Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-95-04

Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache

Victoria Hall

March 1995

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ☐ Intelligent Engineering Systems
- ☐ Intelligent User Interfaces
- ☐ Computer Linguistics
- ☐ Programming Systems
- ☐ Deduction and Multiagent Systems
- ☐ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache

Victoria Hall

DFKI-D-95-04

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-8902 C4).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1995
This work may not be copied or reproduced in whole of part for any commercial purpose. Permission to copy in whole or part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Deutsche Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.
ISSN 0946-0098

Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache

Victoria Hall

17. März 1995

Abstract

Sorts are incorperated into RELFUN as distinguished unary predicates usable as first-class citizens. Inheritance is integrated into the unification process via a taxonomy defined by a partial-order relation on the sorts. Sort knowledge describes which individuals belong to a sort (ABOX) and which order dependencies exist between the sorts (TBOX). This special knowledge should be recognizable as such. The seperation can be reached by a special (second-order) predicate `subsumes` or by partitioning the knowledge base. Focusing on partitioning, we can choose a dynamic (run-time) or static (compile-time) model for the unification involving sorted terms. In the static model a precompiler generates an internal structure representing the reflexive-transitive sort closure and the defined individuals of each sort. Using this structure, unification is reduced to list intersectiom (sort-sort unification, succeeding with the glb) or a membership test (sort-individual unification). It also provides elementary validation operations for taxonomy acylicity, unique glb's, and the equivalence of intensional (TBOX) and exensional (ABOX) glb's.

Inhaltsverzeichnis

1 Einleitung **2**

2 Sorten als First-Class Citizens **4**

3 Sortenbedingungen **10**
- 3.1 Bedingungen an die Ordnung über den Sorten 10
- 3.2 Bedingungen an die Instantiierung der Sorten 12

4 Dynamische GLB-Berechnung **14**
- 4.1 Modell_1: Gemeinsame Sorten-/Operatorbasis mit `subsumes`-Prädikat 14
- 4.2 Modell_2: Getrennte Sorten-/Operatorbasis und dynamischer GLB . . 16

5 Statische GLB-Berechnung **18**
- 5.1 Horizontale Vorcompilation 18
- 5.2 Modell_3: Getrennte Sorten-/Operatorbasis und statischer GLB . . . 22

6 Ausblick **23**

A Unifikation **28**
- A.1 Unifikation mit „:"-Notation für getypte Variablen 28
- A.2 Modell_1 32
- A.3 Modell_2 33
- A.4 Modell_3 34
  - A.4.1 Vorcompilation 34
  - A.4.2 Tools zur Prüfung der Restriktionen 40
  - A.4.3 UNSUBSUMES und RESUBSUMES 43
  - A.4.4 Unifikation mit :-Notation für getypte Variablen 44

B Erweiterung um Sortbase **46**

C Ein Beispieldialog über RTPLAST **49**

Kapitel 1

Einleitung

Strukturierungstechniken haben in der modernen Programmierung immer mehr an Bedeutung gewonnen. Datentypen sind mittlerweile Bestandteil fast jeder prozeduralen Sprache [Wir85]. Objekt-orientierte Sprachen wie etwa SmallTalk[GR85], die auf dem Konzept der Klassenhierarchien aufbauen, verbreiten sich zunehmend. Auch in Anwendungen, z.B. Spracherkennung oder Automatisches Beweisen, bedient man sich mehr und mehr des Typ-Konzepts. Strukturierung erscheint für größere Softwareanwendungen nicht mehr nur wünschenswert, sondern sogar unabdingbar. Auf der anderen Seite bietet sich für komplexere Programme die deklarative Programmierung, etwa im Sinne von purem PROLOG, aus Gründen der Erweiterbarkeit und Lesbarkeit an. Daher halten Typ-Konzepte auch verstärkt Einzug in die logische Programmierung. Die flexible Integration „mehrstufiger Typen“ (Sorten) in die relational-funktionale Sprache RELFUN [BEH$^+$93] ist das Thema dieser Arbeit.

Konzeptionell werden Individuen (mit gleichen Eigenschaften) zu je einer Sorte zusammengefaßt. Sorten werden zunächst extensional beschrieben, d.h. alle Individuen der Sorten werden explizit aufgezählt und dann extensional oder intensional benutzt. Durch die Einführung von Sorten kann das Behauptungswissen (Wissen bzgl. des Anwendermodells)[1] strukturiert werden: auf der Menge der Sorten wird eine explizite Ober-/Untersorten-Relation definiert, wodurch eine partielle Ordnung („order-sorted“) entsteht. Man erhält so ein neues Abstraktionsniveau, welches durch die Sorten beschrieben und benannt wird [Mey94]. Damit steigt bei der Abarbeitung von Regeln mit Typrestriktionen die Effizienz, da sich der Suchraum bei der Lösungsfindung drastisch reduziert[Wal84, Bei87, Sti83]. Benutzt man getypte Variablen (d.h. Variablen, die nur an Elemente einer gegebenen Sorte gebunden werden können) im Kopf einer Regel, wird nur versucht die Regel anzuwenden, wenn die Argumentterme von den entsprechenden Sorten sind. In diesem Sinn kann man Sorten als Constraints betrachten [Tsa94]. Aus der Sicht von KL-ONE [BS85] entspricht eine hier direkt zu definierende Sortenhierarchie einer Taxonomie, wie sie eine vorgeschaltete *Klassifikation* aus einer Menge intensionaler Konzept- oder Sortendefinitionen automatisch

[1] Formal soll als Behauptungswissen das gesamte Wissen, das weder die Taxonomie beschreibt noch Sorten instantiiert, bezeichnet werden.

produzieren könnte [2]. Analog könnte die Zuordnung von Individuen zu Sorten durch eine vorgeschaltete *Realisation* im Sinne von KL-ONE automatisiert werden. Es stellt sich aber heraus, daß von diesem Ausgangspunkt aus eine Reihe weiterer interessanter Problemstellungen existieren sowie Dienstleistungen und Inferenzen einführbar sind, wie dies z.T. schon in LIFE [AKP, AK88] gezeigt wurde.

Für die Erweiterung der Unifikation zur Sortenunifikation benötigt man:

1. einen Test, der prüft, ob *ein Individuum zu einer Sorte* gehört. Über die partielle Ordnung, die auf den Sorten definiert wurde, kann man diese Instanzenprüfungen von Sorten prinzipiell nach „unten" oder „oben" (wir werden die Richtung nach „unten" benutzen, d.h. „rückwärts" schließen) weiterreichen.

2. eine GLB-Berechnung (greatest lower bound, Infimum), welche die *größte gemeinsame Untersorte von zwei gegebenen Sorten* bestimmt. Diese kann dynamisch (Modell_1, Modell_2) oder statisch (Modell_3) erfolgen.

In deklarativen Sprachen wie RELFUN kann man Sorten nach dem Muster endlicher Domänen als First-Class Citizens einführen [Bol94]; das bedeutet, Sorten werden in die Sprache in natürlicher Weise als vollwertige Objekte integriert, indem man sie weitgehend wie gewöhnliche Terme behandelt. Man kann sie z.B. an logische Variablen binden und als Funktionswert zurückgeben (siehe Kapitel 2).

Auf die möglichen Anforderungen an die Ordnung, die die Taxonomie beschreibt, und an die Interpretation der Sorten wird kurz in Kapitel 3 eingegangen. Anschließend werden die beiden Modelle mit dynamischer GLB-Berechnung vorgestellt. Modell_1 benutzt ein second-order Prädikat, `subsumes`, und Modell_2 verwendet Partitionierung zur Abgrenzung des Sortenwissens vom Behauptungswissen (siehe Kapitel 4). Modell_3 ist eine Modifikation des zweiten Modells. Es benutzt eine effizientere statische GLB-Berechnung. In Anhang C wird ein realistisches Beispiel aus dem Bereich recyclingrelevanter Materialien, RTPLAST [Buh94], gezeigt.

[2] Diese Arbeitsteilung wurde z.B. in COLAB [BHHM93] an der TAXON-CONTAX-Schnittstelle zur Constraintpropagierung über hierarisch strukturierten Domänen erprobt.

Kapitel 2

Sorten als First-Class Citizens

In RELFUN werden Sorten als unäre Prädikate aufgefaßt, die auf besondere Weise definiert und benutzt werden.

Um ein Prädikat als Sorte zu benutzen, wird einfach eine Variante seines Namen verwendet; es gibt also keine getrennte Deklaration[1] der Sorten. Um die Sorten (-namen) von Konstanten (kein Präfix) und Variablen („_"-Präfix bzw. großer Anfangsbuchstabe) unterscheiden zu können, zeichnet man Sorten mit einem neuen Präfixzeichen („$") aus.

`dog`	ist eine Konstante
`_dog` oder `Dog`	ist eine Variable
`$dog`	ist eine Sorte

Defaultmäßig sind Variablen in RELFUN nicht typisiert; jetzt können sie durch einen „:"-Infix *getypt* werden, also in ihrem Wertebereich eingeschränkt werden.

`X : $dog` Die Variable `X` kann nur noch Werte der Sorte `$dog` annehmen.

Eine Sorte `$s`$_1$ subsumiert eine Sorte `$s`$_2$, wenn die Sorte `$s`$_1$ alle Individuen der Sorte `$s`$_2$ (und evtl. zusätzliche Individuen) enthält. Mit anderen Worten, die Sorte `$s`$_2$ ist spezieller als die Sorte `$s`$_1$. Über diese Subsumtionsrelation kann man die Sorten in Beziehung zueinander setzen und erhält so eine Taxonomie. Es erscheint natürlich, diese Taxonomie mittels Hornregeln, für die entsprechenden Prädikate, zu definieren.

`veb(X) :- dog(X)` wobei die Prädikate `veb` (als Abkürzung von vertebrate) und `dog` die Sorten `$veb` und `$dog` beschreiben. Die Regel drückt dann aus, daß `$dog` eine direkte Untersorte von `$veb` ist.

Allerdings sind so die subsumtionsbeschreibenden Regeln für taxonomische Prädikate nicht mehr unterscheidbar von DATALOG-Regeln mit einer Prämisse für gewöhnliche unäre Prädikate. Dies ist für die (effiziente) Verwaltung der Sorten jedoch notwendig

[1] Sorten werden hier nur implizit deklariert, d.h. wenn der Bezeichner einer Sorte erstmals auftritt, ist aus dem Text des Bezeichners bzw. aus der Stelle des Programms, an der er auftritt, eindeutig erkennbar, wie die fehlende Deklaration lauten müßte.

(siehe Kapitel 4). Um eine Unterscheidung zu erreichen, kann man das Wissen partionieren, d.h. die taxonomischen Regeln in eine gesonderte Partition auslagern (hier: zur bisherigen `operatorbase` mit dem Behauptungswissen gibt es eine `sortbase` mit Sortenwissen) und/oder statt mit Hornregeln durch ein ausgezeichnetes „second-order" Prädikat (hier: `subsumes`[2]) beschreiben (siehe Abbildung 2.1).

`subsumes(veb, dog).` Man beachte, daß `$dog` als Untersorte von `$veb` definiert wird[3].

ausgezeichnete Prädikate →

Partitionierung ↓

	keine speziellen Prädikate	spezielles Prädikat subsumes
Operatorbase	keine effiziente Verwaltung	Modell_1
Operatorbase Sortbase	Modell_2	Modell_1&_2

Abbildung 2.1: Zwei unabhängige Dimensionen zur Trennung des Sortenwissens (sortbase/ subsumes-Prädikat) vom Behauptungswissen (operatorbase/ normale Prädikate)

Gleich welche Notation man wählt, wird in der ausgezeichneten Partition bzw. innerhalb des ausgezeichneten second-order Prädikates auf die textuelle Unterscheidung ($-Präfix) von Konstanten und Sorten verzichtet. Man kann vereinbaren, daß alle unären Prädikate der Sortenpartition bzw. die Argumente der `subsumes`-Klauseln für die spätere Verwendung mit einem $-Präfix gedacht sind.

In der Hornnotation können Sorten sowohl intensional als auch extensional benutzt werden. Intensionaler Gebrauch von Sorten soll hier bedeuten, daß Sorten als geschlossene Lösungen geliefert werden (`X is $dog` bindet[4] `X` an `$dog`; genauso `pet(X)`, falls das Behauptungswissen `pet(Y: $dog)` gespeichert ist). Unter extensionalem Gebrauch wird hier das Aufzählen der Individuen des Sortenprädikats verstanden (`dog(X)` bindet `X` an `fido` und `lassy`, falls das Sortenwissen `dog(fido)` und `dog(lassy)` gespeichert wurde). Führt man ein spezielles second-order Prädikat `subsumes` zur Beschreibung der Sortenbeziehungen ein, erreicht man zwar eine Abgrenzung zum Behauptungswissen, gleichzeitig wird aber der Gebrauch von Sorten i.a. auf intensionale Art beschränkt[5].

[2] `subsumes` ist ein binäres Prädikat der Prädikatenlogik 2. Stufe über unäre Prädikate 1. Stufe

[3] `subsumes`-Notation und Hornregel-Notation sind gerichtet in dem Sinn, daß erkennbar ist, was Ober- und was Untersorte ist.

[4] `is` unifiziert i.A. seine linke und rechte Seite.

[5] Da die Individuen mittels unärer Fakten den Sorten zugeordnet werden, können die Sorten, für die Individuen definiert werden extensional, benutzt werden. Allerdings werden dann nur die Individuen aufgezählt, die (direkt) für diese Sorte definiert wurden, da `subsumes(P, Q)` anders als `P(X) :- Q(X)` nicht für (indirekte) Folgerungen ausgenutzt wird.

Wünschenswert ist es jedoch, Sorten sowohl intensional als auch extensional nutzen zu können (siehe Abbildung 2.2).

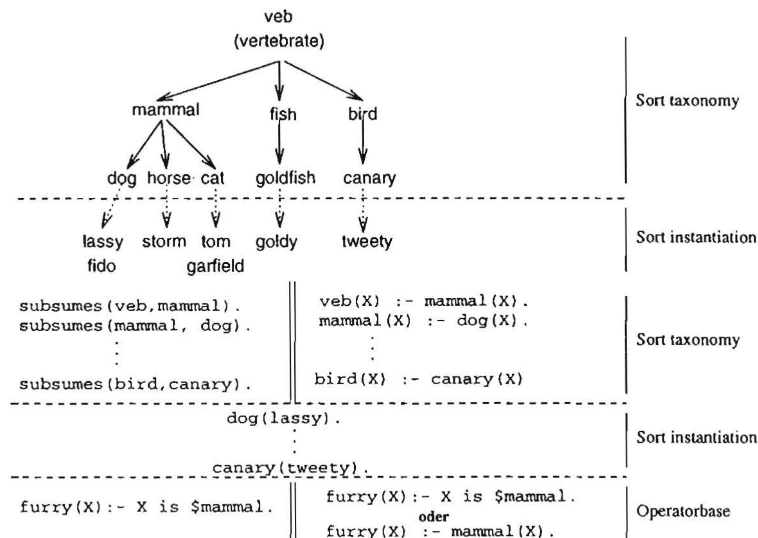


Abbildung 2.2: Die Sortentaxonomie (*sort taxonomy*) wird hier einmal mit `subsumes`-Notation und einmal mit Hornregeln beschrieben. Die Wurzel `$veb` bildet die allgemeinste Sorte der Taxonomie. Nach „unten" werden die Sorten immer spezieller. Die Zuweisung der Individuen zu einer Sorte (*sort instantiation*) geschieht in beiden Fällen über unäre Prädikate. Die Behauptungsregel „Alle Säugetiere haben Fell" (`furry(X) :- X is $mammal`, hier wäre auch `furry(X : $mammal)` möglich) in der `Operatorbase` kann nur im Fall der Hornregelnotation `furry(X) :- mammal(X)` extensional benutzt werden.

Eine zweite Möglichkeit der Abgrenzung besteht in der Strukturierung des Wissens mittels Partitionierung. Die Einteilung des Wissens in Partitionen ergibt sich aus dem inhaltlichen Zusammenhang. So erhält man eine Partition mit Behauptungswissen `operatorbase` und eine Partition mit dem zusätzlichen „Sortenwissen"[6] `sortbase`. Dabei können die `operatorbase` und die `sortbase` noch weiter untergliedert sein. Die `sortbase` könnte noch in Taxonomisches-/ und Instantiierungs-(KL-ONE: assertionales) Wissen unterteilt werden. Da sich die Notation des Taxonomisches-/ und Instantiierungs-Wissens eindeutig unterscheidet (*Instantiierungen*: unäre Fakten; *Taxonomien*: binäre Fakten [subsumes-Notation] oder unäre DATALOG-Regeln mit einer Prämisse), ist diese weitere Partitionierung zum Zweck der Trennung verschiedenartigen Wissens nicht nötig. Bei dem Partitionskonzept ist zu beachten, daß Wissen auf keinen Fall überschrieben werden darf, also die verschiedenen Partitionen disjunkt sein müssen. Dies wird durch die Voraussetzung, daß gleichnamige Prädikate nur in einer Partition definiert werden dürfen, sichergestellt (siehe Abbildung 2.3).

[6]Sortenwissen definiert die Sortenhierarchie und die Instantiierung der darin enthaltenen Sorten.

Man beachte, daß durch eine Unterscheidung in verschiedenartiges Wissen der Benutzer stets wissen muß, was als Sortenwissen und was als Behauptungswissen definiert wurde. Außerdem ergibt sich, bei der Erstellung der Wissensbasis, die Frage, welche unären DATALOG-Prädikate überhaupt taxonomisch wertvoll sind, d.h berechtigt sind als Sorte modelliert zu werden[NG94]. Hier bedeutet das, welches Wissen gehört in die `sortbase` bzw. muß mit `subsumes`-Fakten beschrieben werden.

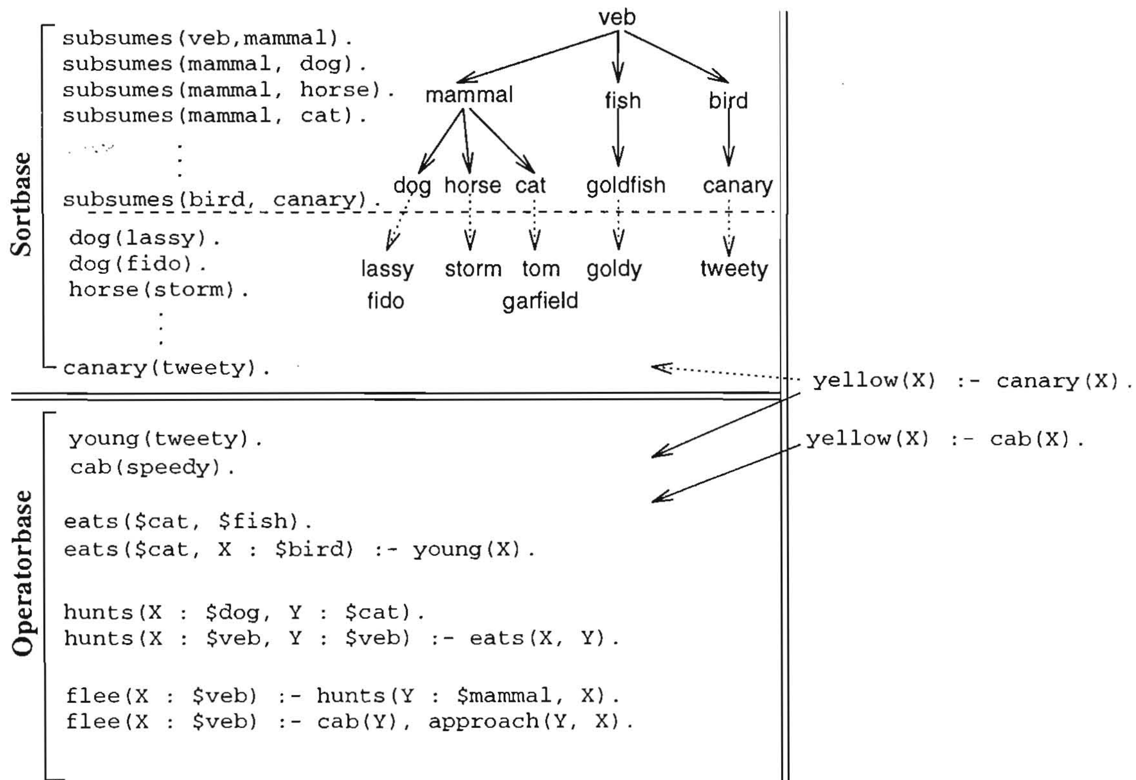


Abbildung 2.3: Die Klausel `yellow(X) :- cab(X)` gehört sicherlich zum Behauptungswissen. `yellow(X) :- canary(X)` dagegen könnte man inhaltlich dem Behauptungswissen oder dem Sortenwissen zuordnen. Dadurch, daß das Prädikat `yellow` aber bereits in der Operatorbase definiert ist, muß auch `yellow(X) :- canary(X)` in die Operatorbase aufgenommen werden.

Die Individuen einer Sorte werden explizit aufgezählt. Dies geschieht durch unäre Fakten:

`dog(lassy)` Das Individuum `lassy` wird der Sorte `$dog` zugewiesen.

Dabei werden Individuen von Sorten an ihre übergeordneten Sorten weitergereicht:

`veb(lassy).` Mit obiger Instanzenzuweisung muß das Individuum `lassy` nicht mehr explizit für die Sorte `$veb` angegeben werden, da `lassy` bereits als Individuum von `$dog` definiert wurde und `$dog` eine Untersorte von `$mammal` und diese wiederum eine Untersorte von `$veb` ist (siehe Abbildung 2.3).

Eine Sorte repräsentiert alle Individuen ihrer Untersorten und zusätzlich noch die Individuen, die direkt für diese Sorte definiert wurden.

Man beachte, daß die Sorten hier extensional (durch Aufzählen der Individuen) beschrieben werden. Potentiell unendlichen Sorten können also nicht beschreiben werden.

Sorten werden in RELFUN auf natürliche Weise integriert, was bedeutet, daß sie genauso wie „gewöhnliche“ Terme verwendet werden. Sie können also an

- eine logische Variable gebunden werden
- als Wert einer Funktion zurückgegeben werden
- anonym als Argument eines Literals oder in einer Struktur auftreten

Um Sorten als solche First-Class Citizens[Bol94], d.h. vollwertige Sprachobjekte, einzubauen, muß die Unifikation zu einer ordnungssortierten Unifikation erweitert werden. Die Sorteninformation soll während der Unifikation benutzt werden [BBDV91].

Unifikation von Sorten (o.B.d.A wird Sorte als 2. Term benutzt):

1. Term	2. Term	Unifikationsergebnis	
VAR	$PRED	$PRED	mit Seiteneffekt VAR : $PRED
CONST	$PRED	CONST	wenn CONST Individuum der Sorte $PRED, d.h. PRED(CONST) gilt
		unbekannt,	sonst
dom$[C_1 \ldots C_n]$	$PRED	dom$[C'_1 \ldots C'_m]$	mit $C'_1 \ldots C'_m$ sind Individuen der Sorte und $\forall i \in \{1 \ldots m\} \exists k \in \{1 \ldots n\} : C'_i = C_k$
		unbekannt,	sonst
exc$[C_1 \ldots C_n]$	$PRED	verboten	
$\$PRED_1$	$\$PRED_2$	$\$PRED_3$	wobei $\$PRED_3$ GLB($\$PRED_1$, $\$PRED_2$) ist

Bei der Unifikation einer freien Variablen mit einer Sorte wird die Variable in ihrem Wertebereich eingeschränkt, d.h. getypt. Ist die Variable bereits gebunden, verhält sich die Unifikation entsprechend dem Inhalt der Variable. Unifiziert man zwei Sorten, wird von allen gemeinsamen Untersorten (common lower bounds) die

allgemeinste/ größte Untersorte (GLB - greatest lower bound, Infimum) zum Ergebnis der Unifikation[7].

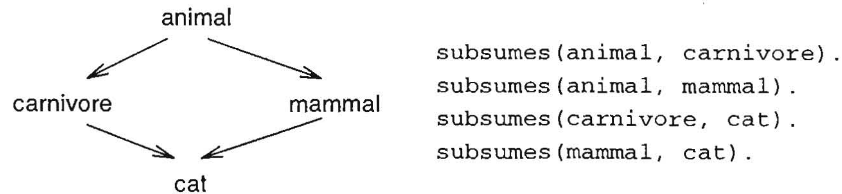


Abbildung 2.4: Hier wird `$cat` vom Benutzer als GLB der Sorten `$carnivore` und `$mammal` definiert.

`X is $mammal, X is $carnivore` Die Variable `X` wird zunächst an die Sorte `$mammal` und mit der zweiten Zuweisung an die Sorte `$carnivore` gebunden. Hier müssen die Sorten `$mammal` und `$carnivore` unifiziert werden. Das Ergebnis der Unifikation ist der GLB `$cat` dieser Sorten.

Man kann endliche Sorten[8] als benannte endliche Domänen[9] verstehen. Wird also eine endliche Domäne mit einer (endlichen) Sorte unifiziert, wird der Schnitt zwischen der endlichen Domäne und der Sorte gebildet. Als Resultat erhält man eine neue endliche Domäne mit denjenigen Elementen der ursprünglichen Domäne, die gleichzeitig auch Individuen der Sorte sind. In dem Spezialfall einer Konstanten (einelementige endliche Domäne) ist die Unifikation nur erfolgreich, wenn die Konstante ein Individuum der Sorte ist.

`X is $dog, X is dom[fido, fifi, lassy]` unifiziert mit `X = dom[fido, lassy]`.

`X is $dog, X is fido` unifiziert mit `X = fido`.

Die Unifikation einer endlichen Exklusion („negierte" endliche Domäne) [Bol94] mit einer potentiell unendlichen Sorte stößt auf große Implementationsprobleme, trotzdem müßte man bei endlichen Sorten eine solche Unifikation erlauben[10]. Natürlich sollen sich unendliche und endliche Sorten gleich verhalten; deshalb muß hier auf die Unifikation von endlichen Sorten mit Exklusionen verzichtet werden[11].

[7] Da der GLB zweier disjunkter Sorten nicht existiert, scheitert eine entsprechende Unifikation.

[8] Letztlich kann man nur endliche Sorten beschreiben, da Sorten extensional definiert werden.

[9] Endliche Domänen sind verwandt mit endlichen Mengen und Aufzählungstypen.

[10] Wenn man endliche Sorten als benannte Domänen versteht, muß folglich auch die Unifikation endlicher Sorten mit Exklusionen erlaubt sein (endliche Domänen sind mit Exklusionen unifizierbar [Bol94]).

[11] Im Rahmen dieser Arbeit wurden zwar nur endliche Sorten implementiert, doch die Erweiterung zu unendlichen Sorten wird theoretisch berücksichtigt.

Kapitel 3

Sortenbedingungen

Für die Sortenunifikation ist die GLB-Berechnung elementar. Damit diese sinnvoll ist, müssen einige Bedingungen an die Ordnung, die die Hierarchie der Sorten aufbaut und an die Instantiierung der Sorten, die durch das assertionale Wissen gegeben ist, erfüllt sein.

3.1 Bedingungen an die Ordnung über den Sorten

Um von einer Sorte auf die nächste schließen zu können, muß die Ordnung auf Sorten

- **Transitivität** (z.B. `subsumes(veb, mammal)` und `subsumes(mammal, dog)` $\rightarrow$ `subsumes(veb, dog)`)

erfüllen. Außerdem sollte die Taxonomie

- **Zyklenfreiheit**[1] (z.B. subsumes(veb_1, veb_2) und subsumes(veb_2, veb_1) $\rightarrow$ $veb_1 = veb_2$ [= hier im Sinne von identisch])

aufweisen, denn daraus resultiert u.a. das praktische Problem von Endlosschleifen bei der Berechnung des GLB's zweier Sorten[2]. Doch nicht nur aus praktischen sondern auch aus semantischen Gründen sollte man Zyklen verbieten. Was bedeutet, daß eine Sorte zugleich Sub- und Supersorte einer Sorte ist?

Jedoch ist ein Sonderfall der Zyklen nähmlich die

- **Schlaufen**[3] (z.B. `subsumes(veb, veb)`)

[1]Zyklenfreiheit wird in der Literatur auch oft als Antisymmetrie bezeichnet.

[2]Allerdings kann man durch Mitprotokollieren der bereits betrachteten Sorten bei dieser Berechnung Endlosschleifen erkennen. Will man diesen zusätzlichen Aufwand verhindern, muß man eine zyklenfreie Taxonomie fordern.

[3]Schlaufen werden in der Literatur auch oft durch die Eigenschaft der Reflexsivität beschrieben.

semantisch betrachtet anschaulich.

Ist die verwendete Ordnung über den Sorten eine partiellen Ordnung, sind genau diese Restriktionen erfüllt!

Weiter kann man

- **Eindeutigkeit**

fordern, d.h. daß für zwei beliebige Sorten aus der Taxonomie höchstens ein GLB definiert wurde.

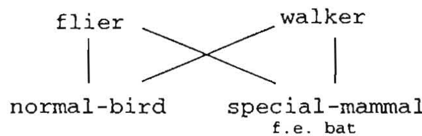


Abbildung 3.1: Mehrdeutige Taxonomie

Mehrdeutige Taxonomien entstehen häufig dadurch, daß die Hierarchie unter verschiedenen Gesichtspunkten aufgebaut wird. Hier z.B. werden die Sorten, die unter einem biologischen Aspekt Fortpflanzung gebildet wurden (`$special-mammal`, `$normal-bird`), mit den Sorten, die unter dem Gesichtspunkt Fortbewegung entstanden sind (`$walker`, `$flier`), in eine Taxonomie aufgenommen.
Es gibt prinzipiell mehrere Möglichkeiten zur Lösung dieses Dilemmas:

1. **Konsistent gebildete Hierarchie**
   Bei der Entstehung der Taxonomie wird auf die Verträglichkeit[4] der Gesichtspunkte, unter denen die Sorten gebildet werden, geachtet.

2. **Vervollständigen**
   Die Hierarchie wird nachträglich vervollständigt, d.h. es werden zusätzliche Sorten an mehrdeutigen Stellen eingefügt. Hier z.B. kann man die Sorte `$walker-or-flier` unter `$walker`, `$flier` und über `$special-mammal`, `$normal-bird` einfügen.

3. **Lazy GLB-Evaluation**
   Eine Konjunktion von Sorten (hier realisiert durch Unifikation) wird nicht sofort ausgeführt („lazy"), sondern die beteiligten Sorten werden z.B. als endliche Menge mitgeführt und erst bei Anwendung auf Individuen ausgewertet.

Ist für zwei beliebige Sorten aus der Taxonomie kein GLB definiert, könnte man dies auch durch Einführen einer leeren Sorte als GLB ausdrücken. So wird gewährleistet, daß genau ein GLB für zwei beliebige Sorten definiert wurde (unterer Halbverband/ inf-Halbverband/ meet-semi-lattice).

[4]Die Frage, welche Gesichtspunkte verträglich sind, ist eng mit der Frage, welche Prädikate überhaupt zu einer Sorte erhoben werden dürfen, verbunden.

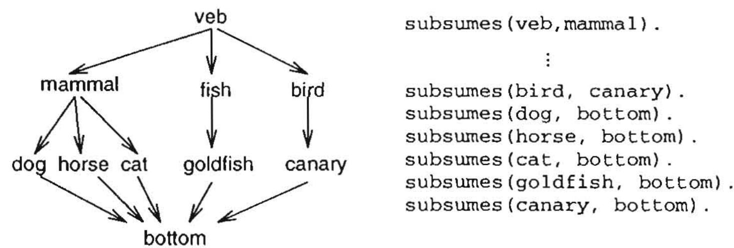


Abbildung 3.2: Unterer Halbverband

3.2 Bedingungen an die Instantiierung der Sorten

Da Sorten hier als *logische* Sprachelemente implementiert werden sollen, ist es nötig,

- **Striktheit/ Bewohntheit**

zu fordern, d.h. jede Sorte beschreibt mindestens ein Individuum [5,6]. Es gibt auch Ansätze [Wei94] in denen Sorten nicht-strikt sein müssen. Dabei wird die sortierte Unifikation um einen Checker erweitert, der testet, ob das Ergebnis einer Unifikation zweier Sorten die leere Sorte ist.

Weiter sollte die Instantiierung die

- **Vollständigkeit**[7]

erfüllen, d.h. der intensionale Schnitt zweier Sorten (GLB) ist gleich dem extensionalen Schnitt, das ist der Schnitt über der Menge von Instanzen, die durch die Sorte repräsentiert werden. Je nach Instantiierung der Sorten kann die Kommutativität von Unifikationskonjunktionen verletzt werden.

Der intensionale Schnitt in Abbildung 3.3 der Sorten `$a` und `$b` liefert die Sorte `$c`. Betrachtet man den extensionalen Schnitt von `$a` und `$b`, erhält man die Menge {2,3}, die der Instantiierung {3} der Sorte `$c` widerspricht. Benutzt man den benutzerdefinierten GLB `$c` dennoch fälschlicherweise als GLB, so liefern folgende Konjunktionen von Literalen eine Kommutativitätsverletzung bei einer Links-Rechts-Abarbeitung:

- `X is $a, X is $b, X is 2` → Unifikation scheitert.
- `X is $a, X is 2, X is $b` → Unifikation erfolgreich.

[5] Denn sonst könnte man das *extralogische* Prädikat `var` aus PROLOG realisieren.`var(X) :- X is $a.` wobei die Sorte `$a` nicht-strikt (unbewohnt) ist. Falls die Variable `X` an irgendeinen (nicht variablen) Wert gebunden ist (inklusive `$a`), wird ein FAIL ausgelöst; ist die Dereferenzierung von `X` frei, ist die Regel erfolgreich.

[6] Erlaubt man nicht-strikte Sorten, kann nicht garantiert werden, daß das Herbrand-Universum nicht leer ist. Das führt zu einem Modellierungsfehler [CGH94].

[7] Vollständigkeit wird hier i.S.v. „Intensional-Extensionale Schnittgleichheit" verwendet.

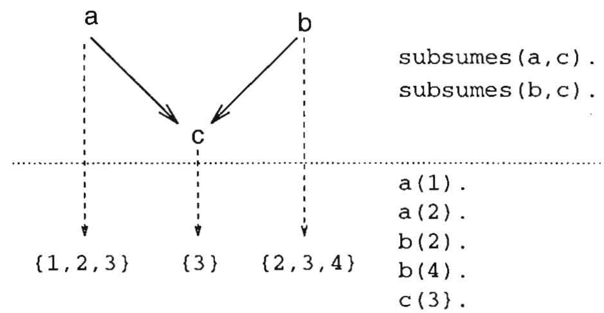


Abbildung 3.3: Unvollständige Taxonomie

Man beachte, daß durch die Voraussetzung der Vollständigkeit auch die Bedingung der Eindeutigkeit gewährleistet ist, da der extensionale Schnitt eindeutig ist.

All diese Voraussetzungen sind bei einer **sauberen** Modellierung automatisch erfüllt, könnten aber bei inkrementellem Aufbau der Taxonomie entstehen. Unbedingt erfüllt sein müssen aus logischer und semantischer Sicht die Restriktionen Striktheit und Zyklenfreiheit. In Spezialfällen könnte man eventuell auf Vollständigkeit und Eindeutigkeit verzichten. Es ist vorstellbar, daß der definierte GLB aus semantischen Gründen nicht unbedingt gleich dem extensionalen Schnitt sein muß, nur sollten sich Benutzer und Entwickler über die Folgen im klaren sein. Ist die Eindeutigkeit verletzt, es gibt also „mehrere GLB's“, wird bei der hier vorgestellten Implementierung, der zuerst gefundene GLB zurückgeliefert. Welcher GLB zuerst gefunden wird, ist jedoch von der Reihenfolge der taxonomiebeschreibenden Regeln oder entsprechenden `subsumes`-Fakten abhängig.

Kapitel 4

Dynamische GLB-Berechnung

4.1 Modell_1: Gemeinsame Sorten-/Operatorbasis mit subsumes-Prädikat

In diesem ersten Modell existiert nur eine Partition, in der das gesamte Wissen gesammelt ist. Wie bereits erwähnt, ist es sinnvoll, das Sortenwissen vom Behauptungswissen abzugrenzen[BBDV91].
Beispiel:

WB:
```
veb(X) :- mammal(X).
mammal(X) :- dog(X).
veb(X) :- fish(X).
fish(X) :- goldfish(X).
⋮
mammal(X) :- furry(X).
furry(X) :- hair_all_over(X).
hair_all_over(X) :- hair_on_head(X), hair_on_body(X).
hair_on_head(john).
⋮
```

Zeichnet man die taxonomiebeschreibenden Regeln nicht aus, sondern betrachtet alle einprämissigen DATALOG-Regeln über unäre Prädikate als solche, wird das Prädikat `furry` als Untersorte der Sorte `$mammal` verstanden. Die Berechnung der Anfrage `$mammal is $fish` (entsprechend `glb(mammal,fish)`) wächst dann kombinatorisch. Es wird nicht nur versucht, `glb(dog,fish)` zu berechnen, sondern zusätzlich noch die Anfragen `glb(furry,fish)`, `glb(hair_all_over,fish)`, ..., bevor schließlich ein FAIL ausgelöst wird. Diese Trennung des Sortenwissens vom Behauptungswissen soll hier durch das spezielle second-order Prädikat subsumes erreicht werden.

WB:
```
subsumes(veb, mammal).
subsumes(mammal, dog).
subsumes(veb, fish).
subsumes(fish, goldfish).
⋮
mammal(X) :- furry(X).
furry(X) :- hair_all_over(X).
hair_all_over(X) :- hair_on_head(X), hair_on_body(X).
hair_on_head(john).
⋮
```

Durch eine Abgrenzung des Sortenwissens vom Behauptungswissen wird die Komplexität der Anfragebearbeitung verringert. Jetzt scheitert die gleiche Anfrage bereits nach wenigen Rechenschritten.

Die GLB-Berechnung

Der GLB wird in diesem Modell zur Laufzeit berechnet. Die GLB-Berechnung `greatest-lower-bound(` $Sort_1$`,` $Sort_2$`)` wird auf eine CLB-Berechnung (Berechnung aller gemeinsamer Untersorten von $sort_1$ und $sort_2$ - Common Lower-Bounds) abgestützt [AKBLN89].

```
greatest-lower-bound(X, Y) :- & remove-subsumed-lbs(all-lbs(X,Y)).

lb(X, X) :- & X.
lb(X, Y) :- subsumes(X, Z) & lb(Z, Y).
lb(X, Y) :- subsumes(Y, Z) & lb(X, Z).

all-lbs(X, Y) :- & remove-duplicates(tupof(lb(X,Y))).

remove-subsumed-lbs(Lbs) :- & rsl(Lbs, []).

rsl([], Rlbs) :- & Rlbs.
rsl([Lb | Lb-rest], Rlbs) :-
  & rsl(rsl1(Lb, Lb-rest), tup(Lb | rsl1(Lb, Rlbs))).

rsl1(Lb, []) :- & [].
rsl1(Lb, [Lb1 | Rest]) :- subsumes+(Lb, Lb1) ! & rsl1(Lb, Rest).
rsl1(Lb, [Lb1 | Rest]) :- & tup(Lb1 | rsl1(Lb, Rest)).

subsumes+(X,Y) :- subsumes(X,Y).
subsumes+(X,Y) :- subsumes(X,Z), subsumes+(Z,Y).
```

Bei der CLB-Berechnung `lb(`$Sort_1$`,` $Sort_2$`)` wird mittels der `subsumes`-Fakten ein Weg zwischen $sort_1$ und $sort_2$ gesucht. Da die `subsumes`-Relation nicht kommutativ, sondern gerichtet ist (siehe Seite 2), existieren zwei Regeln `lb(X, Y)`, um

diesen Weg zu finden. Die erste Regel scheitert, wenn das erste Argument an eine Sorte ohne Untersorten gebunden ist; erst dann wird die zweite Regel benutzt. Findet auch die zweite Regel keine gemeinsame Sorte, wird per Backtracking weiter nach einem Weg gesucht. Die meistens CLB's sind mehrfach herleitbar, deshalb werden die mehrfach auftauchenden CLB's in `all-lbs` mit `remove-duplicates` aus der Liste (tupof) der `lb`-Werte entfernt.

Nachdem so alle CLB's berechnet wurden, wird aus diesen der größte/ allgemeinste ausgewählt. Die Funktion `remove-subsumed-lbs` entfernt aus der Liste aller CLB's diejenigen CLB's, die von einem anderen CLB subsumiert werden. Dazu löscht `rsl` mit `rsl1` einen CLB aus der Liste aller CLB's (Rest des 1. Arguments), wenn er von dem aktuell betrachteten CLB (Kopf vom 1. Arguments) subsumiert (transitiv: `subsumes+`) wird. Im 2. Argument von `rsl` befinden sich die „Subsumierer". Es bleibt noch zu prüfen ob der aktuell betrachtete CLB einen dieser Subsumierer subsumiert (wiederum mit `rsl1`), gegebenfalls wird ein solch subsumierter CLB gelöscht.

Wenn die Taxonomie eindeutig und vollständig ist, werden so alle CLB's, die ungleich dem GLB[1] sind, entfernt.

```
constant-in-sort(Const, Sort):- Sort(Const) &Const.
constant-in-sort(Const, Sort):- subsumes(Sort, Subsort),
                                &constant-in-sort(Const, Subsort).
```

Mit der Funktion `constant-in-sort` wird geprüft, ob eine Konstante direkt oder indirekt in einer Sorte enthalten ist. Dazu muß gegebenenfalls die Untersorte gefunden werden, für die das Individuum definiert wurde.

Analog wird der Test, ob eine Domäne in einer Sorte liegt, durch mehrere `constant-in-sort`-Anfragen realisiert.

Bei der Unifikation mit mindestens einer Sorte wird dann vom RELFUN-Interpreter aus die entsprechende Anfrage in RELFUN selbst gestellt (siehe Anhang A.2).

4.2 Modell_2: Getrennte Sorten-/Operatorbasis und dynamischer GLB

Im zweiten Modell wird neben der Partition `operatorbase` für das Behauptungswissen eine weitere Partition `sortbase` für das assertionale und taxonomische Wissen bereitgestellt. Dazu werden folgende RELFUN-Kommandos benötigt (siehe Anhang B).

- `consult-sortbase` zum Laden einer Sortbase
- `sortbase` zum Betrachten der Sortbase

[1] Im Falle einer nicht-eindeutigen Taxonomie wird die Liste „aller GLB's" zurückgegeben.

- destroy-sortbase zum Löschen der Sortbase
- browse-sortbase bei Vorhandensein einer graphischen Oberfläche wird ein Sortbrowser gestartet

Die Idee der GLB-Berechnung bleibt aus Modell_1 erhalten. Behält man die subsumes-Notation trotz des spezifischen Sortenmoduls bei, ändern sich auch der Algorithmus zur GLB-Berechnung und der Test, ob eine Konstante als Individum der Sorte definiert wurde, nicht.

Es wird im Unterschied zu Modell_1 bei der Unifikation von Sorten lediglich die sortbase als Suchraum benötigt (siehe Anhang A.3).

Da die subsumes-Fakten nun nicht mehr für die Trennung der sortenbeschreibenden Regeln und restlichen Hornregeln (und somit auch nicht mehr für die GLB-Berechnung) notwendig sind, kann man mit dem RELFUN-Kommmando unsubsumes die subsumes-Fakten in entsprechende Hornregeln übersetzen und somit alle Sorten auch extensional als Prädikate verwenden (siehe Anhang A.4.3). Deshalb muß der Suchraum für Prädikate um die Sortbase erweitert werden (siehe Abbildung 4.1).

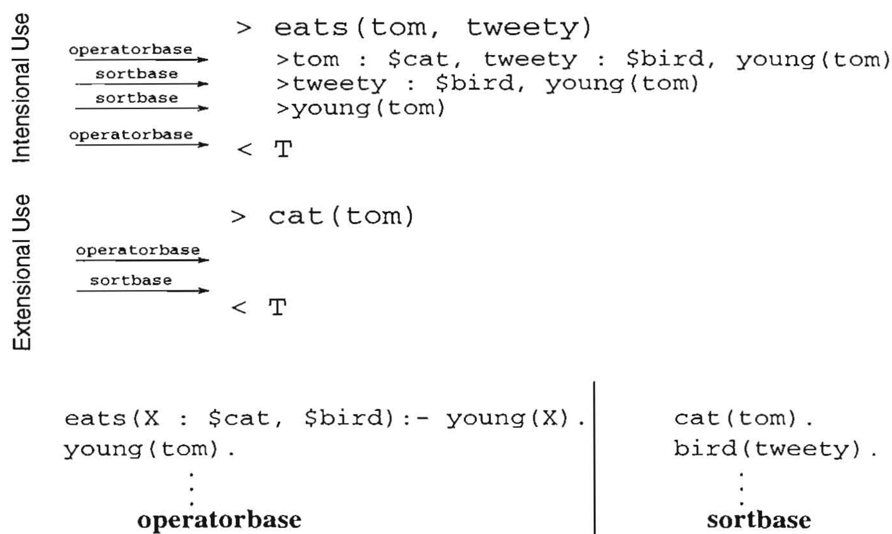


Abbildung 4.1: Bei der Unifikation von Sorten wird nur die Sortbase als Suchraum betrachtet, ansonsten die operatorbase und die sortbase

Wird das taxonomische Wissen mit einprämissigen Hornregeln notiert, ändern sich die Algorithmen nur insofern, als der GLB die Untersorten nicht mehr mittels der subsumes-Fakten, sondern mit dem RELFUN clause-Builtin[2] daß für die entsprechenden Regeln findet. Auch hier wird bei der Unifikation von Sorten natürlich der Suchraum auf die Sortbase eingeschränkt.

[2]Das clause-Prädikat ist ein second-order Prädikat. Hier: clause(P(X) :- Q(X).) = t $\Leftrightarrow$ P(X) :- Q(X) oder '(hn (_p _x) (_q _x))) in der betrachteten Partition existiert

Kapitel 5

Statische GLB-Berechnung

Bei der statischen GLB-Berechnung wird das Sortenwissen vorcompiliert, d.h. bei Benutzung des Sortenwissens greift man auf eine effiziente, fixe, interne Datenstruktur zu. Dadurch können auch verletzte Restriktionen wie etwa Vollständigkeit und Eindeutigkeit, (siehe Kapitel 3) erkannt werden und der Benutzer bzw. der Entwickler kann gewarnt werden.

5.1 Horizontale Vorcompilation

Unabhänig davon wie das Sortenwissen in der Sortbase notiert wurde, ob mit dem second-order Prädikat `subsumes` oder durch Hornregeln, wird dieses, in mehreren Schritten, in folgende interne Datenstruktur, eine Liste von Sortenlisten[1], überführt:

```
((sorte_1 (SUBSUMES ..) (INDIVIDUALS ..)
                                (SUBSUMES* ..) (INDIVIDUALS* ..))
   .
   .
   .
 (sorte_n (SUBSUMES ..) (INDIVIDUALS ..)
                               (SUBSUMES* ..) (INDIVIDUALS* ..)) )
```

Eine Sortenlisten beschreiben *eine* Sorte vollständig.

Die `SUBSUMES`-Liste enthält alle direkt definierten Untersorten der Sorte. Für alle Terminalsorten ist sie bis auf das Schlüsselwort `SUBSUMES` leer. In der nächsten Liste (`INDIVIDUALS`-Liste) befinden sich alle für diese Sorte direkt definierten Individuen. Auch diese Liste kann bis auf das Schlüsselwort `INDIVIDUALS` leer sein. In der Sortenhierarchie bestehen Abhängigkeiten unter den Sorten bzgl. der Instanzierung, daß soll heißen das z.B. `$veb` seine Individuen nur in Abhängigkeit von den Individuen der Sorte `$mammal` bestimmen kann. Eine solche Abhängigkeitschicht der Taxonomie

[1]Sortenliste: (sortenname (SUBSUMES ..) (INDIVIDUALS ..) (SUBSUMES* ..) (INDIVIDUALS* ..))

kann man als „Stratum“ sehen. Unter der stratifizierten Liste der Taxonomie soll hier eine Liste der Sorten, deren Reihenfolge die Abhängigkeiten der Sorte wiederspiegelt, verstanden werden.

- `(dog horse cat goldfish canary mammal fish bird pet)` ist die stratifizierten Liste der `vertebrate`-Taxonomie

Aus der `SUBSUMES-/ INDIVIDUALS`-Liste und der „stratifizierten“ Liste der Taxonomie wird die reflexive, transitive Hülle bezüglich der Untersorten und Individuen (`SUBSUMES*-/INDIVIDUALS*`-Liste) gebildet. Dabei wird die `SUBSUMES*`-Liste stratifiziert.

Der Algorithmus

1. Schritt

Zuerst wird aus der Partition mit Sortenwissen `sortbase` für jede Sorte eine reduzierte Sortenliste erstellt.

```
((sorte_1 (SUBSUMES ..) (INDIVIDUALS ..))
  .
  .
  .
 (sorte_n (SUBSUMES ..) (INDIVIDUALS ..)) )
```

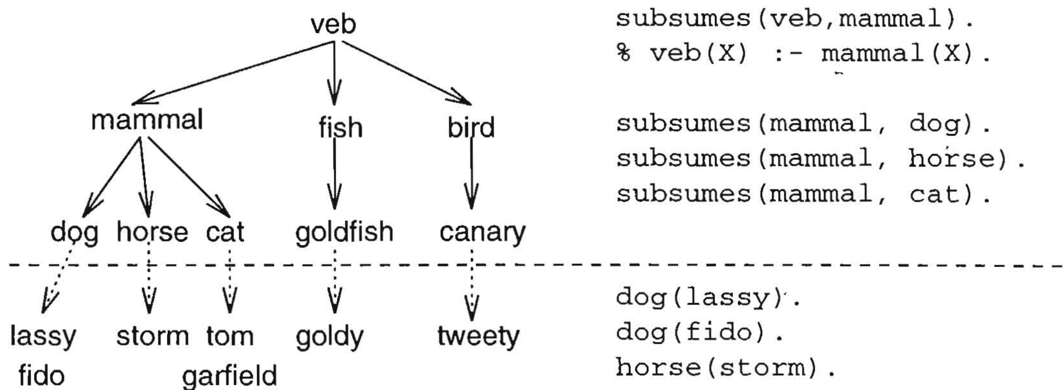
Dabei werden alle mehrfach aufgezählten Individuen und Untersorten nur einmal in die entsprechende Liste aufgenommen.

2. Schritt

Zunächst werden die reduzierten Sortenlisten um die `SUBSUMES*` - und `INDIVIDUALS*`-Listen erweitert. Die Sternlisten werden mit den `SUBSUMES-` bzw. `INDIVIDUALS`-Listen initialisiert und dann schrittweise um die indirekt definierten Untersorten bzw. Individuen ergänzt. Dabei ist die Reihenfolge, in der die Sortenlisten vervollständigt werden, von Bedeutung. Durch eine Abarbeitung der Sorten, schichtenweise der Taxonomie entsprechend, von unten nach oben, müssen die fehlenden Untersorten und Individuen nicht per Tiefensuche ermittelt werden, sondern können direkt aus den entsprechenden Stern-Listen übernommen werden. Zuletzt werden die `SUBSUMES*`-Listen aller Sorten noch entsprechend der stratifizierten Liste der Taxonomie sortiert.

Es wird also für jede Sorte die stratifizierte, reflexive und transitive Hülle bezüglich der Subsorten (`SUBSUMES*`) und die reflexive, transitive Hülle bezüglich der Individuen (`INDIVIDUALS*`) gebildet (siehe Anhang A.4.1).

Beispiel:



Das Ergebnis der Vorcompilation ist die folgende Datenstruktur:

```
( (VEB (SUBSUMES MAMMAL FISH BIRD)
      (INDIVIDUALS)
      (SUBSUMES* VEB BIRD FISH MAMMAL CANARY GOLDFISH CAT HORSE DOG)
      (INDIVIDUALS* LASSY FIDO FURY TOM GARFIELD GOLDY TWEETY))
  (MAMMAL (SUBSUMES DOG HORSE CAT)
          (INDIVIDUALS)
          (SUBSUMES* MAMMAL CAT HORSE DOG)
          (INDIVIDUALS* LASSY FIDO FURY TOM GARFIELD))
  (DOG (SUBSUMES) (INDIVIDUALS LASSY FIDO)
      (SUBSUMES* DOG) (INDIVIDUALS* LASSY FIDO))
  (HORSE (SUBSUMES) (INDIVIDUALS FURY)
         (SUBSUMES* HORSE) (INDIVIDUALS* FURY))
  (CAT (SUBSUMES) (INDIVIDUALS TOM GARFIELD)
      (SUBSUMES* CAT) (INDIVIDUALS* TOM GARFIELD))
  (FISH (SUBSUMES GOLDFISH) (INDIVIDUALS)
       (SUBSUMES* FISH GOLDFISH) (INDIVIDUALS* GOLDY))
  (GOLDFISH (SUBSUMES) (INDIVIDUALS GOLDY)
            (SUBSUMES* GOLDFISH) (INDIVIDUALS* GOLDY))
  (BIRD (SUBSUMES CANARY) (INDIVIDUALS)
       (SUBSUMES* BIRD CANARY) (INDIVIDUALS* TWEETY))
  (CANARY (SUBSUMES) (INDIVIDUALS TWEETY)
          (SUBSUMES* CANARY) (INDIVIDUALS* TWEETY)))
```

Diese Vorcompilation scheitert mit einer Fehlermeldung, wenn die Taxonomie Zyklen enthält. Sind die restlichen Voraussetzungen nicht erfüllt, terminiert die Vorcompilation ohne Fehlermeldung. Jedoch kann durch einfache Tests die Gültigkeit der Restriktionen nachgewiesen werden. Man kann mit unvollständigen oder nicht eindeutigen Taxonomien weiterarbeiten. Allerdings werden dabei unvollständige Taxonomien nicht vervollständigt und die GLB-Berechnung wählt den textuell zuerst gefundenen GLB, wenn die Taxonomie mehrdeutig ist.

Die GLB-Berechnung ist nun auf einen einfachen Links-Rechts-Vergleich der jeweiligen SUBSUMES*-Listen reduziert worden, da beim Vorcompilieren die SUBSUMES*-Listen stratifiziert bzw. entsprechend der stratifizierten Liste der Taxonomie geordnet wurden. Dadurch ist der GLB zweier Sorten in einer eindeutigen Taxonomie genau das erste gemeinsame Element der entsprechenden SUBSUMES*-Listen. Liegt eine nicht eindeutige Taxonomie vor, wird die Sorte als GLB zurückgeliefert, die textuell zuerst in der SUBSUMES*-Liste vorkommt. Wenn eine der Eingabesorten unter der anderen liegt, ist sie selbst der GLB; daher ist es in den SUBSUMES*-Listen notwendig, Sorten reflexiv als ihre eigenen Untersorten aufzuführen.
Beispiel, daß der GLB das erste gemeinsame Element der SUBSUMES*-Listen ist:

- X is $veb, X is $dog
  (VEB ... (SUBSUMES* VEB BIRD FISH MAMMAL CANARY GOLDFISH CAT HORSE DOG))
  (DOG ... (SUBSUMES* DOG))
  → TRUE, X = $dog

- X is $dog, X is $cat
  (DOG ... (SUBSUMES* DOG))
  (CAT ... (SUBSUMES* CAT)
  → FAIL

Genauso ist der Test, ob ein Individuum bzw. eine endliche Domäne zu einer Sorte gehört, auf einen bzw. mehrere Tests, die prüfen, ob das Individuum in der INDIVIDUALS*-Liste enthalten ist, vereinfacht worden.
Beispiel:

- X is lassy, X is $veb
  (VEB ... (INDIVIDUALS* LASSY FIDO FURY TOM GARFIELD GOLDY TWEETY))
  → TRUE, X = LASSY

- X is lassy, X is $cat
  (CAT ... (INDIVIDUALS* TOM GARFIELD))
  → FAIL

- X is dom[fido,goldy], X is $veb
  (VEB ... (INDIVIDUALS* LASSY FIDO FURY TOM GARFIELD GOLDY TWEETY))
  → TRUE, da in(fido,veb) und in(goldy,veb) X = dom[fido,goldy]

- X is dom[goldy],X is $dog
  (DOG ... (INDIVIDUALS* LASSY FIDO))
  → FAIL, da in(goldy,dog) gescheitert

5.2 Modell_3: Getrennte Sorten-/Operatorbasis und statischer GLB

Bei Modell_3 handelt es sich um eine Erweiterung von Modell_2. Anstelle der dynamischen GLB-Berechnung tritt eine statische GLB-Berechnung[2], wie oben beschrieben. Dazu wurden folgende RELFUN-Kommandos (siehe Anhang A.4.2) eingeführt:

- `compile-sortbase` erzeugt aus der Sortenpartition (`sortbase`)[3] eine interne Datenstruktur (siehe oben) mit direkten/indirekten Untersorten und Individuen, wobei die Sortbase selbst nicht modifiziert wird. Hierbei werden Zykel erkannt.

- `unique-sortbase` bildet für alle Sortenpaare den intensionalen Schnitt und liefert eine Fehlermeldung, falls ein Schnitt mehr als eine Sorte enthält.

- `complete-sortbase` vergleicht jeden extensionalen Schnitte zweier Sorten mit der Instantiierung des entsprechenden intensionalen Schnittes. Auch hier wird eine Fehlermeldung erzeugt, falls eine Abweichung festgestellt wird.

- `strict-sortbase` liefert eine Fehlermeldung, wenn eine `INDIVIDUALS*`-Liste leer ist.

Dieses Modell ist das effizienteste der vorgestellten Modelle, da die GLB-Berechnung und der Test, ob eine Konstante als Instanz einer Sorte definiert wurde, auf den Vergleich von Listen reduziert wurden und nicht mehr wie in den anderen Modellen zur Laufzeit berechnet werden müssen. Allerdings ist es auch das unflexibelste Modell. Möchte man das Sortenwissen verändern, z. B. eine Sorte oder ein Individuum in das Sortenwissen aufnehmen oder streichen, muß das gesamte Sortenwissen neu vorcompiliert werden[4]. Wenn man aber von „fast“ starrem Sortenwissen ausgeht, wie es im Normalfall vorzufinden ist, kann man diesen Nachteil zu Gunsten der Effizienzsteigerung (von ca. einem Faktor 125 im best-case) akzeptieren.

Von dem um Sorten erweiterten WAM wird ebenfalls auf die hier erzeugte, interne Datenstruktur zugegriffen. Durch diese Erweiterung der WAM verhalten sich Sorten im Interpreter genau so wie im Emulator. Dieses Thema wird in einer am DFKI in Arbeit befindlichen Diplomarbeit behandelt.

---

[2]Ebenso ist der Test, ob eine Konstante als Individuum einer Sorte definiert wurde, statisch.

[3]Dabei ist die gewählte Notation, also Hornregeln oder Subsumes-Fakten, unwichtig.

[4]Die momentane Implementierung erlaubt keine Änderung in der compilierten Sortbase-Datenstruktur. Dies ist durch intelligente Einfüge-/ Löschfunktion, die die interne vorcompilierte Datenstruktur verändern, erreichbar. Dazu sind die Listen `SUBSUMES` und `INDIVIDUALS` nötig; deshalb werden sie schon jetzt in der Sortbase-Datenstruktur mitgeführt.

Kapitel 6

Ausblick

In den drei vorgestellten Modellen wurden unendliche Sorten vernachlässigt. Ihre Integration ist aber in allen Fällen durch kleine Modifikationen möglich.

Das Problem von Modell_1 und Modell_2 besteht darin, das Behauptungswissen vom Sortenwissen abzugrenzen, damit eine effiziente Benutzung der Sorten erreicht werden kann. Erlaubt man nun auch unendliche Sorten, die durch Regeln beschrieben werden, kann man auch diese zusätzlichen, sortenbeschreibenden Regeln vom Behauptungswissen trennen.

Beispiel:

```
$even
```

Die Sorte der geraden, ganzen Zahlen wird beschrieben durch:

```
even(X : $integer) :- Y is div(X, 2), Y is $integer.
```

Will man eine Auszeichnung der unendlichen Sorten erreichen, sollte das in Modell_1 durch Einführung eines weiteren ausgezeichneten second-order Prädikates zum bisherigen `subsumes(Sorte, Sorte)` für die Taxonomiebeschreibung und in Modell_2 durch weitere Partitionierung der Sortbase (in z.B. tbase [taxonomisches Wissen] und abase [assertionales Wissen]) erfolgen. In Modell_3 muß man zusätzlich für unendlichen Sorten die internen Datenstruktur erweitern.

Beispiel:

```
((sorte1 (SUBSUMES ..) (EXT-INDIVIDUALS ..) (INT-INDIVIDUALS ..)
        (SUBSUMES* ..) (EXT-INDIVIDUALS* ..) (INT-INDIVIDUALS* ..))
 (sorte2  ...)
  .  .  .)
```

Dabei sind in der `EXT-INDIVIDUALS`-Liste (extensional beschriebene Instanzen) wie bisher die direkt angegebenen Individuen und in der `INT-INDIVIDUALS`-Liste (intensional beschriebene Instanzen) die neuen, durch Regeln beschriebenen Individuen bzw. die Prädikatnamen der beschreibenden Regeln aufgelistet. Analog zu der bisherigen Datenstruktur ist dann die `EXT-INDIVIDUALS*`-Liste bzw. `INT-INDIVIDUALS*`-Liste die transitive, reflexive Hülle zu `EXT-INDIVIDUALS` bzw. `INT-INDIVIDUALS`.

Man beachte, daß eine Taxonomie mit unendlichen Sorten nicht auf Vollständigkeit überprüft werden kann. Die GLB-Berechnung bliebe unverändert und der Test, ob ein Individuum zu einer Sorte gehört, müßte um den Test, ob das Individuum zu einer unendlichen Sorte gehört, erweitert werden. Es müßte also das Individuum auf das Prädikat, das die unendliche Sorte beschreibt, angewandt werden.
Eine Aufgabe der Exploration im Sinne von VEGA[1] könnte darin bestehen, aus extensionalen Beschreibungen die zugehörige Sorte (falls existent) zu finden (z.B. durch nicht verallgemeinernde Antiunifikation). In der jetzigen Implementierung der Sorten wird z.B. die Unifikation einer endlichen Domäne mit einer Sorte auf Basis der Individuen durchgeführt. Das Ergebnis einer solchen Unifikation ist natürlich extensional. Wünschenswert ist es aber, eine intensionale Beschreibung zu erhalten, falls sie existiert.

- `X is $dog, X is dom[lassy, fido]`
  $\rightarrow$ `TRUE, X = dom[lassy, fido]`
  Ebenfalls richtig und abstrakter wäre `X = $dog`

Die hier verwendeten Sorten sind lediglich eine strukturierte Zusammenfassung von Individuen, wie sie z.B. ein KL-ONE-Classifier liefert, dabei wird auf eine vorausgehende Beschreibung der Eigenschaften der Sorten verzichtet. Beschreibt man Sorten nicht extensional, wie bisher, sondern intensional durch die charakteristischen Attribute erhält man rekord-ähnliche Datenstrukturen (Feature-Typen)[Mey]. Zu überlegen ist, ob die Sorten zu Feature-Typen erweitert werden sollen.Für solche Typen, die nicht nur durch die Sortenkonstante (Namen), sondern zusätzlich durch Eigenschaften definiert werden, werden eine Deklaration und ein Typ-Checker notwendig, die bisher bei den einfachen Sorten nicht benötigt wurden. Vorteilhaft ist die Zuordnung von Eigenschaften, die nicht mehr extensional jedem Objekt, sondern auch intensional der Sorte zugeordnet werden können. Wird eine Eigenschaft für einen Typ definiert, wird diese Eigenschaft an alle „subsumierten " Typen vererbt, ohne daß sie für diese „subsumierten " Typen definiert sein muß.

Weiter ist zu überlegen, ob die hier beschriebene Taxonomie zu einer Terminologie, wie sie durch KL-ONE-artige Sprachen realisiert wird, erweitert werden soll [Ric89] [BBH+90]. Dazu gehört u.a. :

- **Erweiterung der Verknüpfungen von Sorten.** Zusätzlich zu der bisher verwendeten Subsumtions-Relation und der Konjunktion (GLB), gibt es z.B. Disjunktion und Negation (systemdefinierte Relationen) und Rollen (benutzerdefinierte binäre Relationen).

- **Erweiterung von Sorten zu Konzepten.** Konzepten liegen eine intensionale Definition und eine interne Datenstruktur zugrunde (beschrieben u.a. durch Rollen). Also genügt für Konzepte, anders als bei Sorten, nicht mehr die Beschreibung durch ein Symbol.

[1] Das VEGA-Projekt behandelt die Validierung und Exploration von Wissensbanken durch globale Analyse

Literaturverzeichnis

[AK88] Hassan Aït-Kaci. An operational overview of LIFE. *Unknown*, 1988.

[AKBLN89] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, Roger Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115 - 146, 1989.

[AKP] Hassan Aït-Kaci, Andreas Podelski. LIFE: Logic, Inheritance, Functions, Equations. Copy of slides.

[BBDV91] A. Bockmayr, C. Brozka, P. Deussen, I. Varsek. KA-PROLOG: Erweiterung einer logischen Programmiersprache und ihre effiziente Implementierung. *Informatik Forschung und Entwicklung*, S. 128 -140, Juni 1991. deutsch.

[BBH+90] Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann. Concept Logics. Technischer Bericht RR-90-10, DFKI / Kaiserslautern, P.B. 2080 , D-675 Kaiserslautern, 1990.

[BEH+93] Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, Werner Stein. RELFUN Guide: Programming with Relations and Functions Made Easy. Document D-93-12, DFKI, Juli 1993.

[Bei87] Christoph Beierle. Types, Modules and Databases in the Logic Programming Languaga PROTOS-L. In C.-R. Rollinger K. H. Blaesius, U. Hedstueck (Hrsg.), *Sorts and Types in Artifical Intelligence*, 1987.

[BHHM93] Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer. COLAB: A Hybrid Knowledge Compilation Laboratory. DFKI Research Report RR-93-08, DFKI GmbH, P.O. Box 2080, 67608 Kaiserslautern, Germany, Januar 1993. Also to appear in *Annals of Operations Research*.

[Bir73] Garrett Birkhoff. *Lattice Theory*, Band XXXV von *Colloquim Publications*. American Mathematical Society, Providence, Rhode Island, 3 edition, 1973.

[Bol94] Harold Boley. Finite Domains and Exclusions as First-Class Citizens. In Roy Dyckhoff (Hrsg.), *Proceedings of the 4th International Workshop on Extensions of Logic Programming, ELP '93, St. Andrews, Scotland, 1993*, Band 798 von *LNAI*. Springer, 1994.

[BS85] R. J. Brachman, J. G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–216, 1985.

[Buh94] Ulrich Buhrmann. Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien. Document D-94-06, DFKI Kaiserslautern, Postfach 3049, 67608 Kaiserslautern, April 1994.

[CGH94] Armin B. Cremers, Ulrike Griefahn, Ralf Hinze. *Deduktive Datenbanken – Eine Einführung aus der Sicht der logischen Programmierung.* Friedr. Vieweg & Sohn Verlagsgesellschaft, Braunschweig/Wiesbaden, 1994.

[GR85] Adele Goldberg, David Robinson. *SMALLTALK-80 The Language And It's Implementation.* Addison-Wesley Publishing Company, 1985.

[Kau71] Arnold Kaufmann. *Einführung in die Graphentheorie.* Orientierung und Entscheidung. Oldenbourg Verlag, 1971.

[Mey] G. Meyer. Polymorphic Feature Types. Kapitel 6, S. 169–203.

[Mey94] Gregor Meyer. *KI-94 Workshops*, S. 317. J. Kunze, H. Stoyan, 1994.

[NG94] Pierdaniele Giaretta Nicola Guarino, Massimilliano Carrara. An Ontology of the Meta-Level Categories. In Pietro Torasso Jon Doyle, Erik Sandwall (Hrsg.), *Principles of Knowledge Representation and Reasoning*, S. 270–280, 1994.

[Ric89] Michael M. Richter. *Prinzipien der Künstlichen Intelligenz.* Teubner Verlag, 1989. In German.

[Sti83] Mark E. Stickel. Theory Resolution: Building in Nonequational Theories. In *Proceedings of the Third National Conference on Artificial Intelligence*, S. 391–397. AAAI, August 1983.

[Tsa94] Panagiotis Tsarchopoulos. Type constraints: a practical perspective. In *Workshop on Declarative Programming and Specification*, Bad Honnef, Germany, May 1994.

[Wal84] Christoph Walther. A mechanical solution of Schubert's Steamroller by many-sorted resolution. Technischer Bericht A31-84, Universität Karlsruhe, Institut für Informatik I, P.B.6380, D-7500 Karlsruhe, 1984.

[Wei94] Christoph Weidenbach. Unification in Sort Theories and its Applications. Technischer Bericht MPI-I-94-211, Max-Plank-Institut für Informatik, March 1994.

[Wir85] Nikolaus Wirth. *Programmieren in Modula2*. Springer Verlag, 1985.

Anhang A

Unifikation

A.1 Unifikation mit „:“-Notation für getypte Variablen

Prinzipiell ist die Unifikation aller Modelle gleich. Sie unterscheidet sich nur in den Fällen in denen Sorten auftreten. Deshalb werden nur die Funktionen `dom-intersection`, `sortbase-individualsp` und `sortbase-glb`.

```
(defun typ-t (x) (and (consp x) (eq 'typ (car x))))
(defun typ-tt (x) (and (typ-t x) (= 2 (length x))))

(defun typed-expr2bnd (term)          ; (expr1 : expr2) -> (bnd expr1 expr2)
  (cond ((typed-expr-t term)
         (mk-bnd (typed-expr-term term) (typed-expr-type term)))
        (t term)))


(defun unify
       (x y environment)

  (let ((x (typed-expr2bnd (ultimate-assoc x environment)))    ; typed-expr2b
        (y (typed-expr2bnd (ultimate-assoc y environment))))   ; see above

    (cond ((or (bnd-t x) (bnd-t y))
           (unify-bnd (un-bnd x)
                      (un-bnd y)
                      (variable-if-bnd x)
                      (variable-if-bnd y)
                      environment))
          ((equal x y) environment)
```

```
        ((or (anonymous-p x)
             (anonymous-p y))
         environment)
        ((vari-t x) (cons (list x y) environment))
        ((vari-t y) (cons (list y x) environment))
        ((and (dom-t x) (exc-t y)) (and (dom-exc x y) environment))
        ((and (exc-t x) (dom-t y)) (and (dom-exc y x) environment))
        ((and (dom-t x) (dom-t y)) (and (dom-intersection x y) environment))
        ((and (exc-t x) (exc-t y)) environment)

        ((and (dom-t x) (typ-t y)) (and (dom-in-sort x (s-type y))
                                        environment))
        ((and (typ-t x) (dom-t y)) (and (dom-in-sort y (s-type x))
                                        environment))

        ((dom-t x) (and (member y (cdr x) :test #'equal) environment))
        ((dom-t y) (and (member x (cdr y) :test #'equal) environment))
        ((exc-t x) (and (not (member y (cdr x) :test #'equal)) environment))
        ((exc-t y) (and (not (member x (cdr y) :test #'equal)) environment))

        ((and (typ-t x) (atom y))
             (and (sortbase-individualsp y (s-type x)) environment))
        ((and (atom x) (typ-t y))
             (and (sortbase-individualsp x (s-type y)) environment))
        ((and (typ-t x) (typ-t y))
             (and (mk-type (sortbase-glb  (s-type x) (s-type y)))
              environment))

        ((or (atom x) (atom y)) nil)
        (t
         (let ((new-environment (unify (car x)
                                       (car y)
                                       environment)))
           (and new-environment
                (unify-args (cdr x) (cdr y) new-environment)))))))

(defun s-type (arg) (and (typ-t arg) (cadr arg)))

(defun mk-type (sort-name)
  (when sort-name
        (and (atom sort-name) (list 'typ sort-name))))
```

```
(defun unify-bnd (x y xvar yvar environment)
  (cond ((and (dom-t x) (exc-t y))
         (let ((differ (dom-exc x y)))
           (and differ (unify-bnd-env differ xvar yvar environment))))
        ((and (exc-t x) (dom-t y))
         (let ((differ (dom-exc y x)))
           (and differ (unify-bnd-env differ xvar yvar environment))))
        ((and (dom-t x) (dom-t y))
         (let ((inter (dom-intersection x y)))
           (and inter (unify-bnd-env inter xvar yvar environment))))
        ((and (exc-t x) (exc-t y))
         (unify-bnd-env (exc-union x y) xvar yvar environment))

        ((and (typ-t x) (dom-t y))
         (let ((sort-dom (dom-in-sort y (s-type x))))
              (and sort-dom (unify-bnd-env sort-dom xvar yvar environment))))
        ((and (dom-t x) (typ-t y))
         (let ((sort-dom (dom-in-sort x (s-type y))))
              (and sort-dom (unify-bnd-env sort-dom xvar yvar environment))))

        ((and (dom-t x) (not (vari-t y)))
         (and (member y (cdr x) :test #'equal)
              (unify-bnd-env y xvar yvar environment)))
        ((and (dom-t y) (not (vari-t x)))
         (and (member x (cdr y) :test #'equal)
              (unify-bnd-env x xvar yvar environment)))
        ((and (exc-t x) (not (vari-t y)))
         (and (not (member y (cdr x) :test #'equal))
              (unify-bnd-env y xvar yvar environment)))
        ((and (exc-t y) (not (vari-t x)))
         (and (not (member x (cdr y) :test #'equal))
              (unify-bnd-env x xvar yvar environment)))

        ((and (typ-t x) (typ-t y))
         (let ((sort-sort (mk-type (sortbase-glb (s-type x) (s-type y)))))
              (and sort-sort (unify-bnd-env sort-sort xvar yvar environment))))
        ((and (typ-t x) (not (vari-t y)))
         (and (sortbase-individualsp y (s-type x))
              (unify-bnd-env y xvar yvar environment)))
        ((and (typ-t y) (not (vari-t x)))
         (and (sortbase-individualsp x (s-type y))
              (unify-bnd-env x xvar yvar environment)))
```

```
        (t (let ((new (unify x y environment)))
              (and new (unify-bnd-env (if (vari-t x) x y) xvar yvar new))))
))


(defun ultimate-assoc
       (x environment)
  (cond ((vari-t x)
         (let ((binding (assoc x environment :test #'equal)))
           (cond ((null binding) x)
                 ((dom-t (cadr binding)) (mk-bnd x (cadr binding)))
                 ((exc-t (cadr binding)) (mk-bnd x (cadr binding)))

                 ((typ-t (cadr binding)) (mk-bnd x (cadr binding)))

                 (t (ultimate-assoc (cadr binding)
                                    environment)))))
        (t x)))




(defun ultimate-instant
       (x environment)
 (catch :bndclash    ; thrown at directly from (recursive) ultimate-instant-rec
  (cond ((bnd-t x)
         (let ((bndnew (unify (cadr x) (caddr x) environment)))
              (cond (bndnew (ultimate-instant-rec (cadr x) bndnew))
                    (t 'unknown))))

        ((typed-expr-t x)  ; still special treatment of (expr1 : expr2)
         (let ((bndnew (unify (car x) (caddr x) environment)))
              (cond (bndnew (ultimate-instant-rec (car x) bndnew))
                    (t 'unknown))))

        (t (ultimate-instant-rec x environment)))))


(defun ultimate-instant-rec ; could pair instantiated term with new environment
       (x environment)    ; for bnd-bnd composition, intermediate instantiation
```

```
  (cond ((vari-t x)                                   ; and throw eliminat
         (let ((binding (assoc x environment :test #'equal)))
           (cond ((null binding) x)
                 (t (ultimate-instant-rec (cadr binding)
                                          environment)))))
        ((atom x) x)
        ((bar-t x)                                  ;  (cddr x) should be nil
         (let ((barinst (ultimate-instant-rec (cadr x) environment)))
              (cond ((tup-t barinst) (cdr barinst))
                    (t (list (car x) barinst)))))   ;  (car x) is "|"
        ((bnd-t (car x))
         (let ((bndnew (unify (cadr (car x)) (caddr (car x)) environment)))
              (cond (bndnew (cons (ultimate-instant-rec (cadr (car x)) bndnew
                                  (ultimate-instant-rec (cdr x) bndnew)))
                    (t (throw :bndclash 'unknown)))))  ; escape from cons nes

        ((typed-expr-t (car x))  ; still special treatment of (expr1 : expr2)
         (let ((bndnew (unify (car (car x)) (caddr (car x)) environment)))
              (cond (bndnew (cons (ultimate-instant-rec (car (car x)) bndnew)
                                  (ultimate-instant-rec (cdr x) bndnew)))
                    (t (throw :bndclash 'unknown)))))  ; escape from cons nest

        (t
         (cons (ultimate-instant-rec (car x) environment)
               (ultimate-instant-rec (cdr x) environment)))))
```

A.2 Modell_1

Die Unifikation von Domänen und Sorten wird auf mehrere Konstanten-Sorten-Unifikationsschritte reduziert. Eine solche Konstanten-Sorten-Unifikation ruft die RELFUN-Relation `constant-in-sort` auf. Ebenso wird bei der Sorten-Sorten-Unifikation eine RELFUN-Relation `greatest-lower-bound` benutzt.

```
(defun dom-in-sort (dom sort)
  (and dom
       (mk-dom (remove nil (const-list-in-sort (cdr dom) sort)))))

(defun const-list-in-sort (const_list sort)
  (cond ((null const_list) nil)
(t (cons (sortbase-individualsp (car const_list) sort)
                  (const-list-in-sort (cdr const_list) sort)))))
```

```
(defun sortbase-individualsp (const sort)
  (let ((res (and-process  (list (list 'constant-in-sort const sort))
                           '((bottom))
                           (collect-databases)
                           1
                           'once)))
       (ultimate-instant (un-inst (car res)) (cadr res))))
```

```
(defun sortbase-glb (sort1 sort2)
  (let ((res (and-process (list (list 'greatest-lower-bound sort1 sort2))
                          '((bottom))
                          (collect-databases)
                          1
                          'once)))
       (cadr (ultimate-instant (un-inst (car res)) (cadr res)))))
```

A.3 Modell_2

Prinzipiell bleibt die Unifikation aus Modell_1 erhalten. Es wird lediglich der Suchraum bei der Unifikation von Sorten auf die sortbase eingeschränkt. (Die Unifikation einer Sorte mit einer Domäne muß nicht angepaßt werden, da sie auf die Unifkation mit einer Konstanten zurückgreift.)

```
(defun sortbase-individualsp (const sort)
  (let ((res (and-process  (list (list 'constant-in-sort const sort))
                           '((bottom))
                           (list *rfi-sortbase*)
                           1
                           'once)))
       (ultimate-instant (un-inst (car res)) (cadr res))))
```

```
(defun sortbase-individualsp (sort1 sort2)
  (let ((res (and-process (list (list 'greatest-lower-bound sort1 sort2))
                          '((bottom))
                          (list *rfi-sortbase*)
                          1
                          'once)))
       (cadr (ultimate-instant (un-inst (car res)) (cadr res)))))
```

A.4 Modell_3

Für die Unifikation im dritten Modell sind einige Vorverarbeitungen nötig. Zuerst muß die `sortbase` vorcompiliert werden A.4.1. Dann kann man die `sortbase` noch auf Vollständigkeit und Eindeutigkeit prüfen (A.4.2). Ist das Sortenwissen in `subsumes` -Notation gegeben kann es in Hornregeln überführt werden und andersrum (A.4.3). Erst nach der horizontalen Compilation kann unifiziert werden (A.4.4).

A.4.1 Vorcompilation

```
(defvar *rfi-sortbase*        nil  "taxonomic knowledge ")

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

(defun compile-sortbase (sortbase)
  (setq *subsumes-individuals* (and (compile1-sortbase sortbase)
                             (classify-subsumes* (classify-sortbase)))))


;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

(defvar *subsumes-individuals* nil)

;*************************************************************************
; Explicit relations: subsumes and individuals
;*************************************************************************

;The structure of *subsumes-individuals* after the first compilation step
;(compile1-sortbase) :
;( (sort1 (subsumes subsort11 subsort12 ...) (individuals ind11 ind12 ...))
;  (sort2 (subsumes subsort21 subsort22 ...) (individuals ind21 ind22 ...))
;  (sort3 (subsumes subsort31 subsort32 ...) (individuals ind31 ind32 ...))
;              ...
;  (sortn (subsumes subsortn1 subsortn2 ...) (individuals indn1 indn2 ...))
;)
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *


(defun compile1-sortbase (sortbase)
  (cond (sortbase
         (let ((clause (car sortbase)))
              (cond ((subsumes-t clause)
```

```
                (let ((sort (cadadr clause))
                      (subsort (caddr (cadr clause))))
                     (add-sort-and-subsort sort subsort)))
               ((horn-sort-rel-t clause)
                (let ((sort (caadr clause))
                      (subsort (caaddr clause)))
                     (add-sort-and-subsort sort subsort)))
               ((individual-assignment-t clause)
                (let ((sort (caadr clause))
                      (individual (cadadr clause)))
                     (cond ((dom-t individual)
                            (mapcar #'(lambda (dom-individual)
                                              (add-sort-and-individual
                                               sort dom-individual))
                                        (cdr individual)))
                           (t ;individual is a constant
                              (add-sort-and-individual sort
                                                       individual)))))))
        (compile1-sortbase (cdr sortbase)))
       (t *subsumes-individuals*)))

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

(defun add-sort-and-subsort (sort subsort)
  (cond ((null *subsumes-individuals*)
         (initial-subsumes-individuals1 sort subsort))
        ((member-tree sort *subsumes-individuals*)
         (assert-sub sort subsort))
        (t (assert-sort1 sort subsort))))


(defun add-sort-and-individual (sort individual)
  (cond ((null *subsumes-individuals*)
         (initial-subsumes-individuals2 sort individual))
        ((member-tree sort *subsumes-individuals*)
         (assert-individuals sort individual))
        (t (assert-sort2 sort individual))))

(defun initial-subsumes-individuals1 (sort subsort)
  (setq *subsumes-individuals*
        (list (list sort (list 'subsumes subsort) (list 'individuals))
              (list subsort (list 'subsumes) (list 'individuals)))))
```

```
(defun initial-subsumes-individuals2 (sort individual)
  (setq *subsumes-individuals*
        (list (list sort (list 'subsumes) (list 'individuals individual)))))


;Asserts a sort in the *subsumes-individuals* with the subsorts of the
;'Subsumes-List' and asserts the subsorts in the *subsumes-individuals*,
;if not existent yet
(defun assert-sort1 (sort subsort)
  (setq *subsumes-individuals*
        (append *subsumes-individuals*
                (list
                 (list sort (list 'subsumes subsort) (list 'individuals)))))
  (cond ((member-tree subsort *subsumes-individuals*) t)
        (t (assert-subsort subsort))))

;Asserts a sort in the *subsumes-individuals* with the individual in the
;'individual-list'
(defun assert-sort2 (sort individual)
  (setq *subsumes-individuals*
        (append *subsumes-individuals*
                (list
                 (list sort (list 'subsumes) (list 'individuals individual))))))


;Asserts a subsort in the 'subsumes-list', if not existent yet and
;asserts this subsort in the *subsumes-individuals*.
(defun assert-sub (sort subsort)
  (setq *subsumes-individuals*
        (mapcar #'(lambda (list)
                          (cond ((and (equal sort (car list))
                                      (not (member subsort (cadr list))))
                                 (list (car list)
                                       (append (cadr list) (list subsort))
                                       (caddr list)))
                                (t list)))
                *subsumes-individuals*))
  (cond ((member-tree subsort *subsumes-individuals*) t)
        (t (assert-subsort subsort))))

;Assert a subsort in the *subsumes-individuals*
(defun assert-subsort (subsort)
  (setq *subsumes-individuals*
```

```
      (append *subsumes-individuals*
              (list (list subsort
                          (list 'subsumes)
                          (list 'individuals))))))



;Assert an individual of the sort in the 'individual-list'
(defun assert-individuals (sort individual)
  (setq *subsumes-individuals*
        (mapcar #'(lambda (list) (cond ((equal sort (car list))
                                        (list (car list)
                                              (cadr list)
                                              (append (caddr list)
                                                      (list individual))))
                                       (t list)))
                *subsumes-individuals*)))
;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
(defun horn-sort-rel-t (clause)
    (= (length clause) 3) )

(defun individual-assignment-t (clause)
  (= (length clause) 2))

(defun member-tree (sort subsumes-individuals)
  (and subsumes-individuals
       (if (equal sort (caar subsumes-individuals)) t
           (member-tree sort (cdr subsumes-individuals)))))

;************************************************************************
; Transitive closure: subsumes* and individuals*
;************************************************************************

(defun classify-sortbase ()
  ;1. Add to the *subsumes-individuals* the 'subsumes*-list' with the
  ;   sorts of the 'subsumes-list' and the 'individual*-list'
  ;   with the individuals of the 'individual-list'
  (setq *subsumes-individuals* (extend-sub-ind-list *subsumes-individuals*))

  ;2. Add all elements from the stratified list
  ;    a. 'subsumes-list' is empty -> finished
  ;    b. 'subsumes-list' is not empty ->
  ;         add all elements from the 'subsumes-list'
```

```
  ;            (expand 'subsumes*-list' with the 'subsumes*-list'
  ;            (of each element and 'individual*-list' analogue)
  (expand-sort-list (stratify-list *subsumes-individuals*)))


(defun extend-sub-ind-list (sub-ind-list)
  (mapcar #'(lambda (sort-list)
                  (append sort-list
                          (list (cons 'subsumes* (cons (car sort-list)
                                                       (cdadr sort-list)))
                                (cons 'individuals* (cdaddr sort-list)))))
                  sub-ind-list))

;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
(defvar *stratified-list* nil)

(defun stratify-list (subsumes-list)
    (setq *stratified-list* (sort-depexpr (mk-subsumes-list subsumes-list))))

(defun mk-subsumes-list (sub-ind-list)
    (mapcar #'(lambda (x) (cons (car x) (cdadr x)))
                  sub-ind-list))


;* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
(defun expand-sort-list (stratified-list)
  (cond ((null stratified-list) *subsumes-individuals*)
        (t    (let ((sort (car stratified-list)))
                 (setq *subsumes-individuals*
                       (mapcar #'(lambda (sort-list)
                                  (cond ((sort-list-t sort-list sort)
                                         (expand-sub*-ind* sort-list))
                                        (t sort-list)))
                               *subsumes-individuals*))
                 (expand-sort-list (cdr stratified-list))))))


(defun expand-sub*-ind* (sort-list)
  (let ((sort (car sort-list))
        (sub-list (get-sub-list sort-list))
        (ind-list (get-ind-list sort-list))
        (sub*-list (get-sub*-list sort-list))
        (ind*-list (get-ind*-list sort-list)))
```

```
    (list sort sub-list ind-list
          (expand-sub* sub*-list (cdr sub-list))
          (expand-ind* ind*-list (cdr sub-list)))))


(defun expand-sub* (sub*-list sub-list)
  (cond ((null sub-list) sub*-list)
        (t (expand-sub* (append-not-twice sub*-list
                                          (get-sub*-list
                                           (get-sort-list *subsumes-individuals*
                                                          (car sub-list))))
                        (cdr sub-list)))))

(defun expand-ind* (ind*-list sub-list)
  (cond ((null sub-list) ind*-list)
        (t (expand-ind* (append-not-twice ind*-list
                                          (get-ind*-list
                                           (get-sort-list *subsumes-individuals*
                                                          (car sub-list))))
                        (cdr sub-list)))))



(defun get-sort-list (sub-ind-list sort)
  (when sub-ind-list
        (let ((sort-list (car sub-ind-list)))
             (cond ((equal (car sort-list) sort) sort-list)
                   (t (get-sort-list (cdr sub-ind-list) sort))))))


(defun get-sub*-list (sort-list)    ; returned 'lists' start with sub/ind[*] tag
  (cadddr sort-list))

(defun get-ind*-list (sort-list)
  (car (last sort-list)))

(defun get-sub-list (sort-list)
  (cadr sort-list))

(defun get-ind-list (sort-list)
  (caddr sort-list))
```

```
(defun append-not-twice (sub*-list1 sub*-list2)
  (cond ((null sub*-list2) sub*-list1)
        (t (cond ((member (car sub*-list2) sub*-list1 :test 'equal)
                  (append-not-twice sub*-list1 (cdr sub*-list2)))
                 (t (append-not-twice (append  sub*-list1
                                               (list (car sub*-list2)))
                                      (cdr sub*-list2)))))))


(defun sort-list-t (sort-list sort)
  (equal (car sort-list) sort))


;******************************************************************************


(defun classify-subsumes* (sub-ind-list)
  (mapcar #'(lambda (sort-list)
                (let ((sort (car sort-list))
                      (sub-list (get-sub-list sort-list))
                      (ind-list (get-ind-list sort-list))
                      (sub*-list (get-sub*-list sort-list))
                      (ind*-list (get-ind*-list sort-list)))
                     (list sort sub-list ind-list
                           (sort-subsumes* sub*-list
                                           *stratified-list*)
                           ind*-list)))
          sub-ind-list))


(defun sort-subsumes* (a b)
    (sort (copy-list a) #'(lambda (y x) (member y (member x b)))))
```

A.4.2 Tools zur Prüfung der Restriktionen

```
(defun all-sorts (sub-ind-list) ; select sorts from a subsumes-individuals lis
  (mapcar #'(lambda (sort-list) (car sort-list)) sub-ind-list))

(defun combine-all-sorts (list1 list2) ; pair two lists
  (cond ((null list1) nil)
        ((null list2) (combine-all-sorts (cdr list1) (cddr list1)))
```

```
        (t (cons (list (car list1)
                       (car list2))
                 (combine-all-sorts list1 (cdr list2))))))

;************************************************************************
; Check the taxonomy for completeness (intensional glb = extensional glb)
;************************************************************************

(defun complete-taxonomy ()
  (complete-taxonomy1
   (combine-all-sorts (all-sorts *subsumes-individuals*)
                      (cdr (all-sorts *subsumes-individuals*)))))

(defun complete-taxonomy1 (comb-sort-list)
  (if  comb-sort-list
       (let ((sort1 (caar comb-sort-list))
             (sort2 (cadar comb-sort-list)))
            (cond ((int-ext-intersection-equal sort1 sort2)
                   (complete-taxonomy1 (cdr comb-sort-list)))
                  (t (rf-format "Taxonomy is not complete: ~a ~a ~%"
                                sort1 sort2))))
       t))

(defun int-ext-intersection-equal (sort1 sort2)
  (let ((int-intersection (intensional-intersection sort1 sort2))
        (ext-intersection (extensional-intersection sort1 sort2)))
       (cond ((null (set-difference
                     int-intersection
                     (cdr (get-ind*-list
                           (get-sort-list
                            *subsumes-individuals* ext-intersection)))))
              t)
             (t nil))))

(defun extensional-intersection (sort1 sort2)
  (sortbase-glb sort1 sort2))

(defun intensional-intersection (sort1 sort2)
  (intersection (cdr (get-ind*-list
                      (get-sort-list *subsumes-individuals* sort1)))
                (cdr (get-ind*-list
                      (get-sort-list *subsumes-individuals* sort2)))))
```

```
;*************************************************************************
; Check the taxonomy for uniqueness (i.e., there is at most one glb defined)
;*************************************************************************

(defun unique-glb ()
 (unique-glb1
  (combine-all-sorts (all-sorts *subsumes-individuals*)
                     (cdr (all-sorts *subsumes-individuals*)))))

(defun unique-glb1 (comb-sort-list)
  (if comb-sort-list
      (let ((sort1 (caar comb-sort-list))
            (sort2 (cadar comb-sort-list)))
           (cond ((<= (length (dyn-glb sort1 sort2)) 1)
                  (unique-glb1 (cdr comb-sort-list)))
                 (t   (rf-format "Taxonomy is not well defined: ~a ~a ~%"
                                  sort1 sort2))))
      t))

(defun dyn-glb (sort1 sort2)
  (remove-subsumed-clbs (clb sort1 sort2)))

(defun clb (sort1 sort2)
   (intersection (cdr (get-sub*-list
                       (get-sort-list *subsumes-individuals* sort1)))
                 (cdr (get-sub*-list
                       (get-sort-list *subsumes-individuals* sort2)))))

(defun remove-subsumed-clbs (clbs)
  (rsl clbs nil))

(defun rsl (list1 list2)
  (cond ((null list1) list2)
        (t (let ((clb (car list1))
                 (clb-rest (cdr list1)))
                (rsl (rsl1 clb clb-rest)
                     (cons clb (rsl1 clb list2)))))))

(defun rsl1 (clb list2)
  (cond ((null list2) nil)
        ((subsumes+ clb (car list2)) (rsl1 clb (cdr list2)))
        (t (cons (car list2) (rsl1 clb (cdr list2))))))
```

```
(defun subsumes+ (sort1 sort2)
  (member sort2 (get-sub*-list
                     (get-sort-list *subsumes-individuals* sort1))))
```

A.4.3 UNSUBSUMES und RESUBSUMES

```
;*************************************************************************
; Replace subsumes facts by ordinary rules   V. Hall
;*************************************************************************

(defun unsubsumes (database)
  (mapcar #'(lambda (clause)
              (cond ((subsumes-t clause)
                     (list 'hn
                           (list (cadadr clause) '(vari x))
                           (list (cadr (cdadr clause)) '(vari x))))
                    (t clause)))
          database))

(defun subsumes-t (clause)  ; should be fact of the form (hn (subsumes p q))
  (and (hn-t clause)
       (equal (caadr clause) 'subsumes)))



;*************************************************************************
; Replace ordinary rules by subsumes facts   V. Hall
;*************************************************************************

(defun resubsumes (database)
  (mapcar #'(lambda (clause)
              (cond ((sortrule-t clause)
                     (list 'hn
                           (list 'subsumes
                                 (caadr clause)
                                 (caaddr clause))))
                    (t    clause)))
          database))


(defun sortrule-t (clause)  ; should be rule of the form (hn (p _x) (q _x))
  (and (hn-t clause)
```

```
      (= (length clause) 3)
      (= (length (cadr clause)) 2)
      (= (length (caddr clause)) 2)))
```

A.4.4 Unifikation mit :-Notation für getypte Variablen

```
(defun dom-in-sort (dom sort)
  (dom-intersection dom (sortbase-ind sort)))


(defun sortbase-ind (sort)
  (and (sortbase-sortp sort)
       (let ((ind*-list (get-ind*-list
                          (get-sort-list *subsumes-individuals* sort))))
         (cond (ind*-list (cons 'dom (cdr ind*-list)))
               (t nil)))))

(defun sortbase-sortp (sort)
  (sort-defined sort *subsumes-individuals*))

(defun sort-defined (sort list)
  (and list
       (cond ((equal sort (caar list)) t)
             (t (sort-defined sort (cdr list))))))

(defun sortbase-individualsp (const sort)
  (cond ((member const
                 (get-ind*-list (get-sort-list *subsumes-individuals* sort)))
         const)
        (t nil)))


(defun sortbase-glb (sort1 sort2)
  (let ((subsumes1 (cdr (get-sub*-list
                         (get-sort-list *subsumes-individuals* sort1))))
       (subsumes2 (cdr (get-sub*-list
                         (get-sort-list *subsumes-individuals* sort2)))))
       (glb subsumes1 subsumes2)))

(defun glb (sub-list1 sub-list2)
  (and sub-list1
```

```
(cond ((member (car sub-list1) sub-list2) (car sub-list1))
      (t (glb (cdr sub-list1) sub-list2)))))
```

Anhang B

Erweiterung um Sortbase

```
(defvar *sortstyle*          'static "static or dynamic sorts ")


(defun collect-databases ()
  "Returns a list of all databases."
  (list *rfi-sortbase* *rfi-prelude* *rfi-database* *tracebase*) )

(defun rfi-command (userline)
   .
   .
   .
          ((eq com 'consult-sortbase)
           (rfi-sortbase-cmd-consult userline))
          ((eq com 'complete-taxonomy)
           (if (eq *sortstyle* 'static)
               (complete-taxonomy)
               (rf-error "- running dynamic sort model")))
          ((eq com 'unique-glb)
           (if (eq *sortstyle* 'static)
               (unique-glb)
               (rf-error "- running dynamic sort model")))
          ((eq com 'unsubsumes)
           (setq *rfi-sortbase* (unsubsumes *rfi-sortbase*)))
          ((eq com 'resubsumes)
           (setq *rfi-sortbase* (resubsumes *rfi-sortbase*)))
          ((eq com 'compile-sortbase)
           (if (eq *sortstyle* 'static)
                   ( compile-sortbase *rfi-sortbase*)
                   (rf-error "- running dynamic sort model")))
```

```
        ((eq com 'sortstyle)
         (rfi-cmd-sortstyle userline))
        ((eq com 'destroy-sortbase)
         (setq *subsumes-individuals* nil)
         (setq *rfi-sortbase* nil))
        ((eq com 'sortbase)
         (rfi-cmd-1 (cadr userline) *rfi-sortbase*))
        ((eq com 'browse-sortbase)
         (if (eq *sortstyle* 'static)
             (when *tcl* (browse-sortbase (cadr userline)))
             (rf-error "- running dynamic sort model")))
  .
  .
  .
)


(defun rfi-cmd-sortstyle (userline)
  (let ((error-p nil) )
       (cond ((= 2 (length userline))
              (cond ((eq (second userline) 'static)
                     (setq *sortstyle* 'static))
                    ((eq (second userline) 'dynamic)
                     (setq *sortstyle* 'dynamic))
                    (t (setq error-p t))))
             ((= 1 (length userline))
              (rf-print *sortstyle*))
             (t (setq error-p t)))
       (if error-p
           (progn
            (rf-terpri)
            (rf-princ-like-lisp "Error.  Use:")
            (rf-terpri)
            (rf-princ-like-lisp "   sortstyle static")
            (rf-terpri)
            (rf-princ-like-lisp "or")
            (rf-terpri)
            (rf-princ-like-lisp "   sortstyle dynamic")
            (rf-terpri)
            (rf-princ-like-lisp "or")
            (rf-terpri)
            (rf-princ-like-lisp "   sortstyle")
            (rf-terpri)))))
```

```
(defun rfi-sortbase-cmd-consult (userline)
  (let* ((filename (rfi-extension (cadr userline) (rf-or-rfp))))
    (if (probe-file filename)
        (setq *rfi-sortbase* (append *rfi-sortbase*
                                     (rfi-cmd-consult-1 filename)))
      (rf-error "(rfi-sortbase-cmd-consult): " filename " file doesn't exist!
```

Anhang C

Ein Beispieldialog über RTPLAST

```
rfi-p> destroy-sortbase
rfi-p> destroy
rfi-p> style lisp
rfi-l>
rfi-l> ;-----------------------------------------------------------------------
rfi-l> ; A Declarative RELFUN Knowledge Base on Plastics Production/Recycling
rfi-l> ;                              Partitioned
rfi-l> ;  Into a Sort Base (Inheritance) and an Inference Base (Resolution)
rfi-l> ;-----------------------------------------------------------------------
rfi-l>
rfi-l> ;
rfi-l> ; (c) Harold Boley, Ulrich Buhrmann, Victoria Hall         20 Sept. 94
rfi-l> ;     Michael Herfert, Michael Sintek
rfi-l> ;
rfi-l> ; Sorts are regarded as unary predicates defined in a KB partition
rfi-l> ; and usable in relational queries and variable annotations
rfi-l>
rfi-l> consult-sortbase rtplast-taxo-sub
; Reading file "/home/vega/appl-kb/rtplast-taxo-sub.rf" ..
rfi-l> compile-sortbase
rfi-l> browse-sortbase thermoplastic
rfi-l> sp
rfi-p> sortbase
subsumes(plastic, polykondensate).
subsumes(plastic, polymerisate).
subsumes(plastic, polyaddukte).
subsumes(plastic, thermoplastic).
subsumes(thermoplastic, pc).
subsumes(thermoplastic, pa).
subsumes(thermoplastic, lineare_polyester).
subsumes(thermoplastic, polyphenylene).
subsumes(polykondensate, pc).
```

```
subsumes(polykondensate, pa).
subsumes(polykondensate, lineare_polyester).
subsumes(polykondensate, polyphenylene).
subsumes(lineare_polyester, petb).
subsumes(lineare_polyester, pbtp).
subsumes(polyphenylene, ppo).
subsumes(thermoplastic, polyolefine).
subsumes(thermoplastic, vinylpolymere).
subsumes(thermoplastic, styrolpolymere).
subsumes(thermoplastic, polymethacrylester).
subsumes(thermoplastic, polyacetal).
subsumes(thermoplastic, flourkunststoffe).
subsumes(polymerisate, polyolefine).
subsumes(polymerisate, vinylpolymere).
subsumes(polymerisate, styrolpolymere).
subsumes(polymerisate, polymethacrylester).
subsumes(polymerisate, polyacetal).
subsumes(polymerisate, fluorkunststoffe).
subsumes(polyolefine, pe).
subsumes(polyolefine, pp).
subsumes(polyolefine, pib).
subsumes(polyolefine, pb).
subsumes(polyolefine, pmp).
subsumes(pp, hostalen).
subsumes(styrolpolymere, ps).
subsumes(styrolpolymere, sb).
subsumes(styrolpolymere, san).
subsumes(styrolpolymere, asa).
subsumes(styrolpolymere, abs-k).
subsumes(abs-k, novodur).
subsumes(fluorkunststoffe, ptfe).
subsumes(fluorkunststoffe, pctfe).
subsumes(fluorkunststoffe, pfep).
subsumes(fluorkunststoffe, pvf).
subsumes(fluorkunststoffe, pvdf).
subsumes(thermoplastic, pur).
subsumes(thermoplastic, chlorierter_polyaether).
subsumes(polyaddukte, pur).
subsumes(polyaddukte, chlorierter_polyaether).
subsumes(hostalen, hostalen-ppk-1).
subsumes(hostalen, hostalen-ppn-1).
subsumes(novodur, novodur-rec-1).
novodur-rec-1(novodur_r_5320).
novodur-rec-1(novodur_r_5322).
hostalen-ppk-1(hostalen_ppk_1060_f1).
hostalen-ppn-1(hostalen_ppn_1060).
```

```
hostalen-ppn-1(hostalen_ppn_1060_f).
hostalen-ppn-1(hostalen_ppn_1060_f1).
hostalen-ppn-1(hostalen_ppn_1060_f3).
hostalen-ppn-1(hostalen_ppn_2060).
hostalen-ppn-1(hostalen_ppn_4160).
subsumes(attribute, simple-attribute).
subsumes(attribute, numerical-attribute).
subsumes(numerical-attribute, simple-numerical-attribute).
subsumes(numerical-attribute, numerical-attribute-n/qmm-prototype).
subsumes(numerical-attribute-n/qmm-prototype, numerical-attribute-n/qmm).
subsumes(numerical-attribute, numerical-attribute-kj/qm-prototype).
subsumes(numerical-attribute-kj/qm-prototype, numerical-attribute-kj/qm).
subsumes(numerical-attribute, numerical-attribute-with-conditions).
subsumes(numerical-attribute-with-conditions, melting_index-prototype).
subsumes(melting_index-prototype, melting_index).
subsumes(numerical-attribute-with-conditions, volume_per_unit_time-prototype).
subsumes(volume_per_unit_time-prototype, volume_per_unit_time).
subsumes(attribute, interpol-attribute).
subsumes(interpol-attribute, simple-interpol-attribute).
subsumes(attribute, rule-attribute).
subsumes(rule-attribute, simple-rule-attribute-1).
subsumes(rule-attribute, simple-rule-attribute-2).
simple-attribute(identifier).
simple-attribute(cardinality).
simple-attribute(sort).
simple-attribute(method_for_test).
simple-attribute(additives).
simple-attribute(measurement).
simple-attribute(exists-with-conditions).
simple-attribute(recyclable).
simple-attribute(recycled).
simple-numerical-attribute(temperature).
simple-numerical-attribute(density).
simple-numerical-attribute(volume_per_unit_time_mvr_1_temperature).
simple-numerical-attribute(volume_per_unit_time_mvr_1_stress).
simple-numerical-attribute(volume_per_unit_time_mvr_2_temperature).
simple-numerical-attribute(volume_per_unit_time_mvr_2_stress).
simple-numerical-attribute(melting_index_mfi_1_temperature).
simple-numerical-attribute(melting_index_mfi_1_stress).
simple-numerical-attribute(melting_index_mfi_2_temperature).
simple-numerical-attribute(melting_index_mfi_2_stress).
simple-numerical-attribute(melting_index_mfi_3_temperature).
simple-numerical-attribute(melting_index_mfi_3_stress).
simple-numerical-attribute(vicat_a/50).
simple-numerical-attribute(vicat_b/50).
simple-numerical-attribute(yield_elangation).
```

```
simple-numerical-attribute(dimensional_stability_hdt/a).
simple-numerical-attribute(dimensional_stability_hdt/b).
numerical-attribute-n/qmm(yield_stress).
numerical-attribute-n/qmm(tension_module_of_elasticity).
numerical-attribute-n/qmm(tension_module_of_criep_1_h).
numerical-attribute-n/qmm(tension_module_of_criep_1000_h).
numerical-attribute-n/qmm(ball_thrust_hardness).
numerical-attribute-kj/qm(tension_notched-bar_impact_strength).
numerical-attribute-kj/qm(notched-bar_impact_strength).
numerical-attribute-kj/qm(izod-impact_strength_23).
numerical-attribute-kj/qm(izod-impact_strength_0).
numerical-attribute-kj/qm(izod-impact_strength_-30).
numerical-attribute-kj/qm(izod-notched-bar_impact_strength_23).
numerical-attribute-kj/qm(izod-notched-bar_impact_strength_0).
numerical-attribute-kj/qm(izod-notched-bar_impact_strength_-30).
melting_index(melting_index_mfi_1).
melting_index(melting_index_mfi_2).
melting_index(melting_index_mfi_3).
volume_per_unit_time(volume_per_unit_time_mvr_1).
volume_per_unit_time(volume_per_unit_time_mvr_2).
simple-interpol-attribute(stress-strain).
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % Load inference base
rfi-p> sl
rfi-l> consult rtplast-inf
; Reading file "/home/vega/appl-kb/rtplast-inf.rf" ..
rfi-l> sp
rfi-p> % Load some auxiliary predicates e.g. MEMBER
rfi-p> consult aux
% Reading file "/home/vega/appl-kb/aux.rfp" ..
rfi-p>
rfi-p> % A function mapping materials and production processes to products
rfi-p> % Unary predicates are marked as sorts by a '$'-prefix
rfi-p> % listing used-for-rule
rfi-p>
rfi-p>
rfi-p> % Show all atributes of novodurs
rfi-p> listing  Attribute($novodur, V)
producer($novodur, bayer).
recyclable($novodur, horizontal-recycling).
izod-impact_strength_23($novodur-rec-1, 60).
```

```
izod-impact_strength_-30($novodur-rec-1, 40).
izod-notched-bar_impact_strength_23($novodur-rec-1, 12).
izod-notched-bar_impact_strength_-30($novodur-rec-1, 6).
ball_thrust_hardness($novodur-rec-1, 90).
recycling-product($novodur-rec-1, true).
yield_stress(novodur_r_5320, 38).
yield_elangation(novodur_r_5320, 2.1).
tension_module_of_elasticity(novodur_r_5320, 2000).
dimensional_stability_hdt/a(novodur_r_5320, 90).
dimensional_stability_hdt/b(novodur_r_5320, 95).
yield_stress(novodur_r_5322, 40).
yield_elangation(novodur_r_5322, 2.3).
tension_module_of_elasticity(novodur_r_5322, 2200).
dimensional_stability_hdt/a(novodur_r_5322, 96).
dimensional_stability_hdt/b(novodur_r_5322, 100).
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % Show additives of materials
rfi-p> listing additives(Material, V)
additives(hostalen_ppk_1060_f1, flameproofing_agent).
additives($hostalen-ppn-1, antistatic_additive).
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % A relation defining materials' E-module (tension module of elasticity)
rfi-p> listing tension_module_of_elasticity
tension_module_of_elasticity(hostalen_ppk_1060_f1, 1300).
tension_module_of_elasticity($hostalen-ppn-1, 1300).
tension_module_of_elasticity(novodur_r_5320, 2000).
tension_module_of_elasticity(novodur_r_5322, 2200).
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % For each novodur-rec-1, a subsort of novodur, find its
rfi-p> % tension_module_of_elasticity
rfi-p> % Variable annotations are marked by a ':'-infix
rfi-p> tension_module_of_elasticity(Ident : $novodur-rec-1, E-module)
```

```
true
Ident = novodur_r_5320
E-module = 2000
rfi-p> more
true
Ident = novodur_r_5322
E-module = 2200
rfi-p> more
unknown
rfi-p>
rfi-p>
rfi-p> listing ball_thrust_hardness
ball_thrust_hardness(hostalen_ppk_1060_f1, 67).
ball_thrust_hardness($hostalen-ppn-1, 68).
ball_thrust_hardness($novodur-rec-1, 90).
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % Use of sorted variables
rfi-p>
rfi-p>
rfi-p> % Find each novodur-rec-1 with its tension_module_of_elasticity
rfi-p> % and an unspecified ball_thrust_hardness
rfi-p> tension_module_of_elasticity(I : $novodur-rec-1, E-module),
ball_thrust_hardness(I, Bh)
true
I = novodur_r_5320
E-module = 2000
Bh = 90
rfi-p> more
true
I = novodur_r_5322
E-module = 2200
Bh = 90
rfi-p> more
unknown
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % Find each novodur-rec-1 with its tension_module_of_elasticity
```

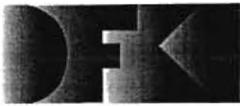
```
rfi-p> % and a given ball_thrust_hardness of 90
rfi-p>
rfi-p> tension_module_of_elasticity(I : $thermoplastic, E-module),
ball_thrust_hardness(I, 90)
true
I = novodur_r_5320
E-module = 2000
rfi-p> more
true
I = novodur_r_5322
E-module = 2200
rfi-p> more
unknown
rfi-p>
rfi-p>
rfi-p> % Let's find a material for producing electrical devices.
rfi-p>
rfi-p> used-for(Material, Verarbeitungsform[elektrogeraete])
true
Material = hostalen_ppk_1060_f1
Verarbeitungsform = spritzgiessen
rfi-p> more
true
Material = hostalen_ppk_1060_f1
Verarbeitungsform = pressen
rfi-p> more
true
Material = $hostalen-ppn-1
Verarbeitungsform = pressen
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % A function computing the recyclability of materials
rfi-p> listing horizontal-recycling
horizontal-recycling(Plastic-id) :-
        additives(Plastic-id : $pp, flameproofing_agent) &
        only_in_closed_circle.
horizontal-recycling(Plastic-id) :-
        additives(Plastic-id : $pp, Additives),
        naf(Additives is flameproofing_agent) &
        possible.
horizontal-recycling(Plastic-id) :-
        additives(Plastic-id : $abs-k, flameproofing_agent) &
```

```
         only_in_closed_circle.
horizontal-recycling(Plastic-id) :-
         additives(Plastic-id : $abs-k, Additives),
         naf(Additives is flameproofing_agent) &
         possible.
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p>
rfi-p> % Now let's find a recyclable electrical device
rfi-p>
rfi-p> used-for(Material, Verarbeitungsform[elektrogeraete]),
recyclable(Material,Rule), Rule(Material)
only_in_closed_circle
Material = hostalen_ppk_1060_f1
Verarbeitungsform = spritzgiessen
Rule = horizontal-recycling
rfi-p> more
only_in_closed_circle
Material = hostalen_ppk_1060_f1
Verarbeitungsform = pressen
Rule = horizontal-recycling
rfi-p> more
possible
Verarbeitungsform = pressen
Rule = horizontal-recycling
Material = $hostalen-ppn-1
rfi-p>
rfi-p>
rfi-p> pause()

rfi-p> bye
true
rfi-p>
rfi-p> % 'Terminological' knowledge about the attributes themselves
rfi-p> listing Att(ball_thrust_hardness, Value)
cardinality($numerical-attribute, <=[1]).
sort($numerical-attribute, real).
measurement($numerical-attribute-n/qmm-prototype, /[n, ^[mm, 2]]).
method_for_test(ball_thrust_hardness, [din_53456]).
rfi-p> cardinality(ball_thrust_hardness, Fun[Val]), Fun(Val,1)
true
Fun = <=
Val = 1
```



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

-Bibliothek, Information
und Dokumentation (BID)-
PF 2080
67608 Kaiserslautern
FRG

Telefon (0631) 205-3506
Telefax (0631) 205-3210

e-mail
dfkibib@dfki.uni-kl.de
WWW
http://www.dfki.uni-sb.de/dfkibib

Veröffentlichungen des DFKI

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder per anonymous ftp von ftp.dfki.uni-kl.de (131.246.241.100) im Verzeichnis pub/Publications bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or by anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) in the directory pub/Publications.
The reports are distributed free of charge except where otherwise noted.

DFKI Research Reports

RR-94-39
Hans-Ulrich Krieger
Typed Feature Formalisms as a Common Basis for Linguistic Specification.
21 pages

RR-94-38
Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, Stephen P. Spackman.
DISCO–An HPSG-based NLP System and its Application for Appointment Scheduling.
13 pages

RR-94-37
Hans-Ulrich Krieger, Ulrich Schaefer.
TDL - A Type Description Language for HPSG, Part 1: Overview.
54 pages

RR-94-36
Manfred Meyer
Issues in Concurrent Knowledge Engineering. Knowledge Base and Knowledge Share Evolution.
17 pages

RR-94-35
Rolf Backofen
A Complete Axiomatization of a Theory with Feature and Arity Constraints
49 pages

RR-94-34
Stephan Busemann, Stephan Oepen, Elizabeth A. Hinkelman, Günter Neumann, Hans Uszkoreit
COSMA - Multi-Participant NL Interaction for Appointment Scheduling
80 pages

RR-94-33
Franz Baader, Armin Laux
Terminological Logics with Modal Operators
29 pages

RR-94-31
Otto Kühn, Volker Becker, Georg Lohse, Philipp Neumann
Integrated Knowledge Utilization and Evolution for the Conservation of Corporate Know-How
17 pages

RR-94-23
Gert Smolka
The Definition of Kernel Oz
53 pages

RR-94-20
Christian Schulte, Gert Smolka, Jörg Würtz
Encapsulated Search and Constraint Programming in Oz
21 pages

RR-94-18
Rolf Backofen, Ralf Treinen
How to Win a Game with Features
18 pages

RR-94-17
Georg Struth
Philosophical Logics—A Survey and a Bibliography
58 pages

RR-94-16
Gert Smolka
A Foundation for Higher-order Concurrent Constraint Programming
26 pages

RR-94-15
Winfried H. Graf, Stefan Neurohr
Using Graphical Style and Visibility Constraints for a Meaningful Layout in Visual Programming Interfaces
20 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

RR-94-13
Jana Koehler
Planning from Second Principles—A Logic-based Approach
49 pages

RR-94-12
Hubert Comon, Ralf Treinen
Ordering Constraints on Trees
34 pages

RR-94-11
Knut Hinkelmann
A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-10
Knut Hinkelmann, Helge Hintze
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-08
Otto Kühn, Björn Höfling
Conserving Corporate Knowledge for Crankshaft Design
17 pages

RR-94-07
Harold Boley
Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-06
Dietmar Dengler
An Adaptive Deductive Planning System
17 pages

RR-94-05
Franz Schmalhofer, J.Stuart Aitken, Lyle E. Bourne jr.
Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-03
Gert Smolka
A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-02
Elisabeth André, Thomas Rist
Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-01
Elisabeth André, Thomas Rist
Multimedia Presentations: The Support of Passive and Active Viewing
15 pages

RR-93-48
Franz Baader, Martin Buchheit, Bernhard Hollunder
Cardinality Restrictions on Concepts
20 pages

RR-93-46
Philipp Hanschke
A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning
81 pages

RR-93-45
Rainer Hoch
On Virtual Partitioning of Large Dictionaries for Contextual Post-Processing to Improve Character Recognition
21 pages

RR-93-44
Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, Martin Staudt
Subsumption between Queries to Object-Oriented Databases
36 pages

RR-93-43
M. Bauer, G. Paul
Logic-based Plan Recognition for Intelligent Help Systems
15 pages

RR-93-42
Hubert Comon, Ralf Treinen
The First-Order Theory of Lexicographic Path Orderings is Undecidable
9 pages

RR-93-41
Winfried H. Graf
LAYLAB: A Constraint-Based Layout Manager for Multimedia Presentations
9 pages

RR-93-40
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt, Andrea Schaerf
Queries, Rules and Definitions as Epistemic Statements in Concept Languages
23 pages

RR-93-38
Stephan Baumann
Document Recognition of Printed Scores and Transformation into MIDI
24 pages

RR-93-36
Michael M. Richter, Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Gabriele Schmidt
Von IDA bis IMCOD: Expertensysteme im CIM-Umfeld
13 Seiten

RR-93-35
Harold Boley, François Bry, Ulrich Geske (Eds.)
Neuere Entwicklungen der deklarativen KI-Programmierung — *Proceedings*
150 Seiten

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

RR-93-34
Wolfgang Wahlster
Verbmobil Translation of Face-To-Face Dialogs
10 pages

RR-93-33
Bernhard Nebel, Jana Koehler
Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis
33 pages

RR-93-32
David R. Traum, Elizabeth A. Hinkelman
Conversation Acts in Task-Oriented Spoken Dialogue
28 pages

RR-93-31
Elizabeth A. Hinkelman, Stephen P. Spackman
Abductive Speech Act Recognition, Corporate Agents and the COSMA System
34 pages

RR-93-30
Stephen P. Spackman, Elizabeth A. Hinkelman
Corporate Agents
14 pages

RR-93-29
Armin Laux
Representing Belief in Multi-Agent Worlds via Terminological Logics
35 pages

RR-93-28
Hans-Ulrich Krieger, John Nerbonne, Hannes Pirker
Feature-Based Allomorphy
8 pages

RR-93-27
Hans-Ulrich Krieger
Derivation Without Lexical Rules
33 pages

RR-93-26
Jörg P. Müller, Markus Pischel
The Agent Architecture InteRRaP: Concept and Application
99 pages

RR-93-25
Klaus Fischer, Norbert Kuhn
A DAI Approach to Modeling the Transportation Domain
93 pages

RR-93-24
Rainer Hoch, Andreas Dengel
Document Highlighting — Message Classification in Printed Business Letters
17 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy
Comparative Study of Connectionist Simulators
20 pages

RR-93-22
Manfred Meyer, Jörg Müller
Weak Looking-Ahead and its Application in Computer-Aided Process Planning
17 pages

RR-93-20
Franz Baader, Bernhard Hollunder
Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-18
Klaus Schild
Terminological Cycles and the Propositional μ-Calculus
32 pages

RR-93-17
Rolf Backofen
Regular Path Expressions in Feature Logic
37 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz
Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster
PLUS - Plan-based User Support Final Project Report
33 pages

RR-93-14
Joachim Niehren, Andreas Podelski, Ralf Treinen
Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-13
Franz Baader, Karl Schlechta
A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-12
Pierre Sablayrolles
A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-11
Bernhard Nebel, Hans-Jürgen Bürckert
Reasoning about Temporal Relations: A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf
Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-09
Philipp Hanschke, Jörg Würtz
Satisfiability of the Smallest Binary Program
8 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer
COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux
Concept Logics with Function Symbols
36 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux
On Skolemization in Constrained Logics
40 pages

RR-93-05
Franz Baader, Klaus Schulz
Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-04
Christoph Klauck, Johannes Schwagereit
GGD: Graph Grammar Developer for features in CAD/CAM
13 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi
An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist
Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-01
Bernhard Hollunder
An Alternative Proof Method for Possibilistic Logic and its Application to Terminological Logics
25 pages

DFKI Technical Memos

TM-95-01
Martin Buchheit, Rüdiger Klein, Werner Nutt
Constructive Problem Solving: A Model Construction Approach towards Configuration
34 pages

TM-94-04
Cornelia Fischer
PAntUDE - An Anti-Unification Algorithm for Expressing Refined Generalizations
22 pages

TM-94-03
Victoria Hall
Uncertainty-Valued Horn Clauses
31 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores
Text Skimming as a Part in Paper Document Understanding
14 pages

TM-93-05
Michael Sintek
Indexing PROLOG Procedures into DAGs by Heuristic Classification
64 pages

TM-93-04
Hans-Günther Hein
Propagation Techniques in WAM-based Architectures — The FIDO-III Approach
105 pages

TM-93-03
Harold Boley, Ulrich Buhrmann, Christof Kremer
Konzeption einer deklarativen Wissensbasis über recyclingrelevante Materialien
11 pages

TM-93-02
Pierre Sablayrolles, Achim Schupeta
Conflict Resolving Negotiation for COoperative Schedule Management Agents (COSMA)
21 pages

TM-93-01
Otto Kühn, Andreas Birk
Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

DFKI Documents

D-95-02
Andreas Butz
BETTY
Ein System zur Planung und Generierung informativer Animationssequenzen
95 Seiten

D-94-15
Stephan Oepen
German Nominal Syntax in HPSG
— On Syntactic Categories and Syntagmatic Relations —
80 pages

D-94-14
Hans-Ulrich Krieger, Ulrich Schaefer.
TDL - A Type Description Language for HPSG, Part 2: User Guide.
72 pages

D-94-12
Arthur Sehn, Serge Autexier (Hrsg.)
Proceedings des Studentenprogramms der 18. Deutschen Jahrestagung für Künstliche Intelligenz KI-94
69 Seiten

D-94-11
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'94 Workshop: KRDB'94 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
65 pages

Note: This document is no longer available in printed form.

D-94-10
F. Baader, M. Lenzerini, W. Nutt, P. F. Patel-Schneider (Eds.)
Working Notes of the 1994 International Workshop on Description Logics
118 pages

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-94-09
DFKI Wissenschaftlich-Technischer Jahresbericht 1993
145 Seiten

D-94-08
Harald Feibel
IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung
58 Seiten

D-94-07
Claudia Wenzel, Rainer Hoch
Eine Übersicht über Information Retrieval (IR) und NLP-Verfahren zur Klassifikation von Texten
25 Seiten

D-94-06
Ulrich Buhrmann
Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien
117 Seiten

D-94-04
Franz Schmalhofer, Ludger van Elst
Entwicklung von Expertensystemen: Prototypen, Tiefenmodellierung und kooperative Wissensevolution
22 Seiten

D-94-03
Franz Schmalhofer
Maschinelles Lernen: Eine kognitionswissenschaftliche Betrachtung
54 Seiten

Note: This document is no longer available in printed form.

D-94-02
Markus Steffens
Wissenserhebung und Analyse zum Entwicklungsprozeß eines Druckbehälters aus Faserverbundstoff
90 pages

D-94-01
Josua Boon (Ed.)
DFKI-Publications: The First Four Years
1990 - 1993
75 pages

D-93-27
Rolf Backofen, Hans-Ulrich Krieger, Stephen P. Spackman, Hans Uszkoreit (Eds.)
Report of the EAGLES Workshop on Implemented Formalisms at DFKI, Saarbrücken
110 pages

D-93-26
Frank Peters
Unterstützung des Experten bei der Formalisierung von Textwissen INFOCOM - Eine interaktive Formalisierungskomponente
58 Seiten

D-93-25
Hans-Jürgen Bürckert, Werner Nutt (Eds.)
Modeling Epistemic Propositions
118 pages

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-93-24
Brigitte Krenn, Martin Volk
DiTo-Datenbank: Datendokumentation zu Funktionsverbgefügen und Relativsätzen
66 Seiten

D-93-22
Andreas Abecker
Implementierung graphischer Benutzungsoberflächen mit Tcl/Tk und Common Lisp
44 Seiten

Note: This document is no longer available in printed form.

D-93-21
Dennis Drollinger
Intelligentes Backtracking in Inferenzsystemen am Beispiel Terminologischer Logiken
53 Seiten

D-93-20
Bernhard Herbig
Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus
97 Seiten

D-93-16
Bernd Bachmann, Ansgar Bernardi, Christoph Klauck, Gabriele Schmidt
Design & KI
74 Seiten

D-93-15
Robert Laux
Untersuchung maschineller Lernverfahren und heuristischer Methoden im Hinblick auf deren Kombination zur Unterstützung eines Chart-Parsers
86 Seiten

D-93-14
Manfred Meyer (Ed.)
Constraint Processing – Proceedings of the International Workshop at CSAM'93, St.Petersburg, July 20-21, 1993
264 pages

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-93-12
Harold Boley, Klaus Elsbernd, Michael Herfert, Michael Sintek, Werner Stein
RELFUN Guide: Programming with Relations and Functions Made Easy
86 pages

D-93-11
Knut Hinkelmann, Armin Laux (Eds.)
DFKI Workshop on Knowledge Representation Techniques — Proceedings
88 pages

Note: This document is no longer available in printed form.

D-93-10
Elizabeth Hinkelman, Markus Vonerden, Christoph Jung
Natural Language Software Registry (Second Edition)
174 pages

D-93-09
Hans-Ulrich Krieger, Ulrich Schäfer
TDLExtraLight User's Guide
35 pages

D-93-08
Thomas Kieninger, Rainer Hoch
Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse
125 Seiten

D-93-07
Klaus-Peter Gores, Rainer Bleisinger
Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-06
Jürgen Müller (Hrsg.)
Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz, Saarbrücken, 29. - 30. April 1993
235 Seiten

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-93-05
Elisabeth André, Winfried Graf, Jochen Heinsohn, Bernhard Nebel, Hans-Jürgen Profitlich, Thomas Rist, Wolfgang Wahlster
PPP: Personalized Plan-Based Presenter
70 pages

D-93-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1992
194 Seiten

D-93-03
Stephan Busemann, Karin Harbusch(Eds.)
DFKI Workshop on Natural Language Systems: Reusability and Modularity - Proceedings
74 pages

D-93-02
Gabriele Schmidt, Frank Peters, Gernod Laufkötter
User Manual of COKAM+
23 pages

D-93-01
Philipp Hanschke, Thom Frühwirth
Terminological Reasoning with Constraint Handling Rules
12 pages

D-95-04

Document

Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache

Victoria Hall