Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-90-02

Logisches Programmieren

mit Feature-Typen

Georg Seul

Mai 1990

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
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 und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, 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
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

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.

Prof. Dr. Gerhard Barth
Director

Logisches Programmieren mit Feature-Typen

Georg Seul

DFKI-D-90-02

Logisches Programmieren mit Feature-Typen

Georg Seul

Zusammenfassung

Diese Arbeit integriert Feature-Typen in eine logische Programmiersprache mit polymorpher Typdisziplin und Untersorten. Während die Typisierung die Erkennung vieler Programmierfehler zur Compile-Zeit ermöglicht, stellen Feature-Typen abstrakte Zugriffsfunktionen und einen Vererbungsmechanismus zur Verfügung.

Es wird zunächst gezeigt wie die Semantik und die allgemeinen operationalen Methoden (Interpreter, Typ-Checker) einer polymorphen Sortenlogik mit Feature-Typen aussehen können. Die logische Programmiersprache TEL verfügt bereits über ein polymorphes Typkonzept mit Untersorten. Wir beschreiben den Entwurf und die Implementierung einer Erweiterung von TEL um Feature-Typen. Es stellt sich heraus, daß die Einbettung von Feature-Typen in das Modulkonzept von TEL interessante Möglichkeiten im Sinne des "information hiding" bietet.

Dieses Dokument ist die überarbeitete Version der gleichnamigen Diplomarbeit an der Universität Kaiserslautern

Dank

Mein besonderer Dank gilt Werner Nutt, dessen engagierte Unterstützung wesentlich zum Gelingen dieser Arbeit beitrug. Seine objektive und methodische Vorgehensweise war mir stets ein Vorbild. Mein Verständnis von POS-Logik und Feature-Typen wurde durch viele Diskussionen mit ihm geprägt. Zusätzlich motiviert wurde ich durch den Enthusiasmus, den Gert Smolka der Weiterentwicklung 'seiner' Sprache TEL entgegenbrachte. Ich danke weiterhin Ralf Scheidhauer, der mich auf viele Feinheiten bei der Implementierung von TEL09 aufmerksam machte. Außerdem war er, ebenso wie Gebhard Przyrembel, eine große Unterstützung bei technischen Problemen mit der Implementierungsumgebung. Nicht zuletzt möchte ich die gesamte Projektgruppe WINO sowie die Arbeitsgruppe Siekmann erwähnen, die mir stets mit Rat und Tat zur Seite standen.

Inhaltsverzeichnis

1 Einleitung **3**

2 Feature-Typen und POS-Logik **8**

2.1 Feature-Typen in der Literatur 8

2.2 Einführung in POS-Logik 13

2.3 Ein Ansatz zur Integration von Feature-Typen in POS-Logik 15

3 Eine POS-Logik mit Feature-Typen **19**

3.1 Feature-Logik als Constraint-Sprache 20

3.2 Wohlgetypte Feature-Constraints 22

3.3 Feature-Spezifikationen 28

3.4 Lösen von Feature-Constraints 31

3.5 Logisches Programmieren mit Feature-Constraints 38

3.6 Typ-Inferenz 43

4 Einführung in TEL **50**

4.1 Sprachbeschreibung 50

4.1.1 Typen 51

4.1.2 Relationen und Funktionen 51

4.1.3 Wohlgetyptheit 54

4.1.4 Module 55

4.2 Struktur des TEL-Systems 56

4.2.1 Parser 59

4.2.2 Signatur-Analyse 59

4.2.3 Typ-Checker 59
4.2.4 Code-Erzeugung 60

5 Sprachbeschreibung von Feature-TEL **61**

5.1 Deklaration von Feature-Typen 62
5.1.1 Lokale Feature-Typen 62
5.1.2 Import und Export von Feature-Typen 65
5.2 Klauseln, Queries und ihre Abarbeitung 73
5.2.1 Feature-Typen in einfachen Conditions 75
5.2.2 Feature-Terme 76
5.2.3 Queries 78

6 Implementierung von Feature-TEL **81**

6.1 Interne Repräsentation von Feature-Typen 82
6.2 Parser 84
6.3 Signatur-Analyse 87
6.3.1 Fill- und Check-Operationen 88
6.3.2 Aufbau von Import- und Export-Signatur 91
6.4 Typ-Checker 94
6.5 Code-Erzeugung 96
6.5.1 Erzeugung von Prolog-Code zur Abarbeitung von Containments auf Feature-Typen 97
6.5.2 Erzeugung von Prolog-Code zur Abarbeitung von Feature-Aufrufen 98
6.5.3 Erzeugung von Prolog-Code zum Ausdrucken von Query-Antworten 98

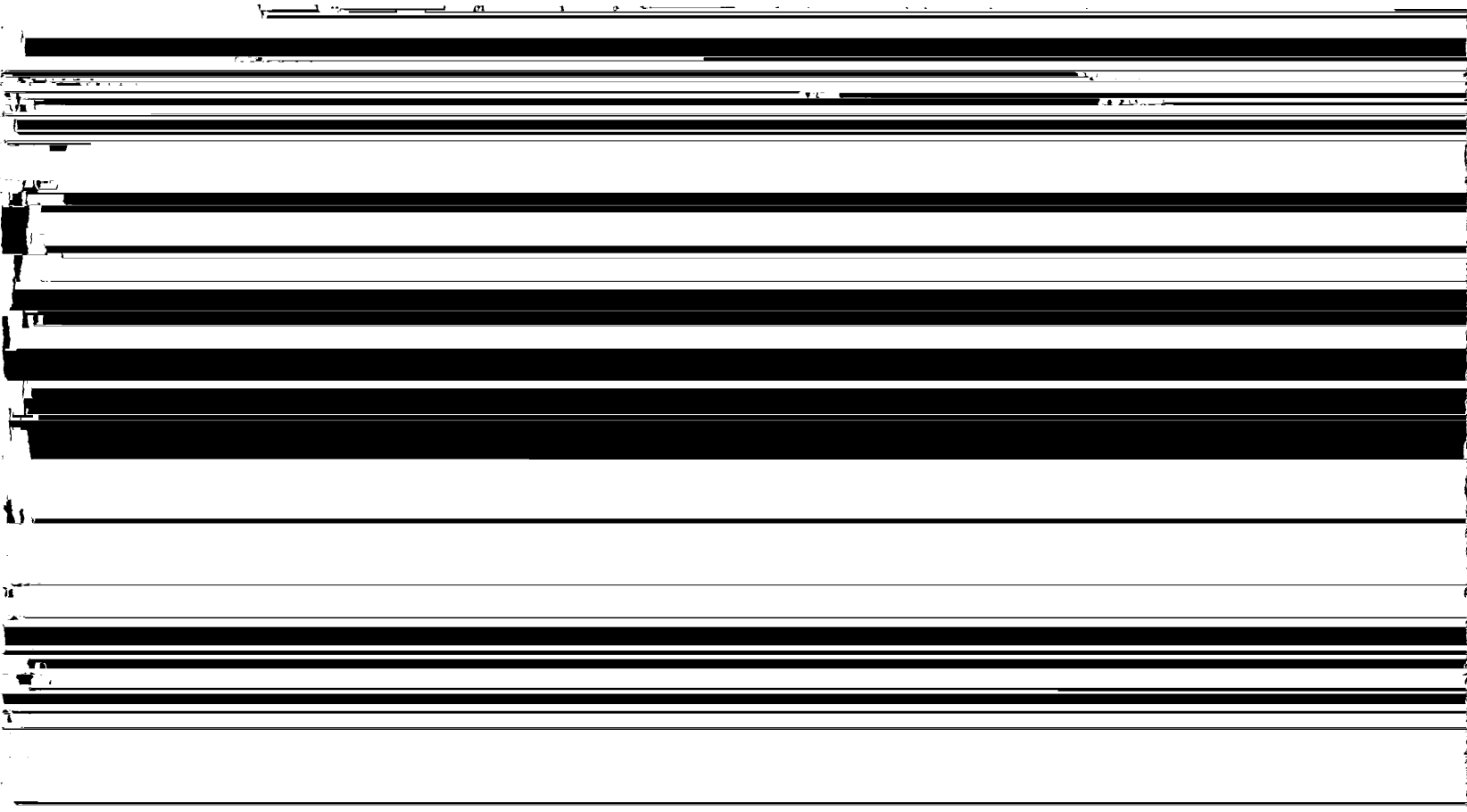
7 Erfahrungen mit Feature-TEL **101**

A Mathematische Grundbegriffe **104**

Kapitel 1

Einleitung

Das Ziel dieser Arbeit ist die Integration von Vererbungshierarchien in die logische Programmiersprache TEL. Vererbungshierarchien werden wir dabei über sogenannte "Feature-Typen" [AS 87] darstellen. Die Sprache TEL [Sm 88] wurde an der Universität Kaiserslautern entwickelt und zeichnet sich durch ein polymorphes Typkonzept aus. Es wird zunächst gezeigt, wie die Semantik und die operationalen Methoden (Interpreter, Typ-Checker) einer polymorphen Sortenlogik mit Feature-Typen aus-



Implementierung der Spracherweiterung von TEL.

Motiviert wurde die vorliegende Arbeit durch die Anforderungen an Implementierungswerkzeuge im Bereich Expertensysteme und Sprachverarbeitung. Dort benötigt man Mechanismen für die Repräsentierung von "strukturiertem Wissen" und den dazugehörigen Inferenzmethoden. Außerdem soll natürlich eine schnelle Systementwicklung unterstützt werden, was insbesondere die effiziente automatische Erkennung von Programmierfehlern voraussetzt. Die vorliegende Arbeit setzt genau

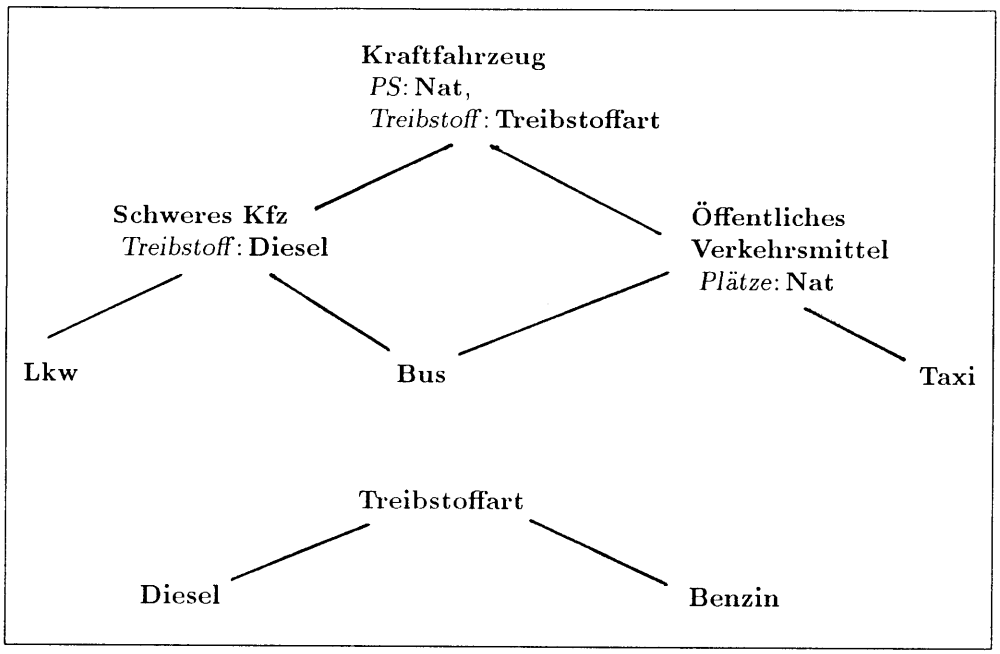


Abbildung 1.1: Eine Vererbungshierarchie

sowohl in der Wissensrepräsentation als auch in Programmiersprachen, so zum Beispiel in Frame-Systemen wie FRL [RG 77] und KL-ONE [BS 85], in semantischen Netzen [Qu 68], sowie im Klassenkonzept von Smalltalk [GR 83].

Bei uns werden Vererbungshierarchien über sogenannte "Feature-Typen" [AS 87] aufgebaut, welche über eine Untertyp-Relation geordnet sind. Man kann sich Feature-Typen als Mengen von Records vorstellen. Jedem Feature-Typ ist eine feste Anzahl von "Features" (Eigenschaften) zugeordnet, welche den Feldern seiner Records entsprechen. Untertypen erben die Features ihrer Obertypen.

Abbildung 1.1 zeigt eine Vererbungshierarchie, mit den Feature-Typen **Kraftfahrzeug**, **Schweres Kfz**, **Öffentliches Verkehrsmittel**, **Lkw**, **Bus**, **Taxi**, **Treibstoffart**, **Diesel** und **Benzin**. Der Typ **Kraftfahrzeug** hat die Features *PS* (vom Typ **Nat**) und *Treibstoff* (vom Typ **Treibstoffart**). Der Typ **Öffentiches Verkehrsmittel** hat drei Features: einmal die von seinem Obertyp **Kraftfahrzeug** ererbten Features *PS* und *Treibstoff*, sowie das explizit deklarierte Feature *Plätze*. Auch der Typ **Bus** hat drei Features: *PS* und *Treibstoff*, welche sowohl von **Schweres Kfz** als auch von **Öffentliches Verkehrsmittel** geerbt werden, sowie das Feature *Plätze*, das nur von **Öffentliches Verkehrsmittel** geerbt wird. Das gleichzeitige Beerben mehrerer Obertypen ist auch als "multiple inheritance" bekannt. Falls ein und dasselbe Feature von zwei verschiedenen Obertypen geerbt wird, so bekommt es den schärferen der beiden Wertebereiche. Beispielsweise erhält das Feature *Treibstoff* für **Bus** den Typ **Diesel** und nicht den allgemeineren Typ **Treibstoffart**. Außerdem ist es möglich, daß ein Feature-Typ den Wertebereich

eines geerbten Features explizit verschärft, wie dies zum Beispiel bei **Schweres Kfz** für *Treibstoff* geschieht.

Wir gehen von einer mengenorientierten Denotation für Feature-Typen aus. Dabei bestehen nichtminimale Feature-Typen (das sind Feature-Typen mit Untertypen) gerade aus den Elementen ihrer Untertypen. Somit ist also jedes Element einer Vererbungshierarchie einem minimalen Typ zugeordnet. Deshalb verlangen wir, daß die minimalen Feature-Typen disjunkt sind. In unserem Beispiel stammt also jedes Element von **Schweres Kfz** entweder aus **Lkw** oder aus **Bus**. Die Elemente eines minimalen Feature-Typs kann man sich als markierte Records vorstellen, deren Felder exakt ihren Features entsprechen. Beispielsweise sind die Elemente von **Lkw** mit "Lkw" markierte Records, die gerade die Felder *PS* und *Treibstoff* haben.

Zur Spezifikation der Untermengen von Feature-Typen verwenden wir sogenannte "Feature-Terme". In unserem Beispiel denotiert der Feature-Term

$$\textbf{Bus}[\mathit{Plätze} \Rightarrow 30]$$

gerade die Menge der mit "Bus" markierten Records, deren *Plätze*-Feld den Wert 30 hat. Der Feature-Term

$$\textbf{Öffentliches Verkehrsmittel}[\mathit{PS} \Rightarrow X{:}\mathit{nat}, \mathit{Plätze} \Rightarrow X]$$

denotiert die Menge aller mit einem minimalen Untertyp von **Öffentliches Verkehrsmittel** markierten Records, deren Felderwerte für *PS* und *Plätze* identisch sind. Eine Bindung verschiedener Features eines Feature-Terms an die gleiche Variable bezeichnet man auch als "Koreferenz". Feature-Unifikation berechnet gerade die Schnittmenge zweier Feature-Terme. Beispielsweise ergibt die Unifikation von

$$\textbf{Öffentliches Verkehrsmittel}[\mathit{PS} \Rightarrow X{:}\mathit{nat}, \mathit{Plätze} \Rightarrow X]$$

und

$$\textbf{Schweres Kfz}[\mathit{PS} \Rightarrow 40]$$

den Feature-Term

$$\textbf{Bus}[\mathit{PS} \Rightarrow 40, \mathit{Plätze} \Rightarrow 40]$$

Falls die Schnittmenge zweier Feature-Terme leer ist, so sind diese nicht unifizierbar. In [AS 87] wird Feature-Unifikation unter dem Kontext einer monomorphen, geordneten Sortenlogik beschrieben.

Polymorphe Typen verstehen wir als "Polymorphically Order-Sorted Types" (kurz: POS-Typen), wie sie in [Sm 89a] beschrieben sind. Dort werden parametrischer Polymorphismus und Sortenlogik (mit Untertypen) kombiniert. POS-Typen werden also über freie Konstruktoren und Untersorten definiert, wobei die definierenden Gleichungen Sortenvariable als Parameter haben dürfen.

Abbildung 1.2 zeigt, wie man auf diese Weise den abstrakten Datentyp "Liste" definieren kann. Man erhält die speziellen Listentypen, indem man die Typvariable

$$
\begin{aligned}
list(T) &:= elist \sqcup nelist(T) \\
elist &:= nil\colon [] \\
nelist(T) &:= cons\colon T \times list(T)
\end{aligned}
$$

$append\colon list(T) \times list(T) \times list(T).$
$append(nil, L, L).$
$append(cons(H, R), L, cons(H, RL)) \leftarrow append(R, L, RL).$

Abbildung 1.2: Ein POS-Programm

T durch den entsprechenden Typ ersetzt. Dabei wird der Hauptvorteil des Polymorphismus deutlich. Allgemeine Operationen auf Listen, wie zum Beispiel "append" brauchen nicht mehr für jeden Typ neu deklariert zu werden. Zusätzlich bringen die Untersorten effizientere Inferenzstrategien. Variable können außer an Werte auch an Typen gebunden werden, so daß mit Hilfe typisierter Unifikation teures Backtracking vermieden werden kann.

Wir typisieren POS-Programme, indem wir den Relationssymbolen Typ-Deklarationen zuordnen, welche beim Typ-Checken und bei der Typ-Inferenz verwendet werden. Der Typ-Inferenzer komplettiert Klauseln um die allgemeinste konsistente Qualifikation der darin vorkommenden Variablen. Zum Beispiel leitet er für die Variablen der zweiten *append*-Klausel in Abbildung 1.2 folgende Typen ab:

$$H\colon T\,,\ R\colon list(T)\,,\ L\colon list(T) \text{ und } RL\colon list(T).$$

Dabei erkennt der Typ-Inferenzer Inkonsistenzen, wie sie etwa aus Tippfehlern resultieren. So kann er beispielsweise nach einem Vertauschen der beiden Argumente von $cons(H, R)$ im Kopf derselben Klausel für H und R keine Typen mehr ableiten.

Die vorliegende Arbeit integriert Feature-Typen in POS-Logik, indem diese (mittels sogenannter "impliziter" Konstruktoren) als algebraische Typen [GT* 78] ausgedrückt werden. Dieser Ansatz liegt nahe, da auch POS-Typen eine modelltheoretische Semantik haben, was eine einheitliche Vorgehensweise ermöglicht.

Die Arbeit ist wie folgt aufgebaut. Im ersten Teil zeigen wir, wie eine Semantik und allgemeine operationale Methoden für die Integration von Vererbungshierarchien in POS-Logik aussehen können. Dazu gehen wir im 2. Kapitel kurz auf verwandte Arbeiten ein und motiveren das Konzept der "Impliziten Konstruktoren". Kapitel 3 ist der theoretische Kern dieser Arbeit. Dort entwickeln wir Syntax, Semantik, sowie allgemeine Constraint-Solving- und Typ-Inferenz-Algorithmen für eine POS-Logik mit Feature-Typen.

Der zweite Teil befaßt sich dann mit der Implementierung von Feature-Typen im TEL-System. Dazu geben wir im 4. Kapitel zunächst eine kurze Einführung in TEL. Kapitel 5 enthält dann eine vollständige Sprachbeschreibung unserer Erweiterung von TEL ("Feature-TEL"). Dabei wird insbesondere die Einbettung von

Feature-Typen in das Modulkonzept von TEL erläutert. Im 6. Kapitel gehen wir schließlich auf die Implementierung der Sprachkonstrukte von Feature-TEL ein. Da Feature-TEL in einem Bootstrapping-Prozeß entwickelt wurde, konnten schon einige praktische Erfahrungen damit gesammelt werden, welche wir im 7. Kapitel dokumentieren. Dort geben wir auch einen Ausblick auf mögliche Erweiterungen.

Kapitel 2

Feature-Typen und POS-Logik

In diesem Kapitel skizzieren wir unseren Ansatz der Integration von Vererbungshierarchien in POS-Logik und gehen auf seine Verwandtschaft mit früheren Arbeiten ein. Das Kapitel ist folgendermaßen aufgebaut. In Abschnitt 2.1 geben wir einen kurzen Rückblick auf die historische Entwicklung von Feature-Typen. Dabei gehen wir insbesondere auf die Arbeit [AS 87] ein. Dort werden Feature-Typen mittels "impliziter" Konstruktoren in eine monomorphe Sortenlogik integriert. Abschnitt 2.2 gibt eine Einführung in POS-Logik [Sm 89a]. Dabei interessiert uns besonders die Vorgehensweise bei der Entwicklung der Semantik und der operationalen Methoden für POS-Typen. Schließlich skizzieren und motivieren wir in Abschnitt 2.3 unseren Ansatz zur Einbettung von Feature-Typen in POS-Logik. Dieser beruht im wesentlichen auf einer Verknüpfung der Arbeiten von [Sm 89a] und [AS 87]. Seine formale Ausarbeitung erfolgt in Kapitel 3.

2.1 Feature-Typen in der Literatur

Das Konzept der Feature-Typen kennen wir sowohl aus der Computerlinguistik als auch aus der Wissensrepräsentation. In der Sprachverarbeitung wurde eine Feature-Constraint-Logik für Unifikationsgrammatiken entwickelt. Einen Überblick geben [Shi 86, Sm 89b]. Im Gegensatz zu "augmented transition networks" ermöglichen

erweitern. Dieser Constraint erzwingt, daß die Werte des Features "*numerus*" für *Artikel* und *Nomen* identisch sein müssen (entweder *plural* oder *singular*).

Grammatikalisches Wissen ist damit auf zwei Ebenen formulierbar, nämlich einmal über kontextfreie Regeln, und außerdem über zu diesen Regeln gehörige Constraints, welche die damit möglichen Ableitungen einschränken, in dem sie gewisse Anforderungen an die in den Regeln enthaltenen linguistischen Objekte stellen. Durch die Zusammenfassung mehrer Gleichungsconstraints zu Feature-Termen erreicht man eine kompakte Darstellung der Regeln [Shi 86]. Dabei ist natürlich Feature-Unifikation die zugrundeliegende Inferenzmethode.

Unabhängig von der Computerlinguistik wurden Feature-Typen zur Repräsentierung von Vererbungshierarchien in der logischen Programmierung entwickelt. Ausgangspunkt war die Sprache LOGIN [AN 86], welche Prolog um sogenannte ψ-Terme erweitert (ψ-Terme entsprechen unseren Feature-Termen). Unifikation von Termen erster Stufe in Prolog wird dabei durch ψ-Term-Unifikation ersetzt. [AN 86] geben eine operationale Beschreibung des Interpreters von LOGIN (inklusive ψ-Term-Unifikation) und skizzieren eine modelltheoretische Semantik von ψ-Termen. Diese wird jedoch nicht auf LOGIN (mit Relationen) erweitert. Außerdem kommt der Zusammenhang zwischen algebraischen Typen und Feature-Typen überhaupt nicht zur Sprache.

Diese Versäumnisse wurden von [AS 87] in einem allgemeineren Rahmen nachgeholt. Dort werden Feature-Typen als algebraische Typen [illegible]

Sortenlogik mit Untertypen und Gleichheit ausgedrückt, was eine einheitliche modelltheoretische Semantik ermöglicht.

Algebraische Typen [GT* 78] werden über eine Signatur und über Gleichheit-

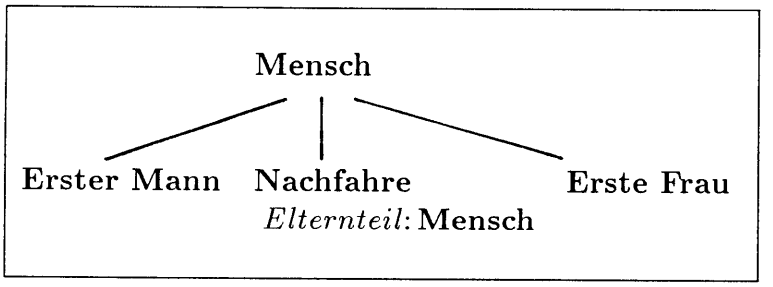


Abbildung 2.1: Der Konstruktor-Typ **Mensch** als Vererbungshierarchie

Untertypen an.

$$\begin{aligned} \textbf{Mensch} &:= \textbf{Erster Mann} + \textbf{Erste Frau} + \textbf{Nachfahre} \\ \textbf{Erster Mann} &:= \{Adam\} \\ \textbf{Erste Frau} &:= \{Eva\} \\ \textbf{Nachfahre} &:= \{Kind\text{:}\ \textbf{Mensch}\} \end{aligned}$$

Die zweite Gleichung sagt aus, daß der Typ **Erster Mann** genau ein Element hat, welches durch den Konstruktor $Adam\text{:} \rightarrow$ **Erster Mann** generiert wird. Die vierte Gleichung sagt aus, daß die Elemente des Typs **Nachfahre** durch Anwendung des Konstruktors *Kind* auf Elemente des Typs **Mensch** erzeugt werden. Die erste Gleichung sagt schließlich aus, daß ein Element von **Mensch** entweder in **Erster Mann**, **Erste Frau**, oder in **Nachfahre** enthalten ist.

In [AS 87] wird insbesondere die Beziehung zwischen Konstruktor-Typen und Feature-Typen charakterisiert. Konstruktor-Typen werden durch sogenannte "Selektoren" vervollständigt, die einen direkten Zugriff auf die Argumente eines Konstruktors ermöglichen. So ist in unserem letzten Beispiel der Selektor des Konstruktors *Kind* gerade sein Inverses im initialen Modell:

$$\begin{gathered} Elternteil\text{:}\ \textbf{Mensch} \rightarrow \textbf{Nachfahre} \\ Elternteil(Kind(N)) = N \end{gathered}$$

Anders ausgedrückt könnte man auch sagen, daß alle Elemente des Typs **Nachfahre** ein Feature $Elternteil\text{:}\ \textbf{Mensch}$ haben. Wenn wir diese Idee weiterverfolgen könnten wir den Typ **Mensch** auch durch die Vererbungshierarchie aus Abbildung 2.1 darstellen. Man beachte, daß jeder Konstruktor-Term in einen Feature-Term übersetzbar ist, zum Beispiel:

$$\begin{aligned} Adam &\quad \textbf{Erster Mann} \\ Eva &\quad \textbf{Erste Frau} \\ Kind(Adam) &\quad \textbf{Nachfahre}[Elternteil \Rightarrow \textbf{Erster Mann}] \end{aligned}$$

$$
\begin{aligned}
\textbf{Kraftfahrzeug} &:= \textbf{Schweres Kfz} + \textbf{Öff.Verkehrsmittel} \\
\textbf{Schweres Kfz} &:= \textbf{Lkw} + \textbf{Bus} \\
\textbf{Öff.Verkehrsmittel} &:= \textbf{Bus} + \textbf{Taxi} \\
\textbf{Lkw} &:= \{KonLkw\text{: } \textbf{Nat} \times \textbf{Diesel}\} \\
\textbf{Bus} &:= \{KonBus\text{: } \textbf{Nat} \times \textbf{Diesel} \times \textbf{Nat}\} \\
\textbf{Taxi} &:= \{KonTaxi\text{: } \textbf{Nat} \times \textbf{Treibstoffart} \times \textbf{Nat}\} \\
\textbf{Treibstoffart} &:= \textbf{Diesel} + \textbf{Benzin} \\
\textbf{Diesel} &:= \{KonDiesel\} \\
\textbf{Benzin} &:= \{KonBenzin\}
\end{aligned}
$$

Variable: P: **Nat**, T: **Treibstoffart**, D: **Diesel**, B: **Benzin**, S: **Nat**

PS: **Kraftfahrzeug** → **Nat**
$PS(KonLkw(P, D)) = P$
$PS(KonBus(P, D, S)) = P$
$PS(KonTaxi(P, T, S)) = P$

Treibstoff: **Kraftfahrzeug** → **Treibstoffart**
Treibstoff: **Schweres Kfz** → **Diesel**
$Treibstoff(KonLkw(P, D)) = D$
$Treibstoff(KonBus(P, D, S)) = D$
$Treibstoff(KonTaxi(P, T, S)) = T$

Plätze: **Öffentliches Verkehrsmittel** → **Nat**
$Plätze(KonBus(P, D, S)) = S$
$Plätze(KonTaxi(P, T, S)) = S$

Abbildung 2.2: Algebraische Spezifikation der Kfz-Vererbungshierarchie

Feature-Typen und Konstruktor-Typen sind offenbar also duale Konzepte. Während Konstruktor-Typen über Konstruktoren spezifiziert sind, geschieht dies bei Feature-Typen über Selektoren. Aus der Definition eines Konstruktors kann man eindeutig (bis auf den Namen) seine Selektoren bestimmen, und umgekehrt. Das bedeutet, daß wir Feature-Typen als algebraische Typen ausdrücken können, wenn wir sie mit versteckten (*impliziten*) Konstruktoren ausstatten. In Übereinstimmung mit Kapitel

Kraftfahrzeug := [*PS*: **Nat**, *Treibstoff*: **Treibstoffart**]
Schweres Kfz := **Kraftfahrzeug**[*Treibstoff*: **Diesel**]
Öffentliches Verkehrsmittel := **Kraftfahrzeug**[*Plätze*: **Nat**]
Lkw := **Schweres Kfz**[]
Bus := **Schweres Kfz** ∗ **Öffentliches Verkehrsmittel**[]
Taxi := **Öffentliches Verkehrsmittel**[]
Treibstoffart := []
Benzin := **Treibstoffart**[]
Diesel := **Treibstoffart**[]

Man beachte, daß diese Gleichungen eindeutig die dazugehörige algebraische Spezifikation aus Abbildung 2.2 bestimmen.

Der Hauptvorteil der Darstellung von Feature-Typen als algebraischen Spezifikationen liegt darin, daß wir eine wohlverstandene modelltheoretische Semantik übernehmen können. Diese baut insbesondere auf den Untersuchungen von Sortenlogik mit Untertypen und Gleichheit auf [Go 78, GT* 78, SN* 87]. In [AS 87] wird eine solche, auf impliziten Konstruktoren beruhende Semantik für Feature-Typen ausgearbeitet. Außerdem wird ein regelbasierter Algorithmus für Feature-Unifikation angegeben und seine Vollständigkeit und Korrektheit bezüglich der entwickelten Semantik gezeigt. Es stellt sich heraus, daß Feature-Unifikation sowohl getypte Unifikation mit Untersorten, als auch ungetypte Unifikation subsumiert.

Das folgende Beispiel soll einen Eindruck von der Arbeitsweise dieses Unifikationsalgorithmus' vermitteln. Wir legen wieder die Vererbungshierarchie aus Abbildung 1.1 beziehungsweise Abbildung 2.2 zugrunde. Zu unifizieren sei die folgende Gleichung.

$$S = \ddot{O}$$

wobei S: **Schweres Kfz**$[PS \Rightarrow 40]$
$\ddot{O}$: **Öffentliches Verkehrsmittel**$[Plätze \rightarrow N$: **Nat**, $PS \rightarrow N]$

Die Betrachtung von Features als Selektoren erlaubt es uns, das Ganze in ein "gewöhnliches" Gleichungssystem zu übersetzen.

$$S = \ddot{O} \;\&\; PS(S) = 40 \;\&\; PS(\ddot{O}) = N \;\&\; Plätze(\ddot{O}) = N$$
wobei S: **Schweres Kfz**, $\ddot{O}$: **Öffentliches Verkehrsmittel**, N: **Nat**

Damit wird klar, daß Feature-Unifikation eigentlich nur eine Kombination von getypter Unifikation mit Untersorten, und dem Auflösen von Gleichungssystemen ist. So wird man hier die Gleichung $S = \ddot{O}$ zu $S = B$ und $\ddot{O} = B$ auflösen und die Vorkommen von S und $\ddot{O}$ durch B ersetzen. Dabei bekommt die Hilfsvariable B den größten gemeinsamen Untertyp von **Scweres Kfz** und **Öffentliches Verkehrsmittel**, nämlich **Bus** als Typ. Damit ist man wiederum in der Lage die Gleichungen $PS(B) = N$ und $PS(B) = 30$ zu reduzieren. Die endgültige Lösung (in Feature-Term-Schreibweise) lautet schließlich:

$$S = B \,\&\, \ddot{O} = B \,\&\, N = 40$$
wobei B: **Bus**[$PS \Rightarrow 40$, *Plätze* $\Rightarrow 40$]

Man beachte, daß die Benutzung von Feature-Termen eine kompakte Darstellung der Lösungsmenge ermöglicht.

Der Nachteil des Ansatzes von [AS 87] liegt darin, daß an die Signaturen von Feature-Typen stark einschränkende Anforderungen gestellt werden. Wir werden darauf im Abschnitt 2.3 und im Schlußkapitel näher eingehen.

Zum Abschluß unseres Literatur-Überblicks über Feature-Typen möchten wir noch kurz die Arbeit [H 88] erwähnen. Dort wird eine allgemeinere Syntax von Feature-Termen als die unsere betrachtet (so ist zum Beispiel auch die Verknüpfung von Feature-Termen mittels logischer Junktoren, sowie die explizite Modellierung von partiellem Wissen über Features möglich). Für diese Feature-Constraint-Sprache wird dann eine Open-World-Semantik entwickelt. Das trägt der Tatsache Rechnung, daß eine eindeutige Beschreibung der realen Welt nie erreicht werden kann, und somit immer nur mögliche Welten betrachtet werden können. Für unsere Zwecke ist dieser Ansatz jedoch kaum brauchbar, denn POS-Typen sind Spezialfälle algebraischer Typen, und werden daher über ein *eindeutiges* initiales Modell beschrieben. Wir werden uns vielmehr auf die eben skizzierte Arbeit von [AS 87] stützen. Die dort beschriebene Reduzierung von Feature-Typen auf algebraische Spezifikationen mittels impliziter Konstruktoren garantiert uns eine einheitliche semantische Betrachtungsweise.

2.2 Einführung in POS-Logik

"Polymorphically Order-Sorted Types" (kurz POS-Typen) kombinieren Parametrischen Polymorphismus [HM* 86] mit Untersorten [SN* 87]. Eine Logik für relationale Programme über POS-Typen wurde in [Sm 89a] entwickelt. Diese Arbeit ist zentral für unser weiteres Vorgehen, und wird deshalb hier kurz skizziert.

Die Verknüpfung von Parametrischem Polymorphismus mit Untersorten bringt die Vorteile beider Konzepte und ist daher von praktischer Bedeutung. Parametrischer Polymorphismus ermöglicht die allgemeine Definition abstrakter Datenstrukturen (wie Listen, Bäumen etc.) über beliebigen anderen Datentypen, ohne sie für jede ihrer Varianten wiederholen zu müssen. Dies gilt insbesondere auch für die dazugehörigen Zugriffsprozeduren. Untersorten implizieren eine Typdisziplin, welche ein automatisches Erkennen vieler Programmierfehler über die aus ihnen resultierenden Typ-Inkonsistenzen ermöglicht. Variable können nicht nur an Werte, sondern auch an Sorten gebunden, und mittels getypter Unifikation (mit Untersorten) abgearbeitet werden. Damit kann die Unvereinbarkeit von Constraints schon auf Typebene erkannt werden, ohne das mittels Backtracking sämtliche Wertekombinationen ausprobiert werden müssen.

POS-Typen sind freie algebraische Typen, und werden über freie Konstrukto-

ren und Typgleichungen spezifiziert. So kann man beispielsweise einen Typ *bool*, wie in normaler Sortenlogik auch, über seine beiden Konstruktoren *true* und *false* definieren:

$$bool := \{true, false\}.$$

Sorten werden durch Sortenterme beschrieben, welche aus Sortenkonstanten, Sortenvariablen und Sortenfunktionen aufgebaut sind. In Abbildung 1.2 ist zum Beispiel *elist* eine Sortenkonstante, T eine Sortenvariable, und *list* eine Sortenfunktion. Programme mit Sortenfunktionen spezifizieren unendlich viele Sorten, zum Beispiel $list(bool)$, $list(list(bool))$, und so weiter. Untersorten-Beziehungen ergeben sich automatisch aus den Typgleichungen, so sind zum Beispiel *elist* und $nelist(T)$ Untertypen von $list(T)$. Wir verlangen von Sortenfunktionen, daß sie monoton sind, das heißt, wenn beispielsweise *nat* ein Untertyp von *int* ist, dann muß auch $list(nat)$ ein Untertyp von $list(int)$ sein. POS-Programme ohne Sortenfunktionen entsprechen gerade logischen Programmen in einer (monomorphen) Sortenlogik mit Untertypen.

Da POS-Typen Spezialfälle algebraischer Typen sind, haben sie eine modelltheoretische Semantik. Ein Modell ordnet den Sorten einer POS-Spezifikation Mengen von Werten zu. Die Untersorten-Relation entspricht dann gerade der Teilmengen-Beziehung (Inklusion). Ein Werteterm ist von einer bestimmten Sorte, wenn er in der dazugehörigen Menge enthalten ist (Membership-Relation). Während eine Sorte einfach die Menge ihrer Werte ist, denotiert ein Typ eine Algebra, die durch Typ-Gleichungen, Sortenfunktionen und Konstruktoren spezifiziert ist. Da man im Kontext von Programmiersprachen aber von Typ-Inferenz, Typ-Checken und Wohlgetyptheit spricht (und nicht etwa von "Wohlgesortetheit") betrachten wir "Typen" und "Sorten" als Synonyme. Die Frage ist nun, ob es für POS-Spezifikationen ein Standardmodell gibt, das heißt, ein Modell dessen gültige Inklusionen und Membership-Relationen auf alle anderen Modelle übertragbar sind. Die Antwort ist "Ja", es gibt ein solches Standardmodell, aber nur für POS-Spezifikationen die spezielle Anforderungen erfüllen. Wenn man die Typ-Gleichungen einer POS-Spezifikation von links nach rechts liest, erhält man ein Typ-Rewriting-System. Von diesem verlangen wir, daß es konfluent und terminierend ist. Dies ist zwar im allgemeinen eine unentscheidbare Eigenschaft, wenn man jedoch die Form der zugelassenen Typgleichungen einschränkt, kann man das Problem in den Griff bekommen. Wir lassen also nur solche Typ-Spezifikationen zu, die ein handhabbares Typ-Rewriting-System spezifizieren. Damit haben wir dann außer einer wohldefinierten Semantik (ein Standardmodell existiert) auch eine effiziente operationale Methode zur Berechnung einer Ordnung auf den Sorten der Spezifikation.

POS-Programme fassen wir als definite Klauseln über einer POS-Constraint-Sprache auf. Sie werden mit Hilfe eines Constraint-Solvers und einer resolutionsähnlichen "Goal-Reduktions-Regel" abgearbeitet. Läßt man beispielsweise nur Gleichungen zwischen Wertetermen erster Ordnung als Constraints zu, so ist Unifikation gerade unser Constraint-Solver und die resultierende Programmiersprache ist Prolog. In [HS 88] wird gezeigt, daß die Semantik der logischen Programme sich direkt aus der Semantik der zugrundeliegenden Constraint-Sprache ergibt.

Als POS-Constraints lassen wir Membership-Relationen, Inklusionen und Gleichungen zwischen Wertetermen zu. Die Lösungen eines POS-Constraints beschreiben wir gerade im Standardmodell der zugrundeliegenden Typ-Spezifikation. In [Sm 89a] wird ein Constraint-Solver in Regelform angegeben, der Typ-Rewriting als eine Basisoperation benutzt. Seine Vollständigkeit und Korrektheit im initialen Modell werden gezeigt. Außerdem wird ein Interpreter für POS-Programme entwickelt, der ebenso wie der Constraint-Solver unnötige Typberechnungen vermeidet.

Um Typ-Inkonsistenzen von POS-Programmen erkennen zu können, verlangen wir, daß Relationen Typdeklarationen haben müssen (siehe auch Abbildung 1.2 : Deklaration von "append"). Ein Typ-Inferenzer soll Klauseln um die allgemeinste konsistente Typ-Qualifikation der unqualifizierten Variablen komplettieren. Falls dies nicht möglich ist, muß eine Typ-Inkonsistenz vorliegen. Es ist bisher ungeklärt, ob ein vollständiger Typ-Inferenzer für POS-Programme existiert. In [Sm 89a] wird jedoch ein für die Praxis akzeptables, schnelles Verfahren vorgestellt.

Die praktische Umsetzung der operationalen Methoden von POS-Logik erfolgte mit der Implementierung der Programmiersprache TEL an der Universität Kaisers-

auf Feature-Terme als Typterme in Typ-Spezifikationen. Wir werden Feature-Typen also über normale POS-Typgleichungen (ohne Feature-Terme) und über die Deklarationen ihrer Features spezifizieren. Wie in Abschnitt 2.1 gezeigt kann man aus der Deklaration der Features (Selektoren) eindeutig die Deklaration der dazugehörigen (impliziten) Konstruktoren gewinnen.

Nachdem wir auf Feature-Terme in Deklarationen verzichten, diskutieren wir nun, ob wir sie in Klauseln zulassen sollen (zum Beispiel in Memberships). Aus [AS 87] wissen wir, daß Feature-Terme keine erhöhte Ausdrucksstärke bringen, sondern

dern nur syntaktischer Zucker für das dazugehörige Constraint-System sind. Für eine formale Betrachtung der Semantik unserer Sprachkombination ist es jedoch sinnvoll diese möglichst primitiv zu halten, deshalb verzichten wir im 3. Kapitel auf Feature-Terme. In unserer praktischen Erweiterung von TEL werden wir allerdings Feature-Terme implementieren, denn ihre kompakte Schreibweise verbessert die Lesbarkeit von Programmen.

Es stellt sich nun die Frage, ob wir polymorphe Feature-Typen zulassen sollen, und ob diese überhaupt sinnvoll sind. Sinnvolle Anwendungen von polymorphen Feature-Typen können wir uns durchaus vorstellen. Beispielsweise wäre ein allgemeiner Diagnose-Datentyp im Rahmen einer Expertensystem-Shell denkbar. Die Typvariablen dieses Diagnose-Typs würde man dann durch die "Symptomstypen" der jeweiligen Anwendung ersetzen. Polymorphe Feature-Typen bringen allerdings auch eine Fülle neuer Probleme beim Checken und Abarbeiten von Typ-Information. So muß man sich etwa klar machen, wie die Zusammenführung von polymorphen Feature-Typen zu verstehen ist. Gegeben seien beispielsweise die folgenden polymorphen Feature-Typen.

$$F(\alpha) := [f: \alpha] \qquad F(\beta) := [g: \beta]$$

Feature-Typ **Taxi** definieren als

$$\textbf{Taxi} := \textbf{Öffentliches Verkehrsmittel}[\mathit{Plätze} \Rightarrow 4].$$

Die Zulassung von Wertetermen in Feature-Typ-Deklarationen hat jedoch schwerwiegende Auswirkungen auf das Typ-Checken. Wie soll man nun Wohlgetyptheit definieren? Ist etwa der Constraint

$$T\colon \textbf{Taxi} \;\&\; \mathit{Plätze}(T) = 17$$

wohlgetypt? Dies widerspräche zumindest der Intuition. Wenn dieser Constraint jedoch nicht wohlgetypt ist, so muß der Typ-Checker solche Inkonsistenzen erkennen und das wird sehr aufwendig. Man betrachte etwa den folgende Constraint:

$$T\colon \textbf{Taxi} \;\&\; r(T, X) \;\&\; \mathit{Plätze}(T) = X\,,$$

wobei r die Deklaration $r\colon \textbf{Kraftfahrzeug} \times \textbf{Nat}$ habe. Um herauszufinden, ob die Gleichung $\mathit{Plätze}(T) = X$ wohlgetypt ist, muß man herausbekommen, ob X den Wert 4 hat. Dazu muß der Typ-Checker den Interpreter simulieren können und ist somit nicht mehr praktikabel (man vergleiche etwa mit dem Typ-Inferenz-Algorithmus aus Kapitel 3). Wir verzichten deshalb auf Werte in Typdeklarationen.

Im Gegensatz zu Konstruktoren können Selektoren (Features) mehrere Ranks haben. Da Werteterme sowohl aus Konstruktoren, als auch aus Features aufgebaut werden sollen, ergeben sich damit zusätzliche Probleme beim Ableiten von Typen für Terme (Abtesten der Membership-Relation). So muß gewährleistet sein, daß auch weiterhin *kleinste* Typen für Werteterme abgeleitet werden können, eine Grundvoraussetzung für effiziente Typoperationen. Dazu stellen wir an die Deklarationen eines Features die Anforderung der Regularität, wie sie auch schon in [AS 87] für monomorphe Typen erhoben worden ist.

Wir kommen nun zur Semantik von Feature-Typen in POS-Logik. Die Hauptfrage dabei ist, welche Feature-Typen mit impliziten Konstruktoren ausgerüstet werden sollen. In [AS 87] werden nur den minimalen Feature-Typen implizite Konstruktoren zugeordnet. Dies wurde damit begründet, daß ansonsten der Dualismus von Feature-Typen und Konstruktor-Typen gestört sei, da nicht mehr jeder Konstruktor-Term als Feature-Term darstellbar wäre. Wir lassen jedoch nur monomorphe Feature-Typen zu und können daher ohnehin polymporphe Konstruktor-Typen nicht darstellen. Die obige Motivation entfällt damit in unseren Fall. Eine Zuordnung von impliziten Konstruktoren auch zu nichtminimalen Feature-Typen, hätte insbesondere den Vorteil, daß die häßliche Minimalitätsbedingung [AS 87] für Feature-Spezifikationen entfiele. Diese verlangt, daß jeder nichtminimale Feature-Typ einen minimalen Feature-Typ unter sich hat, der für die gemeinsamen Features die gleichen Codomains hat. Dadurch wird die Bewohntheit von nichtminimalen Typen erzwungen, was später insbesondere bei der Definition gelöster Formen Bedeutung hat (siehe Satz 3.4.2). Umgekehrt hat eine generelle Ausstattung aller Feature-Typen mit impliziten Konstruktoren den Nachteil, daß ein Feature-Typ immer auch eine von seinen Untertypen unabhängige Denotation hat, was die Darstellung der reinen Vereinigung (Abstraktion) zweier Feature-Typen zu einem Obertyp

unmöglich macht. Dies wäre ein zu hoher Tribut an die Ausdrucksstärke unserer Sprache. Wir werden deshalb genau wie [AS 87] nur minimale Feature-Typen mit impliziten Konstruktoren ausstatten, und die oben beschriebene Minimalitätsbedingung an Feature-Typen stellen. Die Bewohntheit eines nichtminimalen Feature-Typs, unabhängig von seinen Untertypen, modellieren wir, indem wir ihm einen "Dummy-Untertyp" zuordnen. So geben wir in unserem Beispiel aus Abbildung 1.1 dem Typ **Kraftfahrzeug** eine von seinen Untertypen **Schweres Kfz** und **Öffentliches Verkehrsmittel** unabhängige Bedeutung, indem wir ihm einen zusätzlichen Untertyp **Anderes Kfz** zuordnen.

Damit sind die wichtigsten Designentscheidungen für unsere Sprachkombination getroffen. Es besteht dabei offenbar die folgende Beziehung zu unseren beiden "Paten". Wenn wir keine Sortenfunktionen benutzen, so erhalten wir gerade die von [AS 87] [redacted]

Untertypen. Wenn wir umgekehrt keine Features definieren, so ergibt sich POS-Logik [Sm 89a]. Im nächsten Kapitel arbeiten wir unseren Ansatz formal aus.

Kapitel 3

Eine POS-Logik mit Feature-Typen

In diesem Kapitel entwickeln wir formal eine POS-Logik mit Feature-Typen auf der Basis impliziter Konstruktoren. Dabei gehen wir nach dem erweiterten CLP-Schema von [HS 88] vor, und betrachten logische Programme als definite Klauseln über einer Constraint-Sprache. Dadurch können wir uns im wesentlichen auf den Constraint-Anteil unserer Feature-Logik beschränken. Wir entwickeln Syntax, Semantik und einen Constraint-Solver für Feature-Constraints und übertragen diese Ergebnisse dann mittels der in [HS 88] beschriebenen Methodik auf logische Programme.

Dieses Kapitel ist folgendermaßen aufgebaut. In Abschnitt 3.1 entwickeln wir eine Constraint-Sprache für unsere Feature-Logik. Um einen vollständigen und korrekten Constraint-Solver in einer sinnvollen Standard-Interpretation angeben zu können, schränken wir unsere Betrachtung im Abschnitt 3.2 auf spezielle Constraint-Systeme ein, welche unter einer gegebenen "Typ-Spezifikation" wohlgetypt sind. Im darauffolgenden Abschnitt 3.3 werden Typ-Spezifikationen dann zu "Feature-Spezifikationen" erweitert, was den Zusammenhang zwischen impliziten Konstruktoren, Features und Feature-Typen herstellt. Für Feature-Spezifikationen gibt es ein Standard-Modell, wobei die Resultate, die dort gelten auf beliebige andere Modelle der Spezifikation übertragbar sind. Abschnitt 3.4 präsentiert einen vollständigen und korrekten Constraint-Solver für dieses Standardmodell. Schließlich beschäftigen wir uns in Abschnitt 3.5 damit, wie logische Programme über unserer Feature-Constraint-Sprache interpretiert werden, und wann diese wohlgetypt sind. Abschnitt 3.6 stellt einen "Typ-Inferenz"-Algorithmus vor, der versucht solche Programme wohlgetypt zu machen, indem er für unqualifizierte Variable möglichst allgemeine Typen ableitet.

3.1 Feature-Logik als Constraint-Sprache

In diesem Abschnitt entwickeln wir eine Constraint-Sprache für unsere Feature-Logik. Dabei gehen wir von der in [HS 88] beschriebenen Erweiterung des "Constraint-Logic-Programming–Schema"s von Jaffar und Lassez aus.

Der Abschnitt ist folgendermaßen organisiert. Zunächst beschreiben wir die Syntax der in unseren Constraints verwendeten Terme mit Hilfe von sogenannten "Feature-Signaturen". Eine Bedeutung erhalten die Terme durch die Einführung von "Feature-Algebren". Schließlich zeigen wir, wie die Feature-Constraint-Sprache bezüglich einer Feature-Signatur Σ aussieht. Dabei sind deren Interpretationen gerade die Feature-Algebren über Σ.

Eine *Feature-Signatur* $\Sigma = (\mathcal{S}, \mathcal{V})$ besteht aus

- einer Menge $\mathcal{S}$ von *Sortensymbolen*
- einer Menge $\mathcal{V}$ von *Funktionssymbolen.*

Feature-Typen sind spezielle Sortensymbole und werden mit ξ und ζ bezeichnet. Beliebige Sortensymbole benennen wir mit η, ψ und χ. Die Funktionssymbole $\mathcal{V}$ untergliedern sich in *Feature-Symbole* (kurz *Features*, bezeichnet mit f) und Konstruktoren. Diese spalten sich wiederum in *explizite Konstruktoren* und *implizite Konstruktoren* auf. Für explizite Konstruktoren verwenden wir den Buchstaben c, während implizite Konstruktoren mit $\bar{c}$ bezeichnet werden. Beliebige Funktionssymbole beginnen mit g.

Jedem Symbol aus Σ ist eine feste Stelligkeit zugeordnet. Alle Features haben Stelligkeit 1 und alle Feature Typen sind Konstanten, das heißt sie haben Stelligkeit 0. Wir benutzen zwei disjunkte Variablenalphabete: *Sortenvariable*, bezeichnet mit α, β, γ und *Wertevariable*, bezeichnet mit x, y, z. *Σ-Sortenterme (Typterme)* bauen sich aus Sortenvariablen und Sortensymbolen (entsprechend ihrer Stelligkeit) auf. *Σ-Werteterme* bestehen aus Wertevariablen und Funktionssymbolen (entsprechend ihrer Stelligkeit). Wir verwenden im folgenden die Buchstaben σ, τ, μ für Sortenterme (kurz *Sorten* oder *Typen*) und s, t, u für Werteterme. Die Funktion $\mathcal{V}$ liefert für jeden Sorten- oder Werteterm die Menge der darin enthaltenen Variablen.

Nachdem wir die syntaktischen Objekte unserer Constraint-Sprache eingeführt haben, müssen wir ihnen nun eine Bedeutung geben. Dies tun wir mit Hilfe von "Feature-Algebren".

Eine *Feature-Algebra $\mathcal{A}$ über einer Feature-Signatur* $\Sigma = (\mathcal{S}, \mathcal{V})$ besteht aus

- einer Menge von *Sorten* $\mathcal{S}^{\mathcal{A}}$, versehen mit einer partiellen Ordnung "$\leq_{\mathcal{A}}$"
- einer Menge von *Werten* $\mathcal{V}^{\mathcal{A}}$, wobei $\mathcal{S}^{\mathcal{A}}$ und $\mathcal{V}^{\mathcal{A}}$ disjunkt sind
- einer Relation "$:_{\mathcal{A}}$" $\subseteq \mathcal{V}^{\mathcal{A}} \times \mathcal{S}^{\mathcal{A}}$, wobei für alle $a \in \mathcal{V}^{\mathcal{A}}$ ein $A \in \mathcal{S}^{\mathcal{A}}$ mit $a:_{\mathcal{A}} A$ existiert; falls gilt $a:_{\mathcal{A}} A$ und $A \leq_{\mathcal{A}} B$, dann gilt auch $a:_{\mathcal{A}} B$

- einer totalen Funktion

$$\eta^{\mathcal{A}}: \mathcal{S}^{\mathcal{A}} \times \ldots \times \mathcal{S}^{\mathcal{A}} \to \mathcal{S}^{\mathcal{A}}$$

für alle Sortensymbole $\eta \in \Sigma$; $\eta^{\mathcal{A}}$ ist monoton bezüglich $\leq_{\mathcal{A}}$, das heißt:

$$\text{aus } \vec{A} \leq_{\mathcal{A}} \vec{B} \text{ folgt } \eta^{\mathcal{A}}(\vec{A}) \leq_{\mathcal{A}} \eta^{\mathcal{A}}(\vec{B})$$

- einer partiellen Funktion

$$g^{\mathcal{A}}: \mathcal{V}^{\mathcal{A}} \times \ldots \times \mathcal{V}^{\mathcal{A}} \to \mathcal{V}^{\mathcal{A}}$$

für alle Funktionssymbole $g \in \Sigma$.

Wir bezeichnen mit $\mathcal{D}^{\mathcal{A}} := \mathcal{S}^{\mathcal{A}} \cup \mathcal{V}^{\mathcal{A}}$ den *Domain* von $\mathcal{A}$. Eine *$\mathcal{A}$-Belegung* δ bildet alle Sortenvariablen nach $\mathcal{S}^{\mathcal{A}}$ und alle Wertvariablen nach $\mathcal{V}^{\mathcal{A}}$ ab. $ASS^{\mathcal{A}}$ bezeichnet die Menge aller $\mathcal{A}$-Belegungen. Gegeben eine Feature-Algebra $\mathcal{A}$ und eine $\mathcal{A}$-Belegung δ, definieren wir die *Denotation* $\mathcal{A}[\![\cdot]\!]_\delta$ als kleinste partielle Funktion von Σ-Sorten- und Wertetermen nach $\mathcal{D}^{\mathcal{A}}$, die folgenden Gleichungen genügt:

$$\mathcal{A}[\![\alpha]\!]_\delta = \delta(\alpha) \quad \mathcal{A}[\![\eta(\vec{\sigma})]\!]_\delta = \eta^{\mathcal{A}}(\mathcal{A}[\![\vec{\sigma}]\!]_\delta)$$
$$\mathcal{A}[\![x]\!]_\delta = \delta(x) \quad \mathcal{A}[\![g(\vec{s})]\!]_\delta = g^{\mathcal{A}}(\mathcal{A}[\![\vec{s}]\!]_\delta)$$

Nun verfügen wir über sämtliche Hilfsmittel zur endgültigen Definition einer "Feature-Constraint-Sprache über einer Feature-Signatur Σ". Wir beschreiben die Syntax der Constraints über Σ-Sorten- und Werteterme. Als Interpretation übernehmen wir die Feature-Algebren über Σ und die Lösungen von Constraints bezüglich einer Interpretation erhalten wir, indem wir die Denotation von Termen auf Constraints liften.

Zu einer Feature-Signatur Σ konstruieren wir wie folgt die *Feature-Constraint-Sprache* $\mathcal{L}(\Sigma)$:

- $\mathcal{L}(\Sigma)$ hat drei Arten von *(Feature-) Constraints*:
  1. *Inklusionen* $\sigma \sqsubseteq \tau$, wobei σ und τ Σ-Typterme sind
  2. *Memberships (Containments)* $s{:}\sigma$, wobei s ein Σ-Werteterm und σ ein Σ-Typterm ist
  3. *Gleichungen* $s \doteq t$, wobei s und t Σ-Werteterme sind
- Die Interpretationen von $\mathcal{L}(\Sigma)$ sind die Feature-Algebren über Σ

Ein $\mathcal{L}(\Sigma)$-Constraint C heißt *gültig* in einer Interpretation $\mathcal{A}$, falls $\mathcal{A}[\![C]\!] = ASS^{\mathcal{A}}$.Ein $\mathcal{L}(\Sigma)$-Constraint C heißt *erfüllbar* in einer Interpretation $\mathcal{A}$, falls $\mathcal{A}[\![C]\!] \neq \emptyset$. Sei V eine Menge von Variablen. Dann sind die *V-Lösungen* $\mathcal{A}[\![\cdot]\!]^V$ eines $\mathcal{L}(\Sigma)$-Constraints C in einer Interpretation $\mathcal{A}$ definiert als:

$$\mathcal{A}[\![C]\!]^V = \{\delta_{|V} \mid \delta \in \mathcal{A}[\![C]\!]\}$$

wobei $\delta_{|V}$ den Domain von δ auf V einschränkt.

Damit besitzen wir nun eine Constraint-Sprache für unsere Feature-Logik. Sie ist jedoch zu allgemein, um dafür eine vernünftige Standard-Interpretation (in welcher der Constraint-Solver arbeitet) angeben zu können. Deshalb werden wir in den folgenden Abschnitten gewisse Anforderungen an Feature-Constraint-Sprachen stellen.

Allgemeine Annahme. *Für den Rest des Kapitels sei Σ eine feste Feature-*

[redacted]

Signatur und $\mathcal{L}(\Sigma)$ die dazugehörige Feature-Constraint-Sprache.

3.2 Wohlgetypte Feature-Constraints

Nachdem der vorangehende Abschnitt eine allgemeine Feature-Constraint-Sprache eingeführt hat, beschränken wir uns nun auf spezielle, "wohlgetypte" Constraint-Systeme. Für diese existiert ein Standard-Grundtermmodell mit folgender Eigenschaft. Ist ein Constraint im Standard-Grundtermmodell erfüllbar, so ist er in jedem beliebigen Modell erfüllbar. Der Träger unseres Standardmodells ist die Menge der "wohlgetypten" Grundterme. Dabei werden wir Wohlgetyptheit über sogenannte Typ-Spezifikationen definieren. Diese ordnen den Funktionssymbolen Typdeklarationen zu und etablieren über Inklusionen eine Ordnung auf den Typen. Ein Werteterm ist dann wohlgetypt, wenn man mittels der Funktionsdeklarationen und der Typordnung einen Typ für ihn ableiten kann. Um Lösungen von Feature-Constraints eindeutig darstellen zu können (Voraussetzung für einen effizienten Constraint-Solver) ist es wichtig, daß obige Typspezifikationen folgende

[redacted]

zu

$$x{:}\,\sigma \sqcap \tau.$$

Die zweite Voraussetzung garantiert eine eindeutige Lösung für Containments der Form $t{:}\,\alpha$. Die dabei auftretenden Probleme illustrieren wir anhand einiger Beispiele.

Gegeben sei die typische Listenspezifikation

$$\mathit{nil}{:}\,\mathit{list}(\alpha)\ ,\ \mathit{cons}{:}\,\alpha \times \mathit{list}(\alpha).$$

Dann hat *nil* keinen kleinsten Typ und die obige Spezifikation ist somit nicht regulär. Wir lösen das Problem, indem wir generell einen leeren Typ $\bot$ einführen, der durch die Inklusion

$$\bot \sqsubseteq \alpha$$

definiert wird. Dann ist $\mathit{list}(\bot)$ der kleinste Typ von *nil*. Um zu gewährleisten, daß $\bot$ wirklich leer ist, wird gefordert, daß $\bot$ nur in dieser Inklusion vorkommt.

Eine weitere Schwierigkeit ergibt sich, wenn ein Funktionssymbol mehrere Deklarationen hat. Gegeben seien zum Beispiel folgende Deklarationen

$$f{:}\,\sigma_1 \to \mu_1 \text{ und } f{:}\,\sigma_2 \to \mu_2$$

sowie die Inklusionen

$$\sigma \sqsubseteq \sigma_1 \text{ und } \sigma \sqsubseteq \sigma_2.$$

Wir wollen den kleinsten Typ von $f(t)$ ausrechnen, wobei σ der kleinste Typ von t sei. Offenbar könnnen wir beide Deklarationen für f anwenden, wobei zum einen μ_1 und zum anderen μ_2 als kleinster Typ von $f(t)$ herauskommt. Um eindeutig zu sein, fordern wir, daß in solchen Situationen eine weitere Deklaration

$$\sigma_1 \sqcap \sigma_2 \to \mu$$

existieren muß , mit $\mu \leq \mu_1$ und $\mu \leq \mu_2$.

Außerdem brauchen wir *Abgeschlossenheit nach oben*, das heißt, daß für zwei Typen σ und τ mit einem gemeinsamen Obertyp, ein kleinster gemeinsamer Obertyp $\sigma \sqcup \tau$ (das *Supremum von* σ *und* τ) existieren muß. Dies wird zum Beispiel deutlich, wenn man die Funktionsdeklaration

$$g{:}\,\alpha \times \alpha \to \alpha$$

betrachtet und versucht den kleinsten Typ von $g(s,t)$ aus den kleinsten Typen von s und t abzuleiten. Glücklicherweise ergibt sich die Abgeschlossenheit nach oben automatisch aus der Abgeschlossenheit nach unten

Eine ausführliche Darstellung der Anforderungen an polymorphe Typ-Spezifikationen findet man in [Sm 89a]. Hier werden nur die für diese Arbeit benötigten Ergebnisse zusammengefaßt . Der Abschnitt ist wie folgt gegliedert. Zunächst befassen wir uns mit den Inklusionen einer Typ-Spezifikation. Von links nach rechts gerichtet ergeben diese ein Rewriting-System. Wir definieren "Typ-Rewriting-Systeme", welche eine Ordnung auf den Typtermen etablieren, die den obigen Anforderungen genügt. Als nächstes beschäftigen wir uns mit den Anforderungen an die Funktionsdeklarationen von Typ-Spezifikationen. Damit besitzen wir die Hilfsmittel, um Wohlgetyptheit für Werteterme und das Ableiten von kleinsten Typen in Typ-Spezifikationen zu definieren. In der Folge liften wir Wohlgetyptheit von Wertetermen auf Constraint-Systeme. Um zu gewährleisten, daß Typterme, die kein $\bot$ enthalten, eine nichtleere Denotation haben, wird der Begriff der "Bewohntheit" von Typ-Spezifikationen eingeführt. Schließlich gehen wir noch kurz auf die Lösungen von Inklusionssystemen ein.

Die folgende Aussage zeigt, wie man aus einer Menge von Inklusionen ein "korrekt arbeitendes" Rewriting-System auf Sortentermen erhält.

Satz 3.2.1 *Sei $\mathcal{A}$ eine Feature-Algebra und I eine Menge von Σ-Inklusionen. Falls für jede Inklusion $\tau \sqsubseteq \sigma$ aus I, die Menge der in τ enthaltenen Variablen eine Untermenge der in σ enthaltenen Variablen ist, so definiert*

$$R(I) = \{\sigma \to \tau \mid (\tau \sqsubseteq \sigma) \in I\}$$

ein Rewriting-System auf Sortentermen. Falls außerdem alle Inklusionen aus I in $\mathcal{A}$ gültig sind, dann ist

$$\tau \sqsubseteq \sigma \text{ gültig in } \mathcal{A}, \text{ falls } \sigma \to^*_{R(I)} \tau.$$

Sei R ein Rewriting-System auf Sortentermen. Wir schreiben $\sigma \Rightarrow_R \tau$, falls $\sigma \to \tau$ eine Instanz einer Regel aus R ist. Wir liften $\Rightarrow_R$ folgendermaßen auf Sortensymbole. Es gilt $\eta \Rightarrow_R \psi$, falls es $\vec{\sigma}$, $\vec{\tau}$ gibt, so daß $\eta(\vec{\sigma}) \Rightarrow_R \psi(\vec{\tau})$ eine Instanz einer Regel aus R ist. Im folgenden sei $\bot$ ein konstantes Sortensymbol.

Ein *Typ-Rewriting-System* R ist ein endliches, terminierendes Rewriting-System auf Sortentermen, wobei gilt

1. R enthält eine Regel

   $$\alpha \to \bot,$$

   keine andere Regel aus R enthält $\bot$, und alle anderen Regeln aus R haben die Form

   $$\eta(\vec{\alpha}) \to \psi(\vec{\sigma}),$$

   wobei $\vec{\alpha}$ ein Tupel paarweise disjunkter Sortenvariablen ist

2. "$\eta \Rightarrow_R \psi$" terminiert

3. Falls σ die rechte Seite einer Regel aus R ist und sowohl $\sigma \Rightarrow_R^* \eta(\vec{\tau})$ als auch $\sigma \Rightarrow_R^* \eta(\vec{\mu})$ gilt, so muß auch $\eta(\vec{\tau}) = \eta(\vec{\mu})$ gelten

4. Die Menge der Sortensymbole bildet einen Quasi-Verband bezüglich der partiellen Ordnung $\eta \Rightarrow_R^* \psi$.

Satz 3.2.2 *Es ist entscheidbar, ob ein endliches Rewriting-System ein Typ-Rewriting-System ist. Für Typ-Rewriting-Systeme ist es außerdem entscheidbar, ob $\sigma \rightarrow_R \tau$ terminiert.*

Die Rewriting-Relation $\rightarrow_R^*$ definiert eine partielle Ordnung "$\leq$" auf der Menge der Typterme.

Satz 3.2.3 *Gegeben sei ein Typ-Rewriting-System R und zwei beliebige Typterme σ und τ. Falls es einen Typ μ gibt mit $\sigma \leq \mu$ und $\tau \leq \mu$, dann gibt es auch einen kleinsten Typ $\sigma \sqcup \tau$ (das "Supremum" von σ und τ) mit $\sigma \leq (\sigma \sqcup \tau)$ und $\tau \leq (\sigma \sqcup \tau)$. Außerdem exisiert ein größter gemeinsamer Untertyp $\sigma \sqcap \tau$ für σ und τ (das "Infimum" von σ und τ). Sowohl Infima als auch Suprema sind berechenbar.*

Die Typ-Spezifikationen, die wir anstreben, werden genau solche Inklusionen enthalten, die in gerichteter Form ein Typ-Rewriting-System ergeben. Dann ordnet $\Rightarrow_R^*$ die Menge der Typterme in einem Quasi-Verband mit kleinstem Element $\bot$ an.

Um Wohlgetyptheit auf Wertetermen zu spezifizieren, müssen wir nun noch die Funktionssymbole "typisieren". Dies geschieht mit Hilfe von "Funktionsdeklarationen".

Eine *Funktionsdeklaration* für ein Funktionssymbol f ist eine Implikation der Form

$$x_1{:}\sigma_1 \;\&\; \ldots \;\&\; x_n{:}\sigma_n \rightarrow f(x_1, \ldots, x_n){:}\sigma,$$

wobei $n \geq 0$ und $x_1, \ldots, x_n$ paarweise disjunkt sind. Wir benutzen auch die variablenunabhängige Schreibweise

$$f{:}\sigma_1 \cdots \sigma_n \rightarrow \sigma.$$

Wie bereits erwähnt, kann es Probleme geben, falls ein Funktionssymbol mehrere Funktionsdeklarationen hat. Um in solchen Fällen immer eine eindeutige, "kleinste" Deklaration anwenden zu können, fordern wir, daß Funktionsdeklarationen "regulär" sind.

Eine Menge FD von Funktionsdeklarationen für ein Funktionssymbol f heißt *regulär*, falls für zwei beliebige Funktionsdeklarationen

$$f{:}\sigma_1 \cdots \sigma_n \rightarrow \sigma \;\text{ und }\; f{:}\tau_1 \cdots \tau_n \rightarrow \tau$$

aus FD eine der beiden folgenden Bedingungen erfüllt ist:

1. $\sigma_i \sqcap \tau_i$ existiert für alle i und

$$\sigma_1 \sqcap \tau_1 \cdots \sigma_n \sqcap \tau_n \to \mu$$

ist eine Deklaration aus *FD*, wobei $\mu \leq \sigma$ und $\mu \leq \tau$ gilt

2. es existiert ein i mit $\sigma_i = \eta(\cdots)$, $\tau_i = \psi(\cdots)$ und es gibt kein Sortensymbol unterhalb von η und ψ.

Nun sind wir soweit, daß wir "Typ-Spezifikationen" definieren können. Diese etablieren über Inklusionen eine Ordnung auf den Typtermen und ordnen den Funktionssymbolen Deklarationen zu.

Eine *Typ-Spezifikation* $\mathcal{T}$ über einer Constraint-Sprache $\mathcal{L}(\Sigma)$ ist eine endliche Menge von Inklusionen und Funktionsdeklarationen mit folgenden Eigenschaften:

1. Die Inklusionen von $\mathcal{T}$ ergeben ein Typ-Rewriting-System $R(\mathcal{T})$, wenn man jede Inklusion $\tau \sqsubseteq \sigma$ als Rewrite-Regel $\sigma \to \tau$ interpretiert

2. Jede Funktionsdeklaration von $\mathcal{T}$ hat die Form $f\colon \vec{\sigma} \to \eta(\vec{\alpha})$, wobei $\vec{\alpha}$ ein Tupel paarweise disjunkter Sortenvariablen ist, und $\mathcal{V}(\vec{\sigma}) \subseteq \mathcal{V}(\vec{\alpha})$ gilt

3. Kein Funktionsdeklaration aus $\mathcal{T}$ enthält den leeren Typ $\bot$

4. Für alle Funktionssymbole g ist die Menge der dazugehörigen Funktionsdeklarationen regulär und nicht leer.

Allgemeine Annahme. *Im folgenden sei $\mathcal{T}$ eine feste Typspezifikation über unserer Constraint-Sprache $\mathcal{L}(\Sigma)$.*

Nachdem wir mit Hilfe von Typ-Spezifikationen eine Ordnung $\leq$ auf Typtermen definiert haben (die diese in einem Semi-Verband anordnet), und den Funktionssymbolen Ranks zugeordnet haben, sind wir fast soweit, daß wir Wohlgetyptheit auf Termen definieren können; wir müssen nur noch die Typen der benutzten Variablen kennen.

Ein *Präfix* ist eine Konjunktion von Containments der Form

$$x_1\colon \sigma_1 \;\&\; \ldots \;\&\; x_n\colon \sigma_n,$$

Als nächstes zeigen wir, wie man mit Hilfe der Typ-Spezifikation $\mathcal{T}$ und einem Präfix P Typen für Werteterme ableitet. Ein Werteterm soll genau dann wohlgetypt sein, wenn man einen Typ für ihn ableiten kann.

Wir definieren induktiv die zweistellige Relation $\vdash$ auf Präfixen und Containments:

1. $P \vdash x{:}\sigma$ genau dann, wenn $Px \leq \sigma$
2. $P \vdash f(\vec{s}){:}\sigma$ genau dann, wenn eine Instanz $f : \vec{\mu} \to \tau$ einer Funktionsdeklaration von f in $\mathcal{T}$ existiert, mit $P \vdash \vec{s}{:}\vec{\mu}$ und $\tau \leq \sigma$.

Wir sagen, "für s kann man unter P den Typ σ ableiten", falls gilt $P \vdash s{:}\sigma$. Statt $\emptyset \vdash s{:}\sigma$ schreiben wir auch $\vdash s{:}\sigma$.

Satz 3.2.4 *Falls gilt $P \vdash s{:}\sigma$, so gilt auch die Implikation $P \to s{:}\sigma$, das heißt das Ableiten von Typen für Werteterme ist korrekt.*

Wir sagen, daß ein Werteterm s *wohlgetypt unter einem Präfix P* ist, falls es einen Typterm σ gibt, mit $P \vdash s{:}\sigma$. Sei s ein Grundterm, so heißt s *wohlgetypt*, falls $\emptyset \vdash s{:}\sigma$ für einen Typterm σ.

Die obige Definition von Wohlgetyptheit ist sinnvoll, da Werteterme, für die kein Typ abgeleitet werden kann, offenbar eine leere Denotation in der zugrundeliegenden Typ-Spezifikation haben müssen. Wir brauchen nun eine effiziente operationale Methode um Wohlgetyptheit von Wertetermen zu checken. Dazu benutzen wir, daß wohlgetypte Werteterme einen kleinsten Typ haben. Wenn wir kleinste Typen für Werteterme ableiten können, so haben wir also gleichzeitig einen Test auf Wohlgetyptheit. Beim Ableiten von kleinsten Typen gehen wir rekursiv vor. Zunächst leiten wir die kleinsten Typen für die Argumente ab. Dann suchen wir die "genaueste" Deklaration für das Top-Funktionssymbol, die auf diese Typen paßt. Dazu benutzen wir den sogenannten "kleinsten Codomain".

Der *kleinste Codomain lcd* eines Funktionssymbols f (mit Stelligkeit n) bezüglich der Typterme $\sigma_1, \cdots, \sigma_n$ ist definiert als

$$lcd(f,(\sigma_1,\ldots,\sigma_n)) := \min_{\leq}\{\theta\tau \mid f : \tau_1 \cdot\cdot \tau_n \to \tau \text{ ist in } \mathcal{T}, \text{ und } \sigma_1 \leq \theta\tau_1, \ldots, \sigma_n \leq \theta\tau_n\}$$

Die partielle Funktion $\sigma^P[\cdot]$ berechnet den *kleinsten Typ eines Werteterms unter einem Präfix P* :

$$\begin{aligned} \sigma^P[x] &= Px \\ \sigma^P[f(\vec{s})] &= lcd(f, \sigma^P[s_1], ..., \sigma^P[s_n]) \end{aligned}$$

Satz 3.2.5 *In Typ-Spezifikationen gilt: s ist wohlgetypt unter dem Präfix P, genau dann wenn $\sigma^P[s]$ existiert.*

Nun können wir Wohlgetyptheit von Termen leicht über das Ableiten des kleinsten Typs entscheiden. Als nächstes liften wir Wohlgetyptheit auf Constraint-Systeme. Da wir beim logischen Programmieren nur an den Wertevariablen interessiert sind, betrachten wir nur Gleichungen und Containments.

Eine *Gleichung* $s \doteq t$ *heißt wohlgetypt unter einem Präfix* P, falls ein Typterm σ existiert mit $P \vdash s{:}\sigma$ und $P \vdash t{:}\sigma$. Ein *Containment* $s{:}\sigma$ *heißt wohlgetypt unter einem Präfix* P, falls ein Typterm τ existiert mit $P \vdash s{:}\tau$ und $\sigma \leq \tau$. Mit $E\&M$ bezeichnen wir ein Constraint-System, daß aus einer Menge E von Gleichungen und einer Menge M von Memberships besteht. $E\&M$ *heißt wohlgetypt unter einem Präfix* P, falls alle Gleichungen aus E und alle Containments aus M wohlgetypt unter P sind. Diese Definition von Wohlgetyptheit für Constraints ist sinnvoll, da sie eine notwendige Voraussetzung für die Erfüllbarkeit der Constraints unter der gegebenen Typ-Spezifikation ist.

Um zu gewährleisten, daß Typterme, die kein $\bot$ enthalten, keine leere Denotation haben, werden wir an Typ-Spezifikationen die Forderung der "Bewohntheit" stellen.

Wir nennen einen *Typterm* σ *bewohnt*, falls ein Werteterm s und eine Substitution θ existieren, so daß $\vdash s{:}\theta\sigma$. Die *Normalform* $NF[\sigma]$ *eines Typterms* σ erhalten wir, indem wir alle unbewohnten Teilterme von σ durch $\bot$ ersetzen. In [Sm 89a] wird gezeigt, daß Normalformen berechenbar sind. Wir nennen einen *Typterm* σ *normal*, falls $\sigma = NF[\sigma]$ gilt. Ein *Präfix* heißt *bewohnt*, falls für alle x aus $\mathcal{D}P$ Px bewohnt ist. Die *Normalform* $NF[P]$ *eines Präfix* P bilden wir, indem wir $NF[P]x = NF[Px]$ für alle x aus $\mathcal{D}P$ setzen. Ein *Präfix heißt normal*, falls $P = NF[P]$ gilt. Wir sagen eine *Typ-Spezifikation* ist *voll bewohnt*, falls alle Typterme, die kein $\bot$ enthalten, bewohnt sind. In [Sm 89a] wurde gezeigt, daß die volle Bewohntheit von Typ-Spezifikationen entscheidbar ist.

Später werden wir häufig sogenannte "Inklusions-Systeme" lösen müssen. Dies ist insbesondere bei der Typ-Inferenz (siehe Abschnitt 3.6) der Fall. Wir führen hier deshalb noch einige Grundbegriffe ein.

Ein *Inklusions- oder Membership-System* ist eine möglicherweise leere Konjunktion von Inklusionen. Wir stellen Inklusions-Systeme oft in der Vektornotation $\vec{\sigma} \sqsubseteq \vec{\tau}$ dar. Ein *oberer Matcher* eines Inklusions-Systems $\vec{\sigma} \sqsubseteq \vec{\tau}$ ist eine Substitution θ auf Sortenvariablen, so daß $\vec{\sigma} \leq \theta\vec{\tau}$. Wir erweitern die partielle Ordnung $\leq$ folgendermaßen auf Sortensubstitutionen: $\theta \leq \psi$, falls $\theta\alpha \leq \psi\alpha$ für alle $\alpha \in \mathcal{D}\theta$. Wenn ein Inklusions-System einen oberen Matcher hat, so hat es auch einen (bezüglich $\leq$) *kleinsten oberen Matcher* und dieser ist berechenbar [Sm 89a]. Wir schreiben auch $LUM[\vec{\sigma} \sqsubseteq \vec{\tau}]$ für den kleinsten oberen Matcher des Inklusions-Systems $\vec{\sigma} \sqsubseteq \vec{\tau}$.

3.3 Feature-Spezifikationen

Typ-Spezifikationen ermöglichen eine Definition von Wohlgetyptheit bei der kein Unterschied zwischen der Semantik von Feature-Typen (Features, implizite Kon-

struktoren) und Konstruktor-Typen (explizite Konstruktoren) gemacht wird. In diesem Abschnitt erweitern wir Typ-Spezifikationen zu "Feature-Spezifikationen". Diese stellen den Zusammenhang von Feature-Typen, Features und impliziten Konstruktoren her. Wir geben ein Standardmodell für Feature-Spezifikationen an, dessen Resultate auf beliebige andere Modelle übertragbar sind. Unser Constraint-Solver wird Constraints im Standardmodell einer gegebenen Feature-Spezifikation lösen.

Wir wollen nun Typ-Spezifikationen zu Feature-Spezifikationen verschärfen. Feature-Typen sollen eine Vererbungshierarchie denotieren, in der jeder Feature-Typ, zusätzlich zu den eigenen Features, die Features seiner Obertypen erbt. Die Typ-Hierarchie wird dabei, wie gehabt, durch die Inklusionen der Typ-Spezifikation definiert. Um zu gewährleisten, daß nur Feature-Typen Features haben, fordern wir die Unvergleichbarkeit von Konstruktor- und Feature-Typen. Wie gewöhnlich erhalten Konstruktor-Typen ihre Bedeutung im Standard-Grundtermmodell über die Deklarationen der expliziten Konstruktoren. Dabei darf jeder explizite Konstruktor nur eine Deklaration haben, damit er eindeutig einem Konstruktor-Typ zugeordnet ist.

Wir geben Feature-Typen eine Semantik, indem wir sie mittels sogenannter "impliziter Konstruktoren" auf Konstruktor-Typen reduzieren. Dazu ordnen wir jedem "minimalen" Feature-Typ ξ (das heißt: ξ hat keine Untertypen) genau einen impliziten Konstruktor $\bar{\xi}$ zu. Die Stelligkeit von $\bar{\xi}$ entspricht gerade der Anzahl der Features von ξ. Die Deklaration von $\bar{\xi}$ ergibt sich aus den dazugehörigen Feature-Deklarationen. Nichtminimale Feature-Typen sind über die Konstruktoren ihrer Untertypen definiert. Im Gegensatz zu Konstruktoren dürfen Features mehr als eine Deklaration haben. Dies ist sinnvoll, da sie nur als Selektoren dienen. Die Stellen der Features innerhalb der impliziten Konstruktoren werden über eine totale Ordnung auf den Features festgelegt. Sogenannte "Projektionsgleichungen" stellen den direkten Zusammenhang zwischen Features und impliziten Konstruktoren her.

Schließlich stellen wir noch die Bedingung, daß nichtminimale Feature-Typen bewohnt sein müssen. Wie wir noch sehen werden, würde ohne diese Bedingung eine "gelöste Form" nicht immer eine Lösung beschreiben.

Gegeben sei eine Typ-Spezifikation $\mathcal{T}$ und eine totale Ordnung $\leq_{fea}$ auf den Feature-Symbolen. Wir nennen dann f ein *Feature von* ξ, falls es eine Deklaration $f{:}\zeta \to \sigma$ in $\mathcal{T}$ gibt und $\xi \leq \zeta$. Wir sagen f *ist das i-te Feature von* ξ, falls f ein Feature von ξ ist und $\{g \mid g \text{ist Feature von} \xi \text{und} g \leq_{fea} f\}$ genau i Elemente enthält.

Eine *Feature-Spezifikation* $\mathcal{F}$ über einer Constraint-Sprache $\mathcal{L}(\Sigma)$ besteht aus

- einer voll bewohnten Typ-Spezifikation $\mathcal{T}$
- einer totalen Ordnung $\leq_{fea}$ auf der Menge der Feature Symbole
- einer Menge von Gleichungen (*Projektionen*)

und erfüllt die folgenden Anforderungen:

1. Feature-Typen und Konstruktor-Typen sind unvergleichbar bezüglich der in $\mathcal{T}$ definierten Ordnung $\leq$ auf Termen.

2. Für alle Features f existiert mindestens eine Funktionsdeklaration der Form $f{:}\,\xi \to \sigma$ und σ ist ein Sortengrundterm.

3. Für alle Konstruktoren existiert genau eine Funktionsdeklaration.

4. Zu jedem minimalen Feature-Typ ξ gehört genau ein impliziter Konstruktor $\bar{\xi}$ und umgekehrt. Sind $f_1 \leq_{fea} \cdots \leq_{fea} f_n$ die Features von ξ, so hat $\bar{\xi}$ die Deklaration
$$\bar{\xi}{:}\, lcd(f_1, \xi) \cdots lcd(f_n, \xi) \to \xi\,.$$

5. Zu jedem Tripel $(\xi, \bar{\xi}, f)$, wobei ξ minimaler Feature-Typ mit implizitem Konstruktor $\bar{\xi}$ ist, und f das i-te Feature von ξ ist, gehört eine Projektion
$$f(\bar{\xi}(x_1, \ldots, x_n)) = x_i.$$
Umgekehrt ist jede Projektion auch einem solchen Tripel zuzuordnen.

6. Zu jedem Feature-Typ ξ existiert ein minimaler Feature-Typ ζ mit $\zeta \leq \xi$ und $lcd(f, \xi) = lcd(f, \zeta)$ für alle Features f von ξ.

Allgemeine Annahme. *Im folgenden sei $\mathcal{F}$ eine feste Feature-Spezifikation über unserer Constraint-Sprache $\mathcal{L}(\Sigma)$.*

Die Projektionen von $\mathcal{F}$ charakterisieren eine Gleichheit auf den Wertetermen. Wenn wir sie von links nach rechts richten, so entsteht ein Rewriting-System. Die dazugehörige Rewriting-Relation $\to_{\mathcal{F}}$ auf Wertetermen definieren wir wie folgt. Es gilt $s \to_{\mathcal{F}} t$, falls s einen Teilterm $f(\bar{\xi}(s_1, \ldots, s_n))$ enthält, und t aus s entsteht, indem dieser Teilterm durch s_i ersetzt wird, wobei f das i-te Feature von ξ ist. Mit $\downarrow_{\mathcal{F}} t$ bezeichnen wir die Normalform von t bezüglich $\to_{\mathcal{F}}$. Die folgende Proposition sagt aus, daß $\to_{\mathcal{F}}$ gerade die Gleichheit auf Wertetermen spezifiziert, die in allen Modellen einer Feature-Spezifikation gilt.

Satz 3.3.1 *Sei $\mathcal{A}$ ein Modell der Feature-Spezifikation $\mathcal{F}$. Dann gilt $s \doteq t$, falls $\downarrow_{\mathcal{F}} s = \downarrow_{\mathcal{F}} t$.*

Im folgenden wollen wir Constraint-Systeme über einer gegebenen Feature-Spezifikation $\mathcal{F}$ lösen. Dabei werden wir jedoch keine beliebigen Modelle von $\mathcal{F}$ zugrundelegen ("Open-World-Semantik"), sondern uns auf ein Standardmodell beschränken, dessen Ergebnisse auf alle Modelle von $\mathcal{F}$ übertragbar sind. Der Domain dieses Standardmodells ist gerade die Menge der wohlgetypten Grundwerteterme, welche keine Features enthalten. Die Gleichheit auf Wertetermen, die durch die Projektionen von $\mathcal{F}$ charakterisiert wird, werden wir mit Hilfe unserer Rewriting-Relation $\to_{\mathcal{F}}$ erzeugen.

Zu einer Feature-Spezifikation $\mathcal{F}$ konstruieren wir folgendermaßen ein

1. $\mathcal{S}^{\mathcal{I}}$ ist die Menge aller Sortengrundterme.
2. Es gilt $\sigma \leq_{\mathcal{I}} \tau$ genau dann, wenn $\sigma \leq \tau$, und $\tau \in \mathcal{S}^{\mathcal{I}}$.
3. $\mathcal{V}^{\mathcal{I}}$ ist die Menge aller wohlgetypten Grundwerteterme, die nur aus Konstruktoren bestehen.
4. Es gilt $s:_{\mathcal{I}} \sigma$ genau dann, wenn $\vdash s:\sigma$ und $\sigma \in \mathcal{S}^{\mathcal{I}}$.
5. $\xi^{\mathcal{I}}(\vec{\sigma}) := \xi(\vec{\sigma})$.
6. $g^{\mathcal{I}}(\vec{s}) :=\downarrow_{\mathcal{F}} g(\vec{s})$, falls $g(\vec{s})$ wohlgetypt ist.

Satz 3.3.2 *Sei $\mathcal{I}$ die zur Feature-Spezifikation $\mathcal{F}$ gehörige Standardinterpretation. Dann gilt: Die Standardinterpretation $\mathcal{I}$ ist ein Modell von $\mathcal{F}$. Wenn ein Constraint C in $\mathcal{I}$ erfüllbar ist, so ist es auch in jedem anderen Modell von $\mathcal{F}$ erfüllbar.*

Dieser Satz impliziert, daß es genügt, Constraints über einer Feature-Spezifikation $\mathcal{F}$ nur im Standardmodell $\mathcal{I}$ zu lösen. Die Ergebnisse sind dann auf beliebige Modelle von $\mathcal{F}$ übertragbar.

3.4 Lösen von Feature-Constraints

Wir möchten nun Feature-Constraints über einer Feature-Spezifikation $\mathcal{F}$ in deren Standardmodell $\mathcal{I}$ lösen. Dabei beschränken wir uns auf $E\&M$-Systeme, die unter einem Präfix P wohlgetypt sind. Unser Constraint-Solver soll das Startsystem solange transformieren bis ein "gelöstes System" erreicht ist. Dazu beschreiben wir zunächst gelöste Systeme und ihr Verhältnis zu den Lösungen in $\mathcal{I}$. Anschliessend geben wir den Constraint-Solving-Algorithmus in Form eines Regelsystems an, und zeigen seine Korrektheit und Vollständigkeit.

Bei der Einführung unserer Constraint-Sprache definierten wir die Lösungen von Constraints in einer Interpretation mit Hilfe von Belegungen. Wir abstrahieren diese Darstellung nun zu sogenannten "Unifikatoren", welche eine kompaktere Repräsentierung der Lösungen von $E\&M$-Constraints in $\mathcal{I}$ ermöglichen. Diese Darstellung kann jedoch noch implizite Konstruktoren enthalten. In der endgültigen Lösung sollen aber die impliziten Konstruktoren versteckt bleiben, da sie ja nur ein Hilfsmittel waren, um die Bedeutung von Feature-Typen zu definieren. Wir brauchen also eine noch abstraktere Lösungsrepräsentierung, die sogenannte "gelöste Form", bei der implizite Konstruktoren durch Gleichungen mit Features ersetzt werden. Dies hat den zusätzlichen Vorteil, daß nur die Featurewerte eines impliziten Konstruktors, die wirklich belegt sind, in der Lösung auftauchen.

Zyklische Lösungen mit unendlichen Termstrukturen, wie sie zum Beispiel zur Beschreibung von Aït-Kaci's ξ-Termen [AN 86] angebracht wären, können im Rahmen dieser Arbeit nicht betrachtet werden. Das zugrundeliegende Modell müßte dann initial in der Kategorie der "continuous models" [Mö 85] sein.

Wir führen nun "Unifikatoren" zur kompakten Darstellung von Lösungen in $\mathcal{I}$ ein. Diese abstrahieren Belegungen insofern, daß die Bindung von Wertevariablen an Werteterme nicht erzwungen ist. Falls es zur Lösung ausreicht, ist auch eine Bindung von Wertevariablen an Typen (und damit Mengen von Wertetermen) erlaubt.

Ein *Unifikator* ψ_P besteht aus einem normalen und bewohnten Präfix P und einer endlichen und idempotenten Substitution ψ auf Wertevariablen. Ein Unifikator ψ_P heißt *Unifikator eines Constraint-Systems* $E\&M$, falls gilt:

1. $P \vdash \psi M$.
2. Das Gleichungssystem ψE ist wohlgetypt unter P.
3. Das Gleichungssystem ψE ist *trivial*, das heißt es besteht aus Gleichungen der Form $s \doteq s$.

Mit $U[E\&M]$ bezeichnen wir die Menge aller Unifikatoren von $E\&M$. Wir beschreiben nun das Verhältnis zwischen Unifikatoren und Belegungen. Aus einer Belegung δ konstruieren wir einen Unifikator $\psi_P(\delta)$, indem wir die Zuordnung von Variablen zu Wertetermen als Substitution ψ auffassen. Der dazugehörige Präfix P entsteht dadurch, daß Variablen mit dem kleinsten Typ des ihnen durch δ zugeordneten Werteterms qualifiziert werden. Umgekehrt können wir die Substitution ψ eines Unifikators ψ_P als eine Belegung $\delta(\psi_P)$ interpretieren. Die Variablen, die außerhalb des Domains von ψ liegen, die aber im Präfix P mit einen Typ σ qualifiziert sind, belegen wir in δ mit einem beliebigen Grundterm vom Typ σ (dieser existiert, da $\mathcal{F}$ voll bewohnt ist). Der folgenden Satz sagt aus, daß Unifikatoren eine zu Belegungen

äquivalente Repräsentierung von Lösungen darstellen.

Satz 3.4.1 *Es gilt: Aus* $\delta \in \mathcal{I}[\![E\&M]\!]$ *folgt, daß* $\psi_P(\delta) \in U[E\&M]$ *ist. Umgekehrt folgt aus* $\psi_P \in U[E\&M]$ *auch, daß* $\delta(\psi_P) \in \mathcal{I}[\![E\&M]\!]$ *ist.*

Oft sind eigentlich nur gewisse Variablen einer Lösung interessant. Gegeben eine Menge V von Wertevariablen, so bezeichnen wir mit $\psi|_V$ die Einschränkung von ψ auf die Variablen aus V. Wir definieren die *Einschränkung der Unifikatoren eines Constraint Systems* auf V als

$$U^V[E\&M] := \{\psi_P \mid \psi = \psi^*|_V, \text{ wobei } \psi_P^* \in U[E\&M]\}.$$

Ein Werteterm heißt *kanonisch*, falls er keine Features enthält. Werteterme der Form $f(x)$ nennen wir auch *Quasivariable*.

Eine *quasi-gelöste Form* θ_P besteht aus einem normalen und bewohnten Präfix P und einem Gleichungssystem θ. Dabei hat θ die Form:

$$x_1 \doteq s_1, \ldots, x_n \doteq s_n, f_1(y_1) \doteq t_1, \ldots, f_n(y_n) \doteq t_n$$

wobei zum einen die x_i von den y_i verschieden, und zum anderen sowohl die x_i als auch die $f_j(y_i)$ untereinander paarweise verschieden sind.

Um zu entscheiden, ob eine quasi-gelöstes Form θ_P eine zyklische Lösung beschreibt, führen wir eine Abhhängigkeitsrelation $\rightarrow_\theta$ auf den Variablen ein. Es gilt $x \rightarrow_\theta y$, falls es eine Gleichung $s \doteq t$ in θ gibt, so daß x in s, und y in t vorkommt. Eine *gelöste Form* θ_P ist eine quasi-gelöste Form mit azyklischer Abhängigkeitsrelation $\rightarrow_\theta$. Der folgende Satz sagt aus, daß eine gelöste Form auch wirklich eine Lösung beschreibt.

Satz 3.4.2 *Sei θ_P eine gelöste Form. Dann gilt*

$$U[P \,\&\, \theta] \neq \emptyset.$$

Diese Aussage wird durch die Bewohntheit der nichtminimalen Feature-Typen einer

ben sei die folgende Pseudo-Feature-Spezifikation, die alle Bedingungen für Feature-Spezifikationen erfüllt, bis auf die Bewohntheit der nichtminimalen Feature-Typen.

$$\xi \sqsubseteq \zeta \ , \ f{:}\zeta \rightarrow \eta \ , \ f{:}\xi \rightarrow \psi \ , \ \psi \sqsubseteq \eta \ , \ c_1{:}\rightarrow \eta \ , \ c_2{:}\rightarrow \psi$$

Dann ist $Px = \zeta \,\&\, f(x) = c_1$ eine gelöste Form, hat aber keinen Unifikator.

Wir definieren nun, wann gelöste Formen und $E\&M$-Systeme zusammengehören. Eine gelöste Form θ_P heißt *gelöste Form zu einem Constraint-System $E\&M$*, falls gilt

1. $P.f(s) \rightarrow_{ut} P \cup \{x{:}\sigma^P[f(s)]\}.x.f(s) \doteq x$
   wobei x eine neue Variable ist
2. $P.c(\ldots, s_i, \ldots) \rightarrow_{ut} Q.c(\ldots, t_i, \ldots).E$
   falls $P.s_i \rightarrow_{ut} Q.t_i.E$ gilt

Abbildung 3.1: Auffalten von Wertetermen

Der Constraint Solver arbeitet in zwei Phasen:

Auffalten Gestartet wird mit einem $E\&M$-System, das wohlgetypt unter einem normalen und bewohnten Präfix P ist, und in dem keine impliziten Konstruktoren vorkommen. Dieses Constraint System wird durch Auffalten in ein äquivalentes Constraint System $E'\&M'$ in getrimmter Form überführt, das unter einem Präfix P' wohlgetypt ist.

Auflösen Das getrimmte Constraint System $E'\&M'$ (wohlgetypt unter P'), wird in eine ihm zugehörige gelöste Form θ_Q transformiert (falls diese existiert).

Wir betrachten zunächst das Auffalten von $E\&M$-Systemen. Dabei müssen alle Werteterme, die in $E\&M$ vorkommen darauf untersucht werden, ob sie einen "Feature-Subterm" enthalten. Dies ist ein Subterm, der als Topsymbol ein Feature hat. Ist dies bei einem Werteterm der Fall, so muß dieser Feature-Subterm durch eine neue Wertevariable ersetzt werden. Zusätzlich erweitern wir E um eine Gleichung, mit der diese neue Variable an den zu ersetzenden Subterm gebunden wird. Dieser Prozeß wird solange iteriert, bis ein getrimmtes Constraint-System entstanden ist.

Wir geben den Algorithmus zum Auffalten von $E\&M$-Systemen in Regelform an. Zunächst müssen wir das Auffalten von Wertetermen definieren. Die Basisoperation dazu heißt "unfold term". Sie bildet Paare $P.t$, die aus einem Werteterm t (der einen Feature-Subterm enthält) und einem normalen und bewohnten Präfix P bestehen, auf Tripel der Form $P'.t'.E$ ab. Dabei ist t' der Werteterm, der entsteht, wenn man den Feature-Subterm von t auffaltet. Die Gleichung E bindet die beim Auffalten generierte Variable an den aufgefalteten Feature-Subterm, und P' erweitert den Präfix P um eben diese neue Variable. Die Operation *unfold term* wird durch die Relation $\rightarrow_{ut}$ definiert (siehe Abbildung 3.1).

Wir wollen nun $E\&M$-Systeme, die unter einem Präfix P wohlgetypt sind, durch Auffalten in getrimmte Form bringen. Dazu wird auf alle Werteterme, die in den Gleichungen oder Containments von $E\&M$ vorkommen, sukzessive die Operation "unfold term" angewandt. Einzige Ausnahmen sind Quasivariable, die auf der linken Seite einer Gleichung stehen. In jedem solchen Schritt werden der Präfix P und das Gleichungssystem E wie oben beschrieben erweitert. Der Prozeß wird solange iteriert, bis das erweiterte Constraint-System getrimmt ist. Die Grundoperation

1. $P.s \doteq t \,\&\, C \rightarrow_{uc} Q.s' \doteq t \,\&\, C \,\&\, E$
   falls s keine Quasivariable ist,
   und $P.s \rightarrow_{ut} Q.s'.E$ gilt
2. $P.s \doteq t \,\&\, C \rightarrow_{uc} Q.s \doteq t' \,\&\, C \,\&\, E$
   falls $P.t \rightarrow_{ut} Q.t'.E$ gilt
3. $P.t{:}\sigma \,\&\, C \rightarrow_{uc} Q.t'{:}\sigma \,\&\, C \,\&\, E$
   falls $P.t \rightarrow_{ut} Q.t'.E$ gilt

Abbildung 3.2: Auffalten von Constraint-Systemen

"unfold constraint" ist als Rewriting-Relation $\rightarrow_{uc}$ auf Paaren der Form $P.C$ definiert, wobei P ein normaler und bewohnter Präfix und C eine Menge von Constraints (Gleichungen oder Memberships) ist (siehe Abbildung 3.2).

Ein $P.E\&M$-System (wobei $E\&M$ unter P wohlgetypt ist) wird durch Anwendung von $\rightarrow_{uc}$ solange transformiert, bis ein getrimmtes Constraint-System $P'.E' \,\&\, M'$ entsteht, wobei $E' \,\&\, M'$ unter P' wohlgetypt ist. Die nächste Proposition zeigt, daß eine solche Transformation die Lösungsmenge des Constraint-Systems invariant läßt.

Satz 3.4.3 *Falls $E\&M$ wohlgetypt unter einem normalen und bewohnten Präfix P ist, $V \subseteq \mathcal{D}P$, und*

$$P.E \,\&\, M \rightarrow_{uc} P'.E' \,\&\, M'$$

gilt, so ist $E' \,\&\, M'$ wohlgetypt unter P', und es gilt

$$U^V[P \,\&\, E\&M] = U^V[P' \,\&\, E' \,\&\, M'].$$

Im folgenden Satz zeigen wir, daß eine Transformation in ein getrimmtes System immer möglich ist. Dabei ist die Reihenfolge bei der Anwendung der Regeln egal. Falls keine Regel mehr anwendbar ist, liegt ein getrimmtes Constraint-System vor.

Satz 3.4.4 *$\rightarrow_{uc}$ terminiert und ist konfluent modulo Variablenumbenennung. Die Normalformen von $\rightarrow_{uc}$ sind getrimmt.*

Damit ist das Auffalten von $E\&M$-Systemen zu äquivalenten getrimmten Constraint-Systemen berechenbar. Wir wenden uns nun dem Auflösen von getrimmten Constraint-Systemen zu. Die Überführung eines Constraint-Systems in seine gelöste Form ist natürlich nur möglich falls diese existiert. Wir werden unseren Constraint-Solver wiederum in Form eines Regelsystems angeben. Zunächst geben wir eine Basisoperation $\rightarrow_m$ an, die es uns ermöglicht Containments aufzulösen, indem diese in den aktuellen Präfix "hineinmultipliziert" werden. Der so "verschärfte" Präfix

1. $P \,\&\, x{:}\sigma.x{:}\tau \,\&\, M \;\rightarrow_m\; P \,\&\, x{:}(\sigma \sqcap \tau).M$
2. $P.c(\vec{s}){:}\sigma \,\&\, M \;\rightarrow_m\; P.\vec{s}{:}\vec{\sigma} \,\&\, M$

   falls $c{:}\vec{\tau} \rightarrow \tau$ die Funktionsdeklaration von c ist
   und $\vec{\sigma} = gd(c, \sigma)$

Abbildung 3.3: Reduktionsregeln für Containments

beschreibt dann die gleiche Lösungsmenge, wie der ursprüngliche Präfix zusammen mit dem Containment. Um Containments mit zusammengesetzten Wertetermen auf ihre Argumente reduzieren zu können, führen wir die folgende Definition ein. Sei c ein Konstruktor mit Funktionsdeklaration $c{:}\vec{\tau} \rightarrow \tau$. Der *größte Domain* von c bezüglich eines Typs σ ist definiert als

$$gd(c, \sigma) := max_{\geq}\{\theta\vec{\tau} \mid \theta\tau \leq \sigma\},$$

wobei θ eine Substitution auf Sortenvariablen ist. Man versucht also für die Argumente eines Terms t mit Topsymbol c möglichst allgemeine Typen abzuleiten, so daß $t{:}\sigma$ gewährleistet ist. Man beachte, daß dies nicht mit Features als Topsymbol funktioniert, da diese mehrere Funktionsdeklarationen haben können. Durch vorheriges Auffalten können wir aber garantieren, daß in Containments keine Features mehr vorkommen. Nun ist es leicht die *Reduktionsregeln* $\rightarrow_m$ *für Containments* anzugeben. Diese operieren auf Paaren $P.M$, wobei P ein Präfix, und M eine Menge von Containments ist (siehe Abbildung 3.3). Gegeben ein Präfix P und eine Menge von Containments M,so wird M durch sukzessive Anwendungen von $\rightarrow_m$ ganz in den Präfix P "hineinmultipliziert". Der so entstehende "verschärfte" Präfix ist äquivalent zu $P \,\&\, M$.

Nun haben wir die Voraussetzungen, um $E\&M$-Systeme auflösen zu können.

1. $P.\theta.x \doteq x \;\&\; C \;\rightarrow_c\; P.\theta.C$
2. $P.\theta.c(\vec{s}) \doteq c(\vec{t}) \;\&\; C \;\rightarrow_c\; P.\theta.C \;\&\; \vec{s} \doteq \vec{t}$
3. $P.\theta.s \doteq x \;\&\; C \;\rightarrow_c\; P.\theta.C \;\&\; x \doteq s$
   falls s keine Variable oder Quasivariable ist
4. $P.\theta.x \doteq s \;\&\; C \;\rightarrow_c\; Q.x = s \;\&\; \theta.\{x/s\}C$
   falls $x \notin \mathcal{V}s$ und $P.s\colon Px \;\rightarrow_m^*\; Q.\emptyset$ gilt
5. $P.\theta.f(x) \doteq t \;\&\; C \;\rightarrow_c\; Q.f(x) = t \;\&\; \theta.\{f(x)/t\}C$
   falls $x \notin \mathcal{V}s$ und $P.t\colon lcd(f, Px) \;\rightarrow_m^*\; Q.\emptyset$ gilt
6. $P.\theta.s\colon \sigma \;\&\; C \;\rightarrow_c\; Q.\theta.C$
   falls $P.s\colon \sigma \;\rightarrow_m^*\; Q.\emptyset$ gilt

Abbildung 3.4: Reduktionsregeln für Constraints

wendungen geben kann.

Satz 3.4.5 $\rightarrow_c$ *terminiert und ist konfluent modulo Variablenumbenennung.*

Weiterhin gilt, daß unser Constraint-Solver vollständig und korrekt arbeitet.

Satz 3.4.6 [Korrektheit] *Aus* $V \subseteq \mathcal{D}P$ *und*

$$P.E.M \;\rightarrow_c\; P'.E'.M'$$

folgt

$$U^V[P \;\&\; E\&M] = U^V[P' \;\&\; E' \;\&\; M'].$$

Satz 3.4.7 [Vollständigkeit] *Es gilt:* $P \;\&\; E\&M$ *ist erfüllbar in* $\mathcal{I}$ *(unifizierbar), genau dann wenn es* Q, θ *mit* $P.\emptyset.E\&M \;\rightarrow_c^*\; Q.\theta.\emptyset$ *gibt.*

Dies ist eine Verallgemeinerung des Resultats von [Sm 89a] auf Feature-Spezifikationen. Wir haben nun einen berechenbaren, vollständigen, und korrekten Constraint-Solver für $E\&M$-Systeme über Feature-Spezifikationen.

3.5 Logisches Programmieren mit Feature-Constraints

Wir wollen in diesem Abschnitt zeigen, wie logische Programme über Feature-Constraint-Sprachen aussehen, welche Semantik sie haben, und wie sie abgearbeitet

werden. Dabei greifen wir im wesentlichen auf die Ergebnisse von [Sm 89a] und [HS 88] zurück.

Zunächst definieren wir eine relationale Erweiterung unserer Feature-Constraint-Sprache, deren Semantik auf der in Abschnitt 3.3 definierten Standard-Interpretation beruht. "Feature-Programme" bestehen dann aus definiten Klauseln über einer solchen relationalen Feature-Constraint-Sprache. Wie schon bei Feature-Spezifikationen, werden wir auch für Feature-Programme ein Standardmodell angeben. Unser Interpreter soll in diesem Standardmodell arbeiten und greift auf den Constraint-Solver aus Abschnitt 3.4 zurück. Dieser akzeptiert jedoch nur wohlgetypte Constraints. Um dies zu garantieren liften wir Wohlgetyptheit auch auf relationale Atome und Klauseln. Feature-Programme können dann mit Hilfe einer einzigen starken Regel, der sogenannten "Goal-Reduktion", abgearbeitet werden.

Zunächst erweitern wir Feature-Constraint-Sprachen um relationale Atome, aussagenlogische Verknüpfungen, und Quantoren.

Allgemeine Annahme. *Im folgenden sei $\mathcal{R}$ eine feste Menge von Relationssymbolen.*

Wir definieren die *relationale Erweiterung $\mathcal{R}(\mathcal{L})$ der Feature-Constraint-Sprache $\mathcal{L}$* folgendermaßen.

- Die Constraints von $\mathcal{R}(\mathcal{L})$ sind durch die folgenden Regeln definiert.
  1. Jedes Constraint von $\mathcal{L}$ ist auch ist auch ein Constraint von $\mathcal{R}(\mathcal{L})$.
  2. Ist r ein Relationssymbol aus $\mathcal{R}$ und $\vec{x}$ ein Tupel paarweise disjunkter Wertevariablen (wobei die Stelligkeit von r der Länge von $\vec{x}$ entspricht), so ist das *Atom* $r(\vec{x})$ ein $\mathcal{R}(\mathcal{L})$-Constraint.
  3. Die *leere Konjunktion* $\emptyset$ ist ein $\mathcal{R}(\mathcal{L})$-Constraint. Sind F und G $\mathcal{R}(\mathcal{L})$-Constraints, so sind auch die *Konjunktion* $F \,\&\, G$ und die *Implikation* $F \rightarrow G$ $\mathcal{R}(\mathcal{L})$-Constraints.
  4. Sei x eine Wertevariable und F ein $\mathcal{R}(\mathcal{L})$-Constraint. Dann ist auch die *existentielle Quantifizierung* $\exists x.F$ ein $\mathcal{R}(\mathcal{L})$-Constraint.
- Wir konstruieren eine $\mathcal{R}(\mathcal{L})$-Interpretation $\mathcal{A}$, basierend auf einer $\mathcal{L}$-Interpretation $\mathcal{I}$, indem wir den Domain von $\mathcal{I}$ übernehmen, und jedem Relationssymbol r aus $\mathcal{R}$ eine Relation $r^{\mathcal{A}}$ auf $\mathcal{D}^{\mathcal{I}}$ mit der gleichen Stelligkeit zuordnen.
- Die Lösungen von $\mathcal{R}(\mathcal{L})$-Constraints bezüglich einer $\mathcal{R}(\mathcal{L})$-Interpretation $\mathcal{A}$ mit Basis $\mathcal{I}$, sind wie folgt definiert.
  1. $[\![\phi]\!]^{\mathcal{A}} := [\![\phi]\!]^{\mathcal{I}}$, falls ϕ ein $\mathcal{L}$-Constraint ist.
  2. $[\![r(\vec{x})]\!]^{\mathcal{A}} := \{\alpha \in ASS^{\mathcal{A}} \mid \alpha(\vec{x}) \in r^{\mathcal{A}}\}$.
  3. $[\![\emptyset]\!]^{\mathcal{A}} := ASS^{\mathcal{A}}$,
     $[\![F \,\&\, G]\!]^{\mathcal{A}} := [\![F]\!]^{\mathcal{A}} \cap [\![G]\!]^{\mathcal{A}}$,
     $[\![F \rightarrow G]\!]^{\mathcal{A}} := (ASS^{\mathcal{A}} - [\![F]\!]^{\mathcal{A}}) \cup [\![G]\!]^{\mathcal{A}}$.

4. $[\![\exists x.F]\!]^{\mathcal{A}} := \{\alpha \in ASS^{\mathcal{A}} \mid$ es gibt ein $\beta \in [\![F]\!]^{\mathcal{A}}$ so, daß für alle y aus $\mathcal{V}F$ gilt: $y = x$ oder $\alpha(y) = \beta(y)\}$.

Ebenso wie $\mathcal{L}$ nennen wir $\mathcal{R}(\mathcal{L})$ eine "Constraint-Sprache". Beide Sprachen stellen syntaktische Objekte zur Verfügung, die es ermöglichen, Wertemengen für die darin enthaltenen Variablen in einer gegebenen Interpretation zu beschränken. Die Definitionen für "gültig", "erfüllbar", "V-Lösungen", und so weiter, die schon für Feature-Constraint-Sprachen gemacht wurden, können direkt auf ihre relationalen Erweiterungen übertragen werden.

Allgemeine Annahme. *Im folgenden sei $\mathcal{R}(\mathcal{L})$ eine feste relationale Erweiterung der Feature-Constraint-Sprache $\mathcal{L}(\Sigma)$.*

Wir wollen nun beschreiben, wie logische Programme über einer relationalen Feature-Constraint-Sprache aussehen. Dazu führen wir die Begriffe "Feature-Atom" und "Feature-Klausel" ein.

Ein *Feature-Atom* ist ein $\mathcal{R}(\mathcal{L})$-Constraint der Form

$$\exists \vec{x}.(\vec{x} = \vec{s}\ \&\ r(\vec{x})),$$

wobei $\vec{s}$ ein Tupel von *kanonischen* Wertetermen ist, und $\vec{x}$ und $\vec{s}$ variablendisjunkt sind. Da die Bedeutung eines Feature-Atoms unabhängig von den speziellen Variablen $\vec{x}$ ist, benutzen wir auch die Abkürzung $r(\vec{s})$. Im folgenden verwenden wir die Buchstaben A, B für Feature-Atome, und F, G für Konjunktionen von Feature-Atomen.

Eine *Feature-Klausel* ist ein $\mathcal{R}(\mathcal{L})$-Constraint der Form

$$P\ \&\ E\ \&\ A_1\ \&\ \ldots\ \&\ A_n \rightarrow B,$$

wobei $n \geq 0$ und A_1, ..., A_n, B Feature-Atome sind. P und E sind Konjunktionen von $\mathcal{L}$-Constraints. P bezeichnet einen Präfix, und E ein getrimmtes Gleichungssystem. Wir benutzen auch die Schreibweise

$$B \leftarrow P\ \&\ E\ \&\ G.$$

Eine *Feature-Klausel-Spezifikation* ist eine Menge von Feature-Klauseln.

Natürlich könnten wir auch Feature-Symbole in Feature-Atomen zulassen, und beliebige Gleichungssysteme in Feature-Klauseln erlauben. Dies würde jedoch den Formalismus wegen zusätzlicher Auffaltungsoperationen unnötig aufblähen. Wir gehen deshalb schon von der aufgefalteten Form aus. Analog verhält es sich mit Containments auf Werteterme, die Feature-Symbole enthalten. Auch diese kann man herausfalten, so daß man nur noch Containments auf kanonische Werteterme übrigbehält. Diese können wiederum (mittels der Membership-Reduktion $\rightarrow_m$) auf einen äquivalenten Präfix reduziert werden. Es vermindert also auch nicht die Ausdruckskraft, wenn man nur Präfixe anstatt beliebiger Membership-Systeme zuläßt.

"Feature-Programme" werden wir mit Hilfe von Feature-Klauseln beschreiben. Deshalb wollen wir die Semantik von Feature-Klausel-Spezifikationen betrachten. Wir gehen dabei von einer Basis-Interpretation $\mathcal{I}$ für $\mathcal{L}$ aus. In welcher $\mathcal{R}(\mathcal{L})$-Erweiterung von $\mathcal{I}$ soll unser Interpreter für Feature-Programme nun arbeiten? Wir definieren folgendermaßen eine partielle Ordnung auf der Menge der $\mathcal{R}(\mathcal{L})$-Interpretationen mit Basis $\mathcal{I}$. Gegeben zwei $\mathcal{R}(\mathcal{L})$-Interpretationen $\mathcal{A}$ und $\mathcal{B}$ mit Basis $\mathcal{I}$, so gilt $\mathcal{A} \subseteq \mathcal{B}$ genau dann, wenn $r^{\mathcal{A}} \subseteq r^{\mathcal{B}}$ für alle Relationssymbole r aus $\mathcal{R}$. Wie in [HS 88] gezeigt wurde, gibt es für die Menge der $\mathcal{R}(\mathcal{L})$-Modelle (mit Basis $\mathcal{I}$) einer gegebenen Feature-Klausel-Spezifikation ein (bezüglich $\subseteq$) kleinstes Modell.

Satz 3.5.1 *Gegeben seien eine Feature-Klausel-Spezifikation $\mathcal{F}$ (über $\mathcal{R}(\mathcal{L})$) und eine $\mathcal{L}$-Interpretation $\mathcal{I}$. Dann definieren die Gleichungen*

$$r^{\mathcal{A}_0} := \emptyset \ , \ r^{\mathcal{A}_{i+1}} := \{\alpha(\vec{x}) \mid r(\vec{x}) \leftarrow G \in \mathcal{S} \text{ und } \alpha \in [\![G]\!]^{\mathcal{A}_i}\}$$

eine Kette $\mathcal{A}_0 \subseteq \mathcal{A}_1 \subseteq \ldots$ von $\mathcal{R}(\mathcal{L})$-Interpretationen mit Basis $\mathcal{I}$. Die Vereinigung $\bigcup_{i \geq 0} \mathcal{A}_i$ ist das (bezüglich $\subseteq$) kleinste Modell von $\mathcal{S}$ zur Basis $\mathcal{I}$).

Unser Interpreter für "Feature-Programme" wird im kleinsten Modell basierend auf der Standard-Interpretation $\mathcal{I}$ (aus Abschnitt 3.4) arbeiten.

Als nächstes erweitern wir Wohlgetyptheit auf Atome und Klauseln. Dies ist nötig, da wir zum Lösen von $\mathcal{L}$-Constraints auf den Constraint-Solver aus Abschnitt 3.4 zurückgreifen wollen. Dieser akzeptiert jedoch nur wohlgetypte $\mathcal{L}$-Constraints. Wohlgetyptheit für $\mathcal{L}$-Constraints etablieren wir, wie gewohnt, über Feature-Spezifikationen. Im folgenden sei $\mathcal{F}$ eine feste Feature-Spezifikation. Um Relationen zu typen, führen wir sogenannte "Relationsdeklarationen" ein.

Eine *Relationsdeklaration* ist ein $\mathcal{R}(\mathcal{L})$-Constraint der Form

$$r(\vec{x}) \rightarrow_{\exists} \vec{x}{:}\,\vec{\sigma},$$

wobei $\vec{\sigma}$ ein Tupel bewohnter Typterme ist. Da die Gültigkeit einer Relationsdeklaration unabhängig von den speziellen Variablen $\vec{x}$ ist, schreiben wir auch $r{:}\,\vec{\sigma}$. Eine Menge von Relationsdeklarationen heißt *singulär*, falls sie für kein Relationssymbol mehr als eine Deklaration enthält.

Allgemeine Annahme *In der Folge sei D eine feste singuläre Menge von Relationsdeklarationen.*

Bevor wir Wohlgetyptheit nun auf Atome und Klauseln erweitern, führen wir formal den Begriff der (variablenunabhängigen) "Variante" eines Constraints ein.

Eine *Umbenennung* ist eine Bijektion auf der Menge der Variablen, die bis auf endliche viele Ausnahmen die Identität ist. Gegeben eine Umbenennung ϱ und einen

falls alle Vorkommen x von Variablen in ϕ' durch ρx ersetzt wurden. Ein $\mathcal{R}(\mathcal{L})$-Constraint ϕ' heißt *Variante* eines $\mathcal{R}(\mathcal{L})$-Constraints ϕ, falls es eine Umbenennung ρ gibt, so daß ϕ' eine ρ-Variante von ϕ ist.

Ein *Feature-Atom* $r(\vec{s})$ *ist wohlgetypt bezüglich* D unter einem Präfix P, falls D eine Relationsdeklaration $r{:}\,\vec{\sigma}$ enthält, und es eine Substitution θ gibt, mit $P \vdash \vec{s}{:}\,\theta\vec{\sigma}$. Wir nennen eine *Konjunktion von Feature-Atomen* A_1 & ... & A_n *wohlgetypt bezüglich* D unter P, falls alle Atome A_i wohlgetypt bezüglich D unter P sind. Eine *Feature-Klausel* $r(\vec{s}) \leftarrow P$ & E & G *heißt wohlgetypt bezüglich* D, falls sowohl die Konjunktion von Feature-Atomen G, als auch das getrimmte Gleichungssystem E wohlgetypt bezüglich D unter P ist, und eine Variante $r{:}\,\vec{\sigma}$ einer Deklaration aus D existiert, mit $P \vdash \vec{s}{:}\,\vec{\sigma}$.

Diese Definition von Wohlgetyptheit für Klauseln ist verhältnismäßig streng, da der Klauselkopf einer Variante (und nicht einer Instanz) der Relationsdeklaration von r genügen muß . Sie verlangt jedoch lediglich, daß Relationen so genau wie möglich deklariert werden.

Ein *Feature-Programm* S unter einer Feature-Spezifikation $\mathcal{F}$ ist ein Paar (D, C), wobei D eine singuläre Menge von Relationsdeklarationen, und C eine bezüglich D wohlgetypte Menge von Feature-Klauseln ist. Wir verlangen, daß die Interpretationen von S auf dem Standardmodell $\mathcal{I}$ von $\mathcal{F}$ basieren. Die folgende Aussage basiert auf Satz 3.5.1.

Satz 3.5.2 *Sei $S = (D, C)$ ein Feature-Programm. Dann hat C ein kleinstes Modell, und D ist in diesem kleinsten Modell gültig.*

Allgemeine Annahme. *Im folgenden sei $S = (D, C)$ ein festes Feature-Programm unter unserer Feature-Spezifikation $\mathcal{F}$. Mit $\mathcal{S}$ bezeichnen wir das kleinste Modell von S.*

Unser Interpreter für Feature-Programme beantwortet Anfragen (sogenannte "Goals") im kleinsten Modell $\mathcal{S}$ des zugrundeliegenden Feature-Programms S.

Ein *Goal* ist eine Konjunktion P & E & G, wobei P ein normaler und bewohnter Präfix, E ein getrimmtes Gleichungssystem, und G eine Konjunktion von Feature-Atomen ist. E und G sind wohlgetypt bezüglich D unter P. Aus Gründen der Einfachheit gehen wir auch hier schon von der aufgefalteten Form aus.

Unser Interpreter arbeitet ein Goal P & E & G in zwei Schritten ab. Zunächst wird mittels des $\mathcal{L}$-Constraint-Solvers $\rightarrow_c$ das Constraint-System P & E in eine äquivalente gelöste Form überführt (falls P & E unifizierbar).

$$P.\emptyset.E \rightarrow_c^* P'.\theta.\emptyset.$$

Im zweiten Schritt setzt die eigentliche *Goal-Reduktion* ein (siehe Abbildung 3.5), welche mit dem Tripel $P'.\theta.G$ startet. Goal-Reduktion arbeitet generell auf Tripeln der Form $P.\theta.G$, wobei P ein bewohnter Präfix, θ eine Konjunktion von Gleichungen

$P.\theta.r(\vec{s})\ \&\ G \rightarrow_g Q.\psi.G\ \&\ G'$

falls $r(\vec{t}) \leftarrow P'\ \&\ E\ \&\ G'$ eine Variante einer Klausel in C ist,
mit $\mathcal{D}P \cap \mathcal{D}P' = \emptyset$ und $P\ \&\ P'.\theta.\theta\vec{s} \doteq \vec{t}\ \&\ E \rightarrow_c^* Q.\psi.\emptyset$.

Abbildung 3.5: Goal-Reduktion

in gelöster Form, und G eine Konjunktion von Feature-Atomen ist. Die Anwendung θs des Gleichungssystems θ auf einen kanonischen Werteterm s ist wie folgt zu verstehen. Da θ in gelöster Form vorliegt hat es die Gestalt $\{x_i = t_i, f(y_i) = s_i\}$. Wir interpretieren das Gleichungssystem $\{x_i = t_i\}$ als eine Substitution $\{x_i/t_i\}$. θs wendet diese Substitution auf den Werteterm s an. Der Quasivariablen-Anteil von θ bleibt unberücksichtigt, da s kanonisch ist.

Wir zeigen nun, daß Goal-Reduktion den Constraint-Solver *nicht* immer mit wohlgetypten Constraint-Systemen aufruft. Dazu legen wir die Listendeklaration und die "append"-Klauseln aus Abbildung 1.2 zugrunde. Zusätzlich führen wir noch den Typ *bool* ein, der durch die Konstruktoren $true\colon \rightarrow bool$ und $false\colon \rightarrow bool$ bewohnt wird. Dann ist

$$L\colon list(bool)\ \&\ append(nil, cons(true, nil), L)$$

wohlgetypt und somit ein Goal. Bei Goal-Reduktion mit der ersten Klausel von *append* wird der Constraint-Solver mit dem Präfix

$$L\colon list(bool)\ \&\ L'\colon list(\alpha)$$

und den Gleichungen

$$nil \doteq nil\ \&\ cons(true, nil) \doteq L'\ \&\ L \doteq L'$$

aufgerufen. Die beiden letzten Gleichungen sind unter dem mitgegebenen Präfix offenbar nicht wohlgetypt. Der Grund liegt offenbar darin, daß Sortenvariable in Relationsdeklarationen bei der Reduktion nicht instantiiert werden. Es ist klar, daß es sich hier um keine besonders schlimme Form der "Schlechtgetyptheit" handelt. In [Sm 89a] wird gezeigt, daß Constraint-Solver und Interpreter auch für solche "schwach wohlgetypten" Constraints vollständig und korrekt arbeiten. Wir verallgemeinern dieses Resultat für Feature-Spezifikationen.

Satz 3.5.3 [Korrektheit] *Sei $P\ \&\ E\ \&\ G$ ein Goal, das mittels Goal-Reduktion folgendermaßen transformiert werden kann:*

$$P.\emptyset.E \rightarrow_c^* P'.\theta.\emptyset \text{ und } P'.\theta.G \rightarrow_g^* Q.\psi.\emptyset.$$

Dann gilt

$$\mathcal{I}[\![Q\ \&\ \psi]\!] \subseteq \mathcal{S}[\![P\ \&\ E\ \&\ G]\!].$$

Satz 3.5.4 [Vollständigkeit] *Sei $P \,\&\, E \,\&\, G$ ein Goal und $\delta \in \mathcal{S}[\![P \,\&\, E \,\&\, G]\!]$. Dann existieren P', Q, θ und ψ mit*

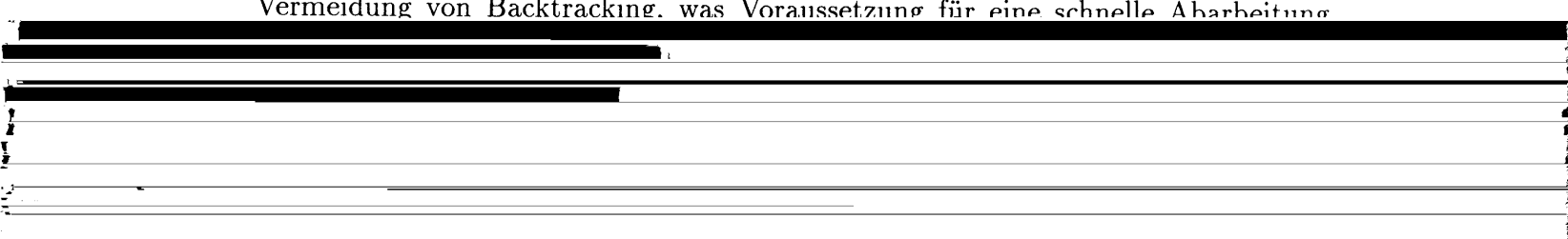
$$P.\emptyset.E \rightarrow_c^* P'.\theta.\emptyset \text{ und } P'.\theta.G \rightarrow_g^* Q.\psi.\emptyset \text{ und } \delta \in \mathcal{I}[\![Q \,\&\, \psi]\!].$$

In [Sm 89a] wird gezeigt, daß nur die Auswahl der Klauseln "don't know" ist, alle anderen Wahlmöglichkeiten aber "don't care" sind. Außerdem wird dort erläutert wie überflüssige Typberechnungen eingespart werden können, was für die praktische Realisierung entscheidend ist.

Hier noch einmal kurz die wichtigsten Ergebnisse dieses Abschnitts. Wir haben Feature-Constraint-Solving in eine logische Programmiersprache eingebettet. Die Semantik von Feature-Programmen basiert dabei auf der Semantik der zugrundeliegenden Feature-Constraint-Sprache. Wir erweiterten dann Wohlgetyptheit auf Klauseln und gaben einen vollständigen und korrekten Interpreter für Feature-Programme an. Dieser greift auf den Constraint-Solver der Basissprache zurück. Im nächsten Abschnitt stellen wir ein Hilfsmittel vor, mit dem man Klauseln wohlgetypt machen kann.

3.6 Typ-Inferenz

Beim Schreiben von Feature-Programmen ist es oft mühsam, einen Präfix anzugeben, unter dem eine Feature-Klausel wohlgetypt wird. Wir zeigen zunächst, daß Wohlgetyptheit entscheidbar ist, das heißt man kann bestimmen, ob eine endliche Folge von syntaktischen Objekten ein Feature-Programm darstellt. In der Folge entwickeln wir einen Typ-Inferenz-Algorithmus, welcher versucht den Präfix einer gegebenen Feature-Klausel derart zu verschärfen, daß diese wohlgetypt wird. Ein Programmierer braucht dann nur noch eine Präfix-Näherung anzugeben. Der vorgestellte Typ-Inferenzer arbeitet jedoch nicht perfekt. So kann er abbrechen, obwohl es eine Präfix-Verschärfung gibt, unter der die vorliegende Klausel wohlgetypt wird. Außerdem wird nicht unbedingt der allgemeinste Präfix berechnet, unter dem eine Klausel wohlgetypt wird. Diese Mängel treten jedoch nur bei polymorphen Deklarationen auf. Desweiteren kann man sie vermeiden, indem man die kritischen Variablen genauer qualifiziert. Der Vorteil unseres Typ-Inferenz-Algorithmus liegt in der Vermeidung von Backtracking, was Voraussetzung für eine schnelle Abarbeitung



und präzise Fehlermeldungen ist. Möglicherweise existiert sogar ein vollständiger Typ-Inferenzer. Es ist jedoch fraglich, ob dieser auch praktikabel wäre.

Wir wollen zunächst die Wohlgetyptheit von Feature-Atomen entscheiden. Dazu verwenden wir die folgende Aussage aus [Sm 89a].

Satz 3.6.1 *Ein Feature-Atom $r(\vec{s})$ ist wohlgetypt bezüglich D unter einem Präfix P, falls es eine Relationsdeklaration $r{:}\,\vec{\sigma}$ in D gibt, so daß $\sigma^P[\vec{s}] \sqsubseteq \vec{\sigma}$ einen oberen Matcher hat.*

Da der kleinste obere Matcher eines Inklusion-Systems berechenbar ist, können wir Wohlgetyptheit von Feature-Atomen entscheiden. Um die Wohlgetyptheit einer Feature-Klausel zu entscheiden, müssen wir feststellen, ob eine Variante ρ der Relationsdeklaration $r{:}\,\vec{\sigma}$ des Klauselkopfes existiert, so daß $P \vdash \vec{s}{:}\,\rho\vec{\sigma}$ gilt. In [Sm 89a] wurde dafür die folgende Aussage benutzt.

Satz 3.6.2 *Es existiert eine Variante $\rho\vec{\tau}$ von $\vec{\tau}$ mit $P \vdash \vec{s}{:}\,\rho\vec{\tau}$ genau dann, wenn $\theta := LUM[\sigma^P[\vec{s}] \sqsubseteq \vec{\tau}]$ existiert und*

1. *Falls $\alpha \in \mathcal{V}\vec{\tau}$, so ist $\theta\alpha$ eine Variable oder $\bot$*
2. *Falls $\alpha, \beta \in \mathcal{V}\vec{\tau}$ verschieden, so gilt entweder $\theta\alpha \neq \theta\beta$ oder $\theta\alpha = \theta\beta = \bot$.*

Korollar 3.6.3 *Es ist entscheidbar, ob eine Feature-Klausel wohlgetypt bezüglich D ist. Außerdem ist es entscheidbar, ob eine Konjunktion $P \,\&\, E \,\&\, G$ ein Goal ist.*

Wir kommen nun zur Typ-Inferenz. Wertevariablen über die noch keine Typinformation vorliegt wollen wir möglichst allgemein qualifizieren. Deshalb erweitern wir unsere Ordnung $\leq$ um das Wildcard-Symbol "_" als größtes Element. Wir definieren wie folgt die partielle Ordnung $\bar{\leq}$ auf der Menge der Typterme:

$$\sigma \bar{\leq} \_\,,\ \bot \bar{\leq} \alpha\,,\ \alpha \bar{\leq} \alpha\,,$$
$$\xi(\vec{\sigma}) \bar{\leq} \tau \text{ falls } \tau \Rightarrow^* \xi(\vec{\tau}) \text{ und } \vec{\sigma} \bar{\leq} \vec{\tau}.$$

Ein Typterm heißt *proper*, falls er nicht das Wildcard-Symbol enthält. Wir zeigen nun, daß Suprema bezüglich $\bar{\leq}$ berechenbar sind. Die folgenden Gleichungen definieren eine totale berechenbare Funktion "$\sigma \bar{\sqcap} \tau$" auf Typtermen.

$$\begin{aligned}
\sigma \bar{\sqcap} _ &= \sigma \\
_ \bar{\sqcap} \tau &= \tau \\
\alpha \bar{\sqcap} \alpha &= \alpha \\
\xi(\vec{\sigma}) \bar{\sqcap} \eta(\vec{\tau}) &= \zeta(\vec{\mu} \bar{\sqcap} \vec{\nu}), \text{ falls } \zeta = \xi \sqcap \eta,\ \xi(\vec{\sigma}) \Rightarrow^* \zeta(\vec{\mu}) \text{ und } \eta(\vec{\tau}) \Rightarrow^* \zeta(\vec{\nu}) \\
\sigma \bar{\sqcap} \tau &= \bot, \text{ falls keine der anderen Gleichungen zutrifft}
\end{aligned}$$

Satz 3.6.4 *Sind σ und τ zwei Typterme, so ist $\sigma \bar{\sqcap} \tau$ das Infimum von σ und τ bezüglich $\bar{\leq}$. Falls σ und τ beide proper sind, so gilt zusätzlich $\sigma \bar{\sqcap} \tau = \sigma \sqcap \tau$.*

Wir kommen nun zum eigentlichen Typ-Inferenz-Algorithmus. Grundoperation ist das Umsetzen eines Membership-Systems in einen Präfix. Dabei wird der allgemeinste Präfix erzeugt, unter dem das Membership-System wohlgetypt ist. Ähnlich wie bei den Reduktionsregeln für Memberships (Abschnitt 3.4) übertragen wir Typanforderungen an zusammengesetzte Terme mittels einer Dekompositionsrelation

auf deren Argumente. Die zugrundeliegende Ordnung auf Typtermen ist dieses Mal jedoch $\bar{\leq}$.

Wir definieren die *Dekompositionsrelation* $M \rightarrow_d M$ *auf Membership-Systemen* wie folgt.

1. $M \,\&\, c(\vec{s}):\sigma \rightarrow_d M \,\&\, \vec{s}:\{\vec{\alpha}/\vec{\sigma}\}\vec{\mu}$

   falls c die Funktionsdeklaration $c:\vec{\mu} \rightarrow \xi(\vec{\alpha})$ hat,
   und $\sigma \Rightarrow^* \xi(\vec{\sigma})$ gilt

2. $M \,\&\, x:\sigma \,\&\, x:\tau \rightarrow_d M \,\&\, x:(\sigma \bar{\sqcap} \tau)$.

Dabei gehen wir davon aus, daß in den Wertetermen der Memberships keine Features mehr vorkommen.

Satz 3.6.5 *Die Dekompositionsrelation* $\rightarrow_d$ *ist konfluent und terminiert.*

Man beachte, daß die in den Memberships enthaltenen Werteterme kanonisch sein müssen, da sonst möglicherweise keine eindeutige Dekomposition existiert. Wir illustrieren dies an folgendem Beispiel. Ein Feature f habe die Deklarationen

$$f:\xi \rightarrow \sigma \ , \ f:\zeta \rightarrow \tau \ , \text{ und } f:\eta \rightarrow \tau \ ,$$

wobei $\zeta \leq \xi$, $\eta \leq \xi$, und $\tau \leq \sigma$ gelte. Die Variable x werde im aktuellen Präfix P mit dem Typ ξ qualifiziert. Gegeben sei die Membership-Relation $f(x):\tau$. Dann können wir ausgehend von der Gültigkeit dieser Membership-Relation unter P keine Rückschlüsse auf den Typ von x ziehen, da es keine eindeutige Deklaration von f gibt.

Um zu gewährleisten, daß der Präfix, den wir aus einem Membership-System mittels $\rightarrow_d$ ableiten, nur normale Typterme enthält, berechnen wir anschließend seine Normalform.

Die folgende Gleichung definiert eine berechenbare, partielle Funktion "$IP_M[\cdot]$" von Membership-Systemen auf normale und bewohnte Präfixe. Dabei greifen wir auf die in Abschnitt 3.2 definierte Normalform NF für Präfixe zurück.

$$IP_M[M] := NF[P] \ ,$$
$$\text{falls } M \rightarrow_d^* P, \text{ und } P \text{ ein bewohnter Präfix ist.}$$

Oft enthalten die Typterme eines Präfix das Wildcard-Symbol. Wie wir wissen, entstammt dieses einer vorläufigen Variablenqualifikation und verhält sich ähnlich einer Typvariable. Beim Ableiten von kleinsten Typen für Werteterme gehen wir von einer minimalen Belegung dieser "Variable" aus, und ersetzen alle Vorkommen von "_" durch "$\bot$". Den so veränderten Präfix bezeichnen wir mit $P_\bot$.

Als nächstes untersuchen wir, wie man einen vorliegenden Präfix so verschärft, daß eine getrimmte Gleichung, beziehungsweise ein Feature-Atom wohlgetypt unter ihm wird. Gegeben sei der Präfix P und die Gleichung $s \doteq t$. Dann ist $P \vdash t{:}\,\sigma^{P_\perp}[s]$ eine hinreichende Bedingung für die Wohlgetyptheit von $s \doteq t$ unter P. Wir erzwingen diese Bedingung, indem wir P zu $IP_M[P \,\&\, t{:}\,\sigma^{P_\perp}[s]]$ verschärfen. Dies funktioniert auch für Gleichungen mit Quasivariablen, wie zum Beispiel $f(x) \doteq t$. Man beachte, daß die Forderung $IP_M[P \,\&\, f(x){:}\,\sigma^{P_\perp}[t]]$ nicht zulässig ist, da dieses Membership einen nichtkanonischen Werteterm enthält. Gegeben sei nun ein Feature-Atom $r(\vec{s})$ mit der Relationsdeklaration $r{:}\,\vec{\tau}$. Dann ist $P \vdash \vec{s}{:}\,\theta\vec{\tau}$ (wobei $\theta = LUM[\sigma^{P_\perp}[\vec{s}] \sqsubseteq \vec{\tau}]$) eine hinreichende Bedingung für die Wohlgetyptheit von $r(\vec{s})$ unter P. Wir erzwingen diese Bedingung, indem wir P zu $IP_M[P \,\&\, \vec{s}{:}\,\theta\vec{\tau}]$ verschärfen.

Wir können also nun einen gegebenen Präfix so verschärfen, daß eine getrimmte Gleichung oder ein Feature-Atom wohlgetypt unter ihm wird. Um eine Folge von Gleichungs- und Atom-Constraints wohlgetypt unter einem Präfix zu machen, reicht es jedoch nicht aus, dieses Verfahren zu iterieren. Das zusätzliche Problen besteht darin, daß vorhergehende Gleichungen möglicherweise eine Typ-Äquivalenz auf den Wertevariablen der nachfolgenden Constraints implizieren. Wir illustrieren dies an einem Beispiel. Gegeben sei die folgende Spezifikation

$$\eta \leq \xi\ ,\ c{:}\,\eta \to \eta\ ,$$

und die Relationsdeklaration $r{:}\,\eta$. Der bisherige Präfix P qualifiziere die Variablen x und y jeweils mit dem Typ ξ. Wir wollen nun P so verschärfen, daß

$$x \doteq y \,\&\, r(x) \,\&\, r(c(y))$$

wohlgetypt unter P wird (ein analoges Beispiel kann man für Gleichungen mit Quasivariablen konstruieren). Wenn wir dazu das oben beschriebene Verfahren für einzelne Constraints einfach iterieren, so erleiden wir beim dritten Constraint Schiffbruch, da wir für $c(y)$ keinen kleinsten Typ ableiten können. Der Grund des Übels liegt darin, daß man beim zweiten Constraint $r(x)$ die Gleichheit von x und y bereits "vergessen" hat. Man wird dort also nur den Typ von x zu η verschärfen, aber keine gemeinsame Typabsenkung von x und y vornehmen. Wir lösen das Problem, indem wir ein Gleichungssystem in gelöster Form mitführen, welches die bisher abgearbeiteten Gleichungsconstraints repräsentiert. Bevor wir nun eine Typverschärfung durch den aktuellen Constraint vornehmen, wenden wir auf diesen das Gleichungssystem an. Es reicht dann, bei einem Gleichungsconstraint nur den Typ der rechten Seite abzusenken, da die linke Seite ja in allen nachfolgenden Constraints durch die rechte Seite ersetzt wird. Das Vorgehen ist sehr ähnlich zum Constraint-Solving; zusätzlich werden jedoch noch Feature-Atome bezüglich ihrer Konsequenzen auf den Präfix berücksichtigt. Wir werden das Verschärfen von Präfixen durch Konjunktionen von Feature-Atomen und Gleichungen nun formal beschreiben.

$$
\begin{aligned}
\theta \uplus (s \doteq t \,\&\, E) &:= (\theta \uplus (s \doteq t)) \uplus E \\
\theta \uplus (c(\vec{s}) \doteq c(\vec{t})) &:= \theta \uplus (\vec{s} \doteq \vec{t}) \\
(\theta \,\&\, x \doteq s) \uplus (x \doteq t) &:= (\theta \,\&\, x \doteq s) \uplus (s \doteq \theta t) \\
\theta \uplus (x \doteq t) &:= \{x/\theta t\}\theta \,\&\, x \doteq \theta t \\
(\theta \,\&\, f(x) \doteq s) \uplus (f(x) \doteq t) &:= (\theta \,\&\, f(x) \doteq s) \uplus (s \doteq \theta t) \\
\theta \uplus (f(x) \doteq t) &:= \{f(x)/\theta t\}\theta \,\&\, f(x) \doteq \theta t \\
\theta \uplus s \doteq t &:= \theta \text{ ,falls sonst keine Gleichung zutrifft.}
\end{aligned}
$$

Dabei ist die Anwendung θt des Gleichungssystems θ auf einen Werteterm t so zu verstehen, daß gleichzeitig alle Vorkommen von linken Seiten aus θ in t durch die dazugehörigen rechten Seiten ersetzt werden. Ebenso bewirkt $\{s/t\}\theta$, daß alle Vorkommen von s in θ durch t ersetzt werden. Offenbar integriert $\theta \uplus E$ eine Menge von Gleichungen E in ein bestehendes (gelöstes) Gleichungssystem, wobei die neuen Gleichungen jedoch nur in einer Richtung gelesen werden, nämlich von links nach rechts.

Die berechenbare, partielle Funktion IP_G bildet Tripel der Form $P.\theta.G$ (wobei P

ein Präfix, θ ein Gleichungssystem in gelöster Form, und G ein Feature-Atom oder eine getrimmte Gleichung ist) auf Paare der Form $Q.\psi$ ab (wobei Q ein Präfix und ψ ein Gleichungssystem in gelöster Form ist). Wir definieren IP_G durch die beiden folgenden Gleichungen.

1. $IP_G[P.\theta.s \doteq t] := IP_M[P \,\&\, \theta t\colon \sigma^{P_\perp}[\theta s]\,].\theta \uplus (s \doteq t)$
2. $IP_G[P.\theta.r(\vec{s})] := IP_M[P \,\&\, \theta\vec{s}\colon \psi\vec{\tau}].\theta$
   falls $r\colon \vec{\tau}$ in D und $\psi = LUM[\sigma^{P_\perp}[\theta\vec{s}] \sqsubseteq \vec{\tau}]$.

Wie bereits motiviert wird also nur die rechte Seite einer Gleichung direkt im Präfix abgesenkt, das gleichzeitige Absenken der linken Seite ist implizit im mitgeführten Gleichungssystem θ enthalten. Zum Schluß des Typ-Inferenz-Prozeßes müssen wir in P mittels θ die Typen aller, in derselben Äquivalenzklasse enthaltenen, Variablen angleichen. Gegeben einen Präfix P und ein Gleichungssystem θ in gelöster Form, so ist P_θ wie folgt definiert.

$$
P_\theta x = \begin{cases} \sigma^{P_\perp}[t] \sqcap Px & \text{falls } x \doteq t \text{ in } \theta \text{ vorkommt} \\ Px & \text{sonst} \end{cases}
$$

Wir erweitern Typ-Inferenz auf Konjunktionen von Gleichungen und Feature-Atomen, indem wir IP_G iterieren. Die berechenbare, partielle Funktion IP_{G^*} ordnet Tripeln der Form $P.\theta.G^*$ (wobei P ein Präfix, θ ein Gleichungssystem in gelöst

2. $IP_{G^*}[P.\theta.G \,\&\, G^*] := IP_{G^*}[IP_G[P.\theta.G].G^*]$

Typ-Inferenz für ganze Klauseln ist nun einfach. Variablen, die in der gegebenen Klausel vorkommen, in ihrem Präfix jedoch nicht qualifiziert sind, ordnen wir zunächst den Typ "_" zu. Der Klauselkopf wird dann ähnlich einem Feature-Atom behandelt (die dazugehörige Deklaration wird hier jedoch nicht instantiiert), und der Klauselrumpf wird durch IP_{G^*} abgearbeitet. Die berechenbare, partielle Funktion IP_C leitet für Feature-Klauseln einen Präfix ab, und ist wie folgt definiert.

$$IP_C[r(\vec{s}) \leftarrow P \,\&\, E \,\&\, G] = IP_{G^*}[IP_M[Q \,\&\, \vec{s}{:}\,\vec{\sigma}].\emptyset.E \,\&\, G]$$
falls r eine Deklaration $r{:}\,\vec{\sigma}$ hat, und
$$Q := P \,\&\, \{x{:}\,\_ \mid x \in (\mathcal{V}\vec{s} \cup \mathcal{V}E \cup \mathcal{V}G) \setminus \mathcal{D}P\}.$$

Dieser Typ-Inferenz-Algorithmus setzt voraus, daß keine Deklaration in D das Wildcard-Symbol enthält. Außerdem gehen wir von einer festen Reihenfolge der Constraints im Klauselrumpf aus. Da jedoch praktische Programme Klauseln gewöhnlich von links nach rechts abarbeiten, ist eine analoge Strategie bei der Typ-Inferenz durchaus sinnvoll. Der folgende Satz sagt aus, daß unser Typ-Inferenzer korrekt arbeitet.

Satz 3.6.6 *Sei $A \leftarrow P \,\&\, E \,\&\, G$ eine Feature-Klausel, für die $Q := IP_C[A \leftarrow P \,\&\, E \,\&\, G]$ existiert. Dann gilt:*

1. *$A \leftarrow Q \,\&\, E \,\&\, G$ ist wohlgetypt bezüglich D*
2. *Falls $x \in \mathcal{D}P$, so gilt $Qx \stackrel{\sim}{\leq} Px$.*

In [Sm 89a] wird anhand des folgenden Beispiels gezeigt, daß der Typ-Inferenzer für polymorphe Deklarationen nicht perfekt arbeitet. Die zugrundeliegende Spezifikation sieht folgendermaßen aus:

$$nat \leq int\ ,\ 1{:}\,nat\ ,\ -1{:}\,int.$$

Das Relationssymbol "m" habe die Deklaration

$$m{:}\,\alpha \times list(\alpha).$$

Wir stellen uns m als Membership-Relation auf Listen vor. Für die Konjunktion von Feature-Atomen

$$m(1, L) \,\&\, m(-1, L)$$

wollen wir einen Präfix berechnen, unter dem diese wohlgetypt werden. Unser Typ-Inferenzer wird den Präfix $L{:}\,list(nat)$ vorschlagen. Unter diesem Präfix würde unsere Konjunktion jedoch unerfüllbar. Dabei macht die schwächere Qualifikation $L{:}\,list(int)$ die Konjunktion wohlgetypt, und sie bleibt erfüllbar. Das Übel liegt offenbar darin, daß immer der schärfste obere Matcher einer Relationsdeklaration

verwendet wird. Man beachte jedoch, daß unser Typ-Inferenzer nach Vertauschen der Atome den allgemeinsten Präfix L: *list*(*int*) berechnet.

Der nächste Satz zeigt, daß unser Typ-Inferenz-Algorithmus für Klauseln ohne polymorphe Deklarationen perfekt arbeitet. Wir nennen eine Relationsdeklaration $r{:}\,\vec{\sigma}$ eine *Grunddeklaration*, falls $\vec{\sigma}$ nur Grundsortenterme enthält.

Satz 3.6.7 *Sei $A \leftarrow P \,\&\, E \,\&\, G$ eine Feature-Klausel, in der jedes Relationssymbol eine Grunddeklaration hat. Weiterhin sei Q' ein Präfix mit $A \leftarrow Q' \,\&\, E \,\&\, G$ ist wohlgetypt bezüglich D, $Q'x \bar{\leq} Px$ für alle x in $\mathcal{D}P$, und $\mathcal{D}Q' = \mathcal{V}A \cup \mathcal{V}P \cup \mathcal{V}E \cup \mathcal{V}G$. Dann existiert $Q := IP_C[A \leftarrow P \,\&\, E \,\&\, G]$, es gilt $Q'x \bar{\leq} Qx$ für alle x in $\mathcal{D}Q'$, und $A \leftarrow Q \,\&\, E \,\&\, G$ ist wohlgetypt bezüglich D.*

Wir beenden dieses Kapitel mit einem kurzem Resümee. Wir haben gezeigt, wie eine Semantik für eine polymorphe, logische Programmiersprache mit Feature-Typen aussehen kann. Dabei führten wir Feature-Typen mittels impliziter Konstruktoren auf Konstruktor-Typen zurück. In der Folge entwickelten wir einen Interpreter und einen Typ-Inferenzer für diese Programmiersprache. Damit verfügen wir über die theoretischen Grundlagen für die praktische Implementierung von Feature-Typen in TEL.

Kapitel 4

Einführung in TEL

TEL, ein Akronym für "Terms, Equations and Logic", ist eine logische Programmiersprache in der POS-Typen und Funktionen integriert sind. TEL wurde an der Universität Kaiserslautern entwickelt und implementiert. Wir geben einen kurzen und deshalb oft vergröberten Überblick über die Version 0.9, auf der Feature-TEL aufsetzen wird. Eine ausführliche Beschreibung findet man in [Sm 88, NS 90].

Das Kapitel ist wie folgt aufgebaut. In Abschnitt 4.1 skizzieren wir die wichtigsten Sprachkonstrukte von TEL aus der Sicht des Programmierers. Abschnitt 4.2 beschreibt dann aus der Sicht des Sprachentwicklers die innere Struktur des TEL-Systems.

4.1 Sprachbeschreibung

TEL ist eine relationale Programmiersprache und unterscheidet sich von Prolog insbesondere durch eine POS-Typdisziplin und durch die Integration von Funktionen. Zusätzlich bietet TEL noch eine Menge nichtlogischer Features – Modulkonzept, File-Handling, Datenbasen – die es zu einer praktikablen Programmiersprache machen.

Ein TEL-Programm besteht im wesentlichen aus Typdeklarationen, Relationsdefinitionen und Funktionsdefinitionen. Während Relationen mittels (getypter) Resolution und Backtracking abgearbeitet werden, werden Funktionen über (getyptes) "innermost rewriting" evaluiert. Das Modulkonzept von TEL unterstützt die inkrementelle Erstellung großer Programme. Wir gehen nun etwas genauer auf die einzelnen Sprachkonstrukte ein.

4.1.1 Typen

Die *Typen* von TEL entsprichen den in Kapitel 2 und Kapitel 3 ausführlich besprochenen POS-Typen. Wir gehen deshalb hier nur noch kurz auf sie ein. Typen werden über Konstruktoren und Untertypen definiert. Man betrachte etwa die folgende rekursive Definition von Bäumen (Bäume sind entweder leer, oder bestehen aus einem Knoten mit einen linken und rechten Unterbaum).

$$\begin{aligned} \texttt{tree(T)} &:= \texttt{nonempty\_tree(T)} +\!+ \{\texttt{etree}\}. \\ \texttt{nonempty\_tree(T)} &:= \{\texttt{netree: tree(T)} \times \texttt{tree(T)} \times \texttt{T}\}. \end{aligned}$$

Der Typ `tree(T)` hat den Untertyp `nonempty_tree(T)` und den Konstruktor

$$\texttt{etree:} \rightarrow \texttt{tree(T)}.$$

Der Typ `nonempty_tree(T)` hat den Konstruktor

$$\texttt{netree: tree(T)} \times \texttt{tree(T)} \times \texttt{T} \rightarrow \texttt{nonempty\_tree(T)}.$$

In TEL gibt es bereits eine Menge eingebauter Typen, wie zum Beispiel `list, pair, int, nat, posint, string`. Einen Typ, der nur über einen Typterm definiert ist, bezeichnen wir als *Abkürzungstyp*. Beispiel:

$$\texttt{nat\_tree} := \texttt{tree(nat)}.$$

Typterme können über Typfunktionen (`tree` und `nonempty_tree`) , Typkonstante (`nat`) und Typvariable (`T`) aufgebaut werden. Beispiel für einen Typ(term):

$$\texttt{tree(nonempty\_tree(nat))}$$

Werte(terme) werden aus Konstruktoren, Funktionssymbolen und Variablen zusammengesetzt. Beispiel:

$$\texttt{netree(etree, netree(etree, etree, 5), 4)}$$

ist ein Wert vom Typ `nonempty_tree(posint)`.

4.1.2 Relationen und Funktionen

Wie in Prolog, werden *Relationen* in TEL über eine Folge von Hornklauseln definiert. Die Klauseln haben die Form

(if-then-else, negation-as-failure, is-list-of, etc.), auf die wir in diesem Rahmen jedoch nicht eingehen können. Zusätzlich zu den Klauseln, hat jede Relation eine *Typ-Deklaration (Rank)*. Diese spezifiziert zum einen die Typen der Argumente, zum anderen beschreibt sie den Datenfluß der Relation, da jedes Argument entweder als Eingabe-, oder als Ausgabeargument gekennzeichnet werden muß. Ausgabeargumente werden mit einem "?" markiert, und müssen im Klauselrumpf gebunden werden. In den Klauselköpfen einer Relation müssen deren Eingabeargumente kanonisch sein, das heißt sie dürfen nur Konstruktoren oder Variablen enthalten. Die Eingabeargumente von Relationsatomen dürfen auch Funktionsaufrufe enthalten, müssen jedoch vollständig gebunden sein. Diese Konventionen zwingen den Programmierer zu einem klaren Entwurf, dessen Konsistenz vom Compiler gecheckt wird. Außerdem wird die Lesbarkeit von Programmen wesentlich verbessert. Falls ein Argument sowohl als Eingabe, als auch als Ausgabe dienen soll, so kann man dies durch die explizite Deklaration von offenen Variablen erreichen. Dieser Mechanismus wird ausführlich in [Sm 88] beschrieben. Wir gehen hier nicht weiter darauf ein.

Zur Illustration von Relationen definieren wir eine Beispiel-Relation, die als Eingabe einen Baum bekommt, und als Ausgabe (falls vorhanden) im Baum enthaltene ganze Zahlen (Typ `int`) liefert.

```
            rel contains_int   :  tree(T) × ?int.
contains_int(netree(_,_,I),I)  ←  I:int.
contains_int(netree(L,_,_),I)  ←  contains_int(L,I).
contains_int(netree(_,R,_),I)  ←  contains_int(R,I).
```

Außer der (erzwungenen) Spezifikation des Datenflußes und der Argumenttypen, bietet TEL weitere Möglichkeiten, um Programme lesbarer und Fehler leichter lokalisierbar zu machen. So kann man eine Relation, die niemals fehlschlagen soll durch Verwendung von **trel** statt **rel** als *total* deklarieren. Dies hat zur Folge, daß bei einem unbeabsichtigten Fehlschlag dieser Relation (aufgrund eines Programmierfehlers) zur Laufzeit sofort mit der entsprechenden Fehlermeldung abgebrochen wird, und nicht etwa Backtracking versucht wird. Analog kann man Relationen (mit **drel**) als *deterministisch* deklarieren, was Backtracking von vorneherein vermeidet. Relationen, die Ein/Ausgabeoperationen anstoßen, müssen (mit **proc**) als *Prozeduren* gekennzeichnet werden.

Relationen werden mittels der aus Prolog bekannten SLD-Resolution abgearbeitet. Da TEL (zur Zeit noch) nach Prolog übersetzt wird, müssen wir (mehr oder weniger) auf getypte Unifikation verzichten. Das bedeutet insbesondere, daß Containments nur abtesten, ob ein (kanonischer) Term von einem bestimmten Typ ist, sie führen aber keine Bindung einer bisher offenen Variable an einen Typ durch, oder verschärfen den aktuellen Typ einer Variablen. Der Typchecker führt allerdings zur Compile-Zeit eine Art getypter Unifikation aus. Wir werden später sehen, daß er

dabei jedoch nicht alle (durch Gleichungen implizierte) gemeinsamen Typabsenkungen von Variablen erkennt. Die Lösung des Problems wäre die Entwicklung einer abstrakten Maschine für TEL. Dies hätte insbesondere den Vorteil, daß das Rechnen mit Typen und Funktionen optimiert werden könnte.

Funktionen werden über sogenannte *Conditional Equations* definiert. Diese haben die Form

$$f(t_1, \ldots, t_n) = t \leftarrow C_1 \;\&\; \ldots \;\&\; C_n.$$

wobei die C_i Conditions sind. Auch Funktionen müssen eine Typ-Deklaration haben, im Unterschied zu Relationen können sie jedoch auch mehrere Ranks haben. Die Ranks müssen jedoch von der Art sein, daß für wohlgetypte Werteterme immer ein eindeutiger kleinster Typ abgeleitet werden kann. Dies wird (analog zu Abschnitt 3.2) durch die Forderung gewährleistet, daß es immer einen eindeutigen "kleinsten" anwendbaren Rank geben muß ("Regularität"). Von den Argumenten einer Funktionsgleichung verlangen wir, daß sie kanonisch sind. Beim Aufruf der Funktion müssen sie (wie die Eingabeargumente von Relationen) vollständig gebunden sein, und enthalten somit nur noch Konstruktoren. Funktionen werden über innermost rewriting abgearbeitet und müssen total und deterministisch sein. Ebenso wie bei Relationen wird dies zur Laufzeit gecheckt, und macht semantische Fehler damit leichter lokalisierbar. Zur Illustration schreiben wir eine Funktion, die die Tiefe eines Baumes berechnet.

```
depth_tree:   tree(T) → nat,
              nonempty_tree(T) → posint.
```

$$\begin{aligned}
\texttt{depth_tree(etree)} &= 0. \\
\texttt{depth_tree(netree(L,R,_))} &= \texttt{depth_tree(L)} + 1 \\
&\leftarrow \texttt{depth_tree(L)} \geq \texttt{depth_tree(R)}. \\
\texttt{depth_tree(netree(L,R,_))} &= \texttt{depth_tree(R)} + 1 \\
&\leftarrow \texttt{depth_tree(L)} \leq \texttt{depth_tree(R)}.
\end{aligned}$$

Man beachte, daß die beiden Ranks von `depth_tree` regulär sind. Für den Term `depth_tree(netree(etree, etree, 1))` kann kann man zum Beispiel den eindeutigen kleinsten Typ `posint` ableiten.

In TEL ist es auch möglich Konstanten zu definieren. Diese heißen *Parameter* und werden wie nullstellige Funktionen definiert.

Wir kommen nun zu *Queries*. Diese haben die Form

$$C_1 \;\&\; \ldots \;\&\; C_n.$$

und sind einfach eine Konjunktion von Conditions. Betrachten wir beispielsweise die folgende Query.

```
contains_int(netree(netree(etree, etree, 2), etree, 4), X).
```

Bevor diese abgearbeitet wird, prüft der Typ-Checker sie auf Typ-Inkonsistenz und leitet anhand der Deklaration von `contains_int` für die Variable `X` den Typ `int` ab. Da, wie bereits ausgeführt, zur Laufzeit keine Typ-Unifikation durchgeführt wird, bekommen wir als Antwort die Belegung

$$\texttt{X} = 4\texttt{: int},$$

obwohl der kleinste Typ von 4 eigentlich `posint` ist. Um Rechenaufwand zu sparen, wird also einfach der vorher abgeleitete Typ übernommen. Wir können nun über Backtracking weitere Antworten anfordern, und erhalten zunächst

$$\texttt{X} = 2\texttt{: int},$$

und schließlich

`failed`

als Antwort.

4.1.3 Wohlgetyptheit

Die Wohlgetyptheit von Klauseln und Queries ist im Großen und Ganzen analog zu Abschnitt 3.6 definiert, wobei Funktionen wie Features behandelt werden. Eine genaue Beschreibung findet man in [Sm 88].

Die Grundoperationen des Typ-Checkens sind das Ableiten von kleinsten Typen für Terme, und das Verschärfen eines Präfix durch ein Containment. Die Regularitätsforderung an Funktionsdeklarationen garantiert die eindeutige Existenz eines kleinsten Typs für jeden Werteterm. Das Ableiten dieses kleinsten Typs wurde bereits beschrieben (Funktion σ^P in Abschnitt 3.2). Beim Verschärfen eines Präfix durch ein Containment werden Funktionen und Konstruktoren unterschiedlich behandelt. Gegeben ein Präfix P und ein Containment $f(\vec{t})\colon\sigma$, so ist ein Rückschluß von σ und der Funktions-Deklaration von f auf die Typen von $\vec{t}$ (wie es bei Konstruktoren geschieht) nicht möglich. Der Präfix bleibt also konstant. Dies liegt darin begründet, daß Funktionen im Gegensatz zu Konstruktoren, keine Typen definieren. Abgesehen davon, kann auch die Existenz mehrerer Ranks eine eindeutige Auswahl unmöglich machen, da die Regularität nicht "rückwärts" gilt. Wir werden diesem Problem im nächsten Kapitel (beim Typ-Checken von Features) wiederbegegnen.

Das Typ-Checken von Klauseln und Queries läuft wieder so ab, daß der Ausgangspräfix beim Durchgehen der Conditions (von links nach rechts) immer weiter verschärft wird. Klauselkopf, relationale Atome, Gleichungen und Containments werden genauso gecheckt, wie in Abschnitt 3.6 beschrieben. Der Typ-Checker merkt sich jedoch (aus Effizienzgründen) die durch die Gleichungen implizierte Äquivalenzrelation auf Variablen *nicht*, so daß er – dadurch implizierte – gemeinsame Typabsenkungen nicht immer durchführt (siehe auch Abschnitt 3.6). Solche Fälle sind

jedoch sehr selten, und können durch die Qualifikation der betroffenen Variablen vermieden werden.

Außerdem testet der Typ-Checker auch noch die Konsistenz des Datenflußes. Dies geschieht, indem überprüft wird, ob die abgeleiteten Typen für Ein- und Ausgabeargumente unter den deklarierten Typen liegen.

4.1.4 Module

TEL stellt ein einfaches nichtparametrisches Modulkonzept zur Verfügung. Dieses unterstützt information hiding, abstrakte Datentypen und getrennte Kompilation, und ermöglicht somit das inkrementelle Erstellen großer Programmpakete.

Ein Modul besteht aus einem *Interface* und einem *Body*. Im Interface ist definiert, welche Module importiert werden, und welche Objekte exportiert werden. Im Body müssen dann diejenigen Export-Objekte implementiert werden, die nicht importiert werden. Dazu nötige Hilfsobjekte kann der Body über zusätzliche lokale Untermodule einführen. Die Idee dabei ist, daß allgemeine, in vielen Modulen eines Bereichs benötigte Datenstrukturen und Operationen über das Interface importiert werden, während spezielle Hilfsoperationen über den Body eingeführt werden. *Views* sind Module ohne Body. Sie ermöglichen eine anwendungsbezogene Sicht auf häufig benutzte Module und filtern die für eine spezielle Anwendung nicht benötigten Objekte heraus. Beim Entwurf eines größeren Programmpaketes wird man zunächst eine Interface- und View-Struktur festlegen. Danach ist dann eine getrennte Kompilation möglich. Dabei müssen die Module hierarchisch angeordnet werden, das heißt ein Modul kann sich (auch auf Umwegen) nicht selbst importieren.

Wir wollen nun den Import und Export von Objekten genauer betrachten. Dazu benötigen wir den Begriff der *Signatur*. Eine Signatur ist eine Menge von Typ-, Relations-, und Funktionsdeklarationen. Wir stellen an Signaturen gewisse Anforderungen, die für die POS-Typdisziplin notwendig sind. Diese entsprechen im wesentlichen den Anforderungen an Typ-Spezifikationen aus Kapitel 3, und beinhalten Abgeschlossenheit (alle vorkommenden Symbole haben eine Deklaration), Wohlfundiertheit des Typsystems, Existenz von Suprema und Regularität der Funktions-Ranks. Eine vollständige Übersicht findet man in [Sm 88]. Wenn eine Signatur alle diese Anforderungen erfüllt, so nennen wir sie *konsistent*.

Jedes Modul definiert drei Signaturen, die alle konsistent sein müssen:

- *Export-Signatur*,
- *Import-Signatur*,
- *Lokale Signatur*.

Ein View besitzt nur eine Import- und Export-Signatur. Die Import-Signatur eines Moduls spezifiziert die über das Interface (den View) des Moduls importierten

Deklarationen, und setzt sich aus den Export-Signaturen der Importmodule zusammen. Die lokale Signatur eines Moduls enthält alle im Body des Moduls benutzbaren Objekte. Sie ist die Vereinigung aus der Import-Signatur des Moduls, den Export-Signaturen der im Body importierten Module, und den lokalen Deklarationen des Bodies. Die Export-Signatur eines Moduls setzt sich schließlich aus den transferierten Objekten der Import-Signatur, sowie aus lokalen Deklarationen im Interface zusammen (letztere müssen im Body implementiert sein). Beim View besteht die Export-Signatur nur aus den Transfers der Import-Signatur.

Die vom Interface, View, oder Body importierten Module werden über Deklarationen der folgenden Form bestimmt:

imports $M_1, M_2, \ldots, M_n$.

Diejenigen Objekte aus der Import-Signatur eines Moduls, die über die Export-Signatur weitergereicht werden sollen, werden durch eine sogenannte *Transfer-Deklaration* spezifiziert. Diese kann beispielsweise folgendermaßen aussehen.

from M: D, T **abstract.**

Dabei ist D der Name eines Objektes aus der zugrundeliegenden Import-Signatur, welches im Modul M definiert wurde. Gleiches gilt für den Typ T, das **abstract** hinter T zeigt jedoch an, daß T als *abstrakter Datentyp* exportiert wird. Das bedeutet, daß seine Untertypen und Konstruktoren nicht in die Export-Signatur übernommen werden. Dies ist zum Beispiel dann sinnvoll, wennn gleichzeitig Operationen auf diesem Typ exportiert werden, der Typ als solcher aber in den importierenden Modulen nicht interessiert.

Die lokalen Deklarationen von Objekten im Interface eines Moduls entsprechen ihren Deklarationen im Body des Moduls (natürlich ohne die dazugehörigen Klauseln). Es gibt jedoch die zusätzliche Möglichkeit einen im Body (konkret) definierten Typ T abstrakt zu exportieren. Dann hat die entsprechende Deklaration im Interface die Form T := `abstract`. Die Wirkung ist wieder die, daß Konstruktoren und Untertypen von T versteckt werden (information hiding). Ein weiterer Vorteil ist der, daß Signaturen klein gehalten werden, was für eine effiziente Abarbeitung wichtig ist.

TEL verwaltet das Modulsystem derart, daß Zyklen und inkonsistente Import- oder Exportdeklarationen erkannt werden. Zu einem Zeitpunkt kann immer nur ein Modul geöffnet sein. Queries dürfen nur Objekte aus der lokalen Signatur des geöffneten Moduls enthalten.

4.2 Struktur des TEL-Systems

Die Version 0.9 des TEL-Systems setzt auf Quintus-Prolog auf und wurde auf Apollo-Workstations unter UNIX 4.2 BSD entwickelt. Kern des TEL-Systems ist ein Compiler, der TEL nach Prolog übersetzt. Ein TEL-Programm hat daher etwa die

gleiche Laufzeitperformanz wie das entsprechende Prolog-Äquivalent. Die Entwicklungszeiten für (größere) TEL-Programme dürften aber, aufgrund der komfortablen Fehlerlokalisierung, wesentlich unter denen von Prolog-Programmen liegen. Der Nachteil der Übersetzung nach Prolog ist der, daß getypte Unifikation mehr oder weniger auf ungetypte Unifikation abgebildet wird (siehe auch Abschnitt 4.1.2). Das langfristige Ziel muß also die Entwicklung einer abstrakten Maschine für TEL sein.

TEL wurde in einem Bootstrapping-Prozeß entwickelt, ist also zum größten Teil selbst in TEL geschrieben. Lediglich das Laufzeit-System ist in Prolog implementiert. Es besteht im wesentlichen aus Code für die Builtins von TEL. Der in TEL geschriebene Teil des TEL-System hat eine klare, funktionale Struktur, welche sich direkt in der Modulhierarchie wiederspiegelt. Dies ermöglicht das separate Kompilieren von Teilfunktionen und führt somit zu einer leichten Wartbarkeit des TEL-Systems.

Wir können in diesem Rahmen leider nur eine stark vergröberte Darstellung der Modulhierarchie des TEL-Systems geben. Eine genauere Beschreibung findet man in [NS 90]. An der Spitze der System-Hierarchie steht die *Manager*-Komponente. Sie steuert den Dialog mit dem Benutzer, verwaltet die aktuelle Modulhierarchie, und stößt die zur Kompilation von Modulen, oder zur Abarbeitung von Queries benötigten Untermodule an. Die wichtigsten Teilkomplexe des TEL-Systems sind:

- Parser
- Signaturanalyse
- Typ-Checker
- Code-Erzeugung.

Sie werden sowohl zur Kompilation von Modulen, als auch zur Abarbeitung von Queries eingesetzt.

Wir besprechen nun, wie das TEL-System die verschiedenen Objekte eines TEL-Programms intern repräsentiert. Der TEL-Compiler legt für jedes Objekt eines Moduls (Typsymbol, Konstruktor, Relatiossymbol, Funktionssymbol) genau einen Eintrag an, in dem sämtliche zu dem Objekt gehörige Informationen abgespeichert werden. Einträge werden über den Datentyp `entry` beschrieben. Dieser hat für jede Objektart einen entsprechenden Untertyp, nämlich `full_type_entry`, `abbreviation_type_entry`, `function_entry`, `parameter_entry`, `relorproc_entry`, und `constructor_entry`. Einträge werden in einer Art Slot-Filler-Notation vorgenommen. Wir illustrieren dies an einem Beispiel. Gegeben sei ein Modul "test", in dem die folgende Typdeklaration vorkomme.

$$\texttt{t1(T)} := \texttt{t2(T)} +\!+ \{\texttt{c1}\}.$$

gleiche Laufzeitperformanz wie das entsprechende Prolog-Äquivalent. Die Entwicklungszeiten für (größere) TEL-Programme dürften aber, aufgrund der komfortablen Fehlerlokalisierung, wesentlich unter denen von Prolog-Programmen liegen. Der Nachteil der Übersetzung nach Prolog ist der, daß getypte Unifikation mehr oder weniger auf ungetypte Unifikation abgebildet wird (siehe auch Abschnitt 4.1.2). Das langfristige Ziel muß also die Entwicklung einer abstrakten Maschine für TEL sein.

TEL wurde in einem Bootstrapping-Prozeß entwickelt, ist also zum größten Teil selbst in TEL geschrieben. Lediglich das Laufzeit-System ist in Prolog implementiert. Es besteht im wesentlichen aus Code für die Builtins von TEL. Der in TEL geschriebene Teil des TEL-System hat eine klare, funktionale Struktur, welche sich direkt in der Modulhierarchie wiederspiegelt. Dies ermöglicht das separate Kompilieren von Teilfunktionen und führt somit zu einer leichten Wartbarkeit des TEL-Systems.

Wir können in diesem Rahmen leider nur eine stark vergröberte Darstellung der Modulhierarchie des TEL-Systems geben. Eine genauere Beschreibung findet man in [NS 90]. An der Spitze der System-Hierarchie steht die *Manager*-Komponente. Sie steuert den Dialog mit dem Benutzer, verwaltet die aktuelle Modulhierarchie, und stößt die zur Kompilation von Modulen, oder zur Abarbeitung von Queries benötigten Untermodule an. Die wichtigsten Teilkomplexe des TEL-Systems sind:

- Parser
- Signaturanalyse
- Typ-Checker
- Code-Erzeugung.

Sie werden sowohl zur Kompilation von Modulen, als auch zur Abarbeitung von Queries eingesetzt.

Wir besprechen nun, wie das TEL-System die verschiedenen Objekte eines TEL-Programms intern repräsentiert. Der TEL-Compiler legt für jedes Objekt eines Moduls (Typsymbol, Konstruktor, Relatiossymbol, Funktionssymbol) genau einen Eintrag an, in dem sämtliche zu dem Objekt gehörige Informationen abgespeichert werden. Einträge werden über den Datentyp `entry` beschrieben. Dieser hat für jede Objektart einen entsprechenden Untertyp, nämlich `full_type_entry`, `abbreviation_type_entry`, `function_entry`, `parameter_entry`, `relorproc_entry`, und `constructor_entry`. Einträge werden in einer Art Slot-Filler-Notation vorgenommen. Wir illustrieren dies an einem Beispiel. Gegeben sei ein Modul "test", in dem die folgende Typdeklaration vorkomme.

$$\texttt{t1(T)} := \texttt{t2(T)} ++ \{\texttt{c1}\}.$$

Dann wird beispielsweise für das Objekt `t1` ein Eintrag T_1 angelegt mit:

$T_1 =$ `full-type-entry`

`name:`	`'t1'`
`module_name:`	`'test'`
`line_number:`	`15`
`code_name:`	`'37t1'`
`formal_arguments:`	[ev('T')]
`given_subtypes:`	[ec(T_2, [ev('T')])]
`constructors:`	[C_1]
`complete_subtypes:`	_
`suprema:`	_

Diese Slot-Filler-Notation wird bisher über Konstruktor-Typen realisiert, wobei die entsprechenden Selektoren (`name`, `module_name`, etc.) für alle Eintragstypen definiert werden mußten. Die Konstruktor-Darstellung des Eintrags T_1 hat beispielsweise die Form:

$$T_1 = \texttt{ft('t1', 'test', 15, ...)}.$$

Offenbar wären Feature-Typen eine wesentlich adäquatere Darstellungsweise für Einträge. Im Verlauf des Bootstrapping-Prozeßes von Feature-TEL werden wir deshalb den Typ `entry` als Feature-Typ reimplementieren (siehe Kapitel 7).

Oft enthalten die Slots von Objekten Verweise auf andere Einträge (siehe den `constructors`-Slot). In unserem Beispiel seien T_2 und C_1 die Einträge für die Objekte `t2` und `c1`. Verweise werden einfach durch Unifikation mit dem entsprechenden Eintrag erzeugt. Dadurch wird zum einen Platz gespart (für jedes Symbol existiert wirklich nur ein Eintrag), zum anderen ist in jedem Eintrag auch die volle Information über die mit ihm verbundenen Einträge enthalten (man braucht diese also nicht irgendwo nachzuschlagen). Die Verpointerung geht sogar soweit, daß Terme auf "Entry-Terme" abgebildet werden, indem die Funktionssymbole durch die dazugehörigen Einträge ersetzt werden. Entry-Terme werden durch den Datentyp `eterm` repräsentiert. Dieser hat den Konstruktor `ev` zur Darstellung von Variablen, und den Konstruktor `ec` zur Repräsentation zusammengesetzter Terme. Dabei ist das erste Argument von `ec` das Topsymbol des Terms und hat den Typ `entry`. Das zweite Argument von `ec` ist die Argumentliste des Terms und hat den Typ `list(eterm)`. Beispielsweise wird der Term `t2(T)` dargestellt als

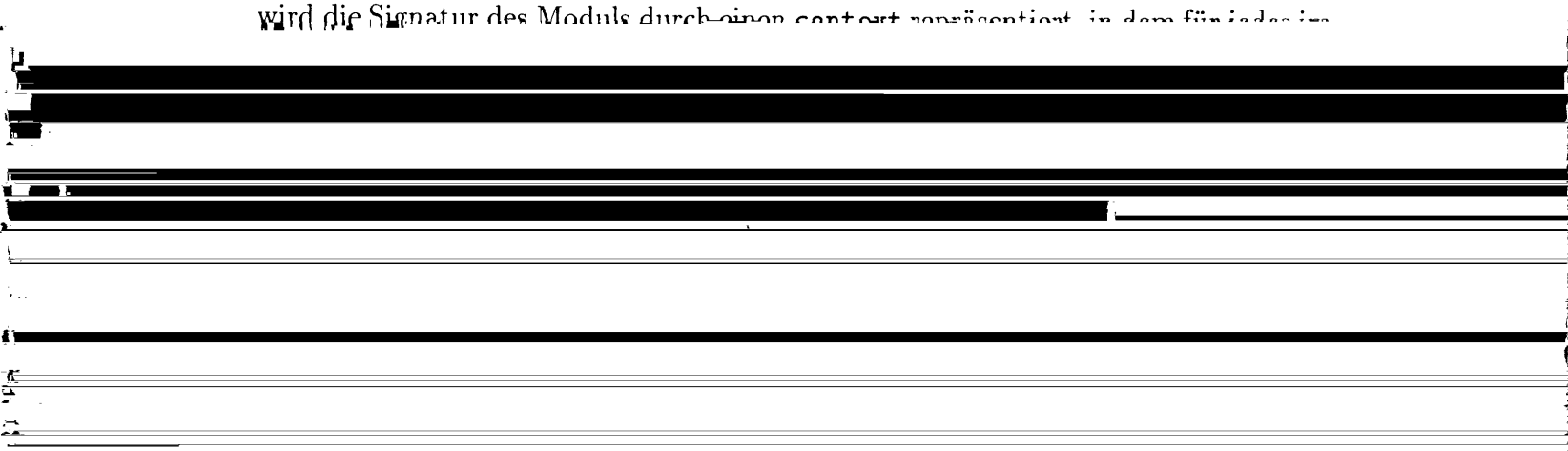
$$\texttt{ec(ft('t2', 'test', 34, ...), [ev('T')])}.$$

Im TEL-System werden Signaturen durch die Datenstruktur `context` repräsentiert. Diese kann man sich als einen, mit dem jeweiligen Modulnamen markierten Suchbaum von Einträgen vorstellen. Als Schlüssel dient dabei der `name`-Slot der Einträge.

Wir beschreiben nun das Zusammenspiel der einzelnen Komponenten des TEL-Systems beim Kompilieren eines Moduls. Das Abarbeiten von Queries läuft sehr ähnlich ab, und wird deshalb nicht mehr gesondert beschrieben.

4.2.1 Parser

Zunächst wird das entsprechende Modul eingelesen und vom Parser in die interne Darstellung des TEL-Systems übersetzt. Diese besteht aus zwei Teilen. Zum einen wird die Signatur des Moduls durch einen `context` repräsentiert, in dem für jedes im



Modulcode vorkommende Objekt ein entsprechender Eintrag vorgenommen wird. Zum anderen wird der Klauselcode des Moduls in die *abstrakte Syntax* von TEL übersetzt. Diese kann man sich als eine Abbildung der BNF-Syntax-Diagramme von TEL auf eine TEL-Typhierarchie vorstellen. Die Verbindung zum Modul-Kontext wird hergestellt, indem Werte- und Typterme in Entry-Terme übersetzt werden. Dies geschieht wieder über Unifikation mit den entsprechenden Signatureinträgen und hat den großen Vorteil, daß an jeder Stelle der Programmrepräsentation die volle Signatur-Information lokal vorhanden ist. Wir illustrieren die Idee an einem kleinen Beispiel. Im Modulcode komme die folgende Gleichung vor:

$$\texttt{c1(f1(X))} = \texttt{Y}.$$

Die abstrakte Syntax davon sieht dann folgendermaßen aus.

$$\texttt{eqcond}(\texttt{ec}(C_1, [\texttt{ec}(F_1, [\texttt{ev('X')}])]), \texttt{ev('Y')})$$

Dabei seien C_1 und F_1 die Signatureinträge für die Objekte `c1` und `f1`.

Der Parser ist im Stile eines endlichen Automaten implementiert, der abhängig vom aktuellem Zustand und dem anliegenden Eingabe-Token in den nächsten Zustand übergeht. Beim Aufbau der internen Repräsentation eines Programms erkennt er Syntaxfehler und Mehrfach-Deklarationen.

4.2.2 Signatur-Analyse

Die Einträge, die der Parser im Modul-Kontext vornimmt, sind in der Regel nicht vollständig (es sind nicht alle Slots gefüllt), sondern spiegeln nur die unmittelbare Quellcode-Information wieder. So wird insbesondere zum Typ-Checken zusätzliche Information benötigt (Untertyp-Beziehungen, Suprema, etc.). Die fehlende Information wird in der Signatur-Analyse vervollständigt. So werden Import-, Export-, und lokale Signaturen aufgebaut und deren Konsistenz gecheckt. Der Konsistenzcheck setzt sich aus einer Reihe unabhängiger Teil-Operationen (Check der Wohlfundiertheit, Berechnen der Untertyp-Relationen, Berechnen der Suprema, etc.) zusammen, die bei der Implementierung in einzelnen Modulen separiert wurden.

gleichzeitig in eine Zwischensprache, die sogenannte *Intermediate-Language*. Diese dient als Bindeglied zwischen Prolog-Code und abstrakter Syntax, und unterscheidet sich von der letzteren im wesentlichen dadurch, daß Funktionsaufrufe aufgefaltet werden. Das Auffalten bewirkt, daß die Terme der Intermediate-Language kanonisch sind, und damit sofort in Prolog-Terme umgesetzt werden können. Zur Illustration zeigen wir, wie die Beispielgleichung aus Unterabschnitt 4.2.1 in Intermediate-Language dargestellt wird.

$$\texttt{fun\_call}(F_1, [\texttt{ev('X')}], \texttt{ev('Z')}) \;\&\; \texttt{eq\_call}(\texttt{ec}(C_1, [\texttt{ev('Z')}]), \texttt{ev('Y')})$$

Dabei bezeichnen F_1 und C_1 wieder die Signatureinträge für die Symbole `f1` und `c1`.

Die modulare Struktur des Typ-Checkers entspricht der hierarchischen Struktur der abstrakten Syntax (Programm → Klausel → Condition → Term). Die Funktion des Typ-Checkers wird durch die Typ-Check-Regeln in [Sm 88] beschrieben, welche sich direkt im Code wiederspiegeln.

4.2.4 Code-Erzeugung

Die Code-Erzeugungs-Komponente setzt Intermediate-Language-Code in entsprechenden Prolog-Code um. Um den Code verschiedener Module unterscheiden zu können, bekommt jedes Modul eine sogenannte *Disambiguierung* (zum Beispiel "37"). Der Codename eines Objekts O, das in Modul M definiert wurde, setzt sich dann aus der Disambiguierung von M und dem Namen von O zusammen. Außer dem Klausel-Code wird auch noch Typ-Code für jedes Modul erzeugt. Dieser sieht im wesentlichen so aus, daß für jeden Typ eine einstellige Relation (bezeichnet mit seinem Codenamen) erzeugt wird. Diese gilt genau dann, wenn für den Argumentterm dieser Typ abgeleitet werden kann. Für ein Containment $s{:}\,\sigma$ wird dann ein Aufruf der zu σ gehörigen Relation mit Argument s erzeugt. Dies geschieht jedoch nur, wenn der bisher abgeleitete Typ von s größer als σ ist. Funktionen werden wie zweistellige Relationen behandelt. Der Code für die Beispielgleichung aus dem vorhergehenden Abschnitt sieht folgendermaßen aus.

$$\texttt{37f1(X,Z)}\,,\ \texttt{c1(Z)} = \texttt{Y}$$

Dabei sei `37` die Disambiguierung des zugrundeliegenden Moduls.

Kapitel 5

Sprachbeschreibung von Feature-TEL

In diesem Kapitel beschreiben wir Syntax und Semantik von Feature-TEL, einer Erweiterung von TEL um Feature-Typen. Beim Design von Feature-TEL gehen wir von dem Grundsatz aus, daß alle früheren Sprachkonstrukte von TEL mit der gleichen Bedeutung weiterexistieren sollten. Damit können alle TEL-Programme auch in Feature-TEL abgearbeitet werden. Weiterhin wollen wir für unsere Feature-Typen weitgehend die in Kapitel 3 beschriebene Form übernehmen, und damit gleichzeitig die dort beschriebene Semantik. Feature-Typen werden also über versteckte (für den Benutzer nicht sichtbare) Konstruktoren implementiert. Da Feature-TEL ebenso wie TEL nach Prolog übersetzt wird, können der in Kapitel 3 beschriebene Interpreter und Constraint-Solver nicht direkt umgesetzt werden. Es gibt jedoch einen Kompromiß, bei dem Bindungen von Wertevariablen an Feature-Typen durch Backtracking über die entsprechenden minimalen Feature-Typen realisiert werden. Somit steht Feature-Unifikation in eingeschränkter Form zur Verfügung. Der Typ-Inferenzer von Feature-TEL orientiert sich weitgehend an dem in Abschnitt 3.6 beschriebenen Algorithmus. Ein interessanter neuer Punkt an Feature-TEL ist die Einbettung von Feature-Hierarchien in ein Modulkonzept. Außerdem lassen wir Feature-Terme in Klauseln zu, wenn auch nur in eingeschränkter Form.

Da der POS-Teil von Feature-TEL genau dem alten TEL (siehe Kapitel 4) entspricht, werden wir hier nur die neuen Konstrukte beschreiben. Dabei gehen wir wie folgt vor. In Abschnitt 5.1 präsentieren wir Syntax und Semantik von Feature-Typ-Deklarationen, insbesondere in Verbindung mit dem Modulkonzept von TEL. Damit kennen wir die Signaturen von Feature-TEL und können uns in Abschnitt 5.2 mit den Constraints (Conditions) über Feature-Typen beschäftigen. Dazu gehören auch Feature-Terme. Wir gehen insbesondere auf Aspekte der Abarbeitung und des Typ-Checkens der neuen Conditions ein.

Wir beschreiben die Syntax von Sprachkonstrukten über eine *BNF-ähnliche Notation*, die wir hier kurz erklären :

1. Syntaktische Kategorien werden *kursiv* gedruckt
2. Jede Syntaktische Kategorie wird über eine Regel der Form

$$C \longrightarrow S_1 \mid S_2 \mid \ldots \mid S_n$$

beschrieben, und kann eine beliebige der alternativen Formen $S_1, \cdots, S_n$ annehmen.

3. Ein Terminal 'T' zeigt an, daß T an dieser Stelle erscheinen muß.
4. Eine optionale Form $[F]$ kann auch weggelassen werden.
5. Eine Listen-Form $\{F\}$ bedeutet, daß die Form F mindestens einmal, oder aber mehrmals und durch ',' getrennt vorkommen muß.
6. Eine Stern-Form $(F)^*$ steht für eine möglicherweise leere Folge von F's.

5.1 Deklaration von Feature-Typen

In diesem Abschnitt beschreiben wir, wie Feature-Signaturen in Feature-TEL aufgebaut werden und welche Anforderungen wir an sie stellen. Dazu schränken wir uns in Unterabschnitt 5.1.1 zunächst auf lokale Deklarationen ein, das heißt wir vernachlässigen das Zusammenspiel mit anderen Modulen. Es wird gezeigt wie man Feature-Hierarchien aufbaut und was dabei zu beachten ist. Unterabschnitt 5.1.2 diskutiert Abstraktionsmöglichkeiten von Feature-Hierarchien beim Modultransfer und erläutert, wie Export- und Import-Signaturen aufgebaut werden.

5.1.1 Lokale Feature-Typen

Wir führen nun die sprachlichen Konstrukte zur Deklaration von Feature-Hierarchien ein. Dabei gehen wir (aus Gründen der Klarheit) zunächst davon aus, daß keine Objekte importiert werden.

Signaturen in Feature-TEL entsprechen genau den in Kapitel 3 beschriebenen Feature-Spezifikationen. Damit haben unsere Feature-Typen eine wohldefinierte Semantik. Wir fassen noch einmal kurz die wichtigsten Designentscheidungen zusammen (siehe auch Abschnitt 2.3).

- Feature-Typen sind monomorph und werden wie gewöhnliche Typkonstante behandelt. Dies bedeutet insbesondere, daß sie in Typtermen, und damit überall wo Typterme verwendet werden, vorkommen dürfen.
- Feature-Typen und POS-Typen sind unvergleichbar.

- In Feature-Deklarationen dürfen weder Feature-Terme noch Werteterme vorkommen.
- Die Semantik von Feature-Typen wird mittels impliziter Konstruktoren auf die von Konstruktor-Typen zurückgeführt. Dabei werden jedoch nur den minimalen Feature-Typen implizite Konstruktoren zugeordnet.
- Features können mehrere Typ-Deklarationen haben.

Unsere Signaturen müssen zusätzlich zu den bisherigen Anforderungen an TEL-Signaturen (siehe Abschnitt 4.1.4) insbesondere auch die in Kapitel 3 beschriebenen zusätzlichen Anforderungen erfüllen, nämlich Regularität der Feature-Ranks und Bewohntheit der nichtminimalen Feature-Typen ("Minimalitätsbedingung"). Unsere Syntax für *Feature-Typ-Deklarationen* entspricht der in Abschnitt 2.1 beschriebenen "textuellen Repräsentation" von Feature-Hierarchien. Sie hat die folgende Form:

Feature-Typ-Dekl $\longrightarrow$ *Feature-Typ-Name* ':=' *Feature-Typ-RS* '.'

Feature-Typ-RS $\longrightarrow$ [*Feature-Typ-Name* ('*' *Feature-Typ-Name*)*]
Feature-Def

Feature-Def $\longrightarrow$ '[' ']' |
'[' { *Feature-Name* ':' *Grundsortenterm* } ']'

Feature-Typ-Name $\longrightarrow$ *Bezeichner*

Feature-Name $\longrightarrow$ *Bezeichner*

Ein *Grundsortenterm* wird nur aus Sortenfunktionen und Sortenkonstanten aufgebaut, kann also auch Feature-Typen enthalten. Wir sehen, daß Feature-Typen über ihre direkten Obertypen und ihre gegebenen (nicht geerbten) Features definiert werden. Daraus ergeben sich direkt die Inklusionsordnung und die Feature-Ranks. Wir illustrieren dies an einem Beispiel welches in Abbildung 5.1 veranschaulicht wird. In einem Modul `uni` seien folgende Feature-Typ-Deklarationen gegeben:

```
        person := [name:string].
  angestellter := person[chef:angestellter].
       dozent := angestellter[chef:professor,
                              schueler:list(student)].
   sekretaerin := angestellter[ ].
       student := person[semester:nat].
        hoerer := student[ ].
 uebungsleiter := student*dozent[schueler:list(hoerer)].
     professor := dozent[ ].
```

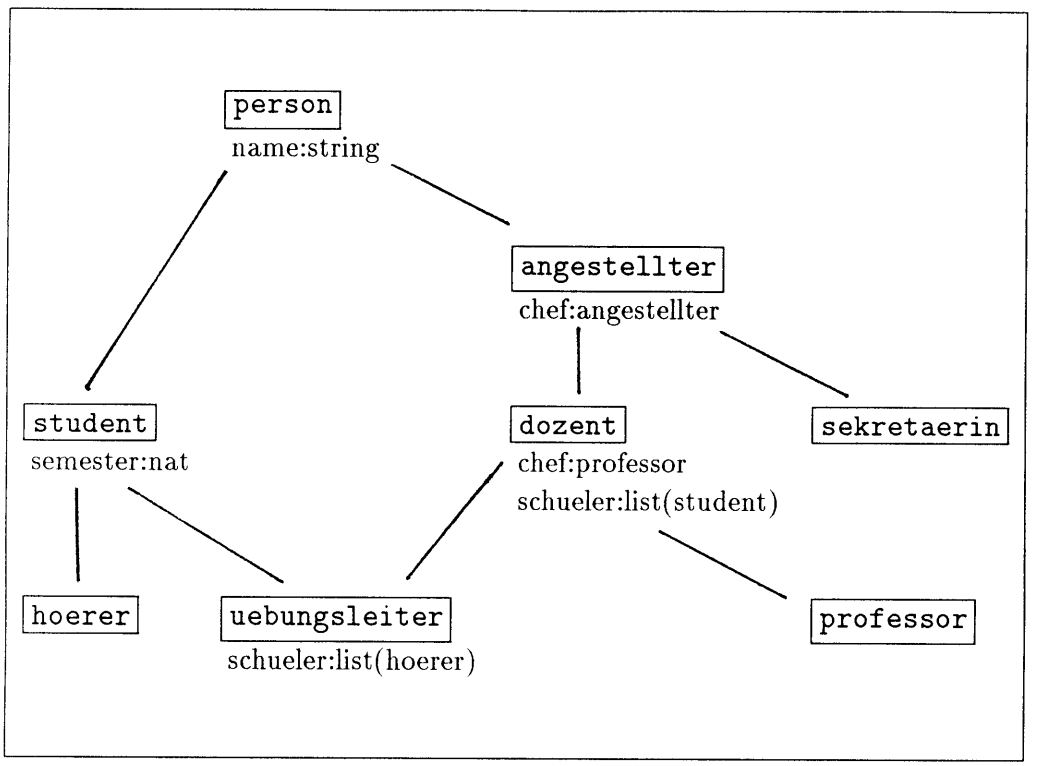


Abbildung 5.1: Signatur von `uni`

Diese Deklarationen spezifizieren die folgenden Feature-Ranks und eine Ordnung auf den Feature-Typen.

$$\begin{aligned}
\texttt{hoerer} &\leq \texttt{student} \\
\texttt{student} &\leq \texttt{person} \\
\texttt{uebungsleiter} &\leq \texttt{dozent} \\
\texttt{professor} &\leq \texttt{dozent} \\
\texttt{dozent} &\leq \texttt{angestellter} \\
\texttt{sekretaerin} &\leq \texttt{angestellter} \\
\texttt{angestellter} &\leq \texttt{person}
\end{aligned}$$

$$\begin{aligned}
\texttt{name} &: \texttt{person} \rightarrow \texttt{string}. \\
\texttt{semester} &: \texttt{student} \rightarrow \texttt{nat}. \\
\texttt{chef} &: \texttt{angestellter} \rightarrow \texttt{angestellter}, \\
&\quad\ \texttt{dozent} \rightarrow \texttt{professor}. \\
\texttt{schueler} &: \texttt{dozent} \rightarrow \texttt{list(student)}, \\
&\quad\ \texttt{uebungsleiter} \rightarrow \texttt{list(hoerer)}.
\end{aligned}$$

Die impliziten Konstruktoren und die Selektorgleichungen für die Features dienen dem Benutzer von Feature-TEL lediglich als Modell zur Illustration der Semantik von Feature-Typen. Sie sind nach außen nicht sichtbar! Intern werden Feature-Typen aber genau über diese impliziten Konstruktoren implementiert.

$$
\begin{aligned}
\texttt{hoerer} &:= \{c_\text{h}\colon \texttt{string} \times \texttt{nat}\} \\
\texttt{uebungsleiter} &:= \{c_\text{u}\colon \texttt{string} \times \texttt{nat} \times \texttt{professor} \times \texttt{list(hoerer)}\} \\
\texttt{professor} &:= \{c_\text{p}\colon \texttt{string} \times \texttt{professor} \times \texttt{list(student)}\} \\
\texttt{sekretaerin} &:= \{c_\text{s}\colon \texttt{string} \times \texttt{angestellter}\}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{name}(c_\text{h}(N, \_)) &= N \\
\texttt{name}(c_\text{u}(N, \_, \_, \_)) &= N \\
\texttt{name}(c_\text{p}(N, \_, \_)) &= N \\
\texttt{name}(c_\text{s}(N, \_)) &= N \\
\\
\texttt{chef}(c_\text{u}(\_, \_, C, \_)) &= C \\
\texttt{chef}(c_\text{p}(\_, C, \_)) &= C \\
\texttt{chef}(c_\text{s}(\_, C)) &= C \\
\\
\texttt{schueler}(c_\text{u}(\_, \_, \_, S)) &= S \\
\texttt{schueler}(c_\text{p}(\_, \_, S)) &= S \\
\\
\texttt{semester}(c_\text{h}(\_, S)) &= S \\
\texttt{semester}(c_\text{u}(\_, S, \_, \_)) &= S
\end{aligned}
$$

Wir fassen zusammen:

> Feature-Typen werden über ihre direkten Obertypen und ihre gegebenen Features deklariert. Diese spezifizieren eindeutig eine Feature-Signatur, welche den Feature-Typen eine wohlfundierte Semantik gibt. Die dazugehörigen impliziten Konstruktoren sind für den Benutzer nicht sichtbar!

5.1.2 Import und Export von Feature-Typen

Wir erläutern nun, wie Feature-Typen in das Modulkonzept von TEL eingebettet werden. Dazu beschreiben wir zunächst das Verhältnis zwischen der lokalen, der Import- und der Export-Signatur eines Moduls bezüglich Feature-Deklarationen.

Anschließend gehen wir dann auf Abstraktionen von Feature-Typen in Transfer-Deklarationen ein. Es gibt dort die Möglichkeit, nicht benötigte Feature-Typen oder Features aus einer Feature-Hierarchie auszublenden. Schließlich wird beschrieben, wie Import-Signaturen aufgebaut werden. Dabei interessiert besonders, wie verschiedene Ausschnitte der gleichen Feature-Hierarchie zusammengesetzt werden.

Lokale Signatur

Die lokale Signatur von Modulen in Feature-TEL ist genauso wie im bisherigen TEL definiert. Sie setzt sich zusammen aus den lokalen Deklarationen (im Modul-Body), der Import-Signatur und den Export-Signaturen der im Body importierten Module. Wir zeigen hier, in welcher Beziehung die lokalen Feature-Deklarationen aus Interface und Body zueinander stehen müssen. Außerdem wird erläutert, in welcher Weise man importierte Feature-Typen in der lokalen Signatur eines Moduls verwenden darf. Wir beginnen mit der ersten Fragestellung.

- Wie exportiert man lokale Feature-Typen?

Lokale Feature-Typen werden im Interface eines Moduls über die in Unterabschnitt 5.1.1 beschriebenen Feature-Deklarationen exportiert. Wir wissen aus Kapitel 4, daß beim Erstellen eines TEL-Programms zunächst die Interface-Struktur festgelegt wird. Dabei müssen die lokalen Feature-Typ-Deklarationen aus dem Interface im Body exakt wiederholt werden. Insbesondere dürfen keine neuen Unter- oder Obertypen dazukommen. Die Feature-Deklarationen müssen ebenfalls unverändert bleiben. Am sichersten ist es, alle lokalen Deklarationen aus dem Interface sofort in den Body zu kopieren, da auch die Reihenfolge der Deklarationen eine Rolle spielt. Während die Beibehaltung der Reihenfolge aus Effizienzgründen verlangt wird (der Vergleich von lokaler- und Export-Signatur kann dadurch auf syntaktischem Level erfolgen), gibt es für die semantische Identität der lokalen Feature-Typ-Deklarationen in Interface und Body gewichtige Gründe. Wir erläutern sie anhand eines Beispiels.

Im Interface des Moduls `uni` sei die Feature-Hierarchie aus Abbildung 5.1 deklariert. Angenommen wir übernehmen im Body alle Deklarationen aus dem Interface und fügen noch folgende Deklaration hinzu:

```
unipraesident := professor[].
```

Dann hat der Typ `professor` im Interface andere implizite Konstruktoren als im Body, und somit eine andere Bedeutung. Module, die `uni` importieren, gehen also von einer anderen Bedeutung aus, als der, die im Body von `uni` implementiert ist. So können in diesen Modulen Operationen, die ebenfalls aus `uni` importiert wurden, dort unbekannte Objekte erzeugen (zum Beispiel den impliziten Konstruktor von `unipraesident`). Damit ginge die Funktionalität der lokalen Operationen verloren. Ähnlich verhält es sich mit Features. Wenn neue Features dazukommen, ändert sich

die Länge der impliziten Konstruktoren. Wenn Feature-Ranks verändert werden, so ändern sich die Typen von Wertetermen. Unsere Forderung ist daher verständlich. Zum Abschluß wiederholen wir sie in verkürzter Form:

Lokale Feature-Hierarchien aus dem Interface müssen im Body exakt

Wir kommen nun zur zweiten Fragestellung.

- In welcher Form dürfen importierte Feature-Typen in lokalen Feature-Typ-Deklarationen vorkommen?

Importierte Feature-Typen werden als normale Sortenkonstante behandelt und dürfen in Typtermen vorkommen, und somit auch in Relations-, Konstruktor-, und Funktions-Deklarationen. Sie dürfen jedoch nicht als Untertypen in POS-Typ-Deklarationen vorkommen, da sonst die Unvergleichbarkeit von Feature-Typen und POS-Typen verletzt wäre.

Es stellt sich nun die Frage, ob man importierte Feature-Typen auch in lokalen Feature-Typ-Deklarationen zulassen soll. Dies ist im Codomain der lokalen Features durchaus zulässig. Was für Folgen hat es jedoch, wenn wir importierte Feature-Typen direkt als Ober/Unter-Typen in eine lokale Feature-Hierarchie einbauen? Man sieht sofort, daß importierte Feature-Typen keine Obertypen von lokalen Feature-Typen sein dürfen, da die importierten Typen sonst zusätzliche implizite Konstruktoren erhielten. So bleibt nur noch die Möglichkeit, importierte Feature

Typen als Untertypen von lokalen Feature-Typen zu deklarieren. Damit würde man sozusagen die importierte Feature-Hierarchie nach oben erweitern. Vorausset-

Die angesprochenen Operationen können nun auf dem gemeinsamen Obertyp `individuum` arbeiten. Ähnliche Fälle erhält man, wenn man die klassischen Beispiele des nichtmonotonen Schließens (Tweety, Pinguin, etc.) auf Feature-Typen überträgt.

Solche Erweiterungen von importierten Feature-Hierarchien bringen jedoch für die Sprach-Entwicklung eine Menge Probleme, wie sich bei einer probeweisen Implementierung der dazu notwendigen Signatur-Operationen herausstellte. So sind die einzelnen Signatur-Checks ohnehin hochgradig voneinander abhängig, insbesondere müssen Checks auf importierten Objekten in der Regel vor Checks auf lokalen Objekten durchgeführt werden. Für die neuen Zwitterobjekte ist zu klären, wo sie in diesem sensiblen Prozeß eingefügt werden können. Dabei ergeben sich eine Menge prinzipieller Fragen, zum Beispiel: Sind die Features der Vereinigungstypen nun importierte oder lokale Objekte? Wann führt man den Regularitätscheck für sie durch? Dürfen die zu vereinigenden Import-Typen gemeinsame Untertypen haben?

Es stellte sich heraus, daß Erweiterungen importierter Feature-Hierarchien an sehr vielen Stellen die Behandlung von Sonderfällen erfordern. Dies macht den Code der Signatur-Analyse langsam und unübersichtlich. Zusätzliche Probleme ergeben sich bei der Code-Erzeugung für die Features solcher Objekte. An eine Iteration dieses Verfahrens mag man gar nicht denken. Was geschieht, wenn mehrere, in verschiedenen Modulen definierte Erweiterungen derselben Hierarchie gleichzeitig von einem darüberliegenden Modul importiert werden?

Offenbar steht der Nutzen solcher Erweiterungen in keinem Verhältnis zu den damit verbundenen Problemen. Außerdem wird durch einen sorgfältigen Entwurf der Interface-Struktur eine solche nachträgliche Erweiterung hinfällig. Wir verzichten deshalb gänzlich auf importierte Feature-Typen als Untertypen lokaler Feature-Typ-Deklarationen. Insgesamt stehen importierte Feature-Typen damit in folgendem Verhältnis zu lokalen Feature-Typen:

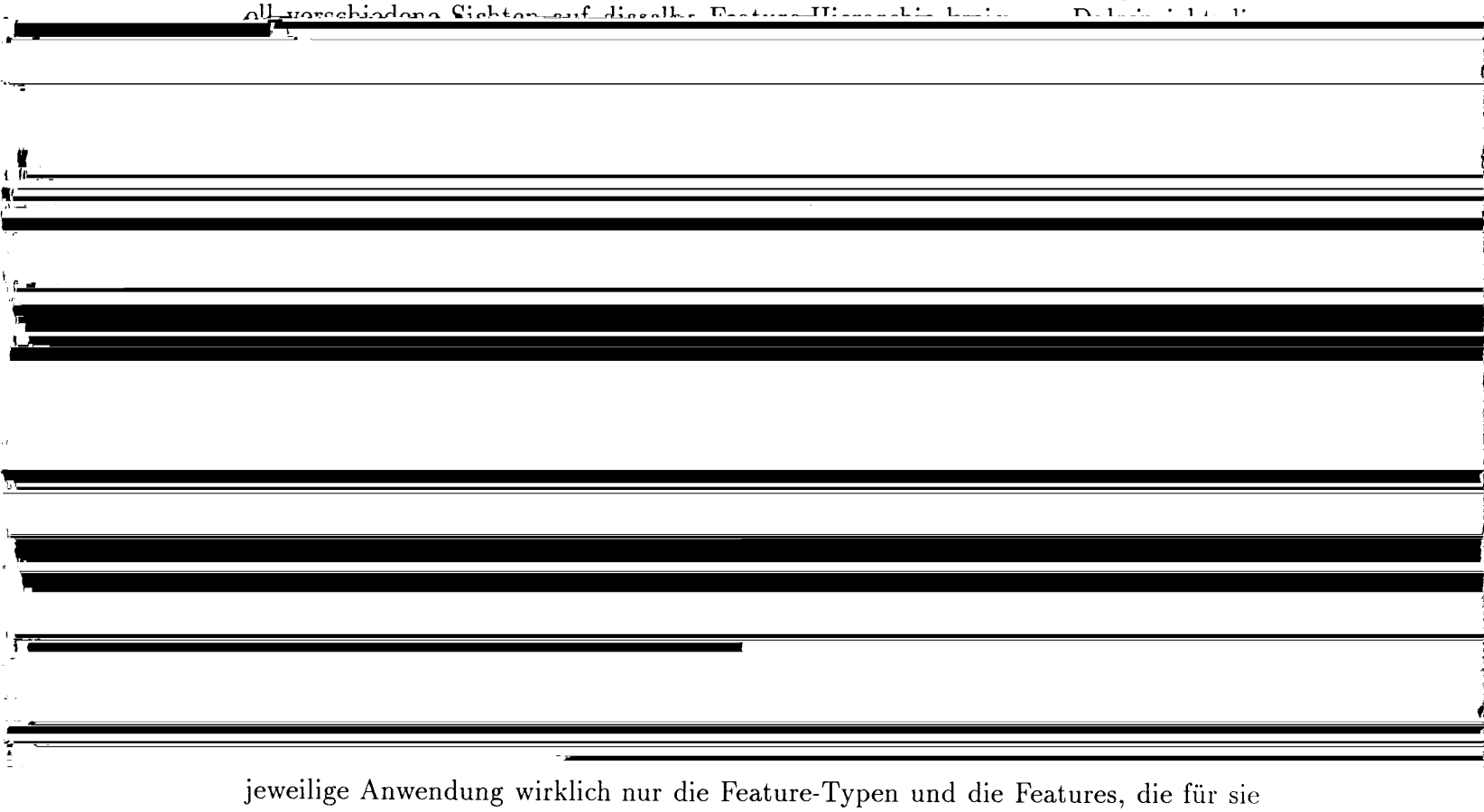
- Importierte Feature-Typen dürfen nicht als Unter- oder Obertypen lokaler Feature-Typen auftreten!
- Jede Feature-Hierarchie ist vollständig in einem Ursprungsmodul definiert!

Export-Signatur

Wie in TEL, setzt sich die Export-Signatur eines Moduls aus den lokalen Deklarationen und den Transfer-Deklarationen zusammen (siehe Kapitel 4). Die lokalen Deklarationen für Feature-Typen entsprechen den Deklarationen im Modul-Body (siehe vorangehender Abschnitt). Die Transfer-Deklarationen spezifizieren diejenigen Objekte der Import-Signatur, welche in die Export-Signatur übernommen werden sollen. Dabei gibt es die Möglichkeit, nicht benötigte Objekte wegzulassen oder Typen als abstrakt zu exportieren. Wir übertragen diesen Mechanismus nun auf Feature-Typen.

Feature-Typen innerhalb von Modul-Hierarchien eignen sich hervorragend zur

Realisierung von "information hiding". Man kann, je nach Anwendung, individuell verschiedene Sichten auf dieselbe Feature-Hierarchie bzw. [illegible]



jeweilige Anwendung wirklich nur die Feature-Typen und die Features, die für sie interessant sind. Umgekehrt kann eine Sicht auch als Schutzmaßnahme für unberechtigten Zugriff auf geschützte Features verstanden werden. Ein weiterer Vorteil dieser Idee liegt darin, daß Signaturen klein gehalten werden.

Bei der Benutzung solcher Sichten geht man in der Regel so vor, daß eine zentrale Feature-Hierarchie in einem Datenstruktur-Modul komplett definiert wird. Ein Modulkomplex, der auf diese Feature-Hierarchie zugreifen will, tut dies dann über einen (dazwischengeschalteten) View, der die (für diesen Komplex) unbenötigten Objekte herausfiltert.

Das Ausblenden von Teilen einer Feature-Hierarchie wird von Feature-TEL voll unterstützt. Es gibt prinzipiell zwei Möglichkeiten der Abstraktion von Feature-Hierarchien.

- Feature-Typen können weggelassen werden
- Features können für bestimmte Feature-Typen weggelassen werden

Damit ist das Bilden beliebiger Ausschnitte einer Feature-Hierarchie möglich. Man muß sich jedoch darüber klar werden, welche Konsequenzen das Weglassen von Features und Feature-Typen auf die Unter/Obertyp-Beziehungen und auf die Feature-Ranks hat.

Wir geben zunächst die Syntax von *Transfer-Deklarationen* in Feature-TEL:

Transfer-Dekl ⟶ `'from'` *Modul-Name* `':'`
{*Bezeichner* [`'abstract'`] |
Feature-Typ-Transfer}

Zusätzlich gilt in der Export-Signatur natürlich die normale Feature-Vererbung von Obertypen auf Untertypen. Dies bedeutet, daß jeder Feature-Typ, der transferiert wird (egal, ob mit oder ohne `with`-Klausel), zumindest die Features seiner Obertypen erhält. Wie sehen nun die Ranks der transferierten Features aus? Wir stellen zunächst fest, daß der bisherige Codomain eines Features bezüglich eines Feature-Typs sich nicht ändern darf! Man betrachte etwa das folgende Beispiel (wir greifen hier und in der Folge immer wieder auf die Feature-Hierarchie des Moduls `uni` aus Abbildung 5.1 zurück!).

```
interface uni-test.
imports uni.
from uni:  angestellter with chef, sekretaerin,
           dozent, professor,
           op1.
...
endinterface.
```

Hier wird das Feature `chef` zusammen mit dem Typ `angestellter` transferiert, jedoch nicht mit dem Typ `dozent`. Somit muß es für `dozent` von `angestellter` geerbt werden. Dabei behalten wir die für `dozent` definierte Rankverschärfung von `chef` (von `angestellter` auf `professor`), obwohl diese nicht direkt transferiert wurde! Wenn wir dies nicht täten, dürften Operationen aus `uni-test` dieses Feature für Elemente vom Typ `dozent` auch mit Werten vom Typ `sekretaerin` belegen. Operationen, die aus dem Modul `uni` importiert werden (zum Beispiel `op1`), gehen aber davon aus, daß dieses Feature mit Werten vom Typ `professor` belegt ist! Damit wäre die ursprüngliche Funktionalität dieser Operationen nicht mehr gewährleistet. Wir müssen also alle Rankverschärfungen übernehmen, auch wenn diese nicht explizit transferiert werden. Die Feature-Ranks in der Export-Signatur ergeben sich dann folgendermaßen. Ein Feature bekommt zunächst für jeden Feature-Typ, für den es definiert ist, einen Rank mit dem gleichen Codomain wie in der Import-Signatur. Anschließend werden dann die redundanten Ranks weggelassen (ein Rank ist redundant, wenn er ohnehin schon geerbt wird).

Wir haben nun gesehen, wie Signaturen für Feature-Typ-Transfers aufgebaut

werden. Die Export-Signatur ist die Vereinigung der durch die Transfer-Deklarationen spezifizierten Signatur mit der durch die lokalen Deklarationen spezifizierten Signatur. Selbstverständlich muß die Export-Signatur wieder konsistent sein (dies wird von der Signatur-Analyse überprüft).

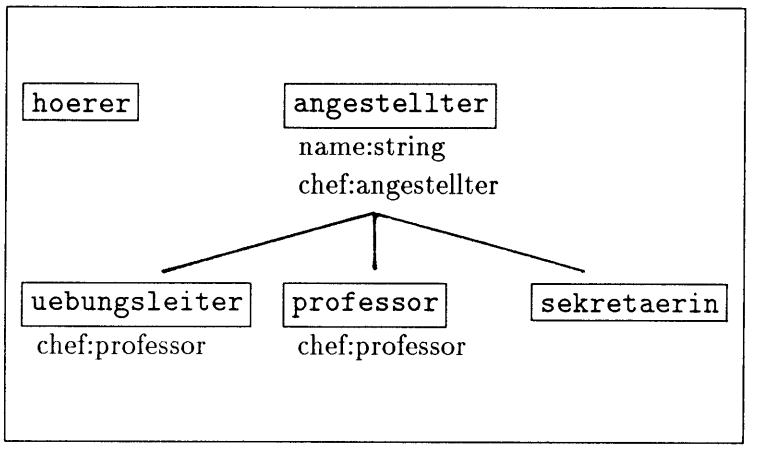


Abbildung 5.2: Export-Signatur von `uni-view1`

```
view uni-view1.
imports uni.
from uni:  angestellter with name chef,
           uebungsleiter,
           professor with chef,
           hoerer, sekretaerin.
endview.
```

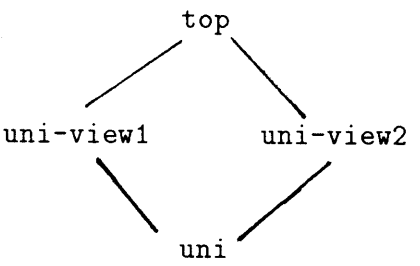
Die dazugehörige Export-Signatur wird durch Abbildung 5.2 veranschaulicht. Man beachte, daß das Feature `chef` für `uebungsleiter` den Typ `professor` hat, obwohl diese Verschärfung nicht explizit transferiert wurde (`dozent` wurde nicht transferiert). Interessant ist auch, daß der Typ `hoerer` jede Verbindung zu den anderen Typen verloren hat (es wurde keiner seiner Unter- oder Obertypen transferiert) und auch selber keine Features mehr hat. Er ist für die Export-Signatur von `uni-view1` ein abstrakter Datentyp. Man kann also offenbar die `abstract`-Klausel für Konstruktor-Typen in gewisser Weise für Feature-Typen simulieren.

Wir fassen die wichtigsten Punkte für die Bildung von Export-Signaturen zusammen:

> Über Transfer-Deklarationen kann man nicht benötigte Feature-Typen und Features aus einer Feature-Hierarchie herausfiltern. Dabei bleiben die alten Untertyp-Beziehungen bestehen. Auch der Codomain eines Features bezüglich eines Feature-Typs bleibt unverändert, selbst wenn der dazugehörige Rank nicht explizit transferiert wird!

Import-Signatur

Die Import-Signatur eines Moduls in Feature-TEL ist (genauso wie im bisherigen TEL) als die Vereinigung der Export-Signaturen der im Interface importierten Module definiert. Dabei kann ein interessanter Fall auftreten: verschiedene Ausschnitte der gleichen Feature-Hierarchie sollen zusammengeführt werden. Man betrachte beispielsweise folgende Modulhierarchie:



Den View `uni-view1` kennen wir bereits aus dem vorangehenden Abschnitt. Mit `uni-view2` bezeichnen wir einen weiteren View auf die Feature-Hierarchie aus `uni`, den wir werden in der Folge noch definieren werden. Das Modul `top` importiert in seinem Interface sowohl `uni-view1` als auch `uni-view2` (ein analoges Problem ergibt sich, wenn beide Views über den Body von `top` importiert werden). Wie sieht die resultierende Feature-Hierarchie der Import-Signatur von `top` aus?

Wir reduzieren das Problem auf das Zusammenmischen zweier Signaturen S_1 und S_2 zu einer Vereinigungs-Signatur S. Dann können wir mehrere Signaturen durch Iteration dieses Verfahrens zusammenmischen. Die Feature-Typen von S ergeben sich aus die Vereinigung der Feature-Typen von S_1 und S_2. Ebenso werden die Untertyp-Beziehungen vereinigt. Dies ist unproblematisch, da Feature-Hierarchien immer komplett in einem Ursprungsmodul definiert sind. Die Features eines Feature-Typs aus S ergeben sich aus der Vereinigung seiner Features in S_1 und S_2. Auch die Ranks der Features werden vereinigt. Dabei werden (aufgrund der Vererbung) redundante Ranks weggelassen. Hier kann es ebenfalls keine Inkonsistenzen geben. Aus dem vorangehenden Abschnitt wissen wir nämlich, daß der Codomain eines Features für einen bestimmten Feature-Typ immer dem Codomain aus dem Ursprungsmodul entspricht.

Zur Illustration definieren wir den View `uni-view2` aus der obigen Modul-Hierarchie, und zeigen wie die Import-Signatur des Moduls `top` aussieht.

```
view uni-view2.
imports uni.
from uni:  person withall,
           hoerer with semester,
           dozent,
           uebungsleiter with schueler.
endview.
```

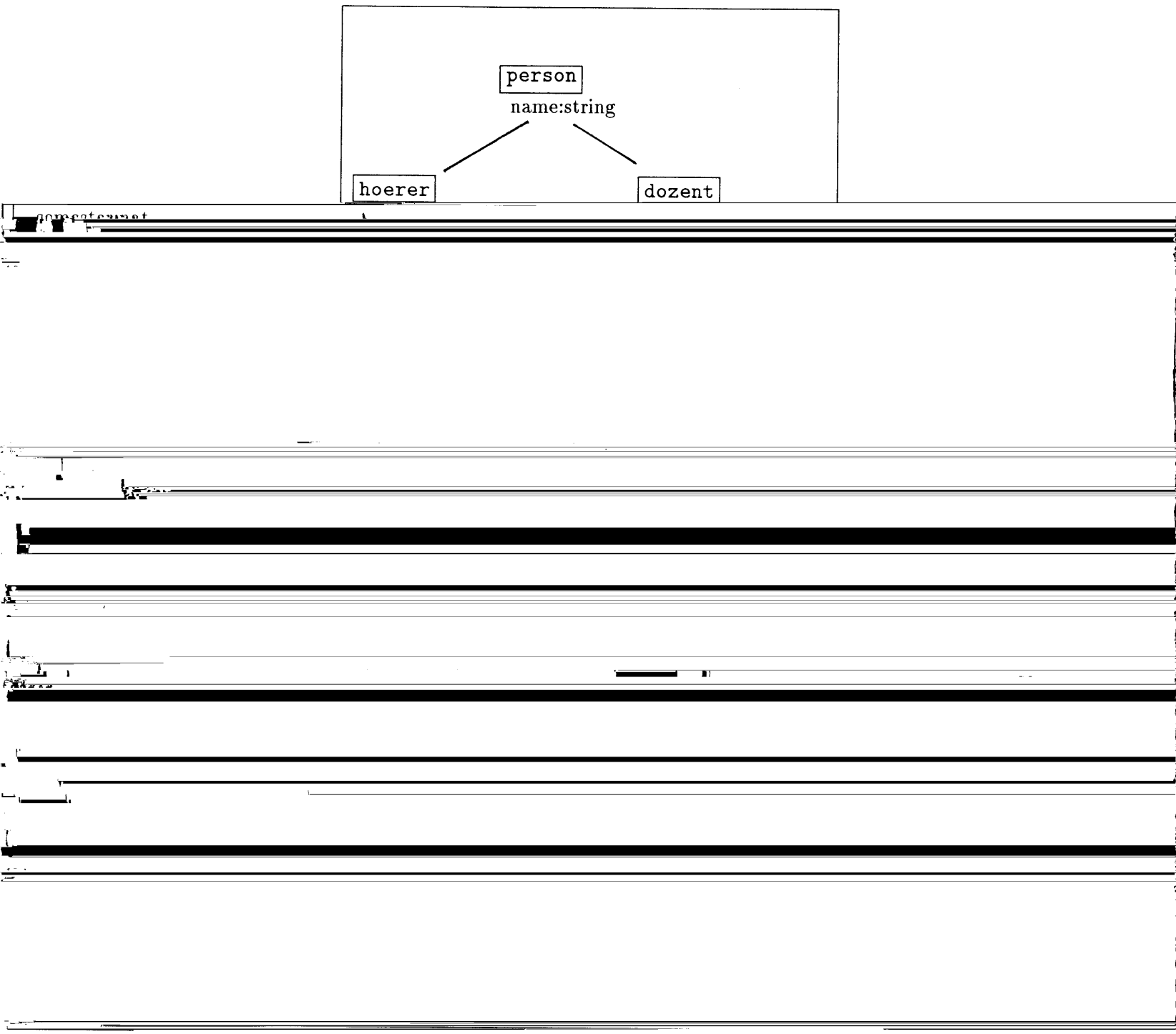


Abbildung 5.3: Export-Signatur von `uni-view2`

Wir veranschaulichen die dazugehörige Export-Signatur in Abbildung 5.3. Die Import-Signatur des Moduls `top` ist in Abbildung 5.4 dargestellt (man vergleiche mit den Abbildungen 5.1, 5.2 und 5.3). Es fällt auf, daß `angestellter` nicht mehr Obertyp von `dozent` ist, daß der Typ `hoerer` nicht mehr abstrakt ist, und daß `dozent` kein Feature `schueler` mehr hat. Dies mutet zwar etwas merkwürdig an, wenn man noch die Ursprungshierarchie aus Abbildung 5.1 im Kopf hat. Bei der Vereinigung von `uni-view1` und `uni-view2` ist diese jedoch völlig vergessen und es sind nur die Untertyp-Beziehungen aus den beiden Views bekannt. Man kann sich den Vorgang anschaulich machen, indem man die graphischen Darstellungen der

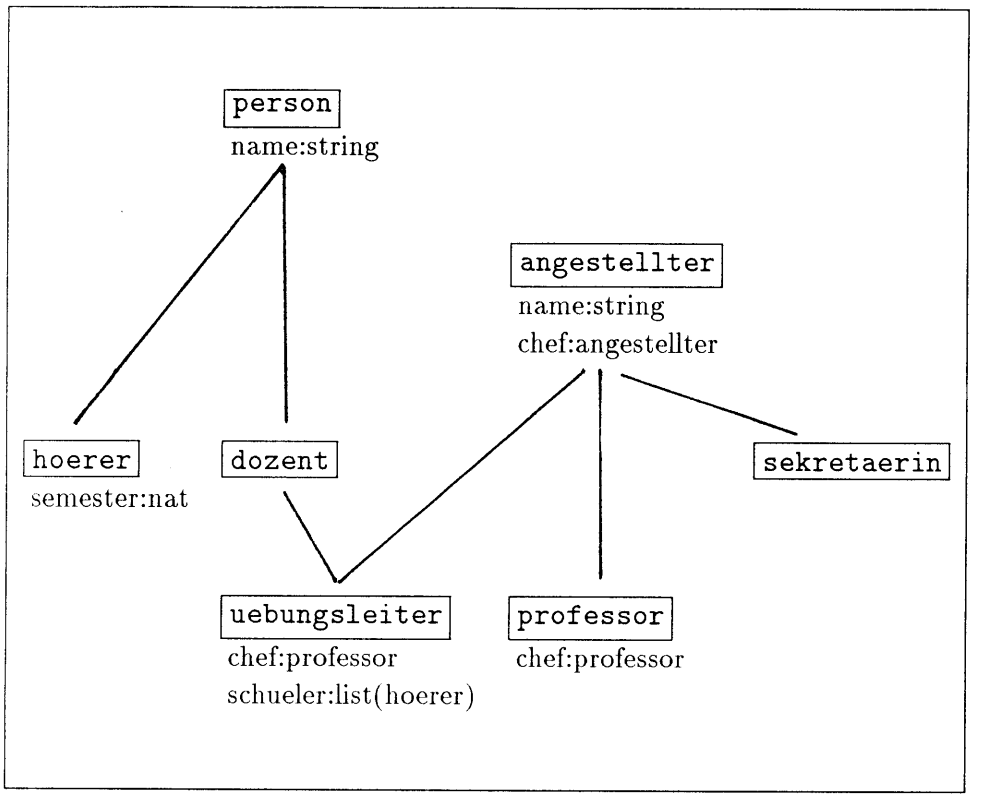


Abbildung 5.4: Import-Signatur von `top`

untersuchen, was sich durch die Feature-Typen an Klauseln und Queries ändert. Dabei gehen wir insbesondere auch auf Fragen des Typ-Checkens und der Abarbeitung ein.

Feature-Typen werden über versteckten Konstruktoren realisiert. Diese sind jedoch für den Benutzer von Feature-TEL *nicht* sichtbar. Dieser kennt nur die Ordnung auf den Feature-Typen und die Feature-Ranks. Features werden bezüglich Typ-Checken und Abarbeitung wie TEL-Funktionen behandelt. Containments auf Feature-Typen sind nicht nur Tests gebundener Werteterme auf Typzugehörigkeit, sondern führen auch Typverschärfungen und Typbindungen ungebundener Wertevariablen aus. Dies geschieht durch Backtracking über die entsprechenden minimalen Feature-Typen. Feature-Terme stellen im Grunde nur eine kompaktere Darstellung von Gleichungen und Containments dar, und werden in eingeschränkter Form zugelassen. Sie dienen insbesondere zur Ausgabe von Feature-Typen in Query-Antworten.

Wir gehen in diesem Abschnitt wie folgt vor. Unter Punkt 5.2.1 behandeln wir Features und Feature-Typen in einfachen Conditions, wie Gleichungen und Containments. Damit haben wir die Voraussetzungen, um Feature-Terme verstehen zu können, welche in Punkt 5.2.2 beschrieben werden. Schließlich zeigen wir unter Punkt 5.2.3, wie Feature-Typen in Queries behandelt werden.

Im folgenden setzen wir stillschweigend voraus, daß Beispiele sich auf die Feature-

Signatur des Moduls `uni` (siehe Abbildung 5.1) beziehen.

5.2.1 Feature-Typen in einfachen Conditions

Abgesehen von Feature-Termen (welche aber eigentlich nur syntaktischer Zucker für eine Folge primitiver Conditions sind) werden durch Feature-Hierarchien keine syntaktisch neuen Conditions geschaffen. Features und Feature-Typen dürfen aber in Typ- und Wertetermen vorkommen, wodurch sich durchaus neue Ausdrucksmöglichkeiten ergeben.

Typterme in Feature-TEL werden wie in TEL über Sortenfunktionen, Sortenkonstante und Sortenvariable aufgebaut. Feature-Typen werden dabei wie POS-Sortenkonstanten behandelt. Werteterme werden aus Konstruktoren, Funktionssymbolen, Feature-Symbolen und Wertevariablen aufgebaut. Wohlgetyptheit von Wertetermen ist wie in Abschnitt 3.2 definiert. Features und Funktionen werden also gleich behandelt. Dabei garantiert die Regularität ihrer Typ-Deklarationen, daß für wohlgetypte Werteterme immer ein kleinster Typ abgeleitet werden kann.

Wir untersuchen zunächst, inwiefern die Präsenz von Features sich auf die Abarbeitung von Wertetermen auswirkt. Prinzipiell kann man sagen, daß Features sowohl beim Typ-Checken als auch bei der Abarbeitung von Programmen genauso wie Funktionen behandelt werden. Ebenso wie Funktionen dienen Features nämlich nicht zur Erzeugung von Elementen eines Datentyps (im Gegensatz zu Konstruktoren). Deshalb kann man vom Typ eines Werteterms mit einem Feature als Topsymbol keine Rückschlüsse auf den Typ des Arguments ziehen. Wir geben ein kleines Beispiel. Aus der Gültigkeit der Containments

```
X : angestellter & chef(X) : professor
```

kann man nicht auf eine Verschärfung des bisherigen Typs von `X` auf `dozent` schließen. Schließlich kann auch eine `sekretaerin` einen `professor` als `chef` haben!

Nicht nur das Typ-Checken, sondern auch die Abarbeitung von Features entspricht genau der von Funktionen. So sind Features ebenfalls total und deterministisch und ihre Eingabe muß beim Aufruf gebunden sein. Die Funktionalität von Features wird durch ihre Selektorgleichungen spezifiziert. Dabei sind die impliziten Konstruktoren für den Benutzer natürlich nicht sichtbar!

Wir kommen nun zur Auswirkung der Präsenz von Feature-Typen in Typtermen. Beim Typ-Checken ändert sich gar nichts. Feature-Typen werden wie monomorphe Konstruktor-Typen behandelt. Ein wesentlicher Unterschied liegt jedoch in der Abarbeitung von Containments.

Containments auf reine POS-Typen bewirken in der aktuellen Implementierung von Feature-TEL und TEL nur einen Typzugehörigkeitstest für gebundene Werteterme, jedoch keine Typverschärfung oder Typbindung für ungebundene Wertevariable (vergleiche 4.1.2). Dies liegt darin begründet, daß Feature-TEL ebenso wie TEL

nach Prolog übersetzt wird und somit getypte Unifikation auf ungetypte Unifikation abgebildet werden muß.

Containments auf Feature-Typen stellen jedoch die einzige Möglichkeit dar, Elemente eines Feature-Typs zu erzeugen, da die Erzeugung durch Konstruktoren entfällt. Somit müssen Containments auf Feature-Typen (im Gegensatz zu Containments auf POS-Typen) auch eine tatsächliche Typbindung ungebundener Wertevariablen bewirken. Diese Typbindung einer Wertevariablen an einen Feature-Typ in einem Containment geschieht operational durch Backtracking über die minimalen Untertypen dieses Feature-Typs. Dabei wird die Bindung an einen minimalen Feature-Typ intern durch Unifikation mit dessen implizitem Konstruktor realisiert. Dies ermöglicht eine praktikable Nutzung von Feature-Typen; Feature-Unifikation steht damit aber nur in eingeschränkter Form zur Verfügung. Das endgültige Ziel muß die Implementierung einer abstrakten Maschine für Feature-TEL sein, welche "echte" Typbindungen vornimmt.

Wir geben ein kleines Beispiel zur Illustration der Wirkung von Containments auf Feature-Typen. Wir wollen zunächst ein Element `S` des Feature-Typs `student` mit den Feature-Werten `'Otto'` für `name` und 17 für `semester` erzeugen. Dies geschieht offenbar durch die Folge

```
S : student & name(S) = 'Otto' & semester(S) = 17.
```

Wenn wir noch die Konjunktion `S : dozent & schueler(S) = nil` anfügen, so gilt nach Abarbeitung der gesamten Folge

```
S : uebungsleiter &
name(S) = 'Otto' & semester(S) = 17 & schueler(S) = nil.
```

Man sieht, daß hier nach außen wirklich Feature-Unifikation durchgeführt wird. Die interne Abarbeitung solcher Constraint-Systeme wird in Kapitel 6 beschrieben. Man beachte übrigens, daß die Typverschärfung von Variablen nicht für Containments auf polymorphe Typen mit Feature-Typen als Argumenten gilt. So bewirkt etwa `X : list(dozent)` nur eine Typ-Abfrage, aber keine Typ-Bindung.

Wir fassen zusammen:

> Containments auf Feature-Typen bewirken eine reale Typverschärfung und nicht nur eine Typ-Abfrage. Dies geschieht durch Backtracking über die dazugehörigen minimalen Feature-Typen. Features werden wie Funktionen getypcheckt und abgearbeitet.

5.2.2 Feature-Terme

Mit Feature-Termen steht eine kompakte Notation für zusammengehörige Folgen von Containments und Gleichungen zur Verfügung. Durch ihre Verwendung wird

insbesondere die Lesbarkeit von Programmen verbessert. Der Typ-Checker und der Interpreter arbeiten Feature-Terme genauso wie die dazugehörige Folge von Containments und Gleichungen ab. Dabei lokalisieren die Fehlermeldungen des Typ-Checkers genau die Stelle im Feature-Term, die zu der inkonsistenten Gleichung (beziehungsweise zu dem inkonsistenten Containment) gehört. In Feature-TEL dürfen Feature-Terme bisher nur in der rechten Seite von Containments auftreten. Diese Isolierung macht die Tatsache bewußt, daß Feature-Terme eine Art Zwitter zwischen

Wert- und Typtermen sind. Wir geben zunächst die Syntax von Feature-Termen an:

Feature-Term ⟶ *Feature-Typ-Name*'['(*Feature-Paar*)*']'
Feature-Paar ⟶ *Cont-Paar* | *Gleich-Paar*
Cont-Paar ⟶ *Feature-Name*':'(*Grundsortenterm* | *Feature-Term*)
Gleich-Paar ⟶ *Feature-Name*'=>' *Werteterm*

Wir illustrieren nun anhand eines Beispiels, wie man aus einem Feature-Term die dazugehörige Folge von Containments und Gleichungen konstruiert. Gegeben sei folgender Feature-Term:

```
X : dozent[  name => 'Otto',
             chef :  professor[  name => 'Joerg',
                                 schueler :  list(hoerer)],
             schueler => nil]
```

Die dazugehörige Folge von Containments und Gleichungen sieht wie folgt aus:

```
X : dozent & name(X) = 'Otto' &
!Y & Y : professor & chef(X) = Y &
name(Y) = 'Joerg' & schueler(Y) : list(hoerer) &
schueler(X) = nil
```

Offenbar werden Gleichungspaare direkt in Gleichungen und Containment-Paare mit gewöhnlichen Typtermen direkt in Containments umgesetzt. Das rekursive Auftreten von Feature-Termen in Containment-Paaren entspricht dem impliziten Auffalten des entsprechenden Features (die neue Variable Y wird eingeführt), wonach sich das

nur den Namen des Typs einer Variable merken, sondern auch alle seine Feature-Belegungen. In dem folgenden Beispiel, müßte man sich für die Variable `X` also nicht nur den Typ `list(uebungsleiter)`, sondern auch noch den Wert des Features `schueler` von `uebungsleiter` merken:

```
X : list(uebungsleiter[schueler => nil]).
```

Wir kommen nun zu möglichen Erweiterungen der bisherigen Implementierung von Feature-Termen in Feature-TEL. Zum einen wäre eine Zulassung von Koreferenzen in Containment-Paaren sinnvoll, wie sie bei den ψ-Termen von Aït-Kaci [AN 86] möglich ist. Wir geben ein Beispiel:

```
dozent[chef => X:uebungsleiter, schueler => X.nil].
```

Solche Koreferenzen bedeuten offenbar nur ein zusätzliches Auffalten und dürften daher unproblematisch sein.

Eine weitere sinnvolle Erweiterung wäre die Zulassung von Feature-Termen im Klauselkopf oder als Argumente von relationalen Atomen. Die letzte Möglichkeit ist insbesondere in Verbindung mit einer Code-Optimierung durch Indexing über die entsprechenden impliziten Konstruktoren interessant. Wir skizzieren dies kurz an einem Beispiel:

```
r(sekretaerin[name => 'Christine'],...)  <-- c1...
r(student[semester => 1],...)  <-- c2...
```

könnte übersetzt werden zu:

```
r(c_s('Christine',...),...)  <-- c1...
r(c_h(...,1),...)  <-- c2...
r(c_u(...,1,...),...)  <-- c2...
```

Wir fassen zusammen:

> Feature-Terme stellen eine kompakte Schreibweise einer Folge von Containments und Gleichungen dar. Sie dürfen bisher jedoch nur als rechte Seite von Containments vorkommen.

5.2.3 Queries

Queries in Feature-TEL sind wie in TEL als Konjunktionen von Conditions definiert. Die Abarbeitung von Conditions wurde bereits in den vorangehenden Abschnitten ausreichendend beschrieben. Wir gehen hier kurz auf die Ausgabeform von Feature-

bleiben sollen, muß die Ausgabe von Feature-Typ-Werten in einer anderen Form erfolgen. Eine Ausgabe der entsprechenden Gleichungen und Containments erscheint nicht sinnvoll, denn Feature-Typ-Werte können ja auch innerhalb von Konstruktor-Termen vorkommen. In diesem Fall wäre eine unübersichtliche Auffaltung nötig. Wir stellen daher die Werte von Feature-Typen durch Feature-Terme dar. Dies ermöglicht eine übersichtliche und kompakte Notation von Feature-Typ-Elementen auch innerhalb von Konstruktor-Termen. Die Repräsentation der Ausgabe in Form von Feature-Termen war ursprünglich auch die Motivation zur Einführung von Feature-Termen in Containments, da Lösungen wieder in der Constraint-Sprache ausgedrückt werden sollten (und nicht in einer fremden Notation). Die Ausgabe von Feature-Termen erfolgt nach den folgenden Grundsätzen:

- Alle gebundenen Features werden ausgegeben.
- Ungebundene Feature-Werte werden nur dann ausgegeben, wenn sie noch an einer anderen Stelle der Query-Antwort vorkommen. Anstelle der unlesbaren Prolog-Mnemos (zum Beispiel "_193467") werden sie mit "$\$N$" explizit als Koreferenzen gekennzeichnet (dabei ist die Nummer N der Koreferenz in der Regel einstellig und somit leicht identifizierbar).
- Bei der Ausgabe werden (durch Transfers bedingte) Abstraktionen von Feature-Hierarchien beachtet. Das bedeutet, daß Feature-Typen oder Features, die im geöffnetem Modul nicht bekannt sind, in der Antwort nicht vorkommen.

Den letzten Punkt schauen wir uns noch etwas näher an. Wenn in der Query-Antwort Features belegt sind, die im geöffnetem Modul wegabstrahiert wurden, so wird dies dem Benutzer durch die Ausgabe von `*abstract-features*` im entsprechenden Feature-Term mitgeteilt. Wenn der Typ der Ausgabevariable im geöffneten Modul wegabstrahiert wurde, so werden seine minimalen (bekannten) Obertypen (eingerahmt von *) angegeben. Die Angabe von * signalisiert dem Benutzer also, daß der angegebene Typ eigentlich nicht der kleinste Typ der Ausgabevariable ist. Wir illustrieren dies an einem Beispiel. Die Ausgabe

```
X : *student∩dozent*[name => 'Otto',*abstract-features*]
```

zeigt an, daß der kleinste Typ der Variable `X` im geöffneten Modul unbekannt ist, aber unter den minimalen bekannten Typen `student` und `dozent` liegt. Außerdem existieren belegte Features von `X`, die im geöffneten Modul nicht bekannt sind. Diese Situation könnte etwa die folgende Ursache haben: Im geöffneten Modul sind der Typ `uebungsleiter` und das Feature `semester` wegabstrahiert. Die Variable `X` ist eigentlich vom Typ

```
X : uebungsleiter[name => 'Otto', semester => 3].
```

Wir fassen zusammen:

Werte von Feature-Typen werden in Queries in Form von Feature-Termen ausgegeben. Dabei werden Koreferenzen und Abstraktionen berücksichtigt.

Kapitel 6

Implementierung von Feature-TEL

Die Implementierung von Feature-TEL setzt auf der Version 0.9 des TEL-Systems auf (siehe Kapitel 4). Feature-TEL wurde in einem Bootstrapping-Prozeß entwickelt. Zunächst wurde das TEL-System (welches ja selbst in TEL geschrieben ist) so erweitert, daß es Feature-TEL-Programme verarbeiten konnte. Anschließend wurde dieses erweiterte System in Feature-TEL reimplementiert. Insbesondere wurden Signatur-Einträge, die zentrale Datenstruktur des TEL-Systems, als Feature-Typen dargestellt (siehe Kapitel 7). Das Feature-TEL-System ist ebenso wie TEL ein Prolog-Savestate (Größenordnung: etwa 1MByte). Der Feature-TEL-Compiler checkt Feature-TEL-Programme zunächst auf Syntax-, Signatur- und Typ-Konsistenz und übersetzt sie anschließend nach Prolog, wo sie abgearbeitet werden.

Die Erweiterung von TEL um Feature-Typen erfordert im wesentlichen die Integration der neuen Objekte in die interne Datenstruktur, sowie die Anpassung der verschiedenen Compiler-Komponenten. Ein großer Teil des alten TEL-Systems bleibt jedoch unberührt, insbesondere die Manager-Komponente.

Wir gehen in diesem Kapitel folgendermaßen vor. In Abschnitt 6.1 beschreiben wir, wie Feature-Typ-Deklarationen in Signatur-Einträgen repräsentiert werden. Abschnitt 6.2 beschäftigt sich mit dem Parser. Dabei gehen wir insbesondere auf die Repräsentation der neuen Konstrukte in der abstrakten Syntax ein. In Abschnitt 6.3 werden die durch die Feature-Typen bedingten Erweiterungen der Signatur-Analyse beschrieben. Dabei liegt der Schwerpunkt auf der Implementierung der Abstraktionsmöglichkeiten für Feature-Hierarchien. Abschnitt 6.4 beschäftigt sich mit dem Typ-Checker und dem Umsetzen der neuen Konstrukte in die Intermediate Language. Schließlich erläutern wir in Abschnitt 6.5 welcher Prolog-Code für Feature-Typen und Features erzeugt wird. Im folgenden setzen wir stillschweigend voraus, daß Beispiele sich auf die Feature-Deklarationen des Moduls `uni` (siehe Abbildung 5.1) beziehen.

6.1 Interne Repräsentation von Feature-Typen

Wie stellen wir die Deklarationen von Feature-Typen intern dar? Aus Abschnitt 4.2 wissen wir, daß das TEL-System für jedes Symbol eines Programms genau einen Signatur-Eintrag anlegt. Durch Verpointerung (Unifikation) der Symbole mit ihren Einträgen ist an jeder Stelle der internen Programmrepräsentation die volle Signatur-Information vorhanden. Einträge stellen eine Art Slot-Filler-Notation dar und werden über den Datentyp `entry` repräsentiert. Für jede Symbolart gibt es einen speziellen Untertyp: `type_entry`, `constructor_entry`, `function_entry`, `relorproc_entry`, `parameter_entry`. Der Untertyp `type_entry` unterteilt sich nochmals in `full_type_entry` und `abbreviation_type_entry`. Es stellt sich nun die Frage, wo Feature-Typen, Features und Implizite Konstruktoren in der `entry`-Hierarchie integriert werden sollen, und welche Slots sie haben.

Feature-Typen und Features können direkt im Programmtext vorkommen, insbesondere in Typ- und Wertetermen. Terme werden intern als `entry`-Terme dargestellt, das heißt, daß Funktionssymbole durch Pointer auf ihre Einträge ersetzt werden. Somit ist es also erforderlich, daß wir unsere `entry`-Hierarchie um einen `feature_type_entry` und einen `feature_entry` erweitern. Implizite Konstruktoren repräsentieren wir nicht als Einträge, da grundsätzlich alle Einträge des jeweilig geöffneten Moduls für den Benutzer sichtbar sind, er aber nichts von der Existenz der impliziten Konstruktoren erfahren soll.

Der Datentyp `feature_type_entry` ist ein Untertyp von `type_entry`, denn allgemeine Operationen auf `type_entry` sollen natürlich auch für `feature_type_entry` anwendbar sein. Zur Darstellung der Attribute von Einträgen verwenden wir eine Art Slot-Filler-Notation (anstatt der unübersichtlichen Konstruktor-Schreibweise). Feature-Typ-Einträge haben die folgenden Slots.

feature_type_entry

```
              name:  string,
       module_name:  string,
       line_number:  nat,
         code_name:  string,
  given_supertypes:  list(type_entry),
    given_features:  list(feature_entry),
    given_subtypes:  list(eterm),
          subtypes:  list(eterm),
           suprema:  list(feature_type_entry##
```

Die Slots `name`, `module_name`, `line_number`,`code_name`, `given_supertypes`, und `given_features` eines Feature-Typ-Eintrags werden beim Parsen seiner Feature-Typ-Deklaration gefüllt, die restlichen Slotwerte werden in der Signatur-Analyse berechnet. Die Ordnung auf Typen wird durch die Slots `given_supertypes`, `given_subtypes`, und `subtypes` repräsentiert. Die letzten beiden Slots sind auch für `full_type_entry` definiert, so daß viele Signatur- und Typcheck-Operationen übernommen werden können. Dies betrifft insbesondere den Test auf Wohlfundiertheit, das Berechnen der allgemeinen Untertypen aus den direkten Untertypen, das Ausrechnen von Suprema und das Vergleichen von Typtermen. Der Slot `given_features` eines Feature-Typs wird nur mit denjenigen Features gefüllt, die direkt in seiner Deklaration vorkommen. Der Slot `complete_features` enthält zusätzlich noch die geerbten Features. Die impliziten Konstruktoren eines Feature-Typs werden im Slot `implicit_constructors` abgelegt.

Wir kommen nun zum `feature_entry`. Dieser ist direkter Untertyp von `entry` und enthält folgende Informationen:

```
feature_entry

        name:  string,
  module_name: string,
  line_number: nat,
    code_name: string,
  given_ranks: list(rank),
expanded_ranks: list(rank).
```

Bis auf `expanded_ranks` werden alle Slots schon beim Parsen gefüllt. Der Slot `given_ranks` enthält alle Ranks eines Features, die explizit in Feature-Typ-Deklarationen vorkommen. Die Ranks im Slot `expanded_ranks` entsprechen denen des Slots `given_ranks`, wobei jedoch die Abkürzungstypen aufgelöst sind. Die Slots von `feature_entry` wurden bewußt von `function_entry` übernommen, weil an Funktionen und Features sehr ähnliche Anforderungen gestellt werden (Regularität der Ranks, Ableiten von kleinsten Typen mittels des "least_codomain", etc.). Operationen, die für Funktionen bereits implementiert wurden, sind durch die Namensgleichheit der Slots nun auch für Features zugänglich.

Man beachte, daß die Definition der Slots von `feature_type_entry` und `feature_entry` bewußt so gewählt wurde, daß keine Redundanzen vorkommen. So braucht man beispielsweise keinen Slot für die Codomains der Features eines Feature-Typ-Eintrags. Diese erhält man, wenn man sich die `expanded_ranks` der Feature-Einträge seines `complete_features`-Slots anschaut.

Wir kommen nun zu den impliziten Konstruktoren. Diese entsprechen dem Typ `impl_constructor`, welcher folgendermaßen definiert ist:

```
impl_constructor := {icons:  codename x
                             list(feature_entry) x
                             rank}.
```

Der Codename eines impliziten Konstruktors und die Folge seiner Features werden bei der Typ-Codeerzeugung für die Selektor-Gleichungen gebraucht. Der Rank eines impliziten Konstruktors ist eigentlich ein redundantes Argument, da er über die Ranks seiner Features gewonnen werden kann. Bei der modulspezifischen Code-Erzeugung (siehe 6.5.1) benötigt man jedoch die Ranks der impliziten Konstruktoren. Man will sich aber nicht mehr mit Typberechnungen beschäftigen, die vorher ohnehin schon in der Signatur-Analyse durchgeführt wurden (nämlich beim Minimalitäts-Check). Die Signatur-Analyse gibt deshalb den impliziten Konstruktoren ihre Ranks als Argument mit.

Um die obigen Definitionen mit Leben zu füllen, zeigen wir in Abbildung 6.1, wie der vollständig gefüllte Signatur-Eintrag zu `student` aussieht (vergleiche mit Abbildung 5.1). Dabei sei "31" die Disambiguierung von `uni`. Andere Signatur-Einträge sind in Großbuchstaben geschrieben, so bezeichnet zum Beispiel PERSON den Signatur-Eintrag für den Feature-Typ `person`.

6.2 Parser

Beim Kompilieren eines Programms erfüllt der Parser zwei Funktionen. Zum einen erzeugt er die Signatureinträge der Programmobjekte und trägt dort die unmittelbare Signatur-Information ein. Zum anderen übersetzt er die Programm-Klauseln in die abstrakte Syntax von Feature-TEL. Dabei ersetzt er jeden Term durch den dazugehörigen `eterm` und stellt so die Verbindung zwischen abstrakter Syntax und Signatur her.

Beim Erzeugen von Signatur-Einträgen erkennt der Parser Doppeldeklarationen, indem er bei einer neuen Deklaration nachschaut, ob schon ein Eintrag gleichen Namens existiert. Einen Sonderfall stellen Features dar, da sie nicht direkt (wie Funktionen), sondern indirekt über Deklarationen von Feature-Typen eingeführt werden. Zudem können sie in mehreren Feature-Typ-Deklarationen vorkommen. Bei einer Feature-Deklaration innerhalb einer Feature-Typ-Deklaration geht der Parser daher folgendermaßen vor: Falls noch kein Eintrag gleichen Namens existiert, so legt er den entsprechenden Eintrag für das Feature an und füllt die Slots `name`, `module_name`, `line_number`, und `given_ranks`. Beim letzten Slot arbeitet er mit offenen Listen, denn es können später ja noch Ranks dazukommen. Falls bereits ein Feature-Eintrag gleichen Namens existiert, so fügt er nur den neuen Rank hinzu. Falls ein Nicht-Feature-Eintrag gleichen Namens existiert so reklamiert er eine Doppeldeklaration.

Die abstrakte Syntax von Feature-TEL wird durch eine Typ- beschrieben, welche genau den BNF-Syntax-Diagrammen von Feature-TEL entspricht. Dabei werden Terme auf `entry`-Terme (`eterm`) abgebildet. Die abstrakte Syntax von Feature-

```
feature_type_entry

                name =  'student',
         module_name =  'uni',
         line_number =  5,
           code_name =  '31student',
    given_supertypes =  PERSON.nil,
      given_features =  SEMESTER.nil
       given_subtypes:  ec(HOERER,nil).
                        ec(UEBUNGSLEITER,nil).nil,
             subtypes:  ec(HOERER,nil).
                        ec(UEBUNGSLEITER,nil).nil,
             suprema =  ANGESTELLTER#PERSON.nil,
   complete_features =  NAME.SEMESTER.nil,
implicit_constructors =  icons('31hoerer.cons',
                                NAME.SEMESTER.nil,
                                rk(tl,
                                   ec(STRING,nil).
                                    ec(NAT,nil).nil,
                                   ec(HOERER,nil))).
                         icons('$31uebungsleiter',...).nil
```

Abbildung 6.1: Der Feature-Typ-Eintrag für `student`

TEL unterscheidet sich von der abstrakten Syntax von TEL nur in zwei Punkten

Zum einen gibt es neue Konstrukte für Transfer-Deklarationen, zum anderen müssen Feature-Terme in Containments repräsentiert werden.

Die abstrakte Syntax für *Transfer-Deklarationen* sieht folgendermaßen aus:

```
transfer_decl  :=  modname##list(export_item).
  export_item  :=  abstract_item ++ concrete_item.
abstract_item  :=  {abs_item :  designator}.
concrete_item  :=  {conc_item :  designator,
                    featy_item :  designator x fea_select}.
   fea_select  :=  {withall, fea_list :  list(designator)}
```

Die Namen der zusammen mit einem Feature-Typ transferierten Features werden durch den Typ `fea_select` spezifiziert. Die "withall"-Klausel ist direkt durch einen Konstruktor repräsentiert. Falls ein Feature-Typ ohne Features transferiert wird , so wird er als "normales" `conc_item` behandelt. In den Transfer-Deklarationen werden keine Einträge sondern nur Bezeichner repräsentiert, da die Signatur-Analyse ohnehin erst prüfen muß, ob die transferierten Einträge in der Import-Signatur vorkommen.

Containments mit Feature-Termen werden als neue Condition in der abstrakten Syntax verankert:

```
                fea_contain_cond:  condno x eterm x fea_term,
                ...}.
```

Dabei bezeichnet das erste Argument von `fea_contain_cond` die Nummer der Condition innerhalb der Klausel, das zweite Argument die linke Seite des Containments (den Werteterm) und das dritte Argument den Feature-Term. Feature-Terme werden folgendermaßen repräsentiert:

```
 fea_term  :=  {ftc :  entry x list(fea_pair)}.
 fea_pair  :=  cont_pair ++ val_pair.
cont_pair  :=  {cp :  entry x eterm,
                fcp :  entry x fea_term}.
 val_pair  :=  {vp :  entry x eterm}.
```

```
ftc(DOZENT,
     vp(NAME,'Otto').
     fcp(CHEF,
          ftc(PROFESSOR,
              cp(CHEF,ec(DOZENT,nil)).nil)).
     nil)
```

Man kann fragen, ob eine explizite Repräsentierung von Feature-Termen in der abstrakten Syntax notwendig ist. Könnte der Parser nicht einfach Feature-Terme direkt in die dazugehörige Folge von Containment- und Gleichungs-Conditions überführen? Die Antwort lautet "Nein!", eine explizite Repräsentierung von Feature-Termen in der abstrakten Syntax ist deshalb notwendig, damit der Typ-Checker (der ja von der abstrakten Syntax ausgeht) sich in seinen Fehlermeldungen auch auf die entsprechenden Stellen im Feature-Term beziehen kann (und nicht auf irgendwelche Conditions).

Da Feature-Terme bisher nur in Containments vorkamen, wurde ihre Abarbeitung nicht in den Term-Parser integriert. Bei einer Erweiterung von Wertetermen durch Feature-Terme müßte jedoch eine integrierte Notation für `fea_term` und `eterm` entwickelt werden, welche eine integrierte Abarbeitung durch den Term-Parser ermöglicht.

6.3 Signatur-Analyse

Wir unterscheiden für Signatur-Einträge zwei Arten von Slots: Zum einen *Deklarations-Slots*, das sind die Slots, die vom Parser gefüllt werden und zur Deklaration eines Objekts absolut notwendig sind. Zum anderen *Hilfs-Slots*, welche in der Signatur-Analyse berechnet werden und zum Typ-Checken sowie zur Code-Erzeugung gebraucht werden. Die Signatur-Analyse besteht aus zwei, sich ergänzenden Aufgabenbereichen:

1. Berechnen der Hilfs-Slots und Durchführen von Konsistenzchecks ausgehend von den Deklarations-Slots einer Signatur ("Fill- und Check-Operationen").

2. Berechnen der Deklarations-Slots für Import- und Export-Signatur.

Zum ersten Bereich gehört zum Beispiel der Regularitätstest oder das Berechnen von Suprema. Der zweite Bereich beschäftigt sich insbesondere mit dem Umsetzen von Transfer-Deklarationen in Signaturen sowie mit dem Zusammenmischen von Signaturen. Wir betrachten nun die durch Feature-Typen bedingten Erweiterungen der Signatur-Analyse in den beiden Bereichen.

6.3.1 Fill- und Check-Operationen

Bei der Definition von `feature_type_entry` und `feature_entry` wurden viele Slots so gewählt, daß eine Übernahme der mit `full_type_entry` und `function_entry` gemeinsamen Signatur-Operationen erfolgen konnte. Wir geben eine Überblick über die wichtigsten dieser Operationen:

- Check auf Wohlfundiertheit der Typ-Ordnung
- Ausrechnen der `subtypes` aus den `given_subtypes`
- Auflösen von Abkürzungstypen
- Berechnen der Suprema
- Vergleichen von Typtermen
- Regularitäts-Check für Ranks.

Folgende Fill- und Check-Operationen sind spezifisch für Feature-Typen:

- Einfache Syntax- und Semantik-Checks der folgenden Art: Kommen Feature-Typen als `given_subtypes` von `full_type`-Einträgen vor? Ist der `super_types`-Slot eines Feature-Typs mit Feature-Typen belegt? Sind die Terme in den Ranks von Features wirklich Typ-Terme? Stimmt ihre Stelligkeit mit den Deklarationen überein?
- Bestimmen der `given_subtypes` aus den `given_supertypes`. Dies geschieht durch einfaches Bilden der Umkehrrelation.
- Berechnen der `complete_features` aus den `given_features`. Dies geschieht über eine Top-Down-Propagierung der Features in der Feature-Typ-Hierarchie. Gleichzeitig können die impliziten Konstruktoren berechnet werden.
- Checken der Bewohntheit der nichtminimalen Feature-Typen (Minimalitätsbedingung)

Interessant ist insbesondere der letzte Punkt. Wir gehen deshalb etwas ausführlicher darauf ein. Zur Wiederholung geben wir noch einmal die genaue Definition der Minimalitätsbedingung:

> Zu jedem nichtminimalen Feature-Typ T muß ein minimaler Untertyp M existieren, so daß für jedes Feature f von T gilt:
>
> $$\mathit{least\text{-}codomain}(f, T) = \mathit{least\text{-}codomain}(f, M).$$

Wir beschreiben nun, wie diese Bedingung im Feature-TEL-System gecheckt wird. Bei näherem Betrachten der obigen Forderung fällt auf, daß die Bewohntheit eines nichtminimalen Feature-Typen insbesondere von der Bewohntheit seiner Untertypen abhängt. Es drängt sich also eine rekursive Struktur des Test-Algorithmus' auf. Dabei besteht jedoch die Gefahr von Doppelberechnungen (ein Typ kann ja Untertyp von mehreren Typen sein). Deshalb bekommt jeder Feature-Typ einen zusätzlichen Slot `mark` vom Typ `info`. Der Typ `info` kann zwei Werte annehmen: `ok` für "Dieser Feature-Typ ist bewohnt", und `fault` für "Dieser Feature-Typ ist nicht bewohnt". Sobald ein Feature-Typ bearbeitet ist, wird sein `mark`-Slot mit dem entsprechenden Ergebnis belegt, so daß bei einem weiteren Zugriff darauf zurückgegriffen werden kann. Auf dem Typ `info` sind die Funktionen `min` und `max` definiert. Sie entsprechen den logischen Operatoren "und" beziehungsweise "oder".

Zur Beschreibung unseres Minimalitäts-Checks ist die Einführung weiterer Primitive erforderlich, auf deren genaue Implementierung wir hier jedoch nicht eingehen wollen. Paare von Features und Typtermen werden durch den Typ `fea_codom` repräsentiert:

```
fea_codom := feature_entry##eterm.
```

Die Funktion `lcd_FT` liefert zu einem Feature-Typ F die Liste seiner Features zusammen mit ihrem kleinsten Codomain bezüglich F.

```
lcd_FT : feature_type_entry --> list(fea_codom).
```

Gegeben seien zwei Paare (f_1, t_1) und (f_2, t_2), jeweils vom Typ `fea_codom`. Dann nennen wir das erste Paar eine *Verschärfung* des zweiten Paars, falls $f_1 = f_2$ und $t_1 \leq t_2$. Die Funktion `vergleiche_codom` hat die folgende Deklaration:

```
vergleiche_codom:  list(fea_codom) x list(fea_codom) --> info.
```

Dabei sollen beim Aufruf von `vergleiche_codom` die Paare der zweiten Liste zumindest die in der ersten Liste vorkommenden Features enthalten. Der Aufruf `vergleiche-codom(L1,L2)` liefert den Wert `fault`, falls es ein Paar `P1` aus `L1` und ein Paar `P2` aus `L2` gibt, so daß `P2` eine Verschärfung von `P1` ist.

Wir können nun den eigentlichen Bewohntheits-Algorithmus für Feature-Typen beschreiben. Die rekursive Top-Funktion heißt `teste_bewohntheit` und bekommt als Eingabe einen Feature-Typ sowie eine Liste von Features mit Typtermen. Bei einem Aufruf `teste_bewohntheit(F, FCS)` bezeichnet `FCS` immer die Liste der Feature-Codomains eines direkten Obertypen `S` von `F`. Es werden dann die beiden folgenden Operationen ausgeführt:

1. Die Bewohntheit des Typs `F` wird abgeprüft. Das Ergebnis wird im `mark`-Slot von `F` abgelegt.

```
teste_bewohntheit:
    feature_type_entry x list(fea_codom) --> info.

 F, FCS |> vergleiche_codom(FCS,lcd_FT(F))
         <-- minimal(F).
 F, FCS |> min(mark(F), vergleiche_codom(FCS,lcd_FT(F)))
         <-- schon_bearbeitet(F).
 F, FCS |> Mes
         <-- S_i = given_subtypes(F) &
             FCF = lcd_FT(F) &
             M_i = teste_bewohntheit(S_i,FCF) &
             mark(F) = max(M_i) &
             Mes = min(mark(F),vergleiche_codom(FCS,FCF)).
```

Abbildung 6.2: Bewohntheitstest für Feature-Hierarchien

2. Es wird festgestellt, ob `F` den aufrufenden Obertyp `S` bewohnt. Das Ergebnis ist der Funktionswert von `teste_bewohntheit(F,FCS)`.

Die erste Aufgabe wird gelöst, indem `teste_bewohntheit` rekursiv auf seine Untertypen angewandt wird. Damit `F` bewohnt ist, muß `teste_bewohntheit` offenbar für mindestens einen seiner Untertypen den Wert `ok` liefern. Falls der Typ `F` minimal ist, so ist er ohnehin bewohnt.

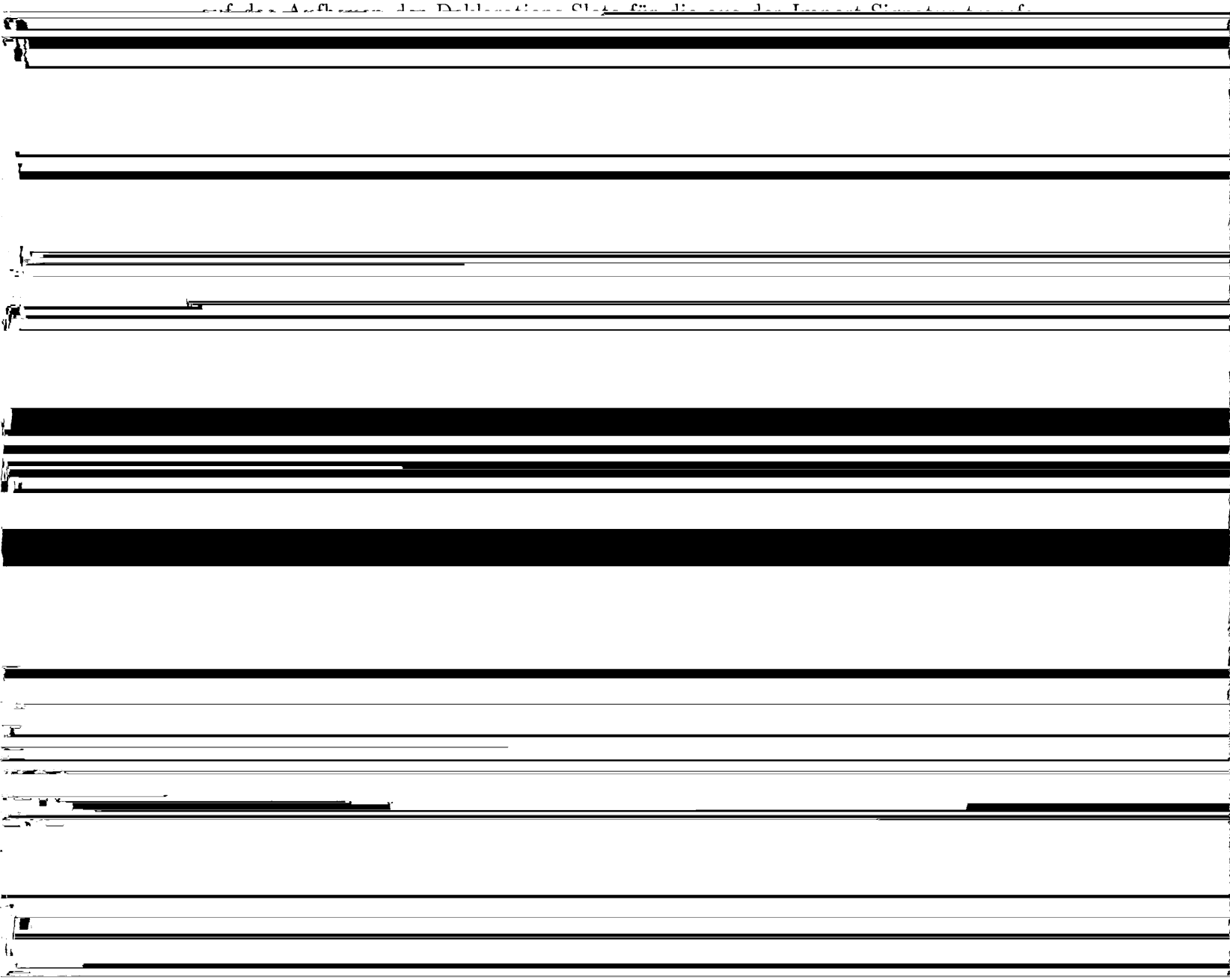
Bei der zweiten Aufgabe ist festzustellen, ob der direkte Obertyp `S` durch `F` bewohnt wird. Dazu müssen gerade die folgenden zwei Bedingungen erfüllt sein. Zum einen muß natürlich `F` selbst bewohnt sein, und zum anderen darf in `F` keine Verschärfung eines Features von `S` auftreten.

Abbildung 6.2 stellt in TEL-Notation die Topfunktion unseres Bewohntheitstests dar. Gegeben die Liste `FTL` der Feature-Typen einer Signatur, die auf Bewohntheit

6.3.2 Aufbau von Import- und Export-Signatur

Aufbau der Export-Signatur

Die Export-Signatur eines Moduls wird aus den lokalen Deklarationen im Interface und aus den Tranfer-Deklarationen aufgebaut. Dabei besteht das Aufbauen des lokalen Teils der Export-Signatur im wesentlichen nur aus syntaktischen Vergleichen mit den Deklarationen der lokalen Signatur. Wir beschränken unsere Betrachtung daher



Beim Transferieren von Feature-Hierarchien aus der Import-Signatur in die Export-Signatur können Feature-Typen oder Features weggelassen werden (siehe Kapitel 5). Die Aufgabe der Signatur-Analyse ist es nun, die neue Ordnung auf den Feature-Typen und die neuen Feature-Ranks auszurechnen. Dabei müssen zunächst die Deklarationsslots gefüllt werden, der Rest kann wieder mit den normalen Fill- und Check-Operationen bearbeitet werden.

Gestartet wird mit einer Liste von Transfer-Deklarationen und mit der vollstän-

eines Eintrags untereinander verglichen (über ihre `subtypes`-Slots!). Die `given_subtypes` sind dann gerade die maximalen Elemente.

3. Berechnen der `given_supers` aus den `given_subtypes`: Geschieht einfach durch Bilden der Umkehrrelation.

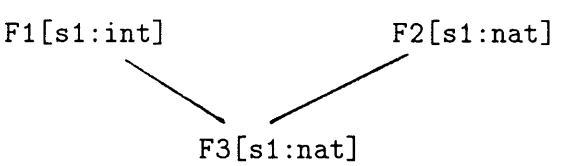
Bei dieser Vorgehensweise werden nicht nur die Deklarations-Slots gefüllt, sondern auch die Hilfs-Slots `given_subtypes` und `subtypes`. Da ihre Information jedoch zum Füllen des `given_supertypes`-Slots benötigt wird, sollte sie nicht weggeworfen werden. Die Verletzung der Aufgabenteilung (Deklarations-Slots/Hilfs-Slots) ist deshalb zu verschmerzen.

Wir kommen nun zum Ausrechnen der `given_ranks` für die transferierten Features. Features werden mittels der `with`- oder `withall`-Klausel zusammen mit ihren Feature-Typen transferiert. In Kapitel 5 wurde gezeigt, daß der Codomain eines Features bezüglich eines Feature-Typen immer dem Codomain des Ursprungsmoduls entsprechen muß. Wenn also ein Feature f zusammen mit einem Feature-Typ T transferiert wird, so müssen wir insbesondere prüfen, ob gleichzeitig Untertypen von T transferiert werden, welche f nicht explizit transferieren aber für die in der Import-Signatur eine Rankverschärfung von f gilt. Wir gehen dabei folgendermaßen vor. Wir sammeln zunächst alle transferierten Untertypen von T und T selbst auf, besorgen uns ihre kleinsten Codomains bezüglich f aus der Import-Signatur und bilden für jedes Element S einen Rank der Form:

$$S \rightarrow \textit{least-codomain}(f, S).$$

Dies tun wir für alle expliziten Transfers von f, so daß wir schließlich für jeden Domain von f in der Export-Signatur einen Rank haben. Jetzt müssen wir nur noch die redundanten Ranks streichen, um die `given_ranks` von f zu erhalten. Wann ist nun ein Rank redundant? Genau dann wenn er schon in einem allgemeineren Rank enthalten ist. Wir sagen ein Rank $T_1 \rightarrow c_1$ *subsumiert* einen Rank $T_2 \rightarrow c_2$, genau dann, wenn $T_1 \geq T_2$ und $c_1 = c_2$. Wir entfernen also aus unserer Rank-Menge alle Ranks, die durch einen anderen Rank subsumiert werden und behalten genau die `given_ranks` übrig.

Bei der Subsumption von Ranks gibt es jedoch noch ein kleines Problem. Es können nämlich gerade die für das effiziente Ableiten von kleinsten Typen kritischen Ranks wegsubsumiert werden. Man betrachte etwa die folgende Feature-Hierarchie.



Dann wird der vom TEL-Regularitäts-Check verlangte Rank

```
s1 :  F3 --> nat
```

von dem Rank

```
s1 :  F2 --> nat
```

subsumiert und fällt damit bei einem Transfer dieser Hierarchie weg! Dies ändert natürlich nichts daran, daß die Deklaration von `s1` regulär ist, da immer noch eindeutige kleinste Typen für wohlgetypte Werteterme existieren. Es gibt auch keine Probleme mit dem Regularitätstest an sich, da dieser für Export-Signaturen ohnehin nicht mehr durchgeführt werden muß (aus der Regularität der Importe kann man auf die Regularität der Transfers schließen!). Das Problem ist das Ableiten der kleinsten Typen durch das TEL-System. Bisher waren die Ranks einer Funktion nach ihrem Domain geordnet (kleinster Domain zuerst). Damit war der *least-codomain* eines Funktionsaufrufs einfach der Codomain des ersten Ranks dessen Domain "paßte", was natürlich eine effiziente Lösung ist. Dabei bedeutet "passen", daß ein kleinster oberer Matcher existiert (siehe Kapitel 3). Wenn aber Ranks möglicherweise wegsubsumiert werden, dann funktioniert diese Methode auf einmal nicht mehr! Es gibt jedoch eine elegante Lösung. Die verbliebenen Ranks werden zunächst nach ihrem Codomain sortiert, und anschließend zusätzlich nach ihrem Domain geordnet (so daß beide Ordnungen gelten). Dies funktioniert wegen der Regularität immer! Bei der so entstandenen neuen Ordnung der Ranks können wir wieder das alte Prinzip anwenden und den Codomain des ersten Ranks, der paßt, als kleinsten Codomain übernehmen. Die Richtigkeit dieser Behauptung läßt sich leicht über einen Widerspruchsbeweis erbringen. Da Feature-Typen monomorph sind können wir auf das Berechnen des kleinsten oberen Matchers verzichten. Der kleinste Typ eines Terms $f(t)$ ist einfach der Codomain des ersten Ranks, dessen Domain größer als der Typ von t ist.

Wir wissen nun, wie man die Deklarations-Slots für die aus der Import-Signatur transferierten Features und Feature-Typen berechnet. Was muß anschließend noch getan werden? Auf jeden Fall müssen die restlichen Slots noch gefüllt werden. Wir können jedoch auf viele Check-Operationen verzichten, da diese schon für die lokale Signatur der importierten Einträge durchgeführt wurden. Im Fall von Features und Feature-Typen sind dies die primitiven Syntax- und Semantik-Tests, der Regularitäts-Check, das Testen der Wohlfundiertheit der Typ-Ordnung und der Bewohntheits-Check für nichtminimale Feature-Typen.

Aufbau der Import-Signatur

Die Import-Signatur entsteht durch Zusammenmischen der Export-Signaturen der im Interface importierten Module. Ein analoges Problem gibt es beim Bilden der Body-Importe. Die gemeinsame Grundoperation ist offenbar das Zusammenmischen zweier komplett gefüllter Signaturen. Dies ist bei Feature-Typen nur dann problematisch, wenn in beiden Signaturen verschiedene Ausschnitte der gleichen Ursprungs-Hierarchie vorkommen. Ebenso wie beim Bilden der Export-Signatur stellt sich die Frage, wie die Deklarations-Slots `given_supertypes` und `given_ranks` berechnet

werden. Wir beginnen wiederum mit dem Ausrechnen der Hierarchie-Slots. Dabei wird für einen Feature-Typ T zunächst der `subtypes`-Slot gefüllt. Dieser ergibt sich einfach aus der Vereinigung aller `subtypes` von T in den zu vereinigenden Signaturen. Anschließend berechnen wir nach bewährtem Muster (siehe Aufbau der Export-Signatur) daraus die `given_subtypes` und die `given_supertypes`. Auch bei der Berechnung der `given_ranks` gehen wir analog zum Aufbau der Export-Signatur vor. Diesmal brauchen wir jedoch nicht für jeden möglichen Domain einen Rank zu erzeugen, sondern es reicht, wenn wir auf die Vereinigung aller in den zu vermischenden Signaturen vorkommenden `given_ranks` eines Features unsere Subsumptionsoperation anwenden.

Ebenso wie bei Export-Signaturen brauchen auch für Import-Signaturen nicht mehr alle Konsistenz-Tests durchgeführt zu werden. Im Falle von Feature-Hierarchien betrifft dies wieder die primitiven Syntax- und Semantik-Checks, den Regularitätstest, den Bewohntheitstest für nichtminimale Feature-Typen und den Test auf Wohlfundiertheit von Feature-Hierarchien.

6.4 Typ-Checker

Der Typ-Checker erfüllt die beiden folgenden Funktionen:

1. Checken der Wohlgetyptheit von Klauseln und Queries
2. Übersetzen der abstrakten Syntax von Klauseln und Queries in die Intermediate-Language.

Was ändert sich dabei durch die Integration von Feature-Typen? Der erste Punkt erforderte nur geringfügige Umstellungen, da Features genauso wie Funktionen getypcheckt werden. Dies wird zusätzlich dadurch unterstützt, daß die Ranks von Features und Funktionen die gleichen Zugriffsfunktionen haben. Somit können die Operationen zur Berechnung des kleinsten Codomains, zum Verschärfen des Präfix bei Containments und zum Auffalten von Feature-Aufrufen einfach von den Funktionen übernommen werden. Lediglich der syntaktische Arity-Check von Wertetermen muß an Features angepaßt werden.

Beim zweiten Punkt stellte sich die Frage, wie die neuen Konstrukte in der Intermediate-Language repräsentiert werden sollen. Da für Containments auf Feature-Typen spezieller Code erzeugt wird, müssen diese auf jeden Fall in der Intermediate-Language von anderen Containments unterschieden werden. Es ist außerdem klar, daß Features in Wertetermen ebenso wie Funktionen aufgefaltet werden müssen, da sie im Prolog-Code wie Relationen behandelt werden. Feature-Terme werden in der Intermediate-Language nicht repräsentiert. Sie werden vom Typ-Checker in eine entsprechende Folge von Gleichungen und Containments übersetzt und dabei gleichzeitig getypcheckt. In den Fehlermeldungen bezieht sich der Typ-Checker direkt auf die entsprechenden Stellen im Feature-Term. Damit er auch

bei einer rekursiven Verschachtelung von Feature-Termen die Fehlerstelle genau lokalisieren kann, wurde für Fehler in Feature-Termen ein rekursives Fehlerformat bereitgestellt, dessen Syntax wir kurz skizzieren.

```
  type_error  :=  ...++ fea_cont_err ++...  .
fea_cont_err  :=  some_error ++ {ftc_err:  fea_info x
                                           list(type_error)}
   fea_info   :=  {no_fea_inf, fea_inf:entry}.
```

Dabei wird über `fea_info` das Feature im Feature-Term spezifiziert, in dem der Fehler auftritt. Über die Rekursion von `type_error` kann die nächste Verschachtelungsstufe ebenso genau spezifiziert werden. Am Schluß der Rekursion steht dann eine ganz normale Fehlermeldung für das entsprechende Containment oder die entsprechende Gleichung. In dieser Meldung können zwar Hilfsvariable aus dem Auffaltungsprozeß des Feature-Terms auftreten; dies ist aber unproblematisch, weil der Benutzer die dazugehörige Stelle im Feature-Term exakt identifizieren kann.

Wir geben nun die oben beschriebenen Zusätze für die Intermediate-Language an. Alle *Feature-Aufrufe* in Wertetermen werden aufgefaltet und als eigenständige Condition dargestellt:

```
condition_call  :=  ...  ++ fea_call ++ ...
     fea_call   :=  {fcall :  entry x
                              variable_eterm x
                              variable_eterm}.
```

Das erste Argument von `fcall` enthält das Feature des Aufrufs, das zweite Argument enthält das Argument des Feature-Aufrufs und das dritte Argument enthält eine Hilfsvariable an die das Ergebnis des Feature-Aufrufs gebunden wird. *Containments auf Feature-Typen* werden wie folgt von anderen Containments differenziert:

```
containment_call := {...,
                     fcont :  constructor_eterm x eterm,
                     ...} .
```

Feature-Terme werden in Listen von Containments, Gleichungen, Announcements und Feature-Aufrufe übersetzt. Wir illustrieren dies an einem Beispiel. Darzustellen sei folgendes Containment auf einen Feature-Term:

```
X : dozent[name => 'Otto',
           chef :  professor[chef :  dozent]]
```

Die Darstellung dieses Feature-Terms in der abstrakten Syntax findet man in Abschnitt 6.2. In der Intermediate-Language wird obiges Containment durch die folgende Liste von Bedingungen ersetzt:

```
fcont(ev(X),ec(DOZENT,nil)).
ann_call(ev(_Y1)).
fea_call(ec(NAME,nil),ev(X),ev(_Y1)).
equal_call(ev(_Y1),'Otto').
ann_call(ev(_Y2)).
fea_call(ec(CHEF,nil),ev(X),ev(_Y2)).
fcont(ev(_Y2),ec(PROFESSOR,nil)).
ann_call(ev(_Y3)).
fea_call(ec(CHEF,nil),ev(_Y2),ev(_Y3)).
fcont_call(ev(_Y3),ec(DOZENT,nil)).nil
```

Beim Auffalten von Feature-Aufrufen werden die Hilfsvariablen `_Yi` durch Annoncements (`ann_call`) eingeführt. Man beachte, daß Containments auf Feature-Terme vom Typ-Checker genauso wie die dazu gehörende Folge von TEL-Conditions gecheckt wird.

6.5 Code-Erzeugung

Für ein Feature-TEL-Modul erzeugt der Compiler zwei Arten von Prolog-Code:

- Typ-Code
- Klausel-Code.

Der Typ-Code regelt zur Laufzeit die Abarbeitung von Containments und das Ausdrucken von Werte- und Typtermen. Der Klausel-Code resultiert aus der Übersetzung der Intermediate-Language-Klauseln des Moduls nach Prolog und stellt den eigentlichen Programm-Code dar. Queries werden ebenfalls nach Prolog übersetzt und vom Prolog-Interpreter abarbeitet.

Was ändert sich nun durch die Einführung von Feature-Typen in TEL? Zunächst bleibt der Prolog-Code für die bisherigen TEL-Konstrukte unverändert. Da Feature-Terme in der Intermediate-Language bereits aufgefaltet sind, fallen nur drei Erweiterungen des Prolog-Codes von Feature-TEL-Modulen an:

1. Code zur Abarbeitung von Containments auf Feature-Typen
2. Code zur Abarbeitung von Feature-Aufrufen
3. Code zum Ausdrucken von Feature-Termen und Feature-Typen in Query-Antworten.

Wir sehen uns diese drei Punkte nun etwas genauer an. Dabei zeigen wir nicht nur, wie der jeweilige Typ-Code aussieht, sondern geben auch Beispiele für die dazugehörigen Aufrufe im Klausel- beziehungsweise Query-Code.

6.5.1 Erzeugung von Prolog-Code zur Abarbeitung von Containments auf Feature-Typen

Containments auf Feature-Typen dienen sowohl als Typzugehörigkeitstest für gebundene Werteterme als auch zur Bindung offener Wertevariable an Feature-Typen.

Für jeden Feature-Typ wird eine einstellige Relation erzeugt, welche für jeden seiner impliziten Konstruktoren genau eine Klausel enthält. In solch einer Klausel wird dann das Topsymbol des Eingabeterms mit dem jeweiligen impliziten Konstruktor unifiziert. Die Bindung einer offenen Variablen an einen Feature-Typ wird durch Backtracking über seine impliziten Konstruktoren gewährleistet. Falls der Eingabeterm bereits gebunden ist, so wird gerade der Typzugehörigkeitstest durchgeführt.

Wir geben ein kleines Beispiel. Für den Feature-Typ `student` aus dem Modul `uni` werden die beiden folgenden Prolog-Klauseln erzeugt:

```
31student(31hoerercons(_,_)).
31student(31uebungsleitercons(_,_,_,_)).
```

Dabei sei `31` die Disambiguierung von `uni`. Das Containment

```
X : student
```

wird in das Prolog-Atom

```
31student(X)
```

übersetzt.

Der Nachteil dieses Ansatzes besteht darin, daß keine echten offenen Variablen eines nichtminimalen Feature-Typs existieren können. So bewirkt etwa die Folge

```
!X & X : student,
```

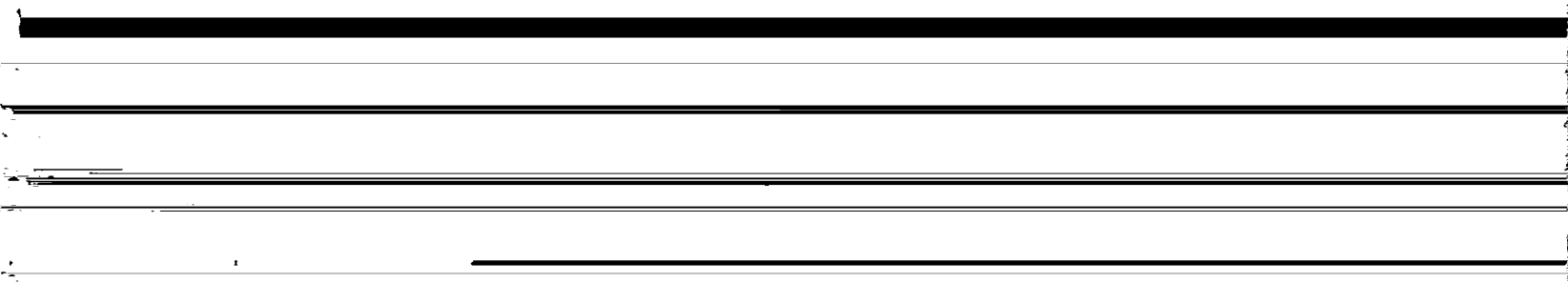
daß die "offene" Variable `X` an den Werteterm

```
31hoerercons(_,_)
```

Erzeugungsfunktion für Feature-Typen angewiesen, die Typbindungen durch Unifikation mit impliziten Konstruktoren realisiert. Selbst wenn man bei einer Übersetzung nach Prolog ungebundene Variablen eines Feature-Typs erzeugen würde, so hätte man immer noch das Problem der Abarbeitung von Feature-Aufrufen auf solche Variablen. Wie kann man einen Feature-Aufruf realisieren ohne eine Bindung an implizite Konstruktoren durchzuführen?

6.5.2 Erzeugung von Prolog-Code zur Abarbeitung von Feature-Aufrufen

Für jedes Feature f wird eine zweistellige Prolog-Relation erzeugt, welche semantisch



den Selektorgleichungen von f entspricht. Dazu werden zunächst die impliziten Konstruktoren aller Feature-Typen, die im Domain von f liegen aufgesammelt. Für jeden solchen impliziten Konstruktor c wird nun eine Klausel angelegt, in der das Ausgabeargument des Feature-Aufrufs mit der zu f gehörigen Argumentposition von c unifiziert wird. Zur Illustration geben wir den Code für das Feature `semester` an:

```
31semester(31hoerercons(_,X),X).
31semester(31uebungsleitercons(_,X,_,_),X).
```

Für den Feature-Aufruf

```
semester(X) = Y
```

wird das Prolog-Atom

```
31semester(X,Y)
```

erzeugt. Man beachte, daß der Typ-Checker garantiert, daß bei einem Aufruf von `31semester` das erste Argument bereits an einen Grundwerteterm aus dem Domain von `semester` gebunden ist! Das bedeutet, daß `31semester` nie fehlschlagen kann. Gleichzeitig kann immer nur mit genau einer Klausel unifiziert werden. Features sind also wirklich total und deterministisch.

6.5.3 Erzeugung von Prolog-Code zum Ausdrucken von Query-Antworten

Um die Probleme beim Ausdrucken von Query-Antworten deutlich zu machen, wer-

auch die kleinsten Typen für die in der Query vorkommenden Variablen abgeleitet. Anschließend wird die Query in ihre Prolog-Darstellung übersetzt, wobei man sich die Zuordnung von Prolog-Variablen zu Query-Variablen merkt. Nun kann der Prolog-Interpreter die Query abarbeiten. Ist keine erfolgreiche Abarbeitung möglich, so wird `failed` ausgegeben, ansonsten werden die Bindungen der Query-Variablen ausgedruckt. Dazu gehen wir von den Bindungen ihrer Prolog-Äquivalente aus. Um einen Prologterm auszudrucken benötigt man seinen Typ. Dies ist wichtig, weil Werte verschiedener Typen ganz verschieden ausgegeben werden. Beispielsweise erfordert das Ausdrucken von negativen Zahlen eine andere Behandlung als das Ausdrucken von Strings. Die Typen der Query-Variablen wurden bereits vom Typ-Checker abgeleitet und sind daher bekannt. Es reicht jedoch nicht aus, nur den Typ des gebundenen Terms zu kennen. Wenn der gebundene Term aus einem Topsymbol und Argumenten besteht, so müssen wir auch die Typen dieser Argumente kennen. Dies bedeutet, daß man zur Laufzeit aus dem Typ und dem Topsymbol eines Werteterms auf die Typen seiner Argumente zurückschließen können muß. Die dazu notwendige Information muß also in Form von Typ-Code bereitgestellt werden. Das gleiche gilt für die Namen der Typsymbole beim Ausdrucken von Typtermen. Dort will man die in den Feature-TEL-Programmen eingeführten Bezeichner verwenden und nicht deren Prolog-Codenamen. Außerdem muß man zur Laufzeit feststellen können, ob ein Typ ein POS-Typ oder ein Feature-Typ ist, denn diese werden verschieden getypcheckt und ausgegeben.

Die angesprochenen Informationen werden für jeden Typ in einer dreistelligen Relation bereitgestellt, deren erste Klausel den Drucknamen und die Typart des Typs enthält. Die restlichen Klauseln ermöglichen den Rückschluß vom Topsymbol eines Terms dieses Typs auf die Typen seiner Argumente. Dabei ist das zweite Argument nur für polymorphe Typen von Bedeutung. Bei Feature-Typen wird es mit `[]` belegt. Wir illustrieren das Gesagte am Beispiel des Feature-Typs `student`.

```
31student(printname,'student',fea_type).
31student(31hoerercons(_1,_2),[],[_1$string,_2$nat])).
31student(31uebungsleitercons(_1,_2,_3,_4),[],[_1$string,..])).
```

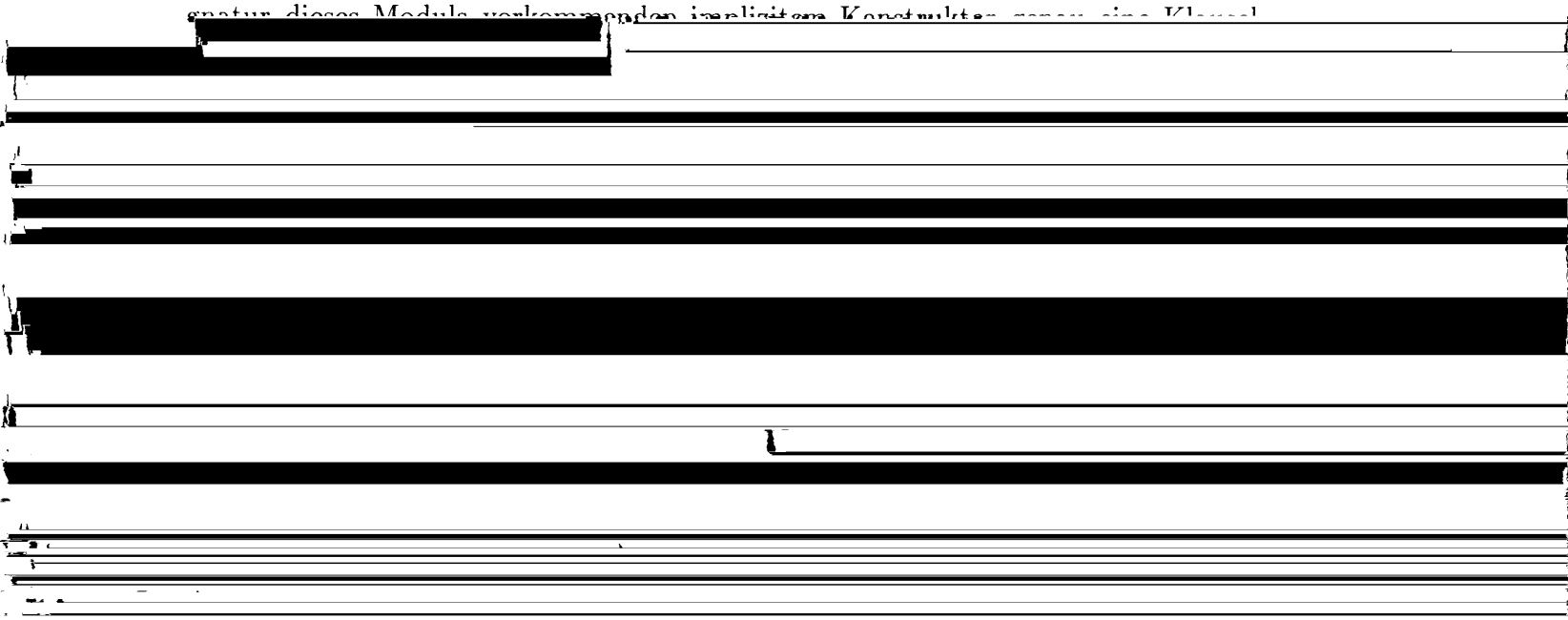
Man beachte, daß `$` als ein Konstruktor benutzt wird, der Terme und die dazugehörigen Typen zu Paaren zusammenfaßt.

Die Elemente von Feature-Typen werden in Form von Feature-Termen ausgegeben (siehe 5.2.3). Das bedeutet, daß wir Prolog-Terme mit implizitem Konstruktor als Topsymbol in den dazugehörigen Feature-Term übersetzen müssen. Wir müssen also zur Laufzeit aus dem Namen eines impliziten Konstruktors die Namen seiner (im geöffnetem Modul) minimalen bekannten Obertypen und die Namen seiner Features herleiten können. Diese Information wird wieder in Form von Typ-Code abgelegt. Dabei sollen auch Abstraktionen und Koreferenzen berücksichtigt werden.

Eine Koreferenz ist eine Prologvariable, die zu keiner Query-Variable gehört, aber mehrfach in den Bindungen der Query-Variablen vorkommt. Koreferenzen werden durch lineares Abarbeiten der Query-Variablen-Bindungen erkannt und an

numerierte Bezeichner gebunden. Innerhalb eines Feature-Terms werden Feature-Werte nur ausgegeben, falls die dazugehörige Stelle im impliziten Konstruktor-Term gebunden ist.

Nun ist es möglich, daß innerhalb von Feature-Termen Features oder Feature-Typen ausgegeben werden, die eigentlich in der lokalen Signatur des geöffneten Moduls wegabstrahiert wurden und somit für den Benutzer nicht sichtbar sein sollen. Deshalb muß unser Typ-Code für implizite Konstruktoren modulspezifisch sein. Für jedes Modul wird eine dreistellige Relation erzeugt, die zu jedem in der lokalen Signatur dieses Moduls vorkommenden impliziten Konstruktor genau eine Klausel



enthält. In dieser Klausel sind die Namen der in dem Modul minimalen bekannten Obertypen dieses Konstruktors sowie die Namen der dazugehörigen Features abgelegt. Falls ein Feature wegabstrahiert wurde, so wird es als `*abstract_feature*` benannt. Ist der im importierenden Modul minimale bekannte Obertyp eines impliziten Konstruktors im Ursprungsmodul nichtminimal, so wird er in * eingeschlossen. Diese Information wird schon während der Signatur-Analyse berechnet und dann direkt an die Code-Erzeugung weitergegeben. Wir illustrieren das Gesagte an einem Beispiel. Ein Modul `uni_top` importiere die Feature-Hierarchie des Moduls `uni` ohne den Feature-Typ `uebungsleiter` und das Feature `semester`. Dann sieht der modulspezifische Code für `uni_top` folgendermaßen aus.

```
42icons(31hoerercons(_,_),hoerer,[name,*abstract_feature]).
42icons(31uebungsleitercons(_,_,_,_),*student∩dozent*,
        [name,*abstract_feature*,...]).
...
```

Dabei sei `42` die Disambiguierung von `uni_top`.

Kapitel 7

Erfahrungen mit Feature-TEL

In diesem abschließenden Kapitel geben wir eine kritische Betrachtung der Möglichkeiten des Feature-TEL-Systems. Dabei greifen wir auf unsere Erfahrungen mit Feature-TEL während des Bootstrapping-Prozeßes zurück.

Die Implementierung von Feature-TEL verlief in zwei Phasen:

1. Implementierung des Feature-TEL-Systems in TEL. Dabei wurde auf dem Code des alten TEL-Systems aufgebaut.

2. Reimplementierung des Feature-TEL-Systems in Feature-TEL.

Die zweite Phase diente gleichzeitig als Test für das Feature-TEL-System.

Die wesentliche Änderung beim Bootstrap war die Reimplementierung von Signatur-Einträgen als Feature-Typen. Die Signatur-Einträge sind die zentrale Datenstruktur des TEL-Systems und somit zog sich die Änderung durch fast alle Module des Systems. Dadurch wurden die implementierten Operationen auf Feature-Typen einem intensiven Test unterzogen.

Die Repräsentierung von Signatur-Einträgen (`entries`) durch Feature-Typen

```
entry:=[name:designator,
        module_name:modname,
        code_name:codename,
        line_number:linenumber].

type_entry:=entry[].

feature_type_entry:=type_entry
                    [given_supertypes:list(type_entry),
                     ...
                     complete_features:list(feature_entry)].

poly_type_entry:=type_entry[formal_args:list(eterm)].

full_type_entry:=poly_type_entry
                    [given_subtypes:list(eterm),
                     ...
                     category:type_category].

abbr_type_entry:=poly_type_entry[given_expansion:eterm,
                                 expansion:eterm].

feature_entry:=entry[given_ranks:list(rank),
                     expanded_ranks:list(rank)].

function_entry:=entry[given_ranks:list(rank),...].

constructor_entry:=entry[given_rank:rank,...].

parameter_entry:=entry[given_codomain:eterm,...].

relorproc_entry:=entry[rp_class:...].
```

Abbildung 7.1: Signatur-Einträge als Feature-Typen

- Abstrakter Zugriff auf die Argumente eines Typs. Die nervtötende Stellenzählerei für Konstruktoren mit vielen Argumenten entfällt ebenso wie die redundante Angabe von unbenötigten Argumenten.
- Das Schreiben von Selektor-Funktionen entfällt.
- Kompakte und übersichtliche Darstellung einer Menge von Feature-Constraints durch Feature-Terme.
- Information-Hiding und Zugriffsschutz durch Abstraktionen von Feature-Typen. Dabei ist insbesondere das Ausblenden bestimmter Typ-Domains für Features nicht auf Konstruktor-Typen übertragbar.

Insgesamt bewirkte die Umstellung der `entries` auf Feature-Typen eine spürbare Reduzierung des TEL-Codes, sowie eine Verbesserung seiner Lesbarkeit und der Fehlerlokalisierung.

Eine Schwachstelle der Implementierung von Feature-TEL ist die Behandlung offener Variablen. Diese werden durch Containments auf Feature-Typen an deren implizite Konstruktoren gebunden. Dabei muß der Benutzer sich allerdings darüber im klaren sein, daß diese Typ-Bindung über Backtracking realisiert ist. Die langfristige Lösung kann deshalb nur eine abstrakte Maschine für Feature-TEL sein, die wirkliche Typ-Bindungen vornimmt. Auch die Forderung der Bewohntheit von nichtminimalen Feature-Typen kann dadurch, daß sie die Einführung von unsinnnigen "Dummy"-Typen erzwingt, zum Hemmschuh werden.

Alles in allem stellt sich Feature-TEL aber als eine praktikable, ausdrucksstarke Programmiersprache dar, die eine adäquate Repräsentierung komplexer Datenstrukturen ermöglicht.

In der Zukunft ist insbesondere die Zulassung von Feature-Termen im Klauselkopf, in Relationsatomen und in Gleichungen geplant. Auch eine Erweiterung der Ausdruckskraft von Feature-Typ-Deklarationen sollte genauer untersucht werden. Dabei kommt etwa die explizite Modellierung von partiell definierten Features in Betracht. Ein weiteres interessantes Thema wäre die Zulassung von numerischen Unterbereichstypen oder gar von beliebigen Werten in Feature-Typ-Deklarationen.

Anhang A
Mathematische Grundbegriffe

Wir geben hier die Definitionen einiger in dieser Arbeit häufig verwendeter mathematischer Grundbegriffe an. Dabei verwenden wir die gleiche Notation wie [Sm 89a].

Gegeben eine zweistellige Relation $\rightarrow$ auf einer Menge M, so bezeichnet $\rightarrow^*$ die reflexive und transitive Hülle von $\rightarrow$. Die Relation $\rightarrow$ *terminiert*, falls es keine unendlichen Ketten $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ gibt.

Terme werden aus Variablen und Funktionssymbolen zusammengesetzt. Einen Term der Form $f(s_1, \ldots, s_n)$ (wobei $n \geq 0$) schreiben wir oft verkürzt als $f(\vec{s})$. Dabei bezeichnet $\vec{s}$ den möglicherweise leeren Tupel $(s_1, \ldots, s_n)$. Die Funktion $\mathcal{V}$ liefert die Menge aller in einem Term enthaltenen Variablen. Statt $\mathcal{V}(s)$ schreiben wir auch $\mathcal{V}s$.

Eine *Substitution* ist eine totale Funktion ψ von Termen auf Terme, so daß für alle Terme $f(\vec{s})$ die Gleichung $\psi f(\vec{s}) = f(\psi\vec{s})$ erfüllt ist. Gegeben eine Substitution ψ, so ist der *Domain* von ψ folgendermaßen definiert:

$$\mathcal{D}\psi := \{x \mid \psi x \neq x \text{ und } x \text{ ist eine Variable}\}.$$

Eine Substitution ψ heißt *endlich*, falls $\mathcal{D}\psi$ endlich ist. Eine Substitution ψ heißt *idempotent*, falls $\psi = \psi\psi$. Ein Term t heißt *Instanz* des Terms s, falls es eine Substitution ψ gibt mit $t = \psi s$.

Eine *Rewriting-Regel* $s \rightarrow t$ ist ein geordnetes Paar zweier Terme s und t, wobei $\mathcal{V}t \subseteq \mathcal{V}s$ gilt. Ein *Rewriting-System* ist eine Menge von Rewrite-Regeln. Gegeben ein Rewriting-System R, so nennen wir $s \rightarrow t$ die *Instanz einer Regel aus* R, falls es eine Substitution ψ und eine Regel $u \rightarrow v$ in R gibt, so daß $s = \psi u$ und $t = \psi v$. Wir schreiben $s \Rightarrow_R t$, falls $s \rightarrow t$ die Instanz einer Regel aus R ist. Wir schreiben $s \rightarrow_R t$, falls s einen Unterterm u hat für den $u \Rightarrow_R v$ gilt und t sich aus s ergibt, indem in s ein (nicht jeder) Unterterm u durch v ersetzt wird. Ein Rewriting-System R heißt *terminierend*, falls die Relation "$s \rightarrow_R t$" terminiert. Ein Rewriting-System heißt *konfluent*, falls für alle Terme u, s und t mit $u \rightarrow_R^* s$ und $u \rightarrow_R^* t$ ein Term v existiert mit $s \rightarrow_R^* v$ und $t \rightarrow_R^* v$.

Abbildungsverzeichnis

1.1 Eine Vererbungshierarchie 4
1.2 Ein POS-Programm 6

2.1 Der Konstruktor-Typ **Mensch** als Vererbungshierarchie 10
2.2 Algebraische Spezifikation der Kfz-Vererbungshierarchie 11

3.1 Auffalten von Wertetermen 34
3.2 Auffalten von Constraint-Systemen 35
3.3 Reduktionsregeln für Containments 36
3.4 Reduktionsregeln für Constraints 37
3.5 Goal-Reduktion 42

5.1 Signatur von `uni` 64
5.2 Export-Signatur von `uni-view1` 71
5.3 Export-Signatur von `uni-view2` 73
5.4 Import-Signatur von `top` 74

6.1 Der Feature-Typ-Eintrag für `student` 85
6.2 Bewohntheitstest für Feature-Hierarchien 90

7.1 Signatur-Einträge als Feature-Typen 102

Literaturverzeichnis

[AN 86] H.Aït-Kaci und R.Nasr, "LOGIN: A Logic Programming Language with Built-in Inheritance", *The Journal of Logic Programming*, S.185-215, 1986

[AS 87] H.Aït-Kaci und G.Smolka, "Inheritance Hierarchies: Semantics and Unification", *MCC Report AI-057-97*, Austin, Texas, 1987

[BS 85] R.Brachmann und G.Schmolze, "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science 9*, S.171-216, 1985

[Go 78] J.A.Goguen, "Order Sorted Algebra" *Semantics and Theorie of Computation: Report Nr. 14*, UCLA Computer Science Department, 1978

[GR 83] A.Goldberg und D.Robson, "Smalltalk-80, The Language and its Implementation", *Addison-Wesley*, 1983

[GT* 78] J.A.Goguen, J.W.Thatcher, und E.G.Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types" in *R.T.Yeh (Ed.): Current Trends in Programming Methodology, Vol. 4 Prentice Hall*, S.80-149, 1978

[H 88] M.Höhfeld, "Eine Schema für constraint-basierte Wissensbanken", *Diplomarbeit*, Universität Kaiserslautern, 1988

[Sch 87] M.Schmitt, "Implementierung einer logischen Programmiersprache mit Gleichheit", *Diplomarbeit*, Universität Kaiserslautern, 1987

[Shi 86] S.M.Shieber, "An Introduction to Unification-Based Approaches to Grammar", *CSLI Lecture Notes 4*, Center for the Study of Language and Information, Stanford University, 1986

[Sm 88] G.Smolka, "TEL: Report and User Manual", *SEKI-Report SR-87-11*, Universität Kaiserslautern, 1988

[Sm 89a] G.Smolka, "Logic Programming over Polymorphic Types", *Dissertation*, Universität Kaiserslautern, 1989

[Sm 89b] G.Smolka, "Feature-Constraint-Logics for Unification Grammars", *IWBS-Report 93*, IBM Deutschland, IWBS Stuttgart, 1989

[SN* 87] G.Smolka, W.Nutt, J.A.Goguen und J.Meseguer, "Order Sorted Equational Computation", *SEKI-Report SR-87-14*, Universität Kaiserslautern, 1987

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:

1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka : On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt: The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost
X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents
A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist
WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta
RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke
A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel
Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-13
Gert Smolka
Residuation and Guarded Rules for Constraint Logic Programming
17 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber: Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay: Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler
Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann
Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel and Rainer Bleisinger
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

DFKI Documents

D-89-01
Michael H. Malburg, Rainer Bleisinger:
HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

D-90-02
Georg Seul: Logisches Programmieren mit Feature -Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

D-91-01
Werner Stein , Michael Sintek
Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause
RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

Logisches Programmieren mit Feature-Typen

Georg Seul

D-90-02
Document