
  -?->
  nonvar(Value),
  melt(Var),
  melt_solution(Vars).
```

4 Ein Anwendungsbeispiel

Wir wollen uns nun anhand einer einfachen Grammatik von der Nützlichkeit der oben eingeführten Prolog-Erweiterung überzeugen. Die Grammatik ist als DCG formuliert, wobei wir davon ausgehen, dass ein dem Compiler vorgeschalteter Makroexpander Klauseln der Gestalt

```
cat0(Tree0) --> cat1(Tree1),...,catN(TreeN).
```

in die entsprechenden Differenzlistenklauseln übersetzt:

```
cat0(Tree0,L0,LN) :-
  cat1(Tree1,L0,L1),
  ...,
  catN(TreeN,LNminus1,LN).
```

Die Grammatik weicht in zwei Punkten von der unvermeidlichen *john loves mary*-Analyse ab: Die Knoten der N-Kopflinie enthalten ein zusätzliches Argument für einen sogenannten Index. Im hier betrachteten einfachsten Fall handelt sich dabei lediglich um eine Genus-Markierung. Außerdem erlauben wir nur Reflexiva, die in ihrer Rektionskategorie c-kommandiert werden. Diese letztere Einschränkung haben wir mit Hilfe des Prädikats `c_commands/2` formuliert, das folgende Bedeutung hat: `?c_commands(Com,Dom)` gelingt, falls es eine Rektionskategorie `xp(Com,Node)` in der aktuellen Lösungsmenge gibt, sodass `dominates(Node,Dom)` gelingt. `dominates/2` formalisiert die übliche Dominanzrelation auf Bäumen.

```
:- set_flag(all_dynamic,on).
```

```
s(s(NP,VP)) -->
  np(NP),
  vp(VP).

np(np(Index,Det,N=n(Index,Lex))) -->
  det(Det),
  n(N).
np(np(Index,empty,PN=pn(Index,Lex))) -->
  pn(PN).
np(np(Index,empty,Pro=pro(Index,Lex))) -->
  pro(Pro).

pn(pn(fem,mary)) --> [mary].
pn(pn(masc,john)) --> [john].

pro(Pro=pro(fem,herself)) --> [herself],
  { ?c_commands(np(fem,Det,N),Pro) }.

pro(Pro=pro(masc,himself)) --> [himself],
  { ?c_commands(np(masc,Det,N),Pro) }.

n(n(fem,woman)) --> [woman].
n(n(masc,man)) --> [man].

det(det(every)) --> [every].
det(det(a)) --> [a].

vp(vp(V,NP)) -->
  v(V),
  np(NP).

v(v(loves)) --> [loves].
v(v(hates)) --> [hates].

:- set_flag(all_dynamic,off).
```

Die Grammatik ist unspektakulär: Sie enthält keine komplexen NPs, keine mehrstelligen oder satzeinbettenden Verben, keine Passivierung und keine Bewegung. Natürlich müssen alle diese Phänomene in einer Bindungstheorie berücksichtigt werden. Hier geht es aber zunächst nur darum, eine überschaubare Anwendung für Testprädikate zu präsentieren. Nachdem wir unseren Metainterpreter mit `loop/1` gestartet haben, können wir die Grammatik testen:

```
[meta_interpreter]: loop.
?- s(Tree,[mary,loves,herself],[]).
s(s(np(fem, empty,
       pn(fem, mary) ),
    vp(v(loves),
       np(fem, empty,
          pro(fem,
              herself) ) ) ),
  [mary, loves, herself], [] )
?- s(Tree,[mary,loves,himself],[]).
?- s(Tree,[every,man,loves,himself],[]).
s(s(np(masc,
       det(every),
       n(masc, man)),
    vp(v(loves),
       np(masc, empty,
          pro(masc, himself) ) ) ),
  [every, man, loves, himself], [])
?- s(Tree,[every,man,loves,herself],[]).
?- s(Tree,[every,man,loves,mary],[]).
s(s(np(masc,
       det(every),
       n(masc, man)),
    vp(v(loves),
       np(fem, empty,
          pn(fem, mary) ) ) ),
  [every, man, loves, mary], [])
?- bye.
yes.
[meta_interpreter]:
```

Betrachten wir nun die Definition von `c_commands/2`. Da unsere Grammatik keine komplexen NPs kennt, ist die Rektionskategorie immer `s(NP,VP)`. Und weil die Verben nicht satzeinbettend sind, brauchen wir auch nicht nach der kleinsten solchen Kategorie zu suchen.

```
:- set_flag(all_dynamic,on).

c_commands(NP,Comp):-
  node(s(NP,VP)),
  dominates(VP,Comp).
```

```
node(s(Subj,Pred)).
node(np(Index,Det,N)).
node(det(Lex)).
node(n(Index,Lex)).
node(pn(Index,Lex)).
node(pro(Index,Lex)).
node(vp(V,Comp)).
node(v(Lex)).

dominates(Node,Node).
dominates(Node,Dom):-
  daughter(Node,Dtr),
  dominates(Dtr,Dom).

daughter(s(X,Y),X).
daughter(s(X,Y),Y).
daughter(np(A,X,Y),X).
daughter(np(A,X,Y),Y).
daughter(vp(X,Y),X).
daughter(vp(X,Y),Y).

:- set_flag(all_dynamic,off).
```

Letztendlich ist eine solche Formulierung von `c_commands/2` aber unbefriedigend. Im nächsten Abschnitt möchte ich zeigen, dass der hier vorgestellte Ansatz auch für eine tiefergehende Analyse geeignet ist.

4.1 Eine kleine Bindungstheorie

Chomskys GB-Theorie besagt, dass Reflexiva in ihrer Rektionskategorie gebunden sein müssen. Diese Einschränkung gilt für die D-Struktur eines Satzes. Wir wollen aber im folgenden noch keine Bewegungen betrachten, sondern erweitern unsere Grammatik lediglich soweit, dass sie folgende Analysen geben kann:

1. [Mary$_i$ loves herself$_i$]
2. [Mary loves himself] *
3. [Mary$_i$ loves every picture of herself$_i$]
4. Mary$_i$ loves [john's picture of herself$_i$] *

5. [John$_i$ talks to Mary about himself$_i$]
6. [John talks to Mary$_i$ about herself$_i$]
7. John knows that [Mary$_i$ loves herself$_i$]
8. John$_i$ knows that [Mary loves himself$_i$] *
9. [John$_i$ knows that every picture of himself$_i$ sucks]
10. John$_i$ knows that [mary's picture of himself$_i$] sucks *

In (1)-(10) sind die Rektionskategorien jeweils durch eckige Klammern angegeben. Wir definieren die Rektionskategorie einer Anapher x als die kleinste gesättigte Kategorie, die ein bindungsfähiges Element enthält, das x c-kommandiert. Dabei sei X ist die kleinste Kategorie mit einer Eigenschaft P gdw sie keine Kategorie dominiert, auf die ebenfalls P zutrifft. Bindungsfähige Elemente sind Kategorien `Phr`, für die `index(Phr,Index)` gelingt, also Nominal- und Präpositionalphrasen. Gesättigte Kategorien sind Phrasen `Phr`, für die `subcat(Phr,[])` gelingt.

Um diese Definition zu verstehen, müssen wir die Strukturen kennen, die von unserer erweiterten Grammatik generiert werden. Sie sind durch folgende Typgleichungen gegeben:

```
Phr=phr(Cat,Dtrs)
Cat=cat(Head,Subcat)
Head=verb|noun(Index,Case)|prep(Index,Prep)
Index=masc|fem|neut
Case=subj|obj|poss
Prep=of|to|about
Subcat=[]|[Cat|Subcat]
Dtrs=empty|dtrs(Phr,Phr)|dtrs(Phr,Phr,Phr)
```

Die Grammatik erzeugt Terme vom Typ `Phr`, für die wir die folgenden Test- und Projektionsfunktionen definieren:

```
:- set_flag(all_dynamic,on).

np(phr(cat(noun(Index,Case),[]),Dtrs)).
pp(phr(cat(prep(Index,Prep),[]),Dtrs)).
vp(phr(cat(verb,[Subj]),Dtrs)).
s(phr(cat(verb,[]),Dtrs)).
```

```
daughter(phr(Cat,dtrs(X,Y)),X).
daughter(phr(Cat,dtrs(X,Y)),Y).
daughter(phr(Cat,dtrs(X,Y,Z)),X).
daughter(phr(Cat,dtrs(X,Y,Z)),Y).
daughter(phr(Cat,dtrs(X,Y,Z)),Z).

cat(phr(Cat,Dtrs),Cat).
head(phr(cat(Head,Subcat),Dtrs),Head).
subcat(phr(cat(Head,Subcat),Dtrs),Subcat).
index(phr(cat(noun(Index,Case),Subcat),Dtrs),Index).
index(phr(cat(prep(Index,Prep),Subcat),Dtrs),Index).
case(phr(cat(noun(Index,Case),Subcat),Dtrs),Case).
prep(phr(cat(prep(Index,Prep),Subcat),Dtrs),Prep).
```

Die modifizierte `pro`-Regel sieht nun so aus:

```
pro(Pro=phr(cat(noun(ProIndex=fem,obj),[]),empty)) --> [herself],
  { ?poss_binder(Ante,Pro),
    index(Ante,AnteIndex),
    AnteIndex=ProIndex }.
```

`?poss_binder(Ante,Pro)` sucht in der aktuellen syntaktischen Struktur nach einer bindungsfähigen Phrase `Ante`, die `Pro` c-kommandiert und die näher an `Pro` liegt, als jede andere Phrase mit dieser Eigenschaft. D.h. so würden wir es gerne formulieren, aber es gibt dabei ein Problem mit den mehrstelligen Verben:

(3) [Mary$_i$ loves every picture of herself$_i$]

(4) Mary$_i$ loves [john's picture of herself$_i$] *

(5) [John$_i$ talks to Mary about himself$_i$]

(6) [John talks to Mary$_i$ about herself$_i$]

Wie (3) und (4) zeigen, zählt ein bindungsfähiges, c-kommandierendes Subjekt als nähergelegener Binder, ein bindungsfähiges, c-kommandierendes Objekt jedoch nicht: In (5) und (6) sind sowohl Subjekt wie Objekt mögliche Binder. Wir müssen die Definition von `poss_binder/2` also modifizieren, indem wir zwischen Subjekt-c-Kommando und Objekt-c-Kommando unterscheiden. Ein möglicher Binder für `Pro` ist eine bindungsfähige Phrase `Ante`, die `Pro` c-kommandiert und die näher an `Pro` liegt, als jedes andere bindungsfähige Subjekt, das `Pro` c-kommandiert:

```
poss_binder(Ante,Phr):-
  c_commands(Ante,Phr),
  not((subj_commands(Com,Phr), c_commands(Ante,Com))).

c_commands(Com,Phr):-
  subj_commands(Com,Phr).
c_commands(Com,Phr):-
  obj_commands(Com,Phr).

subj_commands(Subj,Phr):-
  s(phr(Cat,dtrs(Subj,Pred))),
  dominates(Pred,Phr).
subj_commands(Det,Phr):-
  np(phr(Cat,dtrs(Det,N))),
  np(Det),
  dominates(N,Phr).
obj_commands(Obj1,Phr):-
  vp(phr(Cat,dtrs(V,Obj1,Obj2))),
  dominates(Obj2,Phr).

dominates(Phrase,Phrase).
dominates(Phrase,Dom):-
  daughter(Phrase,Dtr),
  dominates(Dtr,Dom).
```

Um die Satzstrukturen in (1) - (10) behandeln zu können, benötigt unsere Grammatik die Möglichkeit zur Subkategorisierung. Wir verwenden zwei Prinzipien aus der HPSG: das Kopfmerkmalsprinzip (HFP) und das Subkategorisierungsprinzip (SCP). Nach dem HFP erbt die Mutterphrase die Kopfmerkmale der Kopftochter. Da Köpfe vor- oder nachgestellt werden können, haben syntaktische Regeln die allgemeine Form:

```
mother(phr(cat(Head,_),dtrs(Phr0,...,PhrN,HeadPhr))) -->
  dtr0(Phr0),
  ...,
  dtrN(PhrN),
  head_dtr(HeadPhr=phr(cat(Head,_),_)).

mother(phr(cat(Head,_),dtrs(HeadPhr,Phr0,...,PhrN))) -->
  head_dtr(HeadPhr=phr(cat(Head,_),_)),
```

```
  dtr0(Phr0),
  ...,
  dtrN(PhrN).
```

Das Subkategorisierungsprinzip fordert, dass die syntaktischen Kategorien der Komplementtöchter durch die Kopftochter bestimmt werden. Dabei sind vorgestellte und nachgestellte Komplemente zu unterscheiden. Im Gegensatz zur GB und im Einklang mit der HPSG nehmen wir an, dass Subjekte regiert werden:

```
mother(phr(cat(_,[Subj]),dtrs(HeadPhr,Phr0,...,PhrN))) -->
  head_dtr(HeadPhr=phr(cat(_,[Subj,Cat0,...,CatN]),_)),
  comp(Phr0=phr(Cat0,_)),
  ...,
  comp(PhrN=phr(CatN,_)).

mother(phr(cat(_,[]),dtrs(Phr,HeadPhr))) -->
  subj(Phr=phr(Cat,_)),
  head_dtr(HeadPhr=phr(cat(_,[Cat]),_)).
```

Wir betrachten nur einfache Haupt- und Komplementsätze, denn alle anderen Satzformen erhält man nur durch Bewegung. Unverzweigende *NP*s sind Namen und Pronomina. Verzweigende *NP*s bestehen aus einem Determinator und einer $\bar{N}$, d.h. einem Nomen mit abgesättigten Komplementen. Komplemente sind immer obligatorisch. Wir behandeln Eigennamen (John, Mary), prädikative (man, woman) und relationale Nomen (picture). Um Linksrekursion zu vermeiden, lassen wir keine vollen *NP*s als Determinatoren zu, sondern lediglich Eigennamen im Genitiv.

Linksrekursion stellt gegenwärtig ein Problem für uns dar, denn der bisher entwickelte Ansatz ist von der eingesetzten Parsing-Strategie nicht unabhängig: Wenn wir die Position einer Anapher im Parsingbaum schon im Moment ihrer Entdeckung bestimmen wollen, muss dieser Baum top-down generiert werden. Bei einer Bottom-Up-Analyse wird die einbettende Phrase erst fertig konstruiert, *nachdem* die Anapher gefunden worden ist. Wir kommen auf dieses Problem im Abschnitt über Bewegung zurück.

```
s(phr(cat(Head,[]),dtrs(NP,VP))) -->
  np(NP=phr(NPCat,NPDtrs)),
```

```
  vp(VP=phr(cat(Head,[NPCat]),VPDtrs)).

cp(Phrase) -->
  s(Phrase).
cp(Phrase) -->
  [that],
  s(Phrase).

np(phr(cat(Head,[]),dtrs(Det,N))) -->
  det(Det=phr(DetCat,_)),
  n1(N=phr(cat(Head,[DetCat]),_)).
np(Phrase) -->
  pn(Phrase).
np(Phrase) -->
  pro(Phrase).

det(phr(cat(det,[]),empty)) --> [every].
det(phr(cat(det,[]),empty)) --> [a].
det(Phrase) -->
  { case(Phrase,poss) },
  pn(Phrase).

n1(phr(Cat,empty)) -->
  n(phr(Cat,empty)).
n1(phr(cat(Head,[Det]),dtrs(N,PP))) -->
  n(N=phr(cat(Head,[Det,PPCat]),NDtrs)),
  comp(PP=phr(PPCat,PPDtrs)).

pn(phr(cat(noun(fem,Case),[]),empty)) --> [mary].
pn(phr(cat(noun(fem,poss),[]),empty)) --> [mary,s].
pn(phr(cat(noun(masc,Case),[]),empty)) --> [john].
pn(phr(cat(noun(masc,poss),[]),empty)) --> [john,s].

n(phr(cat(noun(fem,Case),[Det]),empty)) --> [woman].
n(phr(cat(noun(masc,Case),[Det]),empty)) --> [man].
n(phr(cat(noun(neut,Case),
          [Det,cat(prep(Gen,of),[])]),empty)) --> [picture].
```

Präpositionen regieren ihr Komplement und übernehmen dabei dessen Index, den sie über das HFP an die Mutter weitergeben. Es ist notwendig, Präpositionalphrasen zu indizieren, weil sie in ihrer Funktion als Präpositionalobjekte als mögliche Binder auftreten können.

```
pp(phr(cat(Head,[]),dtrs(P,NP))) -->
  p(P=phr(cat(Head,[NPCat]),PDtrs)),
  np(NP=phr(NPCat,NPDtrs)).

p(phr(cat(prep(Gen,of),[cat(noun(Gen,obj),[])]),empty)) -->
  [of].
p(phr(cat(prep(Gen,to),[cat(noun(Gen,obj),[])]),empty)) -->
  [to].
p(phr(cat(prep(Gen,about),[cat(noun(Gen,obj),[])]),empty)) -->
  [about].
```

Und hier kommt die *PRO*-Regel: `?poss_binder/2` sucht einen möglichen Binder für die soeben entdeckte *PRO*-Phrase. Dann wird versucht, dessen Index mit dem Index dieser *PRO*-Phrase zu unifizieren. Man beachte: Es handelt sich um Knotenunifikation, nicht um Termunifikation. D.h. nach erfolgreicher Unifikation sind die Indexknoten der Binderphrase und der PRO-Phrase identisch und nicht nur typgleich. Anders ausgedrückt: Nicht nur der Typ (`fem` oder `masc`) der Indizes wird unifiziert, sondern auch die Indizes selbst. Denn unsere Variablen rangieren ja über den Knoten der Lösungsmenge, und das sind nicht Terme, sondern Äquivalenzklassen unifizierter Variablen.

Im Englischen gibt es kein grammatisches, sondern nur ein natürliches Geschlecht. Pronomina müssen mit ihren Vorgängern in Genus, Numerus und Person übereinstimmen. Da wir weder Personalpronomina noch Plural in unserer kleinen Demo-Grammatik berücksichtigt haben, überprüft die Indexunifikation lediglich die Genuskongruenz zwischen Binder und Anapher.

```
pro(Pro=phr(cat(noun(ProIndex=fem,obj),[]),empty)) --> [herself],
  { ?poss_binder(Ante,Pro),
    index(Ante,AnteIndex),
    AnteIndex=ProIndex }.

pro(Pro=phr(cat(noun(ProIndex=masc,obj),[]),empty)) --> [himself],
  { ?poss_binder(Ante,Pro),
    index(Ante,AnteIndex),
    AnteIndex=ProIndex }.
```

Wir haben ein-, zwei- und dreistellige Verben (to suck, to love, to talk) mit Nominal- und Präpositionalkomplementen in der Grammatik, sowie satzeinbettende Verben (to know). Aus Gründen der Übersichtlichkeit verzichten wir auf

die Variation der Verben nach Person, Numerus, Tempus, Modus, Genus verbi und auf infinite Verbformen.

```
vp(phr(Cat,empty)) -->
  v(phr(Cat,empty)).
vp(phr(cat(Head,[SubjCat]),dtrs(V,Obj))) -->
  v(V=phr(cat(Head,[SubjCat,ObjCat]),VDtrs)),
  comp(Obj=phr(ObjCat,ObjDtrs)).
vp(phr(cat(Head,[SubjCat]),dtrs(V,Obj1,Obj2))) -->
  v(V=phr(cat(Head,[SubjCat,Obj1Cat,Obj2Cat]),VDtrs)),
  comp(Obj1=phr(Obj1Cat,Obj1Dtrs)),
  comp(Obj2=phr(Obj2Cat,Obj2Dtrs)).

v(phr(cat(verb,[cat(noun(_,subj),[])]),empty)) -->
  [sucks].
v(phr(cat(verb,[cat(noun(_,subj),[]),
                cat(noun(_,obj),[])]),
      empty)) -->
  [loves].
v(phr(cat(verb,[cat(noun(_,subj),[]),cat(verb,[])]),empty)) -->
  [knows].
v(phr(cat(verb,
          [cat(noun(_,subj),[]),
           cat(prep(_,to),[]),
           cat(prep(_,about),[]) ] ),
      empty) ) -->
  [talks].

comp(Phrase) -->
  np(Phrase).
comp(Phrase) -->
  pp(Phrase).
comp(Phrase) -->
  cp(Phrase).

:- set_flag(all_dynamic,off).
```

5 Anaphernresolution in der HPSG

Wir wollen im folgenden ein nicht-triviales Grammatikfragment des Englischen im Rahmen der HPSG (Head-Driven Phrase Structure Grammar) entwickeln. Dabei stützen wir uns, wo es um die Behandlung von Kongruenz, Rektion, Bewegung, Bindung und Kontrolle geht, auf das HPSG-Buch [32], stellen aber einen eigenständigen Ansatz zur Anaphernbindung vor, der ohne eigene Datenstruktur zur Repräsentation des Diskurses auskommt und sich nahtlos in die Constraint-Solving-Philosophie der HPSG einfügt.

5.1 Die Prinzipien der HPSG

Die HPSG ist nicht nur ein Grammatikformalismus (wie z.B. PATR oder DCG), sondern eine Grammatiktheorie. Eine Grammatiktheorie beschränkt die möglichen Formen syntaktischer Regeln durch sogenannte Prinzipien. Das Ziel einer Grammatiktheorie ist die Erarbeitung dieser Prinzipien, nicht die Beschreibung einer einzelnen Sprache. Natürlich versucht man die Prinzipien so zu wählen, dass die Regeln jeder einzelsprachlichen Grammatik alle Prinzipien der Theorie erfüllen.

Der Wunsch nach einer Aufstellung universeller Prinzipien kann sprachwissenschaftlichem oder technischem Interesse entspringen. Sprachwissenschaftlich ist das Interesse, wenn man damit Chomskys These der angeborenen Universalgrammatik stützen will, die besagt, dass Menschen nur solche Sprachen erlernen können, deren syntaktische Regeln Prinzipien gehorchen, die im menschlichen Gehirn fest verdrahtet sind. Wenn ein Kind beim Erlernen seiner Muttersprache eine Silbenfolge hört und hypothetisch strukturiert, kann es nach Chomsky nur eine stark eingeschränkte Klasse von möglichen Strukturen in Betracht ziehen. Diese Klasse von Strukturen mathematisch zu charakterisieren, ist das Ziel der von ihm gegründeten Theoretischen Linguistik.

Man kann universelle Prinzipien aber auch aus ganz profanen Gründen interessant finden, z.B. als Computerlinguist. Wer mit der Aufgabe konfrontiert ist, in einem theorieunabhängigen Formalismus wie DCG eine umfangreiche Grammatik zu programmieren, wird für jede Regelmäßigkeit, die er in seinen Phrasenstrukturregeln finden kann, dankbar sein. Regelmäßigkeiten erleichtern nicht nur die Fehlersuche, sondern auch die Formulierung von Regeln. Je formaler die Prinzipien vorliegen, desto mehr Arbeit kann dem Autoren bei der Generierung der Regeln vom Computer abgenommen werden. Im idealen Fall sollte der Autor lediglich die Prinzipien angeben, denen seine Grammatik gehorcht, und ein Programm könnte die dazugehörigen Phrasenstrukturregeln generieren.

Eine syntaktische Regel beschreibt die Struktur einer Phrase, d.h. sie legt fest, aus welchen Teilphrasen sie zusammengesetzt sein darf. Das wird üblicherweise

so notiert:

```
Phr0 --> Phr1 ... PhrN
```

d.h. `Phr0`, die Mutterphrase, ist aus den Tochterphrasen `Phr1` bis `PhrN` zusammengesetzt, und zwar in dieser Reihenfolge. Was damit noch nicht festgelegt ist, sind die Kategorien, Funktionen und Bedeutungen der beteiligten Phrasen. Es ist die Aufgabe der syntaktischen Prinzipien, die möglichen Kombinationen festzulegen. Ein solches Prinzip könnte z.B. fordern, dass die Kategorie der Mutterphrase und die Kategorie der ersten Tochterphrase gleich sein müssen. Das ist offenbar eine Gleichheitsbedingung. Eine naheliegende Notation ist daher `cat(Phr0)=cat(Phr1)` und die Regel sieht nun so aus:

```
Phr0 --> Phr1 ... PhrN
    { cat(Phr0)=cat(Phr1) }
```

Das ist natürlich nur eine informelle Notation. In einer Prolog-DCG sähe die Regel so aus:

```
phr(Phr0) -->
  { cat(Phr0,Cat),
    cat(Phr1,Cat) },
  phr(Phr1),
  ...,
  phr(PhrN).
```

Nun stellt sich die Frage, wie `cat/2` definiert werden soll. Aber das ist natürlich davon abhängig, welche Struktur die Terme `Phr0`, `Phr1` haben. Hier kommen wir zu einer theorieabhängigen Eigenschaft einer DC-Grammatik: Die Struktur eines Phrasenterms legt fest, welche Eigenschaften oder Attribute eine Phrase in der gegebenen Theorie hat. Üblicherweise fordert man von einer Phrase mindestens die folgenden Attribute:

- Hauptkategorie (*N*, *V*, *Adj*, *Adv*, *P*, *Conj*, *Det*, *Pro*).
- Unterkategorie (Kasus, Numerus, Genus, Person, Tempus, Modus, Aspekt, Finitheit usw.)
- Valenz (= Subkategorisierungsrahmen)
- Bedeutung (aufgefasst als logischer Ausdruck)
- Lücken

Wir kodieren diese Informationen in Terme vom Typ `Phr`, der durch das folgende Typgleichungssystem festgelegt wird:

```
Phr     = (SynSem,Quants)|(SynSem,Quants,Form)
SynSem  = (Local,NonLoc)
Local   = (Cat,Content)
Cat     = (Head,Subcat)
Head    = noun(Case,RefType)|det(DetType)|verb|prep(PForm,RefType)
Case    = subj|obj
RefType = DetType|ref|wh
DetType = def|indef|quant|poss
PForm   = of|to|with|from
Subcat  = []|[SynSem|Subcat]
Content = (Index,Form)|Form
NonLoc  = (Rel,Slash)
Rel     = []|[Index]
Slash   = []|[Local|Slash]
Quants  = []|[Quant|Quants]
Quant   = every(Index,Form)|exists(Index,Form)|the(Index,Form)
Form    = Atom|(Form and Form)|Quant:Form
```

Für Terme vom Typ `SynSem` gibt es folgende Projektionsfunktionen:

```
rel((_,(Rel,_)),Rel).                       % SynSem --> Rel
slash((_,(_,Slash)),Slash).                 % SynSem --> Slash
gaps((_,Gaps),Gaps).                        % SynSem --> Gaps
content(((_,Content),_),Content).           % SynSem --> Content
index(((_,(Index,_)),_),Index).             % SynSem --> Index
restr(((_,(_,Res)),_),Res).                 % SynSem --> Form
cat(((Cat,_),_),Cat).                       % SynSem --> Cat
head((((Head,_),_),_),Head).                % SynSem --> Head
subcat((((_,Subcat),_),_),Subcat).          % SynSem --> Subcat
det_type((((det(Type),[]),_),_),Type).      % SynSem --> DetType
```

Eine Phrase besteht aus ihren `SynSem`-Eigenschaften und einem Quantoren-Stack (`Quants`). Nominalphrasen haben ein zusätzliches Attribut, das wir später genauer erläutern werden. (Es enthält die atomare Formel, die den Index der NP als Argument enthält. Die Grammatik kann aber so verändert werden, dass das Mitführen dieser Formel überflüssig wird.) Die Quantoren werden zwar, wie in [HPSG] vorgeschlagen, nicht direkt an ihrem Entstehungsort appliziert, dennoch sieht unsere Grammatik eine feste Quantorenfolge entsprechend der Reihenfolge der Komplemente vor.

`SynSem` enthält die syntakto-semantischen Attribute einer Phrase: Kategorie, Valenz, Bedeutung und Lücken. Diese Informationen werden zu einem eigenen `SynSem`-Wert zusammengefasst, weil valenztragende Element in der HPSG für alle vier Attribute subkategorisieren. Das ist für die ersten beiden Attribute nicht weiter erstaunlich. In jeder Grammatiktheorie würde man erwarten, dass z.B. Verben für XPs (also abgesättigte Phrasen) subkategorisieren, deren Kategorie, Kasus (bzw. Präposition) sie darüberhinaus bestimmen. Die Behandlung von Kontroll- und Hebungsverben in der HPSG erfordert aber zusätzlich, dass das Matrixverb Zugriff auf die logischen Argumente des eingebetteten Verbs hat:

```
verb_entry(believe,(((Cat,Content),([],[])),[])):-
  Cat=(verb,[Subj,Obj1,Obj2]),
  cat(Subj,(noun(subj,_),[])),      % Subj:NP[nom]
  cat(Obj1,(noun(obj,_),[])),       % Obj1:NP[acc]
  cat(Obj2,(verb(inf),[Obj1])),     % Obj2:VP[inf]
  index(Subj,Believer),
  content(Obj2,Prop),
  Content=believe(Believer,Prop).
```

`believe` nimmt als sein logisches Subjekt `Believer` den Index seines syntaktischen Subjekts `Subj`. Das Subjekt des eingebetteten Verbs wird mit dem direkten Objekt `Obj1` des Matrixverbs `believe` unifiziert. Dadurch erhält das eingebettete Verb als logisches Subjekt den Index des direkten Objekts von `believe`. Das ist nur möglich, wenn die Bedeutungsinformation, ausgedrückt als logische Formel, in die Werte von `Subj`, `Obj1` und `Obj2` hineingesteckt wird. D.h. die Elemente, die auf der Subkategorisierungsliste `[Subj,Obj1,Obj2]` stehen, müssen auch Bedeutungsinformation enthalten.

Wir haben jetzt gesehen, warum valenztragende Elemente in der HPSG nicht nur Kategorie und Valenz ihrer Komplemente bestimmen, sondern auch Zugriff auf den Inhalt (*engl.* content), d.h. die logische Form ihrer Komplemente haben müssen. Aber warum enthält `SynSem` auch noch Lückeninformation? Das hat mit der Behandlung von Relativsätzen in der HPSG zu tun. Ein Relativsatz wird analysiert als:

$$[_{NP}[_{NP} \text{ the man}_i] \; [_{RP} \; [_{NP[wh]} \text{ who}_i] \; rel \; [_{S/NP[wh]} \text{ I saw } e_i \text{ yesterday}]]]$$

rel ist der phonetisch leere Kopf des Relativsatzes, der für ein Relativpronomen und einen Satz mit NP-Lücke subkategorisiert. Der Lexikoneintrag sieht so aus:

```
rel_entry([],(((Cat,Content),(Rel,[])),[])):-
  Cat=(rel,[RPro,GapS]),
```

```
  RPro=(Gap,(Rel,[])),
  GapS=(((verb,[]),Content),([],[Gap])),
  cat(Gap,(noun(_,wh),[])).

rpro_entry(who,
          ((((noun(_,wh),[]),(Index,true)),([Index],[])),[])).
```

Wir berücksichtigen in unserer Grammatik zwar Relativsätze, behandeln sie aber nicht als Kopf-Komplement-Strukturen, sondern als Kopf-Füller-Strukturen:

$$[_{\mathrm{NP}}[_{\mathrm{NP}} \text{ the man}_i] \; [_{\mathrm{S}} \; [_{\mathrm{NP}[wh]} \text{ who}_i] \; [_{\mathrm{S/NP}[wh]} \text{ I saw } e_i \text{ yesterday}]]]$$

Da wir außerdem keine satzeinbettenden Verben betrachten, entfällt für uns die Notwendigkeit, `SynSem`-Werte auf die Subkategorisierungsliste eines valenztragenden Elements zu setzen. Es wäre ausreichend, Verben, Nomina, Adjektiva und Präpositionen lediglich für `Category`-Werte subkategorisieren zu lassen, wie es z.B. in der GB üblich ist. Aus Gründen der Aufwärtskompatibilität haben wir uns aber entschieden, die Typspezifikation aus [HPSG] zu übernehmen. Immerhin stellt sie einen wichtigen Beitrag der HPSG zur Grammatiktheorie dar.

Die folgenden Prinzipien der HPSG werden von unserer Grammatik berücksichtigt:

- Das Immediate Dominance Principle (IDP)
- Das $\bar{X}$-Prinzip
- Das Head Feature Principle (HFP)
- Das Subcategorization Principle (SCP)
- Das Nonlocal Feature Principle (NFP)
- Das Trace Principle (TP)
- Das Singleton Rel Constraint (SRC)
- Das Weak Coordination Principle (WCP)
- Das Semantics Principle (SP)
- Die Bindungstheorie

Gehen wir sie der Reihe nach durch. Jede Phrase verteilt syntaktische Funktionen an ihre unmittelbaren Teilphrasen. Dabei sind nicht alle denkbaren Verteilungen erlaubt. Die erlaubten Verteilungen nennt man **Schemata**. Wir haben in unserer Grammatik die folgenden sechs Schemata:

- Head-Subject
- Head-Complement
- Head-Adjunct
- Head-Specifier
- Head-Filler
- Coordination

Das IDP fordert nun, dass in jeder Phrasenstruktur – und damit in jeder Phrasenstrukturregel – die syntaktischen Funktionen der unmittelbaren Teilphrasen durch eines dieser Schemata bestimmt werden. Daher können unsere DCG-Regeln nur diese sieben Grundformen annehmen:

```
phr(Mother) -->
 { Head-Subj-Constraints },
  subj(Subj),
  head(Head).
phr(Mother) -->
  { Head-Comp-Constraints },
  head(Head),
  comp(Comp).
phr(Mother) -->
  { Head-Comp-Constraints },
  head(Head),
  comp(Comp1),
  comp(Comp2).
phr(Mother) -->
  { Head-Adjunct-Constraints },
  adjunct(Adjunct),
  head(Head).
phr(Mother) -->
  { Head-Adjunct-Constraints },
  head(Head),
  adjunct(Adjunct).
phr(Mother) -->
```

```
  { Head-Determiner-Constraints },
  det(Determiner),
  head(Head).
phr(Mother) -->
  { Head-Filler-Constraints },
  filler(Filler),
  head(Head).
phr(Mother) -->
  { Coordinate-Structure-Constraints }
  conjunct(Conj1),
  conjunction,
  conjunct(Conj2).

head(Head) -->
  { Head-Category-Constraints },
  phr(Head).
subj(Subj) -->
  { Subj-Category-Constraints },
  phr(Subj).
comp(Comp) -->
  { Comp-Category-Constraints },
  phr(Comp).
det(Det) -->
  { Determiner-Category-Constraints },
  phr(Det).
adjunct(Adjunct) -->
  { Adjunct-Category-Constraints },
  phr(Adjunct).
filler(Filler) -->
  { Filler-Category-Constraints },
  phr(Filler).
conjunct(Conj) -->
  { Conj-Category-Constraints },
  phr(Conj).
```

Wie man sieht, zerfallen die Schemata in zwei Klassen: Die Kopfstrukturen (engl. *headed structures*) und die Koordination. Nicht alle Prinzipien sind auf alle Schemata anzuwenden. Das HFP z.B. gilt nur für Kopfstrukturen, das WCP für die Koordination und das SCP für Kopf-Komplementstrukturen. D.h. für jedes Regelschema sind unterschiedliche Constraints anzusetzen. Darüberhinaus kann nicht jede syntaktische Kategorie jede syntaktische Funktion erfüllen. Also müssen für die einzelnen syntaktischen Funktionen entsprechende Kategoriebeschränkungen

formuliert werden.

Natürlich ist `phr/1` ein zu allgemeiner Schlüssel für eine effektive Formulierung der DCG. Wir wählen als Schlüssel für die Regelköpfe die Kategorie der betrachteten Phrase nach dem $\bar{X}$-Schema, also `np`, `n2`, `n1`, `n` usw. Betrachten wir als Beispiel die Regel für transitive Verben wie sie konkret in der Grammatik steht:

```
vp(((((Head,[Subj]),VPCont),([],Slash)),[])) -->
  { slash(Obj,Slash),
    constraint apply(Obj,Quants,VCont,VPCont)
  },
  v(((((Head,[Subj,Obj]),VCont),([],[])),[])),
  comp((Obj,Quants,VCont)).
```

Die Regel ist eine Instanz des Kopf-Komplement-Schemas: `v/1` bestimmt den Kopf der Phrase, `comp/1` das Komplement. Wie man sieht, wird die Kategorie der Köpfe explizit formuliert, sodass das $\bar{X}$-Schema direkt an der Regelform abzulesen ist. Das Kopfmerkmalsprinzip (HFP) besagt, dass die Kopfmerkmale der Phrase (`Head`) mit den Kopfmerkmalen des Phrasenkopfes identisch sind. Dieses Prinzip ist hier erfüllt. Das Subkategorisierungsprinzip (SCP) besagt, dass die Subkategorisierungsliste der Phrase durch Subtraktion der `SynSem`-Werte der Komplemente von der Subkategorisierungsliste des Phrasenkopfes entsteht. Das ist hier der Fall: `[Subj,Obj]` minus `[Obj]` ergibt `[Subj]`.

Wir überspringen vorläufig die Prinzipien, die bestimmen, wie sich die Bedeutung der Phrase (`VPCont`) aus der Bedeutung der Teilphrasen ergibt, und betrachten die Prinzipien für nichtlokale Merkmale (NFP). Es gibt zwei nichtlokale Merkmale: Relativsatzindizes und Lücken. Das Singleton-Rel-Constraint gilt für jede Regel und besagt, dass die `Rel`-Liste nicht mehr als einen Index enthalten darf. Im betrachteten Fall ist sie leer und das Prinzip daher erfüllt.

Interessanter ist das NFP, dass die Behandlung von Lücken betrifft. Wenn eine Lücke entstanden ist, wird sie solange weitergegeben, bis sie gefüllt werden kann. Da wir in unserer Grammatik nur Komplementbewegung betrachten, entstehen Lücken nur an Komplementpositionen. Eine charakteristische Eigenschaft des Englischen ist, dass nur NPs aus Komplementpositionen heraus bewegt werden können. Eine Lücke entsteht also, wenn eine NP nicht realisiert wird:

```
np((SynSem,[],_)) --> [],
  { empty(SynSem) }.

empty((Local,(_,[Local]))):-
  Local = ((noun(_,_),[]),_).
```

Der SynSem-Wert einer nicht realisierten NP ist dadurch gekennzeichnet, dass sein Local-Wert als einziges Element auf seiner Slash-Liste steht. Das NFP besagt nun, dass die Slash-Liste einer Phrase die Konkatenation der Slash-Listen der Teilphrasen ist, minus dem Local-Wert der Füllertochter, falls vorhanden. Anders ausgedrückt: In einer Kopf-Füller-Struktur muss der Local-Wert der Füllertochter auf der Slash-Liste der Kopftochter vorhanden sein. Die Slash-Liste der Mutter ergibt sich durch Subtraktion dieses Local-Werts von der Slash-Liste der Kopftochter. In der VP-Regel ist dieses Prinzip erfüllt. slash/2 liefert die Slash-Liste des Komplements. Da die Slash-Liste des lexikalischen Kopfs natürlich leer ist, kann die Mutter die Slash-Liste des Komplements direkt übernehmen.

In unserer Grammatik ist der Relativsatz das einzige Beispiel für eine Kopf-Füller-Struktur:

```
rel((((Cat,Cont),([Index],[])),[])) -->
  top(((Gap,([Index],[])),_,Cont)),
  s((((Cat,Cont),([],[Gap])),[])).
```

Die Slash-Liste der Mutter ergibt sich aus der Slash-Liste der Kopftochter durch Subtraktion des Local-Wertes Gap.

Das NFP gilt für alle nichtlokalen Merkmale, also auch für die Rel-Liste. Rel-Indizes entstehen im Lexikon:

```
rpro_entry(who,( ( ((noun(subj,wh),[]),Content),
                   ([X],[]) ),[] ) ):-
  Content = (X{agr(3,_,Gen)},true),
  constraint member(Gen,[masc,fem]).
```

Der Relativsatz-Index X ist ein Metaterm, d.h. eine freie Prolog-Variable, deren Bindungseigenschaften eingeschränkt sind. Ebenso wie Lücken werden Relativindizes an die Mutter weitergegeben, bis sie von der Rel-Liste entfernt werden. Die Entfernungsbedingung ist aber nicht das Auftauchen einer Kopf-Füller-Struktur, sondern die Verwendung des Relativsatzes in attributiver Funktion:

```
postnom_attr((((Cat,(Index,SCont)),([],[])),Quants)) -->
  rel((((Cat,SCont),([Index],[])),Quants)).
```

Das NFP für Relativindizes lautet also: Die Rel-Liste der Mutter ist die Konkatenation der Rel-Listen der Töchter. Beim attributiven Gebrauch eines Relativsatzes wird der Relativindex von der Rel-Liste genommen und mit dem Inhalt des Relativsatzes (SCont) beschränkt ((Index,SCont)).

Im Grunde genommen haben wir für Relativindizes und Lücken zwei Prinzipien angesetzt, da die Entfernungsbedingungen ganz unterschiedlich sind. In [76] wird ein hoher technischer Aufwand betrieben, um die Entfernungsbedingungen einheitlich fassen zu können. Wir weichen an dieser Stelle erheblich von der Standardformulierung ab. Insbesondere ist es in der HPSG nicht erlaubt, die Merkmale einer Phrase in Abhängigkeit von ihrem Gebrauch zu modifizieren. Der Grund dafür ist, dass wir Attribute semantisch gleich behandeln wollen, unabhängig von ihrer syntaktischen Kategorie. Betrachten wir dazu die `n2`-Regel:

```
n2((((N1Cat,(Index,(N1Cont and AttrCont))),(Rel,[])),Quants)) -->
  { N1Cat=(noun(_,Type),_),
    Type ~= poss
  },
  n1((((N1Cat,(Index,N1Cont)),(Rel,[])),[])),
  postnom_attr((((_,(Index,AttrCont)),_),Quants)).
```

Damit wollen wir Sytax und Semantik der folgenden Phrasen erfassen:

1. $[_{N^2}$ man $[_{PP}$ with a car$]]$
2. $[_{N^2}$ man $[_{S}$ who owns a car$]]$

Offenbar ist semantisch dasselbe Prinzip am Werk: Die Bedeutung der N^2-Phrase ist eine Konjunktion der Bedeutungen der Kopf- und Adjunkttochter. Nun haben aber PP- und S[*rel*]-Phrasen unterschiedliche `SynSem`-Werte. Um sie einheitlich behandeln zu können, modifizieren wir sie abhängig vom Gebrauch:

```
postnom_attr((((Cat,(Index,SCont)),([],[])),Quants)) -->
  rel((((Cat,SCont),([Index],[])),Quants)).
postnom_attr(Phrase) -->
  pp(Phrase).
```

Da wir Semantik, Bindungstheorie und das $\bar{X}$-Prinzip im Moment noch zurückstellen wollen, bleiben noch zwei Prinzipien zu besprechen: Das Spurenprinzip und das Koordinationsprinzip. Eine Spur entsteht durch eine nicht realisierte NP. Das Spurenprinzip besagt, dass Spuren von einem lexikalischen Kopf regiert werden müssen. Eine sorgfältige Inspektion der Grammatik zeigt aber, dass NPs nur an Subjekt- oder Objektpositionen generiert werden können. Der triviale Grund dafür ist: Wir lassen keine attributiven, determinierenden oder adverbialen NPs zu. Also wird das Spurenprinzip von unserer Grammatik erfüllt.

Das Koordinationsprinzip nimmt bei uns eine sehr einfache Form an. Die Kopfmerkmale einer koordinierten Phrase müssen mit den Kopfmerkmalen der beiden Konjunkte identisch sein. In unserer Grammatik taucht nur eine einzige koordinierende Regel auf: die Satzkonjunktion.

```
s1(Phrase) --> s(Phrase).
s1((((Head,[]),(LeftCont and RightCont)),([],Slash)),[])) -->
  s((((Head,[]),LeftCont),([],Slash)),[])),
  [and],
  s1((((Head,[]),RightCont),([],Slash)),[])).
s1((((Head,[]),(LeftCont or RightCont)),([],Slash)),[])) -->
  s((((Head,[]),LeftCont),([],Slash)),[])),
  [or],
  s1((((Head,[]),RightCont),([],Slash)),[])).
```

Aus Gründen, die später einsichtig werden, müssen wir von einer Top-Down-Parsingstrategie ausgehen. Um Linksrekursion zu vermeiden, haben wir die Konjunktionsregel entsprechend umformuliert. `s1` steht also nicht für $\bar{S}$, sondern ist lediglich ein weiterer Schlüssel für die `s`-Regel.

5.2 Semantik

Der Begriff 'Inhalt' (engl. *content*) hat in der Situationssemantik und der HPSG eine besondere Bedeutung und unterscheidet sich von 'Interpretation' und 'Bedeutung'. Für uns ist der Inhalt eines Zeichens ein Term vom Typ `Content`. Der Diskursinhalt ist also der `Content`-Term, der dem Diskurs von unserer Grammatik zugeordnet wird. Dabei ist ein Diskurs eine Konjunktion von Sätzen. Zur Erinnerung wiederholen wir hier noch einmal die Typdefinition `Content`:

```
Content  = (Index,Form)|Form
Form     = Atom|(Form and Form)|(Form or Form)|Quant:Form
Quant    = every(Index,Form)|exists(Index,Form)|the(Index,Form)
Quants   = []|[Quant|Quants]
```

5.2.1 Der Inhalt von Nomina und Adjektiven

Die Indizes der HPSG entsprechen den beschränkten Parametern der Situationssemantik. Erinnern wir uns: Ein Parameter teilt mit Variablen die Eigenschaft, dass er an Individuen gebunden werden kann. Und er teilt mit Individuen die Eigenschaft, dass er Wert einer Variablen sein kann. Ein Parameter ist also ein Zwitterwesen zwischen Syntax und Semantik, ähnlich den Diskursreferenten in der DRT. Die Bindungseigenschaften eines Parameters können beschränkt werden: Ein beschränkter Parameter x ist ein Paar (x, σ), wobei σ ein Infon oder eine Proposition ist, die genau x als freien Parameter enthält.

Variable, deren Bindungseigenschaften beschränkt sind: Das erinnert an die Metavariablen in ECL^{i}PS^e. Und tatsächlich werden wir Metavariablen verwenden,

um Indizes zu modellieren. Betrachten wir dazu die Lexikoneinträge der Wortarten, deren Inhalt Indizes sind, nämlich:

```
noun_entry(man,((((noun(_,Type),[Det]),Content),([],[])),[])):-
  Content = (X{agr(3,sing,masc)},man(X)),
  cat(Det,(det(Type),[])).

noun_entry(owner,( ( ((noun(_,Type),[Det,Comp]),Content),
                     ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,_)},own(X,Y)),
  cat(Det,(det(Type),[])),
  subcategorize(Comp,pp(of),Y).

name_entry(Name,( ( ((noun(_,ref),[]),Content),
                    ([],[]) ),[the(X,Res)] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  constraint member(Name,[peter,paul,john]).

rpro_entry(who,( ( ((noun(subj,wh),[]),Content),
                   ([X],[]) ),[] ) ):-
  Content = (X{agr(3,_,Gen)},true),
  constraint member(Gen,[masc,fem]).

ppro_entry(he,( ( ((noun(subj,ref),[]),Content),
                  ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  resolve(X,New).
ppro_entry(himself,( ( ((noun(obj,ref),[]),Content),
                       ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  o_bind(X,New).
```

Appellativa, Eigennamen, Relativ-, Personal- und Reflexivpronomina, also alle Elemente der Kategorie N, haben beschränkte Parameter zum Inhalt. Der Inhalt von `man` z.B. ist `(X{agr(3,sing,masc)},man(X))`, ein Paar aus einer Metavariablen und einem Prologprädikat. Die Bindungseigenschaften von `X` sind also zweifach beschränkt: Erstens kann `X` nur an Indizes gebunden werden, deren Kongruenzmerkmale mit `X` unifizieren, zweitens kann `X` nur an Terme `M` gebunden werden, für die `man(M)` gelingt (eine entsprechende Prolog-Repräsentation der 'Welt' vorausgesetzt).

Namen können, ebenso wie Appellativa im Plural, ohne Artikel verwendet werden. Die quantifizierende Kraft ist dann definit. Das ist der Grund, warum der

Quantor bei Namen, die nicht für einen Determinator subkategorisieren, im Lexikon entsteht:

```
name_entry(Name,( ( ((noun(_,ref),[]),Content),
                    ([],[]) ),[the(X,Res)] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  constraint member(Name,[peter,paul,john]).
```

Wir könnten `Res` und `name(X,Name)` gleich hier im Lexikoneintrag unifizieren. Stattdessen verwenden wir ein Null-Determinator-Constraint in der `n3/1`-Regel (s. Abschnitt 5.2.3). Zur Behandlung von Namen und Determinatoren s. Abschnitt 5.3.1.

Betrachten wir nun die Personal- und Reflexpronomina. Sie unterscheiden sich durch eine zusätzliche Beschränkung von den anderen Nomina: `resolve/2` bei den Personalpronomina und `o_bind/2` bei den Reflexiva. Die Bedeutung dieser Prädikate ist grob gefasst folgende: In `resolve(X,New)` ist `New` die Teilformel des Diskursinhaltes, deren Struktur gerade bestimmt wird, von der aber schon klar ist, dass sie `X` enthalten wird. (In der DRT-Terminologie könnte man sagen: `New` ist die DRS, die gerade bearbeitet wird und `X` als Diskursreferenten enthält.) `resolve(X,New)` gelingt, wenn es einen Index `Y` im Diskursinhalt gibt, der für `X` zugänglich ist, und `X=Y` gelingt. `o_bind/2` gelingt, wenn es einen Index `Y` im Diskursinhalt gibt, der `X` o-kommandiert, und `X=Y` gelingt.

Wir haben die Grammatik so konzipiert, dass die Anaphernresolution 'online' ausgeführt wird, d.h. zu dem Zeitpunkt, wo der Parser auf eine Anapher trifft. Das setzt eine Top-Down-Parsingstrategie voraus, weil sonst der Diskursinhalt zum Zeitpunkt der Resolution nicht als zusammenhängende Formel vorliegt. Beim Bottom-Up-Parsing wird der Diskursinhalt ja erst vollständig zusammengefügt, wenn das letzte Wort des Diskurses erfolgreich analysiert wurde. Natürlich sollte ein Anaphernresolutionsverfahren von der Parsingstrategie unabhängig sein. Das ist aber leicht zu erreichen, wenn der Aufruf der Resolutionsprädikate `resolve/2` und `o_bind/2` zurückgestellt wird. Dazu muss man sie lediglich als Parameterbeschränkungen behandeln, wie es die Theorie ja auch fordert:

```
ppro_entry(he,((((noun(subj,ref),[]),Cont),([],[])),[],New))
:- Cont = (X{agr(3,_,masc)},resolve(X,New)).

ppro_entry(himself,((((noun(obj,ref),[]),Cont),([],[])),[],New))
:- Cont = (X{agr(3,_,masc)},o_bind(X,New)).
```

Dabei ergibt sich allerdings die Frage, bezüglich welchen Programms `resolve/2` bzw. `o_bind` gelingen müssen? Wir waren ja weiter oben davon ausgegangen,

dass für Prädikate wie `man/1`, `owner/2` usw. entsprechende Prologprogramme vorliegen. Die Lösungen von `resolve/2` und `o_bind` sind allerdings vom aktuellen Diskursinhalt abhängig, und der verändert sich mit fortschreitendem Parsing. Betrachten wir dazu im Vorgriff auf Abschnitt 5.4 die Definition von `resolve/2`:

```
resolve(X,New):-
  dis?- antecedent(Ante,Form,New),
  not( (o_commander(Com,X,New), Com == Ante) ),
  Ante=X.
```

`resolve/2` gelingt, wenn `antecedent/3` bezüglich des aktuellen Diskursinhaltes `dis` gelingt. D.h. eigentlich ist `resolve/2` gar kein Prädikat (= Infon), sondern eine Proposition. Ebenso kann man die Beschränkungen für Namen und Appellativa formulieren:

```
noun_entry(man,((((noun(_,Type),[Det]),Content),([],[])),[])):-
  Content = (X{agr(3,sing,masc)},ref?- man(X)),
  cat(Det,(det(Type),[])).

name_entry(Name,( ( ((noun(_,ref),[]),Content),
                    ([],[]) ),[the(X,Res)] ) ):-
  Content = (X{agr(3,sing,masc)},res?- name(X,Name)),
  constraint member(Name,[peter,paul,john]).
```

Das wäre dann so zu lesen, dass der Index `X` an einen Term `M` in der Situation `ref` (formuliert als Prologprogramm) gebunden werden muss, sodass `man(M)` in `ref` gelingt. `ref` steht dann also für die Situation, auf die sich der Sprecher bezieht, die sogenannte beschriebene Situation (*described situation*). `res` könnte dagegen für die Hintergrundsituation (*resource situation*) stehen, die Sprecher und Hörer gemeinsam zugänglich sind, und in der die Individuen eindeutig durch ihren Namen gekennzeichnet sind. Die Grammatik lässt sich leicht in diese Richtung erweitern.

Die Semantik von Adjektiven ist unproblematisch, da wir keine subkategorisierenden Adjektive betrachten:

```
adj_entry(red,((((adj,[]),(X,red(X))),([],[])),[])).
adj_entry(good,((((adj,[]),(X,good(X))),([],[])),[])).
adj_entry(old,((((adj,[]),(X,old(X))),([],[])),[])).
```

Auch Präpositionalphrasen bieten keine grossen Überraschungen. Die Präposition subkategorisiert für den Kasus ihres *NP*-Komplements, übernimmt dessen Typ (`def`, `indef`, `quant`, `poss`, `ref` oder `wh`) und Inhalt, und vererbt sie an die Mutterphrase:

```
pp(((((Head,[]),Content),Gaps),Quants,New)) -->
  { gaps(NP,Gaps)
  },
  p(((((Head,[NP]),Content),([],[])),[])),
  np((NP,Quants,New)).

prep_entry(of,((((prep(of,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).
prep_entry(with,((((prep(with,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).
prep_entry(to,((((prep(to,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).
prep_entry(from,((((prep(from,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).

pcomp_constr(Comp,np(Case,RefType),Cont):-
  cat(Comp,(noun(Case,RefType),[])),
  content(Comp,Cont).
```

5.2.2 Der Inhalt von Verben

Der Inhalt eines n-stelligen Verbs ist ein n-stelliges Prologprädikat samt freier Argumentstellen:

```
verb_entry(sleeps,((((verb(fin,_,_),[Subj]),sleep(X)),
                    ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X).

verb_entry(owns,((((verb(fin,_,_),[Subj,Obj]),own(X,Y)),
                  ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj,np(obj),Y).

verb_entry(sells,((((verb(fin,_,_),[Subj,Obj1,Obj2]),sell(X,Y,Z)),
                   ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj1,np(obj),Y),
  subcategorize(Obj2,pp(to),Z).
```

Terme wie `sleep(X)` oder `own(X,Y)` sind vom Typ `Atom`. Ihre Argumente werden durch `subcategorize/3` mit den Indizes der Verbkomplemente unifiziert. Gleichzeitig werden die syntaktischen Kategorien der Komplemente festgelegt:

```
subcategorize(Comp,np(Case),X):-
  cat(Comp,(noun(Case,_),[])),
  index(Comp,X).
subcategorize(Comp,pp(PForm),X):-
  cat(Comp,(prep(PForm,_),[])),
  index(Comp,X).
```

Zusammen mit dem Subkategorisierungsprinzip sorgen die Lexikoneinträge dafür, dass Verben entsprechend ihrer Valenz mit Komplementen kombinieren.

5.2.3 Der Inhalt von Determinatoren

Der Inhalt eines Determinators ist ein Quantor `Quant(Var,Res)`. `Quant` kann drei Werte annehmen: `every`, `exists` und `the`. Diese Werte nennt man die quantifizierende Kraft (engl. *quantificational force*) des Determinators. Wir haben vier Determinatoren: `a`, `the`, `every` und die Possessivpronomina. Die Lexikoneinträge sehen so aus:

```
det_entry(a,((((det(indef),[]),exists(X,Res)),([],[])),[])).
det_entry(the,((((det(def),[]),the(X,Res)),([],[])),[])).
det_entry(every,((((det(quant),[]),every(X,Res)),([],[])),[])).

det_entry(his,((((det(poss),[]),Content),([],[])),[])):-
  Content=the(Y,(PossRel and Res)),
  PossRel=poss(X{agr(3,_,masc)},Y)
  resolve(X,PossRel).
det_entry(her,((((det(poss),[]),Content),([],[])),[])):-
  Content=the(Y,(PossRel and Res)),
  PossRel=poss(X{agr(3,_,fem)},Y)
  resolve(X,PossRel).
det_entry(its,((((det(poss),[]),Content),([],[])),[])):-
  Content = the(Y,(PossRel and Res)),
  PossRel=poss(X{agr(3,_,neut)},Y)
  resolve(X,PossRel).
```

Die Determinatoren zerfallen in vier syntaktische Unterkategorien: `def` (definit), `indef` (indefinit), `poss` (possessiv) und `quant` (quantifizierend) mit unterschiedlichen syntaktischen Eigenschaften, auf die wir im Kapitel 5.3 zu sprechen kommen werden. Possessivdeterminatoren haben eine zusätzliche Restriktion, die fordert, dass ein im Vordiskursinhalt zu findendes Individuum `X` in einem 'Possessivverhältnis' zu dem eindeutig bestimmten Individuum `Y` steht, das die Eigenschaft

Res(X) hat. Die Eigenschaft Res liefert die N^2, die den Determinator subkategorisiert und die der Determinator spezifiziert. Betrachten wir dazu im Vorgriff auf Abschnitt 5.2.4, wie Determinatoren ihre Restriktoren spezifizieren:

```
n3(((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { constraint (Quant=..[QF,Index,Res]) % Null-Determinator-
  },                                    % Constraint
  n2(((((Head,[]),(Index,Res)),Gaps),[Quant])).

n3(((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { content(Det,Quant),
    constraint specify(Det,Index,Res)
  },
  det((Det,[])),
  n2(((((Head,[Det]),(Index,Res)),Gaps),[])).

specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type = poss),
  Quant=..[QF,Index,(poss(X,Index) and Res)].
specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type ~= poss),
  Quant=..[QF,Index,Res].
```

Zunächst ist bemerkenswert, dass $Det+N^2$ keine maximale Projektion ergibt. Das liegt an unserer Behandlung von Appositionen: N^3 plus Apposition ergibt *NP* (s. Abschnitt 5.3). specify/3 verwendet den N^2-Inhalt zur Restriktion des *Det*-Inhalts. Der so restringierte Quantor wird zur späteren Applikation auf den Quantorenstack gelegt. Subkategorisiert ein Nomen für einen Nulldeterminator, muss es selbst die quantifizierende Kraft bestimmen (Quant). Das Null-Determinator-Constraint sorgt dann dafür, dass der Inhalt der N^2 zum Restriktor des Quantors wird. Bei Possessivpronomina muss berücksichtigt werden, dass ihre Restriktion die Form einer Konjunktion hat. X wird bei der Analyse des Possessivpronomens resolviert.

5.2.4 Das Semantikprinzip

Wir kümmern uns jetzt darum, wie die Inhaltsbestandteile eines Satzes oder Diskurses aus den elemantaren Bestandteilen im Lexikon zusammengesetzt werden. Dabei spielen zwei Attribute der Phrase eine Rolle: Content und Quants. Grundsätzlich ist der Inhalt eines Satzes ein Term vom Typ Form, in dem die Quantoren, die durch die Nominalkomplemente eingeführt worden sind, als Präfix vor dem Nukleus stehen, nämlich dem Inhalt des Hauptverbs. Betrachten wir dazu die *S*- und *VP*-Regel:

```
s(((((Head,[]),SCont),([],SSlash)),[])) -->
  { slash(Subj,SubjSlash),
    constraint append(SubjSlash,VPSlash,SSlash),
    constraint apply(Subj,SubjQuants,VPCont,SCont)
  },
  subj((Subj,SubjQuants,VPCont)),
  vp(((((Head,[Subj]),VPCont),([],VPSlash)),[])).

vp(((((Head,[Subj]),VPCont),([],Slash)),[])) -->
  { slash(Obj,Slash),
    constraint apply(Obj,Quants,VCont,VPCont)
  },
  v(((((Head,[Subj,Obj]),VCont),([],[])),[])),
  comp((Obj,Quants,VCont)).

apply(SynSem,[],Nuc,Nuc).
apply(SynSem,[Quant],Nuc,Quant:Nuc):-
constraint
( restr(SynSem,Res),Res == true ).
apply(SynSem,[Quant],Nuc,Quant:(Res and Nuc)):-
constraint
( restr(SynSem,Res),Res \== true ).
```

Eigentlich würde man erwarten, dass die Quantoren nur in der S-Regel appliziert werden, und zwar in beliebiger Reihenfolge. Schließlich ist es die Idee des Quantorenstacks, eine nichtdeterministische Applikation der Quantoren zu ermöglichen, um die bekannten Skopusambiguitäten in den Griff zu bekommen. Wir haben die Grammatik zwar so konzipiert, dass ein solcher Nichtdeterminismus leicht einzubauen wäre, quantifizieren den Nucleus aber dennoch in der Obliqueheitsreihenfolge der Komplemente. Der Grund dafür ist, dass unser simpler Constraintmechanismus, der ein Goal solange einfriert, bis seine Argumente genügend instantiiert sind, um eine deterministische Reduktion zu ermöglichen, ein echt nichtdeterministisches `apply/4` Prädikat ewig einfrieren würde. Rufen wir aber `apply/4` als Prädikat und nicht als Constraint auf, erzeugt es Wahlpunkte, die zu unendlichem Backtracking führen. Ein Lösungsansatz für dieses Problem wäre, Anaphernresolution und Quantorenapplikation zu verschieben, bis eine erfolgreiche syntaktische Analyse des Diskurses vorliegt. Dann müssten im Generate-and-Test-Verfahren Quantorenpräfixe für die beteiligten Haupt- und Nebensätze generiert und daraufhin getestet werden, ob eine Bindung der Anaphern möglich ist. Wir haben diese Strategie in der vorliegenden Arbeit nicht verfolgt, weil wir uns für die Online-Anaphernresolution interessieren.

Für die Behandlung von Appositionen müssen wir bei `apply/4` eine Fallunterscheidung treffen. Wenn das Komplement eine 'schwere' *NP* ist, d.h. eine *NP*,

die eine Apposition enthält, bildet der Inhalt der Apposition mit dem Inhalt des Hauptverbs den Nucleus. Schwere *NP*s sind daran erkennbar, dass die Beschränkung ihres Index ungleich `true` ist:

```
np(((((Head,[]),(Index,AppCont)),Gaps),[Quant])) -->
  { Head=noun(_,Type),
    constraint member(Type,[ref,def,poss])
  },
  n3(((((Head,[]),(Index,true)),Gaps),[Quant])),
  apposition((((_,(Index,AppCont)),_),[])).
```

Die Behandlung von Appositionen wird in Abschnitt 5.3 genauer besprochen.

Wir müssen semantische Prinzipien für jedes der verwendeten Schemata angeben. Die folgenden Regeln fallen unter das Kopf-Komplement-Schema:

```
n1(((((Head,[Det]),(Index,N1Cont)),Gaps),NQuant)) -->
  { gaps(Comp,Gaps),
    constraint apply(Comp,Quants,NCont,N1Cont)
  },
  n(((((Head,[Det,Comp]),(Index,NCont)),Gaps),NQuant)),
  comp((Comp,Quants,NCont)).

vp(((((Head,[Subj]),VPCont),([],Slash)),[])) -->
  { slash(Obj,Slash),
    constraint apply(Obj,Quants,VCont,VPCont)
  },
  v(((((Head,[Subj,Obj]),VCont),([],[])),[])),
  comp((Obj,Quants,VCont)).

vp(((((Head,[Subj]),VPCont),([],Slashes)),[])) -->
  { slash(Obj1,Slash1),
    slash(Obj2,Slash2),
    constraint append(Slash1,Slash2,Slashes),
    constraint apply(Obj1,Quants1,VCont1,VPCont),
    constraint apply(Obj2,Quants2,VCont,VCont1)
  },
  v(((((Head,[Subj,Obj1,Obj2]),VCont),([],[])),[])),
  comp((Obj1,Quants1,VCont1)),
  comp((Obj2,Quants2,VCont)).

s(((((Head,[]),SCont),([],SSlash)),[])) -->
  {  slash(Subj,SubjSlash),
```

```
    constraint append(SubjSlash,VPSlash,SSlash),
    constraint apply(Subj,SubjQuants,VPCont,SCont)
  },
  subj((Subj,SubjQuants,VPCont)),
  vp(((((Head,[Subj]),VPCont),([],VPSlash)),[])).
```

In der HPSG gilt das Subjekt als Komplement des Verbs, d.h. sein Kasus wird ihm nicht von der Position (wie in der GB) sondern vom Verb zugewiesen. Darüberhinaus sorgt die Kongruenzbeschränkung `agr/3` des Subjektindex in der Subkategorisierungsliste des Verbs für die Kongruenz zwischen Subjekt und Prädikat. Das Semantikprinzip für Kopf-Komplement-Schemata lautet: Der Inhalt der Mutter entsteht durch Applikation der Quantoren der Komplementtöchter auf den Inhalt der Kopftochter, und zwar in der Obliqueheitsreihenfolge, die der Subkategorisierungsrahmen der Komplementtochter vorgibt.

Determinatoren werden ebenfalls von einem Valenzträger subkategorisiert, und zwar von der N^2, deren Inhalt sie als Restriktion verwenden. Formal handelt es sich dabei um Komplemente. Dennoch findet das semantische Kopf-Komplement-Prinzip hier keine Anwendung. Determinatoren spezifizieren ihre Schwester und holen sich dabei den Inhalt der Schwester in den Restriktor ihres Quantors:

```
n3(((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { constraint (Quant=..[QF,Index,Res]) % Null-Determinator-
  },                                    % Constraint
  n2(((((Head,[]),(Index,Res)),Gaps),[Quant])).

n3(((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { content(Det,Quant),
    constraint specify(Det,Index,Res)
  },
  det((Det,[])),
  n2(((((Head,[Det]),(Index,Res)),Gaps),[])).

specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type = poss),
  Quant=..[QF,Index,(poss(X,Index) and Res)].
specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type ~= poss),
  Quant=..[QF,Index,Res].
```

Bei Kopf-Determinator-Konstruktionen wandert der Inhalt des Determinators, ein Quantor, auf den Quantoren-Stack. Wird der Determinator nicht realisiert,

muss der Quantor von der N^2 kommen. In diesem Fall sorgt ein Nulldeterminator-Constraint dafür, dass der Restriktor dieses Quantors mit der Restriktion der Kopftochter unifiziert. Der Index der Mutterphrase bleibt in beiden Fällen unbeschränkt. Der Inhaltsbeitrag einer N^3 ist also ein Quantor und nicht ein beschränkter Parameter.

Koordinationen und Adjunkte haben ähnliche semantische Prinzipien. Bei der Koordination werden die Inhalte der koordinierten Töchter mit dem Inhalt der Konjunktion als Junktor zu einem komplexen Term zusammengesetzt:

```
s1(Phrase) --> s(Phrase).
s1((((Head,[]),(LeftCont and RightCont)),([],Slash)),[])) -->
  s((((Head,[]),LeftCont),([],Slash)),[])),
  [and],
  s1((((Head,[]),RightCont),([],Slash)),[])).
```

Leider lässt ECL^{i}PS^{e}keine variablen Funktoren zu. So muss für jede Konjunktion eine eigene Regel formuliert werden. Natürlich hätte man aber auch ein entsprechendes Constraint verwenden können:

```
s1((((Head,[]),SCont),([],Slash)),[])) -->
  { content(Conj,ConjCont),
    constraint coordinate(ConjCont,LeftCont,RightCont,SCont)
  },
  s((((Head,[]),LeftCont),([],Slash)),[])),
  conjunct((Conj,[])),
  s1((((Head,[]),RightCont),([],Slash)),[])).

conjunct(Entry) --> [Conj],
  { conj_entry(Conj,Entry) }.

conj_entry(and,((((conj,[]),and),([],[])),[])).
conj_entry(.,((((conj,[]),and),([],[])),[])).
conj_entry(or,((((conj,[]),or),([],[])),[])).
...

coordinate(and,LeftCont,RightCont,(LeftCont and RightCont)).
coordinate(.,LeftCont,RightCont,(LeftCont and RightCont)).
coordinate(or,LeftCont,RightCont,(LeftCont or RightCont)).
...
```

Wir verwenden in unserer Grammatik nur zwei Kategorien in attributiver Funktion: $S[+rel]$ und AP. Die entsprechenden Regeln sind aber allgemein genug formuliert, um weitere Kategorien in dieser Funktion aufzunehmen:

```
n2((((N1Cat,(Index,(N1Cont and AttrCont))),(Rel,[])),Quants)) -->
  { N1Cat=(noun(_,Type),_),
    Type ~= poss
  },
  n1((((N1Cat,(Index,N1Cont)),(Rel,[])),[])),
  postnom_attr((((_,(Index,AttrCont)),_),Quants)).

n1((((N1Cat,(Index,(N1Cont and AttrCont))),Gaps),Quant)) -->
  prenom_attr((((_,(Index,AttrCont)),([],[])),[])),
  n1((((N1Cat,(Index,N1Cont)),Gaps),Quant)).

prenom_attr(Phrase) -->
  ap(Phrase).

postnom_attr((((Cat,(Index,SCont)),([],[])),Quants)) -->
  rel((((Cat,SCont),([Index],[])),Quants)).
```

Der Inhalt einer *NP* ist ein beschränkter Index. Die Beschränkung setzt sich aus dem Inhalt des Kopfes und dem Inhalt seiner Attribute und Komplemente zusammen. Da die Beschränkungen jeweils für denselben Index gelten, müssen die Indizes bei jeder Attribuierung unifiziert werden. Die Apposition beschränkt zwar auch den Index ihrer Kopfschwester, aber die Beschränkung darf nicht in die Restriktion des Quantors übernommen werden, da eine Apposition niemals eine restriktive Lesart hat und daher auch nicht mit einer quantifizierenden N^3 kombinieren kann:

1. The old man who owned a red Corvette, a very rich man, died in it.

2. every old man who owned a red Corvette, a very rich man, died in it.*

(1) kann nicht so gelesen werden, dass für das durch die Eigenschaft

```
( man(X) and
  exists(Y, red(Y) and corvette(Y) and owns(X,Y))
  and very-rich(X)
)
```

eindeutig bestimmte Individuum `X` gilt: `died-in(X,Y)`. Vielmehr bedeutet er, dass für das durch die Eigenschaft

```
( man(X) and
  exists(Y, red(Y) and corvette(Y) and owns(X,Y))
)
```

eindeutig bestimmte Individuum `X` gilt: `very-rich(X) and died-in(X,Y)`. Daher wird der Inhalt der Apposition nicht als Restriktion des Quantors verwendet, sondern als Restriktion des `NP`-Index:

```
np(((((Head,[]),(Index,AppCont)),Gaps),[Quant],_)) -->
  { Head=noun(_,Type),
    constraint member(Type,[ref,def,poss])
  },
  n3(((((Head,[]),(Index,true)),Gaps),[Quant])),
  apposition((((_,(Index,AppCont)),_),[])).

apposition((((Cat,(Index,SCont)),([],[])),Quants)) -->
  rel((((Cat,SCont),([Index],[])),Quants)).
```

Taucht eine 'schwere' *NP* als Komplement in einer Kopf-Komplement-Struktur auf, werden sowohl ihr Inhalt als auch ihr Quantor mit `apply/4` auf den Inhalt des Kopfes angewendet. Das erste Argument von `apply/4` enthält den `SynSem`-Wert der *NP*, das zweite Argument die Quantorenliste, das dritte Argument den Inhalt des Kopfes, den sogenannten Nukleus. Die Restriktion des *NP*-Index wandert in den Nucleus und die Quantoren aus der Liste werden vor diesen Nucleus gestellt:

```
apply(SynSem,[],Nuc,Nuc):-
  constraint restr(SynSem,true).
apply(SynSem,[],Nuc,(Res and Nuc)):-
  constraint
  (  restr(SynSem,Res),
    Res~=true
  ).
apply(SynSem,[Quant|Quants],Nuc,Quant:Form):-
  apply(SynSem,Quants,Nuc,Form).
```

Diese Trennung von restriktiver und prädikativer Information wird in Abschnitt 5.3 genauer besprochen.

5.3 Das $\bar{X}$-Prinzip

Wir nehmen für Nominalphrasen vier Ebenen bis zur maximalen Projektion an. Eine vollständige Realisierung dieser Ebenen sieht z.B. so aus:

> $[_{NP}[_{N^3}$The $[_{N^2}[_{N^1}$ famous $[_{N^1}$ $[_{N}$ designer$]_{N}$ $[_{PP}$ of expensive clothes$]_{PP}]_{N^1}]_{N^1}$ $[_{S}$ they found dead$]_{S}]_{N^2}]_{N^3}$, $[_{NP}$a fan of young men as it turned out$]_{NP}]_{NP}$, died of a heart attack.

Der erfahrene Blick erkennt sofort zwei Verstöße gegen die Standard-$\bar{X}$-Theorie: Erstens sollte die Adjunktion einer AP die Ebene erhöhen. Die Regel $N^1 \rightarrow APN^1$ folgt nicht dem $\bar{X}$-Schema. Der Grund dafür ist rein technisch. Um die Iteration von pränominalen *AP*s zu ermöglichen, haben wir die entsprechende Regel so formuliert:

```
n1 --> n.
n1 --> n, comp.
n1 --> ap, n1.
```

Wir könnten Adjektiv-Iterationen auch als Koordinationen behandeln, d.h. eine Regel $AP \rightarrow AP, AP$ annehmen. Dann hätten wir aber gegen ein Linksrekursionsproblem zu kämpfen.

Der zweite Verstoß ist grundlegender: Normalerweise wird angenommen, dass die Determinatoren auf der höchsten Ebene erscheinen. Doch unser $\bar{X}$-Schema für *NP*s sieht so aus:

```
np --> n3.
np --> n3, apposition.

n3 --> n2.
n3 --> det, n2.

n2 --> n1.
n2 --> n1, postnom_attr.

n1 --> n.
n1 --> n, comp.
n1 --> prenom_attr, n1.
```

Mit der Positionierung der Determinatoren innerhalb der $\bar{X}$-Hierarchie stecken wir in einer Zwickmühle. Einerseits müssen wir die Apposition als Adjunkt zu einer N-Projektion auffassen, denn eine Struktur wie:

```
s --> np, apposition, vp.
```

fällt unter kein funktionales Schema und verstößt damit gegen das *ID*-Prinzip: Wenn die *VP* der Kopf dieser Struktur ist, was wir doch annehmen wollen, dann handelt es sich hier offenbar weder um ein Kopf-Komplement-Schema (denn `apposition` ist kein Komplement) noch um ein Kopf-Adjunkt-Schema (denn `np` ist kein Adjunkt). Dasselbe Problem stellt sich, wenn wir von dieser Struktur ausgehen:

```
np --> det, n2, apposition.
```

Auch diese Regel fällt unter kein funktionales Schema. Bleibt als letzte Möglichkeit, um Determinatoren auf der höchsten Ebene zu erhalten:

```
np --> det, n3.
n3 --> n2, apposition.
```

Nun gehen wir aber davon aus, dass die Determinatoren ihre Schwestern spezifizieren. Was ja nichts anderes bedeutet, als das der Inhalt der Schwester zum Restriktor des durch den Determinator eingeführten Quantors gemacht wird. Dann haben wir aber das Problem, dass wir für Appositionen eine restriktive Lesart erhalten, - die es jedoch nicht gibt. Die ausformulierte Regel für die oben betrachtete Möglichkeit würde nämlich so aussehen:

```
np(((((Head,[]),(Index,Res)),Gaps),[Quant])) -->
  { content(Det,Quant),
    constraint specify(Det,Index,Res)
  },
  det((Det,[])),
  n3(((((Head,[Det]),(Index,Res)),Gaps),[])).

specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type = poss),
  Quant=..[QF,Index,(poss(X,Index) and Res)].
specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type ~= poss),
  Quant=..[QF,Index,Res].
```

Wir haben uns deshalb dafür entschieden, Determinatoren nicht auf der höchsten Ebene anzusiedeln, sondern eine spezielle Regel für 'schwere' *NP*s anzunehmen, die etwas aus dem Rahmen fällt:

```
np(((((Head,[]),(Index,AppCont)),Gaps),[Quant],_)) -->
  { Head=noun(_,Type),
    constraint member(Type,[ref,def,poss])
  },
  n3(((((Head,[]),(Index,true)),Gaps),[Quant])),
  apposition((((_,(Index,AppCont)),_),[])).
```

Es handelt sich um eine Kopf-Adjunkt-Regel. Also ergibt sich der Inhalt der Mutterphrase als Konjunktion der Inhalte der Tochterphrasen. Der Informationsbeitrag der Kopftochter ist ihr Quantor. Er wird zur späteren Applikation an die Mutter weitergereicht. Mit diesem Trick erreichen wir eine formale Trennung des restriktiven und prädikativen Informationsgehalts einer 'schweren' *NP*: Der restriktive Inhalt wird über den Quantor weitergegeben, der attributive Inhalt beschränkt den Index der *NP*. Die besondere Wirkungsweise des `apply/4`-Constraints sorgt dafür, dass der Quantor vor die Prädikation, d.h. den Verbinhalt, wandert, und die Restriktion des *NP*-Index wie ein Verbinhalt behandelt wird, nämlich prädizierend.

Noch eine kurze Schlussbetrachtung: Eine vernünftige Alternative für die Verteilung der nominalen Attribute im Englischen scheint folgende Regel zu sein:

```
n2 --> n1.
n2 --> prenom_attr, n1.
n2 --> n1, postnom_attr.
n2 --> prenom_attr, n1, postnom_attr.
```

Oder kürzer:

$$N^2 \rightarrow \{preattribute\} N^1 \{postattribute\}$$

Das funktioniert aber nur, solange wir keine postnominalen *PP*-Attribute berücksichtigen wollen:

1. Do you see this man in black with a gun in his hand who behaves like a perfekt idiot?
2. Do you see this man who behaves like a perfekt idiot in black with a gun in his hand?*

Offensichtlich können also nicht alle nominalen Attribute auf derselben Strukturebene angesiedelt werden. Außerdem scheinen sie im Englischen stark eingeschränkt zu sein, was ihre kategorielle Realisierung betrifft. Eine mögliche Formulierung der Regeln sähe also so aus:

```
n3 --> n2 ; n2, rel.
n2 --> n1 ; ap, n1 ; n1, pp ; ap, n1, pp.
n1 --> n0 ; n0, comp.
```

5.3.1 Namen

Wir haben bisher nur einen Teil der lexikalischen Einträge betrachtet, bei denen es um Namen geht. Der Grund dafür ist, dass die Kombination der Determinatoren mit Namen eine subtile Angelegenheit ist. Wir wollen, dass folgende Analysen möglich sind:

1. Gianni Versace has been stabbed.
2. The Gianni Versace has been stabbed.*
3. The famous Gianni Versace has been stabbed.
4. The Gianni who is a famous designer has been stabbed.
5. Every Gianni wants to be as famous as the famous Gianni is.
6. A Gianni Versace seems to have been stabbed.
7. The fashion people are grieving. Their Gianni has been stabbed.

Natürlich ist in (2) nicht die (phonologisch markierte) Lesart '*The* Gianni Versace' gemeint. (3) - (7) zeigen, dass Namen durchaus für Determinatoren subkategorisieren können. Allerdings gibt es semantische Einschränkungen für den definiten Artikel. Nur wenn der Name mit zusätzlichen restriktiven Attributen versehen ist, wie in (3) und (4), kann er mit einem definiten Artikel kombinieren. Wir müssen also drei Fälle unterscheiden: Kein Determinator (1); Definiter Artikel (3)-(4); Alle anderen Determinatoren (5)-(7). Für jeden dieser Fälle haben wir einen Lexikoneintrag:

```
% ===================
% I Kein Determinator
% ===================
name_entry(Name,( ( ((noun(_,ref),[]),Content),
                    ([],[]) ),[the(X,Res)] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  constraint member(Name,[peter,paul,john]).
% ====================
% II Definiter Artikel
% ====================
name_entry(Name,( ( ((noun(_,def),[Det]),Content),
                    ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  cat(Det,(det(def),[])),
```

```
  content(Det,Quant),
  constraint (Quant=..[QF,X,(Phi and Psi)]),
  constraint member(Name,[peter,paul,john]).
% ===============================
% III Alle anderen Determinatoren
% ===============================
name_entry(Name,( ( ((noun(_,Type),[Det]),Content),
                    ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  cat(Det,(det(Type),[])),
  constraint member(Type,[quant,indef,poss]),
  constraint member(Name,[peter,paul,john]).
```

Die Fälle I und III sind unproblematisch: In Fall I entsteht der definite Quantor direkt durch die Verwendung des Namens ohne Determinator. In Fall III wird der Quantor durch den Determinator `Det` beigesteuert. Der Name selbst subkategorisiert lediglich für den Typ des Quantors: `quant`, `indef` oder `poss`. In Fall II subkategorisiert der Name einerseits für den Typ des Determinators, nämlich `def`, anderseits aber auch für dessen Inhalt: Es wird gefordert, dass die Restriktion des Quantors eine Konjunktion ist. Was nur dann der Fall sein kann, wenn der Name zusätzlich attribuiert worden ist. Betrachten wir dazu das Zusammenspiel der entscheidenden Regeln:

```
n3(((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { constraint (Quant=..[QF,Index,Res])
  },
  n2(((((Head,[]),(Index,Res)),Gaps),[Quant])).

n3(((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { content(Det,Quant),
    constraint specify(Det,Index,Res)
  },
  det((Det,[])),
  n2(((((Head,[Det]),(Index,Res)),Gaps),[])).

n2(Phrase) -->
  n1(Phrase).

n2((((N1Cat,(Index,(N1Cont and AttrCont))),(Rel,[])),Quants)) -->
  { N1Cat=(noun(_,Type),_),
    Type ~= poss
  },
```

```
  n1((((N1Cat,(Index,N1Cont)),(Rel,[])),[])),
  postnom_attr((((_,(Index,AttrCont)),_),Quants)).

n1(Phrase) -->
  n(Phrase).

n1((((N1Cat,(Index,(N1Cont and AttrCont))),Gaps),Quant)) -->
  ap((((_,(Index,AttrCont)),([],[])),[])),
  n1((((N1Cat,(Index,N1Cont)),Gaps),Quant)).
```

Nur wenn `n2/1` oder `n1/1` ein Adjunkt realisiert hat, wird in `n3/1` die Restriktion `Res` zu einer Konjunktion. - Wodurch Subkategorisierung und Spezifikation in `n3/1` gelingen können.

5.4 Anaphernresolution und Bindungstheorie

Wir kommen jetzt zum Kern der Grammatik: Der 'online'-Anaphernresolution. Das Prinzip des Verfahrens wurde schon in Abschnitt 3.5 erläutert. Die beim Top-Down-Parsing erzeugte Lösungsmenge steht dem Interpreter bei jedem Reduktionsschritt zur Verfügung. Sie wird als typisierte Merkmalsstruktur repräsentiert, wobei die Knoten für Mengen äquivalenter Variablen stehen. Aus dieser TMS, die ja nichts anderes ist, als die partielle syntaktische Struktur, die der Parser bisher aufgebaut hat, können sich Testprädikate Knoten mit bestimmten gewünschten Eigenschaften auswählen. In unserem Fall ist dies die Eigenschaft, ein zugänglicher Index zu sein.

5.4.1 Dynamische Semantik

In [35] geben Groenendijk und Stokhof eine sogenannte dynamische Semantik für die Prädikatenlogik 1. Stufe. Sie weicht von der klassischen Semantik unter anderem dadurch ab, dass die Bindungsfähigkeit des Existenzquantors über seinen Skopus hinausreicht. In einer Formel wie $\exists x\phi \wedge \psi[x]$ beispielsweise, in der die Variable x frei in ψ vorkommt, ist bei dieser Semantik die Bedeutung von $\psi[x]$ von der Wahl der gebundenen Variablen in $\exists x\phi$ abhängig: Würde man x im linken Konjunktionsglied in y umbenennen (wobei $x \neq y$), so wäre $\exists y\phi \wedge \psi[x]$ nicht bedeutungsäquivalent zu $\exists x\phi \wedge \psi[x]$. Will man den Umbenennungssatz der klassischen Logik, der besagt, das sich die Bedeutung einer Formel nicht ändert, wenn ihre gebundenen Variablen umbenannt werden, erhalten, muss man den Begriff der Bindung in der dynamischen Semantik neu formulieren.

Die Motivation für eine solche Modifikation des Bindungsbegriffs stammt aus der Montague-Grammatik. Mit dem klassischen Bindungsbegriff bekommt man nämlich Probleme, wenn man die sogenannten Eselssätze behandeln will:

(1a) Every farmer who owns a donkey beats it.

(1b) $\forall x((\text{farmer}(x) \wedge \exists y(\text{donkey}(y) \wedge \text{owns}(x, y))) \rightarrow \text{beats}(x, y))$

(2a) If a farmer owns a donkey, he beats it.

(2b) $\exists x(\text{farmer}(x) \wedge \exists y(\text{donkey} \wedge \text{owns}(x, y))) \rightarrow \text{beats}(x, y)$

In der Montague-Grammatik werden indefinite Artikel als Existenzquantoren interpretiert, und das Kompositionalitätsprinzip fordert die unter (b) angegebenen logischen Formen. Aber in (1b) bindet der Existenzquantor nicht das Vorkommen von y in $\text{beats}(x, y)$, sodass die logische Form nicht die Wahrheitsbedingungen für (1a) wiedergibt. Und in (2b) sind sogar x und y frei in $\text{beats}(x, y)$. Die Wahrheitsbedingungen von (1b) und (2b) lassen sich folgendermaßen paraphrasieren: 'Jeder Bauer, der einen Esel hat, schlägt etwas' (1b), bzw. 'Wenn es einen Bauern gibt, der einen Esel hat, schlägt jemand etwas' (2b). In (2b) ist also auch die allquantifizierende Kraft, die indefinite Artikel im Antezedens eines Konditionalsatzes zu haben scheinen, nicht korrekt erfasst.

Wünschenswert wäre also eine Semantik für die oben angegebenen Formeln, in der die folgenden Äquivalenzen gelten:

- $(\exists x\phi \wedge \psi[x]) \sim \exists x(\phi \wedge \psi)$
- $(\exists x\phi \rightarrow \psi[x]) \sim \forall x(\phi \rightarrow \psi)$

Betrachten wir einen natürlichsprachlichen Satz, für den $\exists x P(x) \wedge Q(x)$ eine logische Form ist: 'Jemand schläft. Er stöhnt.' Die Aussage 'Jemand schläft' gibt uns Information über einen Gegenstand a, nämlich, dass a schläft. Und 'Er stöhnt' verschärft diese Information dahingehend, dass a stöhnt. Wir wissen nicht, für welches $d \in D$ dieses a steht, deshalb kann a selbst nicht ein Element von D sein. Wie sollen wir a formal repräsentieren?

Sei $M = (D, I)$ ein Modell. D ist der Individuenbereich und I eine Interpretationsfunktion. Dann ist D^V die Menge aller Variablenbelegungen. Wenn wir wollen, dass uns eine Formel wie $\exists x P(x)$ Information über einen Gegenstand liefert, der kein Individuum aus D ist, so ist es naheliegend, x als diesen Gegenstand zu wählen. Und ebenso naheliegend ist es, die Information über x durch $s = \{i \in D^V \mid i(x) \in P^I\}$ zu repräsentieren. Diese Information muss an $Q(x)$ weitergereicht werden, denn damit $Q(x)$ die Information, die $\exists x P(x)$ über x liefert, verschärfen kann, braucht es Zugriff auf s. Dann lässt sich der Informationsbeitrag von $Q(x)$ in M so definieren:

$$[\![Q(x)]\!]^M(s) = \{i \in s \mid i(x) \in Q^I\}$$

Wir sehen also, dass wir den Informationsbeitrag einer atomaren Formel als eine Funktion auf Mengen von Variablenbelegungen definieren können. Solche Mengen nennen wir Informationszustände. $S^M = \mathcal{P}(D^V)$ ist die Menge aller Informationszustände. Also definieren wir $[\![\phi]\!]^M : S^M \to S^M$ für alle Formeln ϕ. Die dynamische Bedeutung von prädikatenlogischen Formeln erster Stufe ist eine Funktion auf Informationszuständen. Im folgenden schreiben wir die Applikation einer Funktion auf ihr Argument in Postfix-Notation, also xfg statt $g(f(x))$.

Als nächstes überlegen wir uns, wie wir die Konjunktion behandeln wollen. Wir hatten weiter oben gesagt, dass in einer Aussage wie 'Jemand schläft. Er stöhnt.' oder 'Jemand schläft und er stöhnt.' die Information über den eingeführten Gegenstand an das rechte Konjunktionsglied weitergereicht wird. Also definieren wir die Konjunktion als die Verkettung der Bedeutungsfunktionen der Konjunktionsglieder:

$$s[\![\phi \wedge \psi]\!]^M = s[\![\phi]\!]^M[\![\psi]\!]^M$$

In der natürlichen Sprache führt der indefinite Artikel einen neuen Diskursgegenstand ein, über den der Sprecher im nachfolgenden Text weitere Information geben kann. In der prädikatenlogischen Formalisierung steht eine existenzquantifizierte Variable x für einen neuen Diskursgegenstand. Die Verschärfung der Information über x entspricht einer schrittweisen Einschränkung der erlaubten Belegungen für x. Sei s ein Informationszustand in $M = (D, I)$:

$$s[x] = \{j \mid \exists i \in s \exists d \in D : j = i[x/d]\}$$

$i[x/d]$ ist die x, d-Variante von i, d.h. $i[x/d](y) = d$, falls $x = y$ und $i[x/d](y) = i(y)$ sonst. Im Informationszustand $s[x]$ ist die Information, die s für x hat, gelöscht: Für jedes $d \in D$ findet sich in $s[x]$ ein i, sodass $i(x) = d$. Oder anders ausgedrückt: Im Informationszustand $s[x]$ kann x einen beliebigen Wert annehmen, aber alle anderen Variablen nur dieselben Werte wie in s. Damit können wir die Existenzquantifikation so definieren:

$$s[\![\exists x\phi]\!]^M = s[x][\![\phi]\!]^M$$

Wir kommen nun zur Negation. Dazu betrachten wir den Satz: 'Ein Polizist schoss. Er traf nichts.' Seine logische Form ist $\exists x(P(x) \wedge S(x)) \wedge \neg\exists y T(x, y)$. Angenommen wir interpretieren sie im Informationszustand s. Wir wollen einen Informationszustand s' erreichen, der alle Belegungen i enthält, für die gilt: $i(x) \in (P^I \cap S^I)$, aber es gibt keine y-Variante j von i, sodass $(j(x), j(y)) \in T^I$. Wir erhalten s', indem wir die Menge $\{j \mid (j(x), j(y)) \in T^I\}$ von s abziehen. Um diese

Operation verallgemeinern zu können, bilden wir $\downarrow[\![\phi]\!]^M = \{i \mid \{i\}[\![\phi]\!]^M \neq \emptyset\}$ und definieren:

$$s[\![\neg\phi]\!]^M = s - \downarrow[\![\phi]\!]^M$$

Die Bedeutung der restlichen logischen Zeichen soll klassisch sein, d.h. es soll gelten

- $[\![\forall x\phi]\!]^M = [\![\neg\exists x\neg\phi]\!]^M$
- $[\![\phi \vee \psi]\!]^M = [\![\neg(\neg\phi \wedge \neg\psi)]\!]^M$
- $[\![\phi \rightarrow \psi]\!]^M = [\![\neg(\phi \wedge \neg\psi)]\!]^M$

Damit sind wir ausgerüstet, um eine dynamische Semantik für die Prädikatenlogik erster Stufe zu geben. Die Syntax ist wie üblich gegeben: Atomare Formeln sind Ausdrücke der Gestalt $R(x_1 \dots x_n)$, wobei $x_1 \dots x_n \in V$ Variablen sind. Die Menge der Formeln ist die kleinste Menge F, die alle atomaren Formeln enthält und für die gilt: Wenn $\phi, \psi \in F$ und $x \in V$, dann sind $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \exists x\phi$ und $\forall x\phi$ in F.

Sei $M = (D, I)$ ein Modell und $S^M = \mathcal{P}(D^V)$. Die Bedeutung $[\![\phi]\!]$ einer Formel ϕ ist eine Operation auf S^M mit:

- $s[\![R(x_1 \dots x_n)]\!]^M = \{i \in s \mid (i(x_1) \dots i(x_n)) \in R^I\}$
- $s[\![\phi \wedge \psi]\!]^M = s[\![\phi]\!]^M[\![\psi]\!]^M$
- $s[\![\phi \vee \psi]\!]^M = s \cap (\downarrow[\![\phi]\!]^M \cup \downarrow[\![\psi]\!]^M)$
- $s[\![\neg\phi]\!]^M = s - \downarrow[\![\phi]\!]^M$
- $s[\![\phi \rightarrow \psi]\!]^M = \{i \in s \mid \{i\}[\![\phi]\!]^M \subseteq \downarrow[\![\psi]\!]^M\}$
- $s[\![\forall x\phi]\!]^M = \{i \in s \mid \{i\}[x] \subseteq \downarrow[\![\phi]\!]^M\}$
- $s[\![\exists x\phi]\!]^M = s[x][\![\phi]\!]^M$

Wir wollen uns kurz überzeugen, dass die Bedeutungsfestlegungen für Disjunktion, Implikation und Allquantifikation auch zu den gewünschten Äquivalenzen führen. Der Lesbarkeit halber lassen wir das Modell-Superskript weg:

$$\begin{aligned}
s[\![\phi \vee \psi]\!] &= \{i \in s \mid i \in {\downarrow}[\![\phi]\!] \text{ oder } i \in {\downarrow}[\![\psi]\!]\} \\
&= \{i \in s \mid \forall j : j = i \text{ und } j \notin {\downarrow}[\![\phi]\!] \Rightarrow j \in {\downarrow}[\![\psi]\!]\} \\
&= \{i \in s \mid \forall j : j \in (\{i\} - {\downarrow}[\![\phi]\!]) \Rightarrow j \in {\downarrow}[\![\psi]\!]\} \\
&= \{i \in s \mid \forall j : j \in \{i\}[\![\neg\phi]\!] \Rightarrow j \in {\downarrow}[\![\psi]\!]\} \\
&= \{i \in s \mid i \notin \{i \mid \exists j : j \in \{i\}[\![\neg\phi]\!] \text{ und } j \notin {\downarrow}[\![\psi]\!]\}\} \\
&= s - \{i \mid \{i\}[\![\neg\phi]\!] - {\downarrow}[\![\psi]\!] \neq \emptyset\} \\
&= s - \{i \mid \{i\}[\![\neg\phi]\!][\![\neg\psi]\!] \neq \emptyset\} \\
&= s - {\downarrow}[\![\neg\phi \wedge \neg\psi]\!] \\
&= s[\![\neg(\neg\phi \wedge \neg\psi)]\!]
\end{aligned}$$

$$\begin{aligned}
s[\![\forall x\phi]\!] &= \{i \in s \mid \forall j : j \in \{i\}[x] \Rightarrow j \in {\downarrow}[\![\phi]\!]\} \\
&= \{i \in s \mid i \notin \{i \mid \exists j : j \in \{i\}[x] \text{ und } j \notin {\downarrow}[\![\phi]\!]\}\} \\
&= \{i \in s \mid i \notin \{i \mid \{i\}[x] - {\downarrow}[\![\phi]\!] \neq \emptyset\}\} \\
&= \{i \in s \mid i \notin \{i \mid \{i\}[\![\exists x\neg\phi]\!] \neq \emptyset\}\} \\
&= s - {\downarrow}[\![\exists x\neg\phi]\!] \\
&= s[\![\exists x\neg\phi]\!]
\end{aligned}$$

$$\begin{aligned}
s[\![\phi \to \psi]\!] &= \{i \in s \mid \forall j : j \in \{i\}[\![\phi]\!] \Rightarrow j \in {\downarrow}[\![\psi]\!]\} \\
&= \{i \in s \mid i \notin \{i \mid \exists j : j \in \{i\}[\![\phi]\!] \text{ und } j \notin {\downarrow}[\![\psi]\!]\}\} \\
&= s - \{i \mid \{i\}[\![\phi]\!] - {\downarrow}[\![\psi]\!] \neq \emptyset\} \\
&= s - \{i \mid \{i\}[\![\phi]\!][\![\neg\psi]\!] \neq \emptyset\} \\
&= s - {\downarrow}[\![\phi \wedge \neg\psi]\!] \\
&= s[\![\neg(\phi \wedge \neg\psi)]\!]
\end{aligned}$$

Wir zeigen nun mit Induktion, dass $s[\![\phi]\!] = \bigcup_{i\in s}(\{i\}[\![\phi]\!])$ für alle Formeln ϕ:

$$\begin{aligned}
s[\![R(\bar{x})]\!] &= s \cap {\downarrow}[\![R(\bar{x})]\!] \\
&= (\bigcup_{i\in s}\{i\}) \cap {\downarrow}[\![R(\bar{x})]\!] \\
&= \bigcup_{i\in s}(\{i\} \cap {\downarrow}[\![R(\bar{x})]\!]) \\
&= \bigcup_{i\in s}(\{i\}[\![R(\bar{x})]\!])
\end{aligned}$$

$$
\begin{array}{rcl}
s[\![\phi \wedge \psi]\!] & = & s[\![\phi]\!][\![\psi]\!] \\
 & = & \bigcup\{\{j\}[\![\psi]\!] \mid j \in s[\![\phi]\!]\} \\
 & = & \bigcup\{\{j\}[\![\psi]\!] \mid i \in s \text{ und } j \in \{i\}[\![\phi]\!]\} \\
 & = & \bigcup\{\{i\}[\![\phi]\!][\![\psi]\!] \mid i \in s\} \\
 & = & \bigcup_{i \in s}(\{i\}[\![\phi \wedge \psi]\!]) \\
s[\![\exists x \phi]\!] & = & s[x][\![\phi]\!] \\
 & = & \bigcup\{\{j\}[\![\phi]\!] \mid j \in s[x]\} \\
 & = & \bigcup\{\{j\}[\![\phi]\!] \mid i \in s \text{ und } j \in \{i\}[x]\} \\
 & = & \bigcup\{\{i\}[x][\![\phi]\!] \mid i \in s\} \\
 & = & \bigcup_{i \in s}(\{i\}[\![\exists x \phi]\!]) \\
s[\![\phi \to \psi]\!] & = & \{i \in s \mid \{i\}[\![\phi]\!] \subseteq \downarrow[\![\psi]\!]\} \\
 & = & \bigcup_{i \in s}\{j \in \{i\} \mid \{j\}[\![\phi]\!] \subseteq \downarrow[\![\psi]\!]\} \\
 & = & \bigcup_{i \in s}(\{i\}[\![\phi \to \psi]\!])
\end{array}
$$

Die anderen Fälle funktionieren analog. Damit können wir die folgenden beiden 'Eselsäquivalenzen' beweisen:

$$
\begin{array}{rcl}
s[\![\exists x \phi \wedge \psi]\!] & = & s[x][\![\phi]\!][\![\psi]\!] \\
 & = & s[x]([\![\phi]\!][\![\psi]\!]) \\
 & = & s[\![\exists x(\phi \wedge \psi)]\!] \\
s[\![\exists x \phi \to \psi]\!] & = & \{i \in s \mid \{i\}[\![\exists x \phi]\!] \subseteq \downarrow[\![\psi]\!]\} \\
 & = & \{i \in s \mid \{i\}[x][\![\phi]\!] \subseteq \downarrow[\![\psi]\!]\} \\
 & = & \{i \in s \mid \bigcup\{\{j\}[\![\phi]\!] \mid j \in \{i\}[x]\} \subseteq \downarrow[\![\psi]\!]\} \\
 & = & \{i \in s \mid \forall j \in \{i\}[x] : \{j\}[\![\phi]\!] \subseteq \downarrow[\![\psi]\!]\} \\
 & = & \{i \in s \mid \forall j \in \{i\}[x] : j \in \downarrow[\![\phi \to \psi]\!]\} \\
 & = & \{i \in s \mid \{i\}[x] \subseteq \downarrow[\![\phi \to \psi]\!]\} \\
 & = & s[\![\forall x(\phi \to \psi)]\!]
\end{array}
$$

Wir kommen jetzt zum Umbenennungssatz der Dynamischen Semantik. Um ihn formulieren zu können, benötigen wir eine Notation, die es uns erlaubt, namensgleiche Variablen zu unterscheiden. Wir wählen Superskripte. Es soll gelten: $x^i = x^j$ für alle $i, j \in \omega$, aber $x^i \equiv y^j$ gdw $i = j$ und $x = y$. Variable, die Superskripte tragen, nennen wir indiziert. In der Semantik ändert sich dadurch

nichts: $[\![\phi]\!] = [\![\phi']\!]$ falls ϕ' durch die Indizierung von Variablenvorkommnissen in ϕ entsteht. Wir notieren indizierte Variable mit $\nu_1, \nu_2, \nu_3 \ldots$ bzw. ν, ν', ν'', falls wir das Superskript nicht mitteilen wollen. Hat die Variable ν den Namen x und das Superskript i, so gilt: $\nu \equiv x^i$. Weiterhin gilt: $\nu = \nu'$ gdw es x, i, j gibt, sodass $\nu \equiv x^i$ und $\nu' \equiv x^j$.

In den folgenden syntaktischen Definitionen wollen wir namensgleiche Variablen unterscheiden. Wir setzen daher $\bar{\nu} = (x, i)$ falls $\nu \equiv x^i$. Dann können wir simultan induktiv die Menge $\text{fv}\,\phi$ der freien Variablen in ϕ und die Menge $\text{av}\,\phi$ der zugänglichen Variablen in ϕ definieren:

- $\text{fv}\,R(\nu_1 \ldots \nu_n) = \{\bar{\nu}_1 \ldots \bar{\nu}_n\}$
- $\text{av}\,R(\nu_1 \ldots \nu_n) = \emptyset$
- $\text{fv}\,\neg\phi = \text{fv}\,\phi$
- $\text{av}\,\neg\phi = \emptyset$
- $\text{fv}(\phi \wedge \psi) = \text{fv}\,\phi \cup (\text{fv}\,\psi - \text{av}\,\phi)$
- $\text{av}(\phi \wedge \psi) = \text{av}\,\psi \cup (\text{av}\,\phi - \text{av}\,\psi)$
- $\text{fv}(\phi \vee \psi) = \text{fv}\,\phi \cup \text{fv}\,\psi$
- $\text{av}(\phi \vee \psi) = \emptyset$
- $\text{fv}(\phi \to \psi) = \text{fv}\,\phi \cup (\text{fv}\,\psi - \text{av}\,\phi)$
- $\text{av}(\phi \to \psi) = \emptyset$
- $\text{fv}\,\exists\nu\phi = \text{fv}\,\phi - \{\bar{\nu}\}$
- $\text{av}\,\exists x^i\phi = \text{av}\,\phi \cup \{(x, i)\}$ falls es kein j gibt mit $(x, j) \in \text{av}\,\phi$
  $= \text{av}\,\phi$ sonst
- $\text{fv}\,\forall\nu\phi = \text{fv}\,\phi - \{\bar{\nu}\}$
- $\text{av}\,\forall\nu\phi = \emptyset$

Eine Formel ϕ ist zulässig indiziert gdw $\text{fv}\,\phi = \emptyset$ und außerdem gilt:

- Sind $\exists x^i\psi$ oder $\forall x^i\psi$ Teilformeln von ϕ und ist $(x, j) \in \text{fv}\,\psi$, dann ist $i = j$.
- Kommt $\exists x^i\psi$ als Teilformel von ϕ vor, dann gibt es kein anderes Vorkommen einer Teilformel $\exists x^i\chi$ in ϕ.

Ein paar Beispiele sollen den Begriff klarmachen. Wir schreiben ϕ^* falls ϕ eine nicht zulässig indizierte Formel ist:

- $\exists x^1 P(x^1) \wedge Q(x^1)$
- $\exists x^1 P(x^1) \wedge \exists x^2 Q(x^2) \wedge R(x^2)$
- $\exists x^1 P(x^1) \wedge \exists x^2 Q(x^2) \wedge R(x^1)^*$
- $\exists x^1 P(x^1) \wedge \exists y^2 Q(y^2) \wedge R(x^2)^*$
- $\exists x^1 P(x^1) \wedge \exists x^1 Q(x^1) \wedge R(x^1)^*$
- $\exists x^1 (P(x^1) \wedge Q(x^1))$
- $\exists x^1 (P(x^1) \wedge Q(x^2))^*$

Zulässig indizierte Formeln sind genau diejenigen Formeln, die keine freien Variablen enthalten, und bei denen alle Variablen im Bindungsbereich eines Quantors den Index dieses Quantors tragen. Dabei ist der Bindungsbereich des Existenzquantors weiter als sein Skopus, wie die erste der obigen Formeln zeigt. Der Begriff 'Bindungsbereich' ist insofern gerechtfertigt, als für koindizierte Variable ein Umbenennungssatz gilt: Sei ϕ^i_z die Substitution jedes Vorkommnisses einer Variablen mit dem Index i durch z, und z eine Variable, die in ϕ nicht vorkommt. Dann gilt: $\downarrow [\![\phi]\!] = \downarrow [\![\phi^i_z]\!]$. Zum Beispiel:

- $\downarrow [\![\exists x^1 P(x^1) \wedge \exists x^2 Q(x^2) \wedge R(x^2)]\!] = \downarrow [\![\exists x^1 P(x^1) \wedge \exists z Q(z) \wedge R(z)]\!]$
- $\downarrow [\![\exists x^1 P(x^1) \wedge \exists x^2 Q(x^2) \wedge R(x^1)]\!] \neq \downarrow [\![\exists z P(z) \wedge \exists x^2 Q(x^2) \wedge R(z)]\!]$

5.4.2 Anaphernresolution

In der Dynamischen Semantik ist die Zugänglichkeitsrelation eine Beziehung zwischen einer Variablen `X` und einer Teilformel `Sub` in einer Formel `Form`. `X` ist zugänglich für `Sub` in `Form` gdw es eine Teilformel `Q(X,Res)` von `Form` gibt, sodass `Sub` im Skopus von `Q` liegt. Der Sonderfall, der uns interessiert, ist natürlich der, dass `Sub` selbst eine Variable ist, die durch ein anaphorisches Pronomen eingeführt wurde. `Sub` muss dann mit einem zugänglichen Index unifiziert werden. Betrachten wir zunächst die Prologdefinition des dynamischem Skopus:

```
scope(X,(Phi and Psi),Chi):-
  subform(Chi,Psi),
  accessible(X,Phi).
scope(X,Quant:Phi,Psi):-
```

```
  var_res(Quant,X,Res),
  subform(Psi,Phi).
scope(X,Quant:Phi,Psi):-
  var_res(Quant,X,Res),
  subform(Psi,Res).
scope(X,Quant:Phi,Psi):-
  var_res(Quant,Y,Res),
  X~=Y,
  subform(Psi,Phi),
  accessible(X,Res).

subform(Phi,Phi).
subform(Phi,not(Psi)):-
  subform(Phi,Psi).
subform(Chi,(Phi or Psi)):-
  subform(Chi,Phi).
subform(Chi,(Phi or Psi)):-
  subform(Chi,Psi).
subform(Chi,(Phi and Psi)):-
  subform(Chi,Phi).
subform(Chi,(Phi and Psi)):-
  subform(Chi,Psi).
subform(Psi,Quant:Phi):-
  subform(Psi,Phi).
subform(Psi,Quant:Phi):-
  var_res(Quant,X,Res),
  subform(Psi,Res).

accessible(X,(Phi and Psi)):-
  accessible(X,Phi),!.
accessible(X,(Phi and Psi)):-
  accessible(X,Psi).
accessible(X,Quant:Nuc):-
  Quant=..[Q,X,Res],
  dynamic_quant(Q).
accessible(X,Quant:Nuc):-
  Quant=..[Q,Y,Res],
  dynamic_quant(Q),
  X~=Y,
  accessible(X,Res),!.
accessible(X,Quant:Nuc):-
  Quant=..[Q,Y,Res],
  dynamic_quant(Q),
```

```
  X~=Y,
  accessible(X,Nuc).

var_res(Quant,X,Res):-
  nonvar(Quant),
  Quant=..[Q,X,Res],
  quantifier(Q).

quantifier(every).
quantifier(exists).
quantifier(the).

dynamic_quant(exists).
dynamic_quant(the).
```

Das erste Argument von `scope/3` ist eine Variable `X`; das zweite Argument ist eine Formel der Gestalt `Q(Y,Res):Nuc` oder `(Phi and Psi)`; das dritte Argument ist eine Teilformel von `Res`, `Nuc` oder `Psi`. Wir nennen sie `Chi`. Es sind drei Fälle zu unterscheiden:

1. Klassische Bindung: Ist `X == Y` und `Chi` eine Teilformel von `Res` oder `Nuc`, so liegt `Chi` im 'klassischen' Skopus von `Q`.
2. Dynamische Bindung: Ist `X ~= Y` und `Chi` eine Teilformel von `Nuc`, so muss es einen zugänglichen dynamischen Quantor `D(X,R)` in `Res` geben.
3. Dynamische Bindung: Ist `Chi` eine Teilformel von `Psi`, so muss es einen zugänglichen dynamischen Quantor `D(X,R)` in `Phi` geben.

Ein Quantor `Quant` ist zugänglich in einer Formel `Form` gdw er nicht in einer Teilformel `Sub` von `Form` eingebettet ist, die die Gestalt `not(Phi)` oder `(Phi or Psi)` hat. Obwohl unsere Grammatik keine Negation behandelt, haben wir der Ordnung halber in der Definition von `subform/2` die Negation mitberücksichtigt.

Wie wird das Testprädikat `scope/3` aufgerufen? Betrachten wir dazu den Lexikoneintrag eines anaphorischen Pronomens:

```
ppro_entry(he,( ( ((noun(subj,ref),[]),Content),
                  ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  resolve(X,New).

resolve(X,New):-
  dis?- scope(Ante,Old,New),
  Ante=X.
```

Eigentlich würden wir ein Testprädikat der Form `?scope(Ante,Old,New)` erwarten. Aber aus Effizienzgründen testen wir `scope/3` nicht gegen die ganze Lösungsmenge, sondern nur gegen eine Teilmenge davon, den Diskurs `dis`. Und diese Teilmenge repräsentieren wir auch nicht als TMS, sondern verwenden die Prolog-übliche Termrepräsentation. Zu diesem Zweck müssen wir drei Symbole metainterpretieren: `dis`, '`?-`' und '`:`'. '`?-`' und '`:`' stehen für `test/2` bzw. `update/3` aus Abschnitt 3.5. Ihre Semantik wird jedoch ohne das explizite Mitführen der Lösungsmenge formuliert. Betrachten wir zunächst die Semantik von '`:`'.

```
:- module_interface(interpreter).
:- begin_module(interpreter).

(Sol:(Phi,Psi)):-!,
  Sol:Phi,
  Sol:Psi.

(Sol:(Phi;Psi)):-!,
  Sol:Phi;
  Sol:Psi.

(Sol:not(Goal)):-!,
  not(Sol:Goal).

(Sol:(dis?- Goal)):-!,
  content(Sol,Dis)@grammar,
  Dis?- Goal.

(Sol:constraint(Goal)):-!,
  constraint(Goal).

(Sol:Goal):-
  Goal=..[Rel|_],
  predefined(Rel),!,
  call(Goal,grammar).

(Sol:Goal):-
  clause(Goal,Body)@grammar,
  Sol:Body.
```

Es passiert nichts weiter, als dass ein Prologterm `Sol` bei jedem Reduktionsschritt mitgeführt wird, sodass er, falls er Variablen aus `Goal` enthält, schrittweise instantiiert wird. Das entspricht einem Update der Lösungsmengen-Repräsentati-

on dieses Terms. Damit `Sol` auch etwas Sinnvolles enthält, nämlich den aktuellen Zustand der Satzanalyse, muss diese so gestartet werden:

```
[interpreter] Ph:s(Ph,[every,man,who,owns,a,car,loves,it],[]).
```

Das Prompt `[interpreter]` zeigt an, dass wir uns im Modul `interpreter` befinden. `Goal@Module` bedeutet, dass `Goal` im Namensraum von `Module` ausgeführt wird. Wir verwenden zwei Module: `interpreter` und `grammar`, wobei `grammar` natürlich die zu interpretierende Grammatik enthält. `constraint/1` wird übrigens nicht importiert (was auch denkbar gewesen wäre), sondern direkt im Modul `interpreter` definiert.

Zur Interpretetion von `dis` wird mit `content/2` der aktuelle Inhalt des in der Analyse befindlichen Satzes bestimmt. Es handelt sich dabei in jedem Fall um einen partiell instantiierten Term des Typs `Content`. Dieser Term kann nun mit beliebigen Prologprädikaten getestet werden:

```
(Sol?- not(Goal)):-
  Sol?- Goal,!,fail.
(Sol?- not(Goal)):-!.

(Sol?- (Phi,Psi)):-!,
  Sol?- Phi,
  Sol?- Psi.

(Sol?- (Phi,Psi)):-!,
  Sol?- Phi;
  Sol?- Psi.

(Sol?- Goal):-
  freeze(Sol),
  Goal=..[_|Args],
  domain(Sol,Dom),
  members(Args,Dom),
  call(Goal,grammar),
  melt(Sol).

domain(Term,Dom):-
  subcall(setof(Sub,subterm(Sub,Term),Dom),Delayed).

subterm(X,Y):-
  X = Y.
subterm(X,Y):-
```

```
  nonvar(Y),
  Y=..[_|Args],
  subterm_list(X,Args).

subterm_list(X,[Y|_]):-
  subterm(X,Y).
subterm_list(X,[_|Y]):-
  subterm_list(X,Y).

members([],_).
members([X|Y],Z):-
  member(X,Z),
  members(Y,Z).
```

Der Test erfolgt durch systematische Bindung der freien Variablen in `Goal` an Teilterme von `Sol` und anschließenden Aufruf des instantiierten Goals. Dabei müssen die freien Variablen in `Sol` vorher eingefroren werden, denn wir wollen ja überprüfen, ob der Aufruf von `Goal` mit Werten aus `Sol` gelingt, ohne dass `Sol` dabei instantiiert wird.

`domain/2` bildet den Teilterm-Abschluss von `Sol`. (Aus rein technischen Gründen, die mit der ECL^{i}PS^e-internen Behandlung von Metatermen zu tun haben, muss `setof/3` mit `subcall/2` aufgerufen werden.) Nun werden mit `members/2` alle möglichen Bindungen der Variablen in `Goal` an Elemente von `Dom` ermittelt und getestet.

Zurück zur Grammatik. Betrachten wir noch einmal den Lexikoneintrag von `he`:

```
ppro_entry(he,( ( ((noun(subj,ref),[]),Content),
                  ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  resolve(X,New).

resolve(X,New):-
  dis?- scope(Ante,Old,New),
  Ante=X.
```

Der durch das Pronomen `he` eingeführte Index `X` wird resolviert, indem ein geeigneter Vorgänger `Ante` von `X` gesucht und mit `X` unifiziert wird. Beim Aufruf des Tests ist `New` an eine Teilformel des Diskurses gebunden; und zwar an diejenige, deren Instantiierung gerade bestimmt wird. `Ante` und `Old` sind dagegen frei und müssen geeignet gebunden werden. Betrachten wir dazu die S-Regel:

```
s(((((Head,[]),SCont),([],SSlash)),[])) -->
  { slash(Subj,SubjSlash),
    constraint append(SubjSlash,VPSlash,SSlash),
    constraint apply(Subj,SubjQuants,VPCont,SCont)
  },
  subj((Subj,SubjQuants,VPCont)),
  vp(((((Head,[Subj]),VPCont),([],VPSlash)),[])).
```

Angenommen, diese Regel würde als rechtes Adjunkt einer (`s --> s and s`)-Regel aufgerufen werden, die den Satz `[peter,sleeps,and,he,snores]` analysiert. Dann wäre der Inhalt des Diskurses zum Zeitpunkt des Aufrufs:

```
the(_g1,name(_g1,peter)):sleeps(_g1) and SCont
```

`SCont` ist der Teil des Diskurses, der von der `s/1`-Regel näher bestimmt werden soll. Da der Inhalt des Subjektes auf jeden Fall ein Teilterm von `SCont` sein wird, gibt `SCont` die Position an, von der aus nach möglichen Vorgängern des Subjekts zu suchen ist. Im betrachteten Fall ist das Subjekt ein Pronomen mit dem Index `X`, was zu folgender Aufrufkette führt:

```
CALL: resolve(X,SCont)
CALL: dis?- scope(Ante,Old,SCont)
...
CALL: freeze(the(_g1,name(_g1,peter)):sleeps(_g1) and SCont))
RET : freeze(the(_g1{frozen},name(_g1,peter)):sleeps(_g1)
           and SCont{frozen}))
...
CALL: scope(_g1{frozen},
          the(_g1,name(_g1,peter)):sleeps(_g1) and SCont),
          SCont{frozen})
...
CALL: _g1 = X
```

Nach gelungenem Aufruf der `s/1`-Regel für SCont sieht der Diskurs also so aus:

```
the(_g1,name(_g1,peter)):sleeps(_g1) and snores(_g1)
```

Entscheidend ist, dass der aktuelle Diskurs vor dem Aufruf von `members/2` eingefroren wird. Sonst würde `members/2` nämlich die Variable `SCont`, die als Argument von `scope/3` vorkommt, wie eine freie Variable behandeln und an beliebige Teilterme von `dis` binden. Durch das Einfrieren des Diskurses werden alle Variablen in `scope/3`, die auch im Diskurs vorkommen, zu Konstanten. In Abschnitt

3.5 haben wir diese Variablen Parameter genannt, die ungefrorenen Variablen dagegen syntaktische Variablen.

Nach welchem Prinzip wird nun der `New`-Term vererbt? Er gibt ja an, welcher Teilinhalt des Diskurses noch zu berechnen ist, und spielt damit exakt dieselbe Rolle, wie die unvollständigen DRSen in der DRT. (Eine DRS ist unvollständig, wenn sie noch reduzierbare Bedingungen (engl. *conditions*) enthält - wobei Bedingungen nichts anderes sind als abstrahierte Parsingbäume.)

Zunächst stellen wir fest, dass anaphorische Pronomina nur in Subjekt-, Objekt- und *Det*-Position auftauchen. Im letzteren Fall handelt es sich um Possessivpronomina. Sehen wir uns ihren Lexikoneintrag an:

```
det_entry(his,((((det(poss),[]),the(Y,(PossRel and Res))),
                 ([],[])),[])):-
PossRel=poss(X{agr(3,_,masc)},Y),
resolve(X,PossRel).
```

`PossRel` ist die kleinste Teilformel des Diskurses, die `X` enthält, und wird daher als Position gewählt, von der aus nach Vorgängern zu suchen ist. Die entscheidende Frage lautet: Ist der Diskurs im Moment der `X`-Resolvierung zusammenhängend? Es könnte ja sein, dass die Reduktion des Regelkopfes `det_entry/2` nicht ausreicht, um den Inhalt von `his` in die Diskursformel 'einzuhängen'. Das kann im Prinzip aus zwei Gründen passieren: Das zuständige Constraint (z.B. `apply/4`) ist noch nicht genügend instantiiert, um zu feuern; oder das zuständige Constraint (z.B. `subcategorize/3`) ist noch gar nicht aufgerufen worden. Schematisch (und stark vereinfacht) kann man sich das so vorstellen:

```
CALL: s(SCont)
CALL: np(NPCont) vp(VPCont)
delayed goals:
  apply(NPCont,VPCont,SCont)
CALL: det(DetCont) n(NCont) vp(VPCont)
delayed goals:
  apply(NPCont,VPCont,SCont)
  specify(DetCont,NCont,NPCont)
CALL: det_entry(DetCont) n(NCont) vp(VPCont)
CALL: specify(DetCont,NCont,NPCont) n(NCont) vp(VPCont)
delayed goals:
  apply(NPCont,VPCont,SCont)
CALL: resolve(X,DetCont) n(NCont) vp(VPCont)
FAIL: resolve(X,DetCont) n(NCont) vp(VPCont)
```

Hier wäre der Fall eingetreten, dass NPCont durch den Aufruf von det_entry/2 nicht ausreichend instantiiert wird, um apply/3 feuern zu lassen. Als Folge würde DetCont nicht in SCont eingehängt und der Diskurs wäre zum Zeitpunkt des Aufrufs von resolve/2 unzusammenhängend. Die Frage ist: Kann es einen solchen Fall geben?

Wir müssen überprüfen, ob jedesmal, wenn ein resolve(X,New)-Aufruf stattfindet, die Formel New eine Teilformel des Diskurses ist. Zunächst machen wir uns klar, dass der Diskurs immer zusammenhängend wäre, falls jede Regel folgendem semantischen Prinzip gehorchen würde:

```
mother(MotherCont) -->
{ semantic_principle(Dtr1Cont,...,DtrNCont,MotherCont) }
  dtr1(Dtr1Cont),
  ...,
  dtrN(DtrNCont).
```

wobei semantic_principle/N+1 ein Fakt ist, dessen Auswertung nicht verschoben zu werden braucht. Wir haben semantische Prinzipien für Kopf-Komplement, Kopf-Determinator, Kopf-Adjunkt und Koordinationsstrukturen. Für die letzten zwei davon ist die Bedingung erfüllt:

```
head_adjunct_principle(Head,Adjunct,(Head and Adjunct)).
coordination_principle(Left,Right,(Left and Right)).
coordination_principle(Left,Right,(Left or Right)).
```

Lediglich apply/4 und specify/3 sind 'echte' Constraints, deren Aufruf möglicherweise verschoben werden muss. Kann der Fall eintreten, dass zum Zeitpunkt der Auswertung von resolve/2 noch ein verschobenes apply/4 oder specify/3 vorliegt? Betrachten wir dazu die Feuerbedingungen dieser Constraints:

```
specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type = poss),
  Quant=..[QF,Index,(poss(X,Index) and Res)].
specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type ~= poss),
  Quant=..[QF,Index,Res].
```

specify/3 feuert, sobald der Typ des Determinators vorliegt. Wird ein Possessivpronomen resolviert, hat der Regelkopf des Lexikoneintrags schon den Typ poss festgelegt und specify/3 hat gefeuert. Wird ein Personal- oder Reflexivpronomen resolviert, das ja direkt unter *NP* hängt, kommt specify/3 gar nicht erst zur Anwendung, da eine pronominale *NP* keine Kopf-Determinator-Struktur hat.

Was ist mit apply/4?

```
apply(SynSem,[],Nuc,Nuc).
apply(SynSem,[Quant],Nuc,Quant:Nuc):-
  constraint
    ( restr(SynSem,Res),Res == true ).
apply(SynSem,[Quant],Nuc,Quant:(Res and Nuc)):-
  constraint
    ( restr(SynSem,Res),Res \== true ).
```

`apply/4` taucht nur in Kopf-Komplement-Strukturen auf und appliziert die Quantorenlisten der Komplemente, sobald sie vorliegen. Die Listen können entweder leer sein oder genau einen Quantor enthalten. Ist das Komplement ein Relativ- oder Personalpronomen, wird die leere Quantorenliste und der Restriktor `true` beim Aufruf des Lexikoneintrags instantiiert. Also hat `apply/4` bereits gefeuert, bevor `resolve/2` zum Aufruf kommt. Bei allen anderen Komplementen wird zum Zeitpunkt des *NP*-Aufrufs eine einelementige Quantorenliste instantiiert. Handelt es sich um leichte *NP*s, wird der Restriktor zusätzlich mit true instantiiert, ansonsten bleibt er frei. In beiden Fällen feuert `apply/4` unmittelbar nach dem Aufruf der *NP*. Wir sehen also, dass der Diskurs zum Zeitpunkt eines `resolve/2`-Aufrufs immer zusammenhängend ist.

Nachdem klar ist, wie Possessivpronomina resolviert werden, betrachten wir nun Personalpronomina. Personalpronomina tauchen in Subjekt- und Objektposition auf. Ihr Inhalt ist eine Variable, die durch Subkategorisierung zum Argument eines *VP*-Inhaltes wird. Das Problem ist nur, dass aus ihrem Lexikoneintrag nicht hervorgeht, *welcher VP*-Inhalt das ist:

```
ppro_entry(he,((((noun(subj,ref),[]),Content),([],[])),[],New)):-
  Content=(X{agr(3,_,masc)},true),
  resolve(X,New).
```

Der `Content`-Wert ist keine Teilformel des Diskurses, sondern nur ein Teilterm. Er kann also nicht als Positionsargument von `resolve/2` dienen. Selbst wenn wir in der Definition von `scope/3` die Bedingung `subform/2` durch `subterm/2` ersetzen würden, wäre uns nicht geholfen. Denn `he` kann nur in der Subjektposition eines Satzes vorkommen, aber erst durch die Analyse das nachfolgenden Verbs wird `X` mit der ersten Argumentstelle des Verbinhalts unifiziert, und zwar durch das `subcategorize/3`-Constraint im Lexikoneintrag des Verbs. Zum Zeitpunkt des Aufrufs von `resolve/2` im Lexikoneintrag von `he` ist `X` also auch kein Teilterm des Diskurses.

Dieses Problem ist eine Folge unserer ‘online’-Resolvierungsstrategie. Würden wir die Resolution der Anaphern verschieben, bis eine fertige Satzstruktur generiert worden ist, müssten wir uns nicht mit Fragen des Diskurszusammenhangs

beschäftigen und könnten von der Parsingstrategie abstrahieren. So bleibt uns nichts anderes übrig, als im Lexikoneintrag von Personalpronomina eine zusätzliche Argumentstelle einzuführen, die den Inhalt enthält, der vom Parser gerade berechnet wird. Glücklicherweise ist eine Vererbung des 'Positionsterms' `New` nur bei Kopf-Komplement-Strukturen notwendig und folgt einer sehr einfachen Regel:

```
s(((((...),SCont),(...)),...)) -->
  { ..., constraint apply(Subj,SubjQuants,VPCont,SCont) },
  subj((Subj,SubjQuants,VPCont)),
  vp(((((...),VPCont),(...),...)).

vp(((((...),VPCont),(...),...)) -->
  { ..., constraint apply(Obj,Quants,VCont,VPCont) },
  v(((((...),VCont),(...)),...)),
  comp((Obj,Quants,VCont)).


vp(((((...),VPCont),(...)),...)) -->
  { ...,
    constraint apply(Obj1,Quants1,VCont1,VPCont),
    constraint apply(Obj2,Quants2,VCont,VCont1)
  },
  v(((((...),VCont),(...)),...)),
  comp((Obj1,Quants1,VCont1)),
  comp((Obj2,Quants2,VCont)).

n1(((((...),(Index,N1Cont)),...),...)) -->
  { ..., constraint apply(Comp,Quants,NCont,N1Cont) },
  n(((((...),(Index,NCont)),...),...)),
  comp((Comp,Quants,NCont)).
```

Stark schematisiert kann man das Positionsvererbungsprinzip so darstellen:

```
mother(Subj:(Rest=Head)) -->
  subj(Subj,Rest),
  head(Head).

mother(Comp1:(Rest1=(Comp2:(Rest2=(...CompN:(RestN=Head)))))) -->
  head(Head),
  comp(Comp1,Rest1),
  comp(Comp2,Rest2),
  ...,
  comp(CompN,RestN).
```

Erinnern wir uns: Ein Term der Gestalt `Var=Term` hat dasselbe Unifikationsverhalten wie `Term`, nur dass `Var` nach stattgefundener Unifikation an den Inhalt von `Term` gebunden ist. Wir haben diese Notation im Abschnitt 3.9 eingeführt. Damit lässt sich das Positionsvererbungsprinzip so formulieren: Der Positionsterm eines Komplements ist entweder der Inhalt seiner rechten Schwester oder der Inhalt des Kopfes, falls es selbst die letzte Schwester ist.

5.4.3 Bindungstheorie

Im Rahmen unserer kleinen Grammatik von einer Bindungstheorie zu sprechen, ist ein wenig vermessen. Die interessanten Fälle der Bindungstheorie entstehen durch Bewegung, Passivierung, Satzeinbettung und *NP*-Determinatoren, also durch syntaktische Strukturen, die unsere Grammatik gar nicht beherrscht. Wir implementieren dennoch den Grundgedanken der HPSG-Bindungstheorie in einer vereinfachten Form. In der HPSG wird der strukturelle Begriff des c-Kommandos aus der GB durch den funktionalen Begriff des o-Kommandos ersetzt. Eine Phrase `Phr1` o-kommandiert eine Phrase `Phr2`, falls beide Phrasen von einem lexikalischen Element subkategorisiert werden und der `SynSem`-Wert von `Phr1` dem `SynSem`-Wert von `Phr2` auf der Subkategorisierungsliste dieses Elements vorangeht. Da wir, wie gesagt, keine Passivierung behandeln, und deshalb die Reihenfolge der syntaktischen Argumente eines Valenzträgers der Reihenfolge der logischen Argumente seines Inhalts entspricht, können wir o-Kommando als eine Beziehung zwischen den Argumenten eines Verb- oder Nominalinhalts auffassen. `o_commands/3` nimmt als drittes Argument die Argumentliste `Args` eines Verb- oder Nominalinhalts und liefert alle Variablen `C`, `X` aus `Args`, sodass `C` `X` o-kommandiert:

```
o_commands(A,X,[A|Args]):-
meta_member(X,Args).
o_commands(C,X,[_|Args]):-
  o_commands(C,X,Args).

meta_member(X,[Y|_]):-
  X==Y.
meta_member(X,[Y|R]):-
  X\==Y,
  meta_member(X,R).
```

Ein Index `X` ist o-gebunden, falls es einen Index `Y` gibt, der `X` o-kommandiert und `X==Y` gelingt. Eine o-Bindung wird durch den Aufruf `o_bind(X,New)` erzeugt, wobei `New` der Verb- oder Nominalinhalt ist, der `X` als Argument enthält:

```
o_bind(X,New):-
  o_commander(C,X,New),
  C=X.
```

Eine Variable, die nicht o-gebunden ist, heißt o-frei. Unsere einfache Bindungstheorie lautet:

- Reflexiva sind o-gebunden.
- Personal- und Possessivpronomina sind o-frei.

und wird durch folgende Lexikoneinträge realisiert:

```
ppro_entry(him,( ( ((noun(obj,ref),[]),Content),
                   ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  resolve(X,New).

ppro_entry(himself,( ( ((noun(obj,ref),[]),Content),
                       ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  o_bind(X,New).

resolve(X,New):-
  dis?- scope(Ante,Old,New),
  not( (o_commander(Com,X,New), Com == Ante) ),
  Ante=X.
```

5.5 Beispielanalysen

Nachdem der Interpreter und die Grammatik in ihre entsprechenden Module kompiliert worden sind (nämlich `interpreter` und `grammar`), wechseln wir ins Modul `interpreter` und starten unsere Analysen. Der Output ist aus Lesbarkeitsgründen aufbereitet worden. Nur der Inhalt der Diskursvariablen wird angeben, wobei die systemgenerierten Variablen von Hand durch sprechende Variablen ersetzt wurden:

```
?- Ph:s(Ph,[every,man,who,owns,a,car,loves,it],[]).

Dis = every(Man, man(Man) and exists(Car,car(Car)) :
                                 own(Man,Car)
```

```
           ) :
      love(Man,Car)

?- Ph:s1(Ph,[peter,owns,a,car,.,he,loves,it],[]).

Dis = ( the(Peter, name(Peter,peter)) :
        ( exists(Car, car(Car)) :
          own(Peter,Car)
        )
        and
        love(Peter,Car)
      )

?- Ph:s(Ph,[every,owner,of,a,car,loves,it],[]).

Dis = every(Owner,
            exists(Car, car(Car)) :
            own(Owner,Car)
           ) :
      love(Owner,Car)

?- Ph:s(Ph,[peter,who,owns,a,car,loves,it],[]).

Dis = the(Peter, name(Peter,peter)) :
      ( exists(Car, car(Car)) :
        own(Peter,Car) and love(Peter,Car)
      )

?- Ph:s(Ph,[peter,sells,a,car,to,a,woman,who,loves,it],[]).

Dis = the(Peter, name(Peter,peter)) :
      ( exists(Car, car(Car)) :
        ( exists(Woman, woman(Woman) and love(Woman,Car)) :
          sell(Peter,Car,Woman)
        )
      )

?- Ph:s(Ph,[peter,sells,a,car,to,a,woman,he,loves],[]).

Dis = the(Peter, name(Peter,peter)) :
      ( exists(Car, car(Car)) :
        ( exists(Woman, woman(Woman) and love(Peter,Woman)) :
          sell(Peter,Car,Woman)
```

```
        )
      )

?- Ph:s(Ph,[peter,sells,a,car,to,the,owner,of,it],[]).

Dis = the(Peter, name(Peter,peter)) :
      ( exists(Car, car(Car)) :
        ( the(Owner, own(Owner,Car)) :
          sell(Peter,Car,Owner)
        )
      )

?- Ph:s(Ph,[peter,knows,every,owner,of,a,car,he,loves],[]).

Dis = the(Peter, name(Peter,peter)) :
      ( every(Owner,
              exists(Car, car(Car) and love(Peter,Car)) :
              own(Owner,Car)
             ) :
        knows(Peter,Owner)
      )

?- Ph:s(Ph,[every,owner,of,a,car,he,loves,knows,mary],[]).

Dis = every(Owner,
            exists(Car, car(Car) and love(Owner,Car)) :
            own(Owner,Car)
           ) :
      ( the(Mary, name(Mary,mary)) :
        knows(Owner,Mary)
      )

?- Ph:s(Ph,[every,man,
            who,owns,a,car,he,loves,
            sells,it,to,mary],[]).

Dis = every(Man, man(Man) and
            exists(Car, car(Car) and love(Man,Car)) :
            own(Man,Car)
           ) :
      ( the(Mary, name(Mary,mary)) :
        sell(Man,Car,Mary)
      )
```

```
?- Ph:s(Ph,[every,owner,of,him,sleeps],[]).

no (more) solution.
?- Ph:s(Ph,[every,man,loves,himself],[]).

Dis = every(Man, man(Man)) : love(Man,Man))

?- Ph:s(Ph,[every,man,loves,him],[]).

no (more) solution.
?- Ph:s(Ph,[peter,sells,a,car,to,a,woman,who,loves,itself],[]).

no (more) solution.
?- Ph:s(Ph,[peter,sells,a,car,to,a,woman,who,loves,herself],[]).

Dis = the(Peter, name(Peter,peter)) :
      ( exists(Car, car(Car)) :
        ( exists(Woman, woman(Woman) and love(Woman,Woman)) :
          sell(Peter,Car,Woman)
        )
      )
yes.
?- Ph:s(Ph,[peter,sells,a,car,to,a,woman,who,loves,himself],[]).

no (more) solution.
?-
```

5.6 Der vollständige Programmtext

```
:- module_interface(interpreter).
:- export constraint/1.
:- begin_module(interpreter).
:- import setarg/3 from sepia_kernel.
:- dynamic ch_pts/1.

:- op(500,xfy,(?-)).
:- op(500,xfy,(:)).
:- op(1100,xfy,and).

:- meta_attribute(interpreter,
```

```
                  [unify:unify_frozen/2,print:meta_print/2]).

%============
% Meta + Meta
%============
unify_frozen(_{X},Y):- % Linke Seite ohne Attribut
  -?-> var(X),!.
unify_frozen(_{X},Y):- % Rechte Seite ohne Attribut
  -?-> var(Y).
unify_frozen(_{frozen(X)},frozen(Y)):- % Linke Seite aufgetaut
  -?-> X == no,!.
unify_frozen(_{frozen(X)},frozen(Y)):- % Rechte Seite aufgetaut
  -?-> Y == no,!.
unify_frozen(_{frozen(X)},frozen(Y)):- % Beide Seiten gefroren
  -?-> X == Y.                         % oder aufgetaut
%==============
% Nonvar + Meta
%==============
unify_frozen(X,Y):- % Rechte Seite ohne Attribut
  nonvar(X), var(Y),!.
unify_frozen(X,frozen(Id)):- % Rechte Seite aufgetaut
  nonvar(X), Id == no,!.
unify_frozen(X,frozen(Id)):- % Rechte Seite gefroren
  nonvar(X), var(Id), fail.

freeze(X):-
  term_variables(X,Vars),
  freeze_vars(Vars).

freeze_vars([]).
freeze_vars([X|Vars]):-
  freeze_var(X),
  freeze_vars(Vars).

freeze_var(X):-
  free(X),
  add_attribute(X,frozen(Id)).

freeze_var(X{Attr}):-
  -?-> (var(Attr) -> Attr=frozen(Id); setarg(1,Attr,Id)).

melt(X):-
  term_variables(X,Vars),
```

```
  melt_vars(Vars).

melt_vars([]).
melt_vars([X|Vars]):-
  melt_var(X),
  melt_vars(Vars).

melt_var(X):-
  free(X).
melt_var(X{Attr}):-
  -?-> (var(Attr) -> Attr=frozen(no); setarg(1,Attr,no)).

% ===============
% Update und Test
% ===============

(Sol:(Phi,Psi)):-!,
  Sol:Phi,
  Sol:Psi.

(Sol:(Phi;Psi)):-!,
  Sol:Phi;
  Sol:Psi.

(Sol:not(Goal)):-!,
  not(Sol:Goal).

(Sol:(dis?- Goal)):-!,
  content(Sol,Dis)@grammar,
  Dis?- Goal.

(Sol:constraint(Goal)):-!,
  constraint(Goal).

(Sol:Goal):-
  Goal=..[Rel|_],
  predefined(Rel),!,
  call(Goal,grammar).

(Sol:Goal):-
  clause(Goal,Body)@grammar,
  Sol:Body.
```

```
(Sol?- not(Goal)):-
  Sol?- Goal,!,fail.

(Sol?- not(Goal)):-!.

(Sol?- (Phi,Psi)):-!,
  Sol?- Phi,
  Sol?- Psi.

(Sol?- Goal):-
  freeze(Sol),
  Goal=..[_|Args],
  domain(Sol,Dom),
  members(Args,Dom),
  call(Goal,grammar),
  melt(Sol).

domain(Term,Dom):-
  subcall(setof(Sub,subterm(Sub,Term),Dom),Delayed).

subterm(X,Y):-
  X = Y.
subterm(X,Y):-
  nonvar(Y),
  Y=..[_|Args],
  subterm_list(X,Args).

subterm_list(X,[Y|_]):-
  subterm(X,Y).
subterm_list(X,[_|Y]):-
  subterm_list(X,Y).

members([],_).
members([X|Y],Z):-
  member(X,Z),
  members(Y,Z).

% ===================
% Constraint Handling
% ===================

constraint(true):-!.
constraint((Phi,Psi)):-!,
```

```
  constraint(Phi), constraint(Psi).
constraint((Phi;Psi)):-!,
  constraint(Phi); constraint(Psi).

constraint(Goal):-
  Goal=..[=..,Term,[Rel|Args]],!,
  (  var(Term) ->
    (  atom(Rel) ->
         !,call(Goal,grammar);
         !,var(Rel),delay([Term,Rel],constraint(Goal))
    );
    call(Goal,grammar)
  ).

constraint(Goal):-
  Goal=..[Rel|_],
  predefined(Rel),!,
  call(Goal,grammar).

constraint(Goal):-
  choice_points(Goal,Num,grammar),
  (  Num =< 1 ->
      !,cclause(Goal,SubGoals,grammar),
      !,constraint(SubGoals)
    ;
      term_variables(Goal,Vars),
      delay(Vars,constraint(Goal))
  ).

choice_points(Goal,Num,Module):-
  functor(Goal,Functor,Arity),
  length(List,Arity),
  Head=..[Functor|List],
  term_variables(Goal,Vars),
  findall(Head,
          ( subcall(cclause(Head,_,Module),Delayed),
            Head=Goal,
            call_list(Delayed) ),
          Heads),
  length(Heads,Num),
  assert(ch_pts(Num)),
  fail.
```

```
choice_points(Goal,Num,Module):-
  retract(ch_pts(Num)).

cclause(Head,SubGoals,Module):-
  clause(Head,Body)@Module,
  split(Body,Constraints,SubGoals),
  call(Constraints).

call_list([]).
call_list([Goal|Goals]):-
  call(Goal),
  call_list(Goals).

split(constraint(Cons),constraint(Cons),true):-!.
split((constraint(Cons),Goals),constraint(Cons),Goals):-!.
split(Goals,true,Goals).

% =====================
% Predefined Predicates
% =====================

predefined(true).
predefined(=).
predefined(\=).
predefined(~=).
predefined(==).
predefined(\==).
predefined(=<).
predefined(=..).
predefined(~).
predefined(!).
predefined('C').
predefined(findall).
predefined(length).
predefined(functor).
predefined(print).
predefined(nl).
predefined(trace).
predefined(var).
predefined(nonvar).
predefined(infers).
predefined(delay).
predefined(generator).
```

```
predefined(generate).
predefined(subterm_constrained).
predefined(subterm_constrain).
predefined(clause).
predefined(term_variables).



:- module_interface(grammar).
:- begin_module(grammar).
:- import setarg/3 from sepia_kernel.
:- import constraint/1 from interpreter.

:- op(500,fx,constraint).
:- op(500,xfy,(?-)).
:- op(500,xfy,(:)).
:- op(1100,xfy,and).
:- op(1100,xfy,or).

:- meta_attribute(grammar,[unify:unify_attributes/2]).

unify_attributes(_{TermX},TermY):-
  -?-> TermX=TermY.
unify_attributes(X,_):-
  nonvar(X).

% =============
% begin Grammar
% =============
:- set_flag(all_dynamic,on).
% ==========
% XBar-Rules
% ==========

np((SynSem,[],_)) --> [],
  { empty(SynSem) }.

np(Phrase) -->
  ppro(Phrase).

np((SynSem,Quants,New)) -->
  rpro((SynSem,Quants)).
```

```
np((SynSem,Quants,New)) -->
  n3((SynSem,Quants)).

np((((Head,[]),(Index,AppCont)),Gaps),[Quant],_)) -->
  { Head=noun(_,Type),
    constraint member(Type,[ref,def,poss])
  },
  n3((((Head,[]),(Index,true)),Gaps),[Quant])),
  apposition((((_,(Index,AppCont)),_),[])).

n3((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { constraint (Quant=..[QF,Index,Res])
  },
  n2((((Head,[]),(Index,Res)),Gaps),[Quant])).

n3((((Head,[]),(Index,true)),Gaps),[Quant])) -->
  { content(Det,Quant),
    constraint specify(Det,Index,Res)
  },
  det((Det,[])),
  n2((((Head,[Det]),(Index,Res)),Gaps),[])).

n2(Phrase) -->
  n1(Phrase).

n2(((N1Cat,(Index,(N1Cont and AttrCont))),(Rel,[])),Quants)) -->
  { N1Cat=(noun(_,Type),_),
    Type ~= poss
  },
  n1(((N1Cat,(Index,N1Cont)),(Rel,[])),[])),
  postnom_attr((((_,(Index,AttrCont)),_),Quants)).

n1(Phrase) -->
  n(Phrase).

n1(((N1Cat,(Index,(N1Cont and AttrCont))),Gaps),Quant)) -->
  ap((((_,(Index,AttrCont)),([],[])),[])),
  n1(((N1Cat,(Index,N1Cont)),Gaps),Quant)).

n1((((Head,[Det]),(Index,N1Cont)),Gaps),NQuant)) -->
  { gaps(Comp,Gaps),
    constraint apply(Comp,Quants,NCont,N1Cont)
  },
```

```
  n(((((Head,[Det,Comp]),(Index,NCont)),Gaps),NQuant)),
  comp((Comp,Quants,NCont)).

n(Entry) --> [Noun],
  { noun_entry(Noun,Entry) }.

n(Entry) --> [Name],
  { name_entry(Name,Entry) }.

pn(Entry) --> [Name],
  { name_entry(Name,Entry) }.

ppro(Entry) --> [PPro],
  { ppro_entry(PPro,Entry) }.

rpro(Entry) --> [],
  { rpro_entry(empty,Entry) }.

rpro(Entry) --> [RPro],
  { rpro_entry(RPro,Entry) }.

det(Entry) --> [Quant],
  { det_entry(Quant,Entry) }.

ap(Entry) --> [Adj],
  { adj_entry(Adj,Entry) }.

pp((((Head,[]),Content),Gaps),Quants,New)) -->
  { gaps(NP,Gaps)
  },
  p(((((Head,[NP]),Content),([],[])),[])),
  np((NP,Quants,New)).

p(Entry) --> [Prep],
  { prep_entry(Prep,Entry) }.

vp(Phrase) -->
  v(Phrase).

vp(((((Head,[Subj]),VPCont),([],Slash)),[])) -->
  { slash(Obj,Slash),
    constraint apply(Obj,Quants,VCont,VPCont)
  },
```

```
  v(((((Head,[Subj,Obj]),VCont),([],[])),[])),
  comp((Obj,Quants,VCont)).


vp(((((Head,[Subj]),VPCont),([],Slashes)),[])) -->
  { slash(Obj1,Slash1),
    slash(Obj2,Slash2),
    constraint append(Slash1,Slash2,Slashes),
    constraint apply(Obj1,Quants1,VCont1,VPCont),
    constraint apply(Obj2,Quants2,VCont,VCont1)
  },
  v(((((Head,[Subj,Obj1,Obj2]),VCont),([],[])),[])),
  comp((Obj1,Quants1,VCont1)),
  comp((Obj2,Quants2,VCont)).

v(Entry) --> [Verb],
  { verb_entry(Verb,Entry) }.

s(((((Head,[]),SCont),([],SSlash)),[])) -->
  {  slash(Subj,SubjSlash),
    constraint append(SubjSlash,VPSlash,SSlash),
    constraint apply(Subj,SubjQuants,VPCont,SCont)
  },
  subj((Subj,SubjQuants,VPCont)),
  vp(((((Head,[Subj]),VPCont),([],VPSlash)),[])).

s1(Phrase) --> s(Phrase).
s1(((((Head,[]),(LeftCont and RightCont)),([],Slash)),[])) -->
  s(((((Head,[]),LeftCont),([],Slash)),[])),
  conj,
  s1(((((Head,[]),RightCont),([],Slash)),[])).

rel((((Cat,Cont),([Index],[])),[])) -->
  top(((Gap,([Index],[])),_,Cont)),
  s((((Cat,Cont),([],[Gap])),[])).

conj --> [Conj],
  { conjunct(Conj) }.

% ==================
% Syntactic Functions
% ==================
```

```
top(Phrase) -->
  subj(Phrase).
top(Phrase) -->
  comp(Phrase).

subj((SynSem,Quants,New)) -->
  np((SynSem,Quants,New)),
  { head(SynSem,noun(subj,_)) }.

comp((SynSem,Quants,New)) -->
  np((SynSem,Quants,New)),
  { head(SynSem,noun(obj,_)) }.
comp(Phrase) -->
  pp(Phrase).

apposition((((Cat,(Index,SCont)),([],[])),Quants,New)) -->
  rel((((Cat,SCont),([Index],[])),Quants)).

postnom_attr((((Cat,(Index,SCont)),([],[])),Quants,New)) -->
  rel((((Cat,SCont),([Index],[])),Quants)).

% =======
% Lexicon
% =======

noun_entry(man,((((noun(_,Type),[Det]),Content),([],[])),[])):-
  Content = (X{agr(3,sing,masc)},man(X)),
  cat(Det,(det(Type),[])).
noun_entry(woman,((((noun(_,Type),[Det]),Content),([],[])),[])):-
  Content = (X{agr(3,sing,fem)},woman(X)),
  cat(Det,(det(Type),[])).
noun_entry(car,((((noun(_,Type),[Det]),Content),([],[])),[])):-
  Content = (X{agr(3,sing,neut)},car(X)),
  cat(Det,(det(Type),[])).

noun_entry(owner,( ( ((noun(_,Type),[Det,Comp]),Content),
                     ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,_)},own(X,Y)),
  cat(Det,(det(Type),[])),
  subcategorize(Comp,pp(of),Y).

name_entry(Name,( ( ((noun(_,ref),[]),Content),
                    ([],[]) ),[the(X,Res)] ) ):-
```

```
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  constraint member(Name,[peter,paul,john]).
name_entry(Name,( ( ((noun(_,def),[Det]),Content),
                    ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  cat(Det,(det(def),[])),
  content(Det,Quant),
  constraint (Quant=..[QF,X,(Phi and Psi)]),
  constraint member(Name,[peter,paul,john]).
name_entry(Name,( ( ((noun(_,Type),[Det]),Content),
                    ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,masc)},name(X,Name)),
  cat(Det,(det(Type),[])),
  constraint member(Type,[quant,indef,poss]),
  constraint member(Name,[peter,paul,john]).
name_entry(Name,( ( ((noun(_,ref),[]),Content),
                    ([],[]) ),[the(X,Res)] ) ):-
  Content = (X{agr(3,sing,fem)},name(X,Name)),
  constraint member(Name,[mary,sandy,jane]).
name_entry(Name,( ( ((noun(_,Type),[Det]),Content),
                    ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,fem)},name(X,Name)),
  cat(Det,(det(Type),[])),
  content(Det,Quant),
  constraint (Quant=..[QF,X,(Phi and Psi)]),
  constraint member(Name,[mary,sandy,jane]).
name_entry(Name,( ( ((noun(_,Type),[Det]),Content),
                    ([],[]) ),[] ) ):-
  Content = (X{agr(3,sing,fem)},name(X,Name)),
  cat(Det,(det(Type),[])),
  constraint member(Type,[quant,indef,poss]),
  constraint member(Name,[mary,sandy,jane]).

rpro_entry(who,( ( ((noun(subj,wh),[]),Content),
                   ([X],[]) ),[] ) ):-
  Content = (X{agr(3,_,Gen)},true),
  constraint member(Gen,[masc,fem]).
rpro_entry(whom,( ( ((noun(obj,wh),[]),Content),
                    ([X],[]) ),[] ) ):-
  Content = (X{agr(3,_,Gen)},true),
  constraint member(Gen,[masc,fem]).
rpro_entry(empty,( ( ((noun(obj,wh),[]),Content),
                     ([X],[]) ),[] ) ):-
```

```
  Content = (X{agr(3,_,_)},true).
rpro_entry(which,( ( ((noun(_,wh),[]),Content),
                     ([X],[]) ),[] ) ):-
  Content = (X{agr(3,_,neut)},true).

ppro_entry(he,( ( ((noun(subj,ref),[]),Content),
                  ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  resolve(X,New).
ppro_entry(she,( ( ((noun(subj,ref),[]),Content),
                   ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,fem)},true),
  resolve(X,New).
ppro_entry(him,( ( ((noun(obj,ref),[]),Content),
                   ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  resolve(X,New).
ppro_entry(himself,( ( ((noun(obj,ref),[]),Content),
                       ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,masc)},true),
  o_bind(X,New).
ppro_entry(her,( ( ((noun(obj,ref),[]),Content),
                   ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,fem)},true),
  resolve(X,New).
ppro_entry(herself,( ( ((noun(obj,ref),[]),Content),
                       ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,fem)},true),
  o_bind(X,New).
ppro_entry(it,( ( ((noun(_,ref),[]),Content),
                  ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,neut)},true),
  resolve(X,New).
ppro_entry(itself,( ( ((noun(_,ref),[]),Content),
                      ([],[]) ),[],New ) ):-
  Content = (X{agr(3,_,neut)},true),
  o_bind(X,New).

det_entry(a,((((det(indef),[]),exists(X,Res)),([],[])),[])).
det_entry(the,((((det(def),[]),the(X,Res)),([],[])),[])).
det_entry(every,((((det(quant),[]),every(X,Res)),([],[])),[])).

det_entry(his,( ( ((det(poss),[]),the(Y,(PossRel and Res))),
```

```
                  ([],[]) ),[] ) ):-
  PossRel=poss(X{agr(3,_,masc)},Y),
  resolve(X,PossRel).
det_entry(her,( ( ((det(poss),[]),the(Y,(PossRel and Res))),
                  ([],[]) ),[] ) ):-
  PossRel=poss(X{agr(3,_,fem)},Y),
  resolve(X,PossRel).
det_entry(its,( ( ((det(poss),[]),the(Y,(PossRel and Res))),
                  ([],[]) ),[] ) ):-
  PossRel=poss(X{agr(3,_,neut)},Y),
  resolve(X,PossRel).

verb_entry(sleeps,( ( ((verb(fin,_,_),[Subj]),sleep(X)),
                      ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X).

verb_entry(loves,( ( ((verb(fin,_,_),[Subj,Obj]),love(X,Y)),
                     ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj,np(obj),Y).

verb_entry(knows,( ( ((verb(fin,_,_),[Subj,Obj]),knows(X,Y)),
                     ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj,np(obj),Y).

verb_entry(owns,( ( ((verb(fin,_,_),[Subj,Obj]),own(X,Y)),
                    ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj,np(obj),Y).

verb_entry(sells,((((verb(fin,_,_),[Subj,Obj1,Obj2]),sell(X,Y,Z)),
                   ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj1,np(obj),Y),
  subcategorize(Obj2,pp(to),Z).

verb_entry(buys,( ( ((verb(fin,_,_),[Subj,Obj1,Obj2]),buy(X,Y,Z)),
                    ([],[]) ),[] ) ):-
  subcategorize(Subj,np(subj),X),
  subcategorize(Obj1,np(obj),Y),
  subcategorize(Obj2,pp(from),Z).
```

```
adj_entry(red,((((adj,[]),(X,red(X))),([],[])),[])).
adj_entry(good,((((adj,[]),(X,good(X))),([],[])),[])).
adj_entry(old,((((adj,[]),(X,old(X))),([],[])),[])).

prep_entry(of,((((prep(of,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).
prep_entry(with,((((prep(with,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).
prep_entry(to,((((prep(to,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).
prep_entry(from,((((prep(from,RefType),[NP]),Cont),([],[])),[]))
:- pcomp_constr(NP,np(obj,RefType),Cont).

% ============================
% Deterministic Constraints I:
%         Projection Functions
% ============================

rel((_,(Rel,_)),Rel).                      % SynSem --> Rel
slash((_,(_,Slash)),Slash).                % SynSem --> Slash
gaps((_,Gaps),Gaps).                      % SynSem --> Gaps
content(((_,Content),_),Content).          % SynSem --> Content
index(((_,(Index,_)),_),Index).            % SynSem --> Index
restr(((_,(_,Res)),_),Res).                % SynSem --> Restriction
cat(((Cat,_),_),Cat).                      % SynSem --> Category
head((((Head,_),_),_),Head).              % SynSem --> Head
subcat((((_,Subcat),_),_),Subcat).        % SynSem --> Subcat
det_type((((det(Type),[]),_),_),Type).    % SynSem --> Type

% =============================
% Deterministic Constraints II:
%           Category Predicates
% =============================

empty((Local,(_,[Local]))):-
  Local = ((noun(_,_),[]),_).

relation(peter).
relation(paul).
relation(john).
relation(mary).
relation(sandy).
relation(jane).
```

```
relation(man).
relation(woman).
relation(car).
relation(red).
relation(good).
relation(sleep).

relation(name).
relation(love).
relation(own).

relation(sell).
relation(buy).

atom(Atom,Rel,Args):-
  Atom=..[Rel|Args],
  relation(Rel).

conjunct(and).
conjunct(.).

quantifier(every).
quantifier(exists).
quantifier(the).

dynamic_quant(exists).
dynamic_quant(the).

% ==============================
% Deterministic Constraints III:
%         Complement Constraints
% ==============================

pcomp_constr(Comp,np(Case,RefType),Cont):-
  cat(Comp,(noun(Case,RefType),[])),
  content(Comp,Cont).

subcategorize(Comp,np(Case),X):-
  cat(Comp,(noun(Case,_),[])),
  index(Comp,X).
subcategorize(Comp,pp(PForm),X):-
  cat(Comp,(prep(PForm,_),[])),
```

```
  index(Comp,X).

% ============================
% Nondeterministic Constraints
% ============================

apply(SynSem,[],Nuc,Nuc).
apply(SynSem,[Quant],Nuc,Quant:Nuc):-
  constraint
    ( restr(SynSem,Res),Res == true ).
apply(SynSem,[Quant],Nuc,Quant:(Res and Nuc)):-
  constraint
    ( restr(SynSem,Res),Res \== true ).

specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type = poss),
  Quant=..[QF,Index,(poss(X,Index) and Res)].
specify((((det(Type),[]),Quant),([],[])),Index,Res):-
  constraint (Type ~= poss),
  Quant=..[QF,Index,Res].

member(X,[X|_]).
member(X,[Y|R]):-
  constraint(X~=Y),
  member(X,R).

append([],X,X).
append([X|RestX],Y,[X|RestY]):-
  append(RestX,Y,RestY).

% ===============
% Test Predicates
% ===============

resolve(X,New):-
  dis?- scope(Ante,Old,New),
  not( (o_commander(Com,X,New), Com == Ante) ),
  Ante=X.

o_bind(X,New):-
  o_commander(C,X,New),
  C=X.
```

```
o_commander(C,X,New):-
  nonvar(New),
  New=..[Rel|Args],
  o_commands(C,X,Args).

o_commands(A,X,[A|Args]):-
  meta_member(X,Args).
o_commands(C,X,[A|Args]):-
  o_commands(C,X,Args).

meta_member(X,[Y|_]):-
  X==Y.
meta_member(X,[Y|R]):-
  X\==Y,
  meta_member(X,R).

var_res(Quant,X,Res):-
  nonvar(Quant),
  Quant=..[Q,X,Res],
  quantifier(Q).

scope(X,(Phi and Psi),Chi):-
  subform(Chi,Psi),
  accessible(X,Phi).
scope(X,Quant:Phi,Psi):-
  var_res(Quant,X,Res),
  subform(Psi,Phi).
scope(X,Quant:Phi,Psi):-
  var_res(Quant,X,Res),
  subform(Psi,Res).
scope(X,Quant:Phi,Psi):-
  var_res(Quant,Y,Res),
  X~=Y,
  subform(Psi,Phi),
  accessible(X,Res).

subform(Phi,Phi).
subform(Chi,(Phi and Psi)):-
  subform(Chi,Phi).
subform(Chi,(Phi and Psi)):-
  subform(Chi,Psi).
subform(Psi,Quant:Phi):-
  subform(Psi,Phi).
```

```
subform(Psi,Quant:Phi):-
  var_res(Quant,X,Res),
  subform(Psi,Res).

accessible(X,Atom):-
  atom(Atom,Rel,Args),
  member(X,Args),
  var(X).
accessible(X,(Phi and Psi)):-
  accessible(X,Phi),!.
accessible(X,(Phi and Psi)):-
  accessible(X,Psi).
accessible(X,Quant:Nuc):-
  Quant=..[Q,X,Res],
  dynamic_quant(Q).
accessible(X,Quant:Nuc):-
  Quant=..[Q,Y,Res],
  dynamic_quant(Q),
  X~=Y,
  accessible(X,Res),!.
accessible(X,Quant:Nuc):-
  Quant=..[Q,Y,Res],
  dynamic_quant(Q),
  X~=Y,
  accessible(X,Nuc).

:- set_flag(all_dynamic,off).
% ===========
% end Grammar
% ===========
```

Literatur

[1] Aczel, P.: *Non-Well-Founded Sets.* CSLI Lecture Notes 14, Chicago University Press, Chicago 1988.

[2] Aït-Kaci, H.: A lattice-theoretic approach on computation based on a calculus of partially ordered types. Ph.D. thesis, University of Pennsylvania 1984.

[3] Aït-Kaci, H. and Nasr, R.: LOGIN: A logical programming language with built-in inheritance. *Journal of Logic Programming* 3, 187-215, 1986.

[4] Barba, J.: A multidimensional modal translation for a formal system motivated by situation semantics. *Notre Dame Journal of Formal Logic*, 32(4), 598-608, 1991.

[5] Barba, J.: Two formal systems for situation semantics. *Notre Dame Journal of Formal Logic*, 33(?), ???-???, 1992.

[6] Barwise, J.: The Situation in Logic, CSLI Lecture Notes 17 (1988).

[7] Barwise, J. and Cooper, R.: Generalized quantifiers and natural language. *Linguistics and Philosophy* 4, 159-219, 1982.

[8] Barwise, J. and Cooper, R.: Extended Kamp Notation: a graphical notation for situation theory. In: *Situation Theory and Its Applications* III, CSLI Lecture Notes, Chicago University Press, Chicago 1993.

[9] Barwise, J. und Etchemendy, J.: *The Liar.* Oxford University Press 1987.

[10] Barwise, J. and Etchemendy, J.: Information, infons and inference. In: *Situation Theory and its Applications* I, CSLI Lecture Notes 22, 33-78, Chicago University Press, Chicago 1990.

[11] Barwise, J. und Perry, J.: Situations and Attitudes, Bradford Books, MIT Press 1983.

[12] Beaver, D.I.: The kinematics of presupposition, in H. Kamp (ed.), *Presupposition*, ILLC, University of Amsterdam, Dyana-2 deliverable R2.2.A, Part II 1993.

[13] Black, A. W.: *A Situation Theoretic Approach to Computational Semantics.* Ph.D. Thesis, Department of Artificial Intelligence, University of Edinburgh, Edinburgh 1992.

[14] Carpenter, B.: *The Logic of Typed Feature Structures*, Cambridge University Press 1992.

[15] Chierchia, G.: Anaphora and dynamic binding, *Linguistics and Philosophy* 15, 1992.

[16] Chomsky, N.: *Syntactic Structures.* Mouton, The Hague 1957.

[17] Chomsky, N.: *Lectures on Government and Binding.* Foris, Dordrecht 1988.

[18] Colmerauer, A.: Prolog and infinite trees. In: K. L. Clark and S.-A. Tärnlund (eds.), *Logic Programming*, 1-44. Volume 16 of *APIC Studies in Data Processing.* Academic Press, New York 1982.

[19] Colmerauer, A.: Prolog II: Reference manual and theoretical model. Internal Report, Groupe Intelligence Artificielle, Université Aix-Marseilles II, Marseilles 1982.

[20] Colmerauer, A.: Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, 85-99, Tokyo 1984.

[21] Colmerauer, A.: Theoretical model of Prolog II. In: M. von Canegham and D. H. Warren (eds.), *Logic Programming and its Applications*, 1-31, Ablex, Norwood, New Jersey 1987.

[22] Colmerauer, A.: Final Specifications for PrologIII, Esprit P1106 report: Further development of Prolog and its Validation by KBS in Technical Areas, Februar 1988.

[23] Dekker, P.: An update semantics for dynamic predicate logic, in P. Dekker and M. Stokhof (eds.), *Proceedings of the Eighth Amsterdam Colloquium*, ILLC, University of Amsterdam, Amsterdam 1992.

[24] Dekker, P.: *Transsentential Meditations. Ups and Downs in Dynamic Semantics*, Ph.D. thesis, ILLC/Department of Philosophy, University of Amsterdam, Amsterdam 1993.

[25] Dekker, Paul: Predicate Logic with Anaphora, ILLC Research Report LP-94-11, Universiteit van Amsterdam 1994.

[26] Devlin, K.: Logic and Information, Cambridge University Press, Cambridge 1991.

[27] Dörre, J. and Eisele, A.: Feature logic with disjunctive unification. In: *Proceedings of the 13th International Conference on Computational Linguistics*, 100-105, Helsinki 1990.

[28] Dowty, D., Wall, R. and Peters, S.: *Introduction to Montague Semantics.* Studies in Linguistics and Philosophy, H. Reidel, Dordrecht 1981.

[29] Emden, M. van, and Kowalski, R.A.: The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4), 733-743, 1976.

[30] Franz, A.: A parser for HPSG. Technical Report LCL-90-3, Laboratory for Computational Linguistics, Carnegie Mellon University, Pittsburgh, Pennsylvania 1990.

[31] Gawron, M. and Peters, S.: *Anaphora and Quantification in Situation Semantics.* CSLI Lecture Notes 19, Chicago University Press, Chicago 1990.

[32] Gazdar, G., Klein, E., Pullum, G. and Sag, I.: *Generalized Phrase Structure Grammar.* Basil Blackwell, Oxford 1985.

[33] Grätzer, G.: *Lattice Theory: First Concepts and Distributive Lattices.* W.H. Freeman and Company, San Francisco 1971.

[34] Groenendijk, J. and Stokhof, M.: Dynamic Montague Grammar. In: *Quantification and Anaphora* I, DYANA Deliverable R2.2A, 1-37, Centre for Cognitive Science, University of Edinburgh 1991.

[35] Groenendijk, J. and Stokhof, M.: Dynamic predicate Logic, *Linguistics and Philosophy* 14(1), 39-100, 1991.

[36] Guenthner, F.: Features and values. SNS-Bericht 88-40, Seminar für natürlichsprachliche Systeme, Universität Tübingen, Tübingen 1988.

[37] Harel, D.: Dynamic logic. In: D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic* II, 497-604, Reidel, Dordrecht 1984.

[38] Heim, I.: *The Semantics of Definite and Indefinite Noun Phrases*, Ph.D. thesis, University of Massachusetts, Amherst, 1982. Published in 1989 by Garland, New York.

[39] Heim, I.: File change semantics and the familiarity theory of definiteness, in R. Bäuerle, C. Schwarye and A. von Stechow (eds.), *Meaning, Use and Interpretation of Language*, de Gruyter, Berlin 1983.

[40] Heim, I.: Presupposition projektion and the semantics of attitude verbs, *Journal of Semantics* 9, 183-221, 1992.

[41] Hirsch, S.: PATR into Prolog. Master's thesis, Stanford University, Stanford 1987.

[42] Höhfeld, M. and Smolka, G.: Definite relations over constraint languages. LILOG-Report 53, IBM Deutschland GmbH, Stuttgart 1988.

[43] Hopcroft, J. and Ullman, J.: *Introduction to automata theory, languages and computation.* Addison-Wesley, Reading, Massachusetts 1979.

[44] Jaffar, J. und Lassez, J.-L.: Constraint Logic Programming. *14th ACM Symposium on the Principles of Programming Languages*, 111-119, München 1987.

[45] Johnson, M.: A logic of attribute-value structures and the theory of grammar. Ph.D. thesis, Stanford University 1987.

[46] Johnson, M.: *Attribute-Value Logic and the Theory of Grammar.* CSLI Lecture Notes 14, Center for the Study of Language and Information, Stanford 1988.

[47] Johnson, M. and Klein, F.: Discourse, anaphora and parsing. In: *Proceedings of the 11th International Conference on Computational Linguistics*, 669-675, Bonn 1986.

[48] Kamp, H.: A theory of truth and semantic representation, in J. Groenendijk, T. Janssen and M. Stokhof (eds.), *Formal Methods in the Study of Language*, Mathematical Centre, Amsterdam 1981. Reprinted in J. Groenendijk, T. Janssen and M. Stokhof (eds.), *Truth, Interpretation and Information*, Foris, Dordrecht 1984.

[49] Kamp, H. and Reyle, U.: *From Discourse to Logic*, Kluwer, Dordrecht 1993.

[50] Kasper, R.T.: Feature structures: A logical theory with applications to language analysis. Ph.D. thesis, University of Michigan, Ann Arbor, Michigan 1987.

[51] Kasper, R.T.: A unification method for disjunctive feature structures. In: *Proceedings of the 25th Annual Conference of the Association for Computational Linguistics* 235-242, Stanford 1987.

[52] Kasper, R.T. and Rounds, W.C.: The logic of unification in grammar. *Linguistics and Philosophy* 13(1), 35:58, 1990.

[53] Kay, M.: Functional grammar. In: C. Chiarello (ed.), *Proceedings of the 5th Annual Meeting of the Berkeley Linguistic Society*, 142-158, Berkeley 1979.

[54] Kay, M.: Functional unification grammar: a formalism for machine translation. In: *Proceedings of the 10th International Conference on Computational Linguistics*, 75-78, Stanford 1984.

[55] King, P.: A logical formalism for Head-Driven Phrase Structure Grammar. Ph.D. thesis, University of Manchester, Manchester, England 1989.

[56] Landman, F.: Data semantics: An epistemic theory of partial objects. In: F. Landman: *Towards a Theory of Information.* Foris, Dordrecht 1986.

[57] Lassez, J.-L., Maher, M.J. and Marriott, K.G.: Unification revisited. Internal Report RC 12394 (55630), IBM T.J. Watson Research Center, Yorktown Heights, New York 1986.

[58] Lloyd, J.W.: *Foundations of Logic Programming.* Springer Verlag, Berlin 1984.

[59] Martelli, A. and Montanari, U.: An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), 258-282, 1982.

[60] Minsky, M.: A framework for representing knowledge. In: P.H. Winston (ed.): *The Psychology of Computer Vision.* McGraw-Hill, New York 1975.

[61] Montague, R.: The proper treatment of quantification in English. In: R. Thomason (ed.), *Formal Philosophy.* New York, Yale University Press, 1970.

[62] Moshier, D. and Rounds, W.: A logic for partially specified data structures. In: *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 156-167, München 1987.

[63] Mukai, K.: Ph.D. thesis???

[64] Muskens, R.: *Meaning and Partiality.* Ph.D. thesis, University of Amsterdam, Amsterdam 1989.

[65] Nakashima, H., Peters, S. und Schütze, H.: Communication and Inference through Situations. In: Proceedings of IJCAI 1991.

[66] Nakashima, H., Suzuki, H., Halvorsen, P.-K. und Peters, S.: Towards a computational interpretation of situation theory. In: Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS-88, Tokyo 1988.

[67] Nakashima, H. und Tutiya, S.: Inference *in* a situation *about* situations. In: Situation Theory and its Applications, 2, 1991.

[68] Nebel, B. and Smolka, G.: Representation and reasoning with attributive descriptions. IWB Report 81, IBM Deutschland GmbH, Stuttgart 1989.

[69] Paterson, M. and Wegman, M.: Linear unification. *Journal of Computer and System Sciences*, 16, 158-167, 1978.

[70] Pereira, F.: *Logic for Natural Language Analysis.* Ph.D. thesis, University of Edinburgh, Edinburgh 1982. Reprinted as Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, CA.

[71] Pereira, F.C.N. and Shieber, S.M.: The semantics of grammar formalisms seen as computer languages. In: *Proceedings of the 10th International Conference on Computational Linguistics*, 123-129, Stanford 1984.

[72] Pereira, F. C. N. and Shieber, S. M.: *Prolog and Natural-Language Analysis.* CSLI Lecture Notes 10, Center for the Study of Language and Information, Stanford 1987.

[73] Pereira, F. and Warren, D.: Definite Clause Grammars for language analysis. *Artificial Intelligence* 13, 231-278, 1980.

[74] Pereira, F. and Warren, D.: Parsing as deduction. In: *21st Annual Conference of the Association for Computational Linguistics*, 137-144, MIT, Massachusetts 1983.

[75] Pollard, C.J. and Sag, I.A.: *Information-Based Syntax and Semantics: Volume I, Fundamentals.* CSLI Lecture Notes 13, Center for the Study of Language and Information, Stanford 1987.

[76] Pollard, C.J. and Sag, I.A.: *Head-driven Phrase Structure Grammar.* Chicago University Press, Chicago 1994.

[77] Robinson, J.A.: A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12, 23-41, 1965.

[78] Rooth, M.: Noun phrase interpretation in Montague Grammar, file change semantics and situation semantics. In: P. Gardenförs (ed.), *Generalized Quantifiers*, Studies in Linguistics and Philosophy 31, Reidel, Dordrecht 1987.

[79] Rounds, W.C. and Kasper, R.T.: A complete logical calculus for record structures representing linguistic information. In: *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, 39-43, Cambridge, Massachusetts 1986.

[80] Sandt, Rob A. van der: Presupposition projection as anaphora resolution, *Journal of Semantics* 9, 333-377, 1992.

[81] Scott, D.S.: Outline of a mathematical theory of computation. Programming Research Group Technical Monograph PRG-3, University of Oxford, Oxford 1970.

[82] Schütze, H.: The PROSIT Language v0.4, Unveröffentlichtes Manual, CSLI Stanford 1991.

[83] Sells, P.: *Lectures on Contemporary Syntactic Theories.* CSLI Lecture Notes 3, Center for the Study of Language and Information, Stanford 1987.

[84] Shieber, S.M.: *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes 4, Center for the Study of Language and Information, Stanford 1986.

[85] Siekmann, J.H.: Unification theory. *Journal of Symbolic Computation* 7, 207-274, 1989.

[86] Smolka, G.: A feature logic with subsorts. LILOG Report 33, IBM Deutschland GmbH, Stuttgart 1988.

[87] Smolka, G.: Attributive concept descriptions with unions and complements. IWBS Report 68, IBM Deutschland GmbH, Stuttgart 1989.

[88] Smolka, G.: Feature constraint logics for unification grammars. IWBS Report 93, IBM Deutschland GmbH, Stuttgart 1989.

[89] Smolka, G. and Aït-Kaci, H.: Inheritance hierarchies: Semantics and unification. *Journal of Symbolic Computation* 7, 343-370, 1989.

[90] Smullyan, R.: *First Order Logic.* Springer Verlag, Berlin, Heidelberg, New York 1968.

[91] Sterling, L. and Shapiro, E.Y.: *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Massachusetts, 1986.

[92] Stoy, J.E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, Massachusetts, 1977.

[93] Vallduví, E.: The dynamics of information packaging. In E. Engdahl (ed.) *Integrating Information Structure into constraint-based and Categorial Approaches.* ESPRIT Basic Reasearch Project 6852, DYANA-2 Deliverable R1.3.B. ILLC, University of Amsterdam 1994.

[94] Veltman, F.: Data semantics. In: J. Groenendijk, Janssen and M. Stokhof (eds.), *Formal Methods in the Study of Language*, Mathematical Centre Tracts 135, Amsterdam 1981.

[95] Weich, Th.: *Zum Begriff der Täuschung.* Magisterarbeit, Institut für Logik und Wissenschaftstheorie, Universität München, 1991.

[96] Winograd, T.: *Language as a Cognitive Process: Volume 1, Syntax.* Addison-Wesley, Reading, Massachusetts, 1983.

[97] Wittgenstein, L.: *Philosophische Untersuchungen.* Suhrkamp, Frankfurt 19??.

[98] Zeevat, H.: A compositional approach to Discourse Representation Theory. *Linguistics and Philosophy* 12, 95-131, 1989.

[99] Zeevat, H.: *Aspects of Discourse Semantics and Unification Grammar*, Ph. D. thesis, University of Amsterdam 1991.

[100] Zeevat, H.: Presupposition and accomodation in update semantics, *Journal of Semantics* 9, 379-412, 1992.

ISBN 3-930859-15-7