Optimierung und Simplifikation von Anfragen an eine KL-ONE-Wissensbasis

Diplomarbeit

von
Pok-Son Kim

eingereicht bei
Prof. Dr. M. Schmidt-Schauß
Professor für
Künstliche Intelligenz / Softwaretechnologie
Johann Wolfgang Goethe-Universität
Frankfurt am Main
August 1994

Danksagung

Herrn Professor Schmidt-Schauß danke ich für die ausgezeichnete Betreuung meiner Diplomarbeit. Aufrichtig möchte ich mich bei ihm dafür bedanken, daß er mir trotz meiner sprachlichen Schwierigkeiten stets große Geduld im Zuhören und Erklären entgegengebracht hat. Bei Arne Kutzner bedanke ich mich dafür, daß er mir bei Fragestellungen zur deutschen Sprache immer hilfreich zur Seite stand. Herrn S. E. Panitz danke ich für seine stete Hilfsbereitschaft während meiner Arbeit.

Mein größter Dank gilt meiner Mutter sowie meinen Geschwistern, ohne deren großzügige Unterstützung mein Studium in Deutschland nicht möglich gewesen wäre.

Pok-Son Kim

Hiermit bestätige ich, daß ich die vorliegende Arbeit selbständig verfaßt habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 9. August 1994

Pok-Son Kim

Inhaltsverzeichnis

Kapitel 1

Einführung

Terminologische Wissensrepräsentationssprachen sind die Weiterentwicklung eines Wissensrepräsentationssystem mit dem Namen KL-ONE, das Ende der 70er Jahre entwickelt wurde. Die Entwicklung von KL-ONE wurde durch die Debatte über die Rolle der Logik in der Künstlichen Intelligenz motiviert. Eine weitere Motivation für die Entwicklung von KL-ONE bestand darin, die gegensätzlichen Konzepte der semantischen Netzwerke und der Frames zu kombinieren.
Terminologische Repräsentationssprachen besitzen zwei syntaxtische Primitive, Konzepte und Rollen, wobei in der modelltheoretischen Semantik Konzepte als Mengen und Rollen als binäre Relationen interpretiert werden. Desweiteren verfügen Wissensrepräsentationssysteme über mehrere Folgerungsalgorithmen. Mittels eines solchen Algorithmus kann beispielsweise das Retrieval-Problem gelöst werden, das darin besteht die Antwort auf die Frage ,,Was sind die Instanzen eines gegebenen Konzeptes (bzw. Rolle) ?“ zu ermitteln.
Eine Aufgabenstellung meiner Diplomarbeit besteht darin, ein Programm in der nicht strikten funktionalen Programmiersprache **Gofer** zu erstellen, mit dem das Retrieval-Problem gelöst werden kann. Dabei werden Anfragen, die Konzepte oder Rollen darstellen, interaktiv vom Programm ausgewertet, das heißt, es werden die Instanzen eines Konzepts bzw. einer Rolle ausgegeben.

Mengen können mittels Kalkülen dargestellt werden, die im Kontext Boolscher Algebren formalisiert werden. Genauso können Relationen mittels Kalkülen dargestellt werden, die im Kontext von Relationenalgebren formalisiert werden. Konzepte und

Rollen beeinflussen sich gegenseitig. Dies kann mittels Kalkülen über Mengen und Relationen im Kontext Boolscher Module und im Kontext von Peirce-Algebren formalisiert werden. In diesem algebraischen Rahmen kann die Festlegung der Semantik terminologischer Sprachen erfolgen. Auswirkungen davon sind, daß jedes Konzept bzw. jede Rolle als algebraischer Term dargestellt werden kann und die in den Algebren erfüllten Gleichungsaxiome sowie die daraus gefolgerten Gleichungen zum Simplifizieren von den oben genannten Anfragen verwendet werden können. Die solchen Simplifikationen zugrundeliegende Idee ist, daß eine in den obigen Algebren erfüllte Gleichung aus zwei äquivalenten Termen besteht, für die unterschiedliche Reduktionsschritte zur Evaluierung benötigt werden. Den Schwerpunkt meiner Diplomarbeit bildet die Optimierung und Simplifikation von Anfragen an eine KL-ONE Wissensbasis.

Im zweiten Kapitel dieser Arbeit gebe ich eine Einführung in die funktionalen Programmiersprachen. Dabei werden fünf ausgewählte Aspekte funktionaler Sprachen behandelt, die im Zusammenhang mit dem von mir erstellten Programm von Bedeutung sind.
Das dritte Kapitel enthält eine Einführung in die terminologische Repräsentation und die Definition der terminologischen Sprache $\mathcal{ALC}$. Im Rahmen dieser Einführung wird zu Beispielszwecken Wissen über eine Umwelt von Personen mittels $\mathcal{ALC}$ repräsentiert.
Im Rahmen des vierten Kapitels werden die algebraischen Begriffe „Boolsche Algebra“, „Relationenalgebra“, „Boolscher Modul“ und „Peirce-Algebra“ definiert. Ferner werden aus den Axiomen der Boolschen Algebren und Relationenalgebren gefolgerte arithmetische Eigenschaften aufgeführt, die zum Simplifizieren verwendet werden. Auch wird in diesem Kapitel eine zu $\mathcal{ALC}$ erweiterte Sprache $\mathcal{ALCX}$ definiert und deren Semantik algebraisch dargestellt.
Vor der abschließenden Zusammenfassung beschreibe ich im fünften Kapitel Aufbau und Funktionsweise meines Programms. Im Vordergrund steht dabei, wie mein Programm bei der Evaluation bzw. Optimierung einer Anfrage vorgeht.

Kapitel 2

Funktionale Programmiersprachen Miranda, Gofer, Haskell, $\cdots$.

Die Geschichte der funktionalen Programmierung reicht in die 30er Jahre dieses Jahrhunderts zurück. Das Konzept der funktionalen Programmierung basiert auf dem von Church entwickelten λ-Kalkül [CA32, CA41]. Der λ-Kalkül ist ein mathematischer Formalismus zur Beschreibung von Funktionen. Gegenüber anderen mathematischen Formalismen zur Beschreibung von Funktionen zeichnet sich der λ-Kalkül dadurch aus, daß er auch die Formulierung unbenannter Funktionen ermöglicht.

Will man für eine gegebene Funktion die Funktionswerte von einem Computer berechnen lassen, so muß die Funktion zunächst in einer für den Computer verständlichen Form beschrieben werden. Bei einer derartigen Beschreibung wünscht man, die Vorteile der prägnanten mathematischen Definitionsweise beizubehalten. Aus diesem Wunsch heraus hat sich das Konzept der funktionalen Programmierung entwickelt. Funktionale Programmierung ermöglicht Funktionsbeschreibungen auf eine mathematische, vom Berechungsvorgang abstrahierende, Weise.

„Welche Vorteile haben funktionale Sprachen?“
Funktionale Sprachen sind zur Erstellung komplexer Software-Systeme im naturwissenschaftlich-technischen Bereich teilweise besser geeignet als herkömmliche, imperative Sprachen wie Pascal oder Fortran. Dies ist darin begründet, daß funktionale Programmiersprachen bei einer Vielzahl von Problemstellungen eine knappere und prägnantere Programmformulierung ermöglichen als imperative Sprachen. Besonders

gut geeignet sind funktionale Programmiersprachen zur Behandlung mathematisch orientierter Problemstellungen.
Funktionale Sprachen bieten sich als Spezifikationsmedium an, weil ein Problem knapp, präzise und mit angemessenem Abstraktionsgrad formuliert werden kann. Funktionale Sprachen können in diesem Zusammenhang im Bereich des „rapid prototyping" sinnvoll eingesetzt werden. „Rapid prototyping" bedeutet die schnelle Erstellung einer Applikation mit modellhaftem Charakter. „Rapid prototyping" wird eingesetzt, um die Tragfähigkeit der grundlegenden Ideen und Konzepte bei der Realisation eines Softwaresystems zu überprüfen.

Die funktionalen Sprachen können wie folgt charakterisiert werden:
Ein funktionales Programm besteht aus einer Menge von Definitionen, die in einem Skript formuliert werden. Die in einem Skript spezifizierten Definitionen werden als Gleichungen zwischen bestimmten Arten von Ausdrücken formuliert und beschreiben mathematische Funktionen. Programmieren in einer funktionalen Programmiersprache bedeutet Funktionen zu konstruieren, um ein gegebenes Problem zu lösen. Ein funktionales Programm wird ausgeführt, indem ein Ausdruck zur Auswertung an den Computer übergeben wird. Wird bei der Evaluierung einer Funktion bzw. eines Ausdrucks ein Funktionsname erreicht, so wird dieser reduziert, indem die gegebene Funktionsdefinition als Vereinfachungs- bzw. Reduktionsregel benutzt wird.
Eine weitere Chrakterisierung funktionaler Sprache ist, daß die Ausdrücke in funktionaler Programmierung einen wohldefinierten Wert besitzen und die Bedeutung eines Ausdrucks sein Wert ist. Zudem kann ein Ausdruck in einer funktionalen Sprache wie jeder beliebige andere mathematische Ausdruck durch Anwendung algebraischer Gesetze konstruiert und manipuliert werden.

Bei manchen Implementationen funktionaler Programmiersprachen wird die Auswertung von Ausdrücken durch ein Verfahren optimiert, das call by need bzw. lazy evaluation genannt wird. Grundgedanke dieses Verfahrens ist, daß ein Wert nur dann berechnet wird, wenn er gebraucht wird, und eine wiederholte Auswertung identischer Ausdrücke verhindert wird.

Im folgenden werde ich anhand von Beispielen eine Einführung in die Programmierung mittels funktionaler Programmiersprachen geben. Dabei beschränke ich mich auf einzelne ausgewählte Aspekte, die im Zusammenhang mit dem von mir erstellten Programm von Bedeutung sind; Grundkenntnisse über funktionale Programmierung setze ich voraus.

2.1 Benutzerdefinierte Datentypen

Alle funktionalen Programmiersprachen stellen eine Reihe elemtarer Datentypen bereit. In Gofer sind dies beispielsweise Typen zur Darstellung von Zahlen, Wahrheitswerten, Zeichen, Listen, Listen von Zeichen (Strings) und Tupeln. Zusätzlich ist es in funktionalen Programmiersprachen möglich, neue Datentypen zu definieren. Die dabei verwendete Syntax unterscheidet sich bei den einzelnen funktionalen Programmiersprachen. In diesem Abschnitt stelle ich die in Gofer gegebenen Möglichkeiten zur Definition neuer Datentypen vor.

- *Konstruktor ohne Parameter*

  Angenommen, es wird ein Datentyp zur Repräsentation von Wochentagen benötigt. Ein geeigneter Datentyp dafür könnte mittels der Datendeklaration

  ```
  data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  ```

  eingeführt werden. `data` ist ein reserviertes Wort, das die Deklaration eines neuen Datentyps einleitet. `Day` ist der Name des neuen Typkonstruktors. Auf der rechten Seite des Gleichheitszeichens sind, getrennt durch das Symbol „|“, die Elemente des Datentyps `Day` angegeben. Diese Elemente (`Sun, Mon, Tue, Wed, Thu, Fri, Sat`) werden Konstruktoren genannt. Die Namen der Konstruktoren müssen zu allen bisher für Konstruktoren verwendeten Namen verschieden sein und mit einem Großbuchstaben beginnen. Eine einfache Funktion [JM93], die die Elemente von Typ `Day` manipuliert, kann mit pattern matching definiert werden:

  ```
  what_shall_I_do :: Day → String
  what_shall_I_do Sun = ''relax''
  what_shall_I_do Sat = ''go shopping''
  what_shall_I_do _ = ''looks like I'll have to go to work''
  ```

- *Konstruktor mit Parameter*

Beim Typ `Day` sind alle Konstruktoren Konstanten. Es ist jedoch auch möglich einen Datentyp zu definieren, dessen Konstruktoren Parameter haben. In diesem Fall werden die Konstruktoren auch als Konstruktorfunktionen bezeichnet. In einem neuen Datentyp `Temp` wird z.B. die Repräsentation von Temparaturen durch die Konstruktoren, `Centigrade`, `Fahrenheit`, mit dem Parameter `Int` ermöglicht:

```
data Temp = Centigrade Int | Fahrenheit Int
freezing :: Temp → Bool
freezing (Centigrade temp) = temp <= 0
freezing (Fahrenheit temp) = temp <= 32
```

Nach dem Laden der Textdatei, in der die Funktion `freezing` formuliert ist, kann man die im folgenden aufgeführten Anfragen stellen. Eingabe und anschließende Ausgabe des Computers haben folgendes Aussehen:

? freezing (Centigrade 5)
False
(4 reductions, 11 cells)
? freezing (Fahrenheit 31)
True
(3 reductions, 10 cells)

Der Datentypname kann Typvariablen enthalten und in diesem Fall dürfen die Parameter, die in den Konstruktorfunktionen vorkommen, nur aus diesen Typvariablen bestehen:

```
data Set a = K_set [a]
```

Zum Beispiel ist `K_set [1, 2, 3]` ein Element vom Typ `Set Int`. `K_set [1, 2, 3]` bezeichnet die Menge der ganzen Zahlen $\{1, 2, 3\}$. `K_set ['c']` bezeichnet eine einelementige Menge von Typ `Set Char`.

- *Rekursive Datentypen*

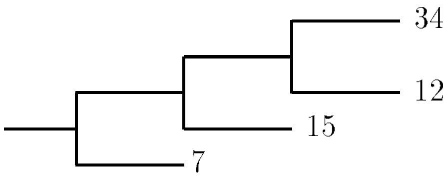
  Ein Datentyp kann rekursiv definiert werden, d.h. bei einer neuen Datentypdefinition kann der Name des zu definierenden Typs auf der rechten Seite erscheinen. Als Beispiel dafür sei die Definition einer Datenstruktur zur Repräsentation von Bäumen unten gezeigt.

```
data Baum a = Blatt a |
              Knoten (Baum a) (Baum a)
```

Beispielsweise hat

```
Knoten (Knoten (Knoten (Blatt 34) (Blatt 12))
               (Blatt 15))(Blatt 7)
```

den Typ `Baum Int` und repräsentiert den folgenden binären Baum.



Im folgenden zeige ich ein Beispiel für eine Funktion, die auf einer Datenstruktur vom Datentyp `Baum a` operiert. Die Funktion gibt als Ergebnis alle in einem Baum enthaltenen Blätter von links nach rechts in Form einer Liste aus:

```
blättern :: Baum a → [a]
blättern (Blatt l) = [l]
blättern (Knoten a b) = blättern a ++ blättern b
```

Zum Beispiel:

?blättern (Knoten (Knoten (Knoten
            Blatt 34 Blatt 12)(Blatt 15)) (Blatt 7))
[34, 12, 15, 7]

Es gibt eine weitere Möglichkeit, den Datentyp `Baum a` zu definieren. Statt der Konstruktorfunktion `Knoten` wendet man einen Infixoperator an, der mit : beginnt:

```
data Baum a = Blatt a |
              Baum a :∧: Baum a
```

Ein Beispielelement dieses Typs sieht wie folgt aus:

```
Blatt 12 :∧: ((Blatt 25 :∧: Blatt 7) :∧: Blatt 9)
```

- *Zusammenfassung*

  Allgemein hat die Definition eines Datentyps in Gofer folgendes Aussehen:

  `data` $Datentyp\ a_1 \cdots a_n = constr_1 \mid \cdots \mid constr_m$

  Dabei ist *Datentyp* der Name des neuen Typkonstruktors mit der Ordnung $n \geq 0$. $a_1, \cdots, a_n$ sind verschiedene Typvariablen, welche die Argumente des Datentypnamen repräsentieren. Die Elemente $constr_1, \cdots, constr_m (m > 1)$ bilden die Konstruktorfunktionen des neu definierten Datentyps *Datentyp*. Bei den Konstruktorfunktionen sind die beiden folgenden Arten zu unterscheiden:

  1. $Name\ Typ_1 \cdots Typ_r (r \geq 0)$, wobei *Name* ein Name ist. Diese Deklaration führt *Name* als eine neue Konstruktorfunktion vom folgenden Typ ein:
     $$Typ_1 \to \cdots \to Typ_r \to Datentyp\ a_1 \cdots a_n$$
  2. $Typ_1 \otimes Typ_2$ wobei $\otimes$ Operator von Konstruktorfunktionen ist. Diese Deklaration führt $(\otimes)$ als eine neue Konstruktorfunktion vom Typ:
     $$Typ_1 \to Typ_2 \to Datentyp\ a_1 \cdots a_n$$

2.2 Polymorphismus

Eine Funktion `quadrat` sei wie folgt deklariert:

```
quadrat :: Int → Int
quadrat x = x × x
```

Bei dieser Funktionsdeklaration ist ein bestimmter Quell- und Zieltyp angegeben. Anstelle einer bestimmten Typangabe kann man eine Typvariable verwenden. Dies gibt die Möglichkeit, Funktionen mit allgemeinen Quell- und Zieltypen zu definieren. Betrachtet sei die folgende Funktionsdefinition:

```
id x = x
```

Diese Gleichung definiert die Identitätsfunktion; sie bildet jedes Element des Quelltyps auf sich selbst ab. Der Typ dieser Funktion kann ein beliebiger passender Typ `a` sein. Bei der Typdeklaration der Funktion `id` kann man anstelle einer bestimmten Typangabe den Typ `a` $\rightarrow$ `a` angeben, wobei `a` eine Typvariable ist. Diese Typvariable `a` kann bei Verwendung der Funktion `id` durch alle möglichen Typen, z.B. `Int, Char, Str,` $\cdots$ ersetzt werden. Der Ausdruck (`id` 3) ist zum Beispiel wohlformuliert und hat den Typ `Int`. Für `a` wurde in diesem Beispiel der Typ `Int` verwendet, so daß sich (`Int` $\rightarrow$ `Int`) als Typ von `id` ergibt.

Falls bei der Definition eine oder mehrere Typvariablen erscheinen, wird davon gesprochen, daß die Funktion einen Polymorphen Typ besitzt.

Beispiele für Funktionen mit polymorphem Typ sind:

```
length :: [a] → Int
(++)   :: [a] → [a] → [a]
concat :: [[a]] → [a]
```

Beispielsweise können wir die beide Länge von sowohl [`Int`] (list of integer) als auch [`Char`] (list of character) berechnen lassen.

```
? length [1..10]
10
(98 reduction, 138 cells)
? length "Hello"
5
(22 reduction, 36 cells)
?
```

2.3 Eine Besonderheit von Gofer - die Typklasse Eq

Eine Typklasse ist eine Familie von Typen (bzw eine Familie von Typtupeln). Jeder in einer Typklasse enthaltene Datentyp wird *Ausprägung* (*engl. instance*) der Typfamilie genannt. Eine besondere Klasse von Typen in Gofer ist die Typklasse `Eq`. Alle in `Eq` enthaltenen Typen zeichnen sich dadurch aus, daß ihre Elemente mittels der Funktion

(==) auf Gleichheit getestet werden können. Diese Funktion (==) hat den Typ

$$(==)::\texttt{a} \longrightarrow \texttt{a} \longrightarrow \texttt{Bool},$$

wobei die beiden Argumente a Typvariablen sind. Die Vereinbarung der Typklasse Eq im *standard prelude* hat folgendes Aussehen:

```
class Eq a where
     (==), (/=) :: a ⟶ a ⟶ Bool
     x /= y       = not (x == y)
```

Diese Typklassen-Vereinbarung ist wie folgt zu lesen:

- In der ersten Zeile wird der Name der Typklasse sowie die verwendeten Typvariablen angegeben.
- In der zweiten Zeile sind die Namen der in der Typklasse Eq verfügbaren Funktionen sowie deren Typ spezifiziert. Diese Funktionen werden als *member functions* der Klasse Eq bezeichnet.
- Die dritte Zeile stellt einen Zusammenhang zwischen den member-Funktionen (==) und (/=) her. Die Semantik dieser Zeile ist, daß die Berechnung eines Ergebnisses der Ungleichheitsfunktion (/=) mit Hilfe der Funktion (==) durch Komplementbildung erzielt wird.

Ein Teil der im standard prelude von Gofer definierten Typen befinden sich in der Klasse Eq, d. h. für diese vordefinierten Typen ist die Funktion (==) zum Testen der Gleichheit bereits vordefiniert. Als Beispiel für mögliche Formen der Definition der Funktion (==) zeige ich einen Ausschnitt aus dem *standard prelude*. In diesem Ausschnitt werden die Funktionen zur Testung der Gleichheit von Zahlen und boolschen Werten definiert.

```
instance Eq Int where (==) = primEqInt

instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

Die beiden Datentypen `Int` und `Bool` werden mittel der obigen Definitionen als Ausprägungen der Typklasse `Eq` vereinbart. Die Funktion primEqInt ist eine primitive Gofer-Funktion. Diese Funktion hat den Typ `Int` $\rightarrow$ `Int` $\rightarrow$ `Bool` und testet zwei Zahlen auf Gleichheit. Die Defintion der Gleichheitsfunktion des Datentyps `Bool` gibt ein Beispiel für die Verwendung von *pattern matching* in Gofer.

Durch geschickte Ausnutzung des Polymorphismus kann die Definition der Funktion `(==)` auf *zusammengesetzte Typen* ausgedehnt werden. Im standard prelude von Gofer wird dies durch die beiden folgenden Funktionsdefinitionen realisiert:

```
instance (Eq a, Eq b) ⇒ Eq (a, b) where
    (x, y) == (u, v) = x == u && y == v

instance Eq a ⇒ Eq [a] where
      [ ]  == [ ] = True
      [ ]  == (y:ys) = False
    (x:xs)== [ ] = False
    (x:xs)== (y:ys) = x == y && xs == ys
```

Die obere der beiden Funktionsdefinitonen berührt den Gleichheitstest bei Tupeln. Voraussetzung für den Test der Gleichheit zweier Tupel gemäß dieser Funktion ist die Vergleichbarkeit der einzelnen Tupelkomponenten. Desweiteren sagt die obere Funktionsdefinition aus, daß für die einzelnen Tupelkomponenten die Funktion `(==)` definiert sein muß.

2.4 Lazy evaluation

Hintergrund des Konzepts der *lazy evaluation* (wörtlich übersetzt, ,,verzögerte" Auswertung) ist, unnötige Berechnungen bei Ausdrucksauswertungen soweit wie möglich zu vermeiden.

Man kann sagen, daß bei ,,*Lazy evaluation*"

1. ein Ausdruck nur dann berechnet wird, wenn sein Wert benötigt wird.

2. ein Ausdruck stets nur einmal berechnet wird; sein Ergebnis wird überall verteilt, wo es gebraucht wird.

Ein funktionales Programm kann in seiner Gesamtheit als ein Ausdruck betrachtet werden, der durch die Programmausführung ausgewertet wird. Die Auswertung eines Ausdrucks entspricht theoretisch gesehen einer wiederholten Reduzierung des Ausdrucks, bis dieser in einer nicht weiter reduzierbaren Form (**Normalform**) vorliegt. Im Rahmen derartiger Reduzierungen kann es bei einer unklugen Auswertungstrategie vorkommen, daß ein identischer Teil in eines reduzierbaren Ausdrucks (**Redex**) wiederholt ausgewertet wird. Diese werde ich im folgenden genauer erläutern.

Wie ich gleich zeigen werde, besitzt ein Ausdruck i. allg. mehrere Redexe. Gegeben seien die folgenden Funktion-Definitionen:

```
double n = n + n
fst (a,b) = a
infinity = 1+infinity
```

Der Ausdruck *double (double 4)* enthält hier zwei Redexe: der gesamte Ausdruck ist ein Redex und der Teilausdruck *double 4* ist ein Redex. Je nachdem welcher Redex reduziert wird, gelangt man zu unterschiedlichen Zwischenergebnissen. Wie bereits früher gesagt ist das Ergebnis einer fortgesetzten Reduktion, die Normalform, eindeutig (Konfluenz oder Church-Rosser-Eigenschaft). Betrachtet sei das folgende Beispiel [BW92].

double$(4+2)$	double$(4+2)$
$\rightarrow$ double 6	$\rightarrow (4+2)+(4+2)$
$\rightarrow 6+6$	$\rightarrow 6+(4+2)$
$\rightarrow 12$	$\rightarrow 6+6$
	$\rightarrow 12$

Der obige Ausdruck kann mittels zweier verschiedener Reduktionsverfahren, der *inneren* bzw. *äußeren* Reduktion, vereinfacht werden. In der linken Reduktionsfolge wird in jedem Schritt jeweils der innerste reduzierbare Ausdruck, d.h. der Ausdruck, der

keinen weiteren reduzierbaren Ausdruck mehr enthält, reduziert.
In der rechten Reduktionsfolge dagegen wird in jedem Schritt jeweils der äußerste reduzierbare Ausdruck, d.h. der Ausdruck, der in keinem anderen reduzierbaren Ausdruck enthalten ist, reduziert. *Die erste Idee der „lazy evaluation“* 1. (siehe Seite 13) kann mit der zuletzt genannten äußeren Reduktion erklärt werden. In diesem Beispiel ist die äußere Reduktion gegenüber einer inneren Reduktion leider nachteilig, da der Term $(4+2)$, der durch die Reduktion von *double* verdoppelt wird, zweimal berechnet werden muß. Die Differenz der erforderlichen Reduktionsanzahl kann beliebig erhöht werden, wenn man den Ausdruck $(4 + 2)$ durch einen Ausdruck ersetzt, für dessen Reduzierung mehr Schritte erforderlich sind.

Im folgenden zeige ich ein anderes Beispiel, bei dem sich die *äußere*-Reduktion als vorteilhaft erweist.

fst (5,double 4)	fst (5,double 4)
→ fst (5,4+4)	→ 5
→ fst (5, 8)	
→ 5	

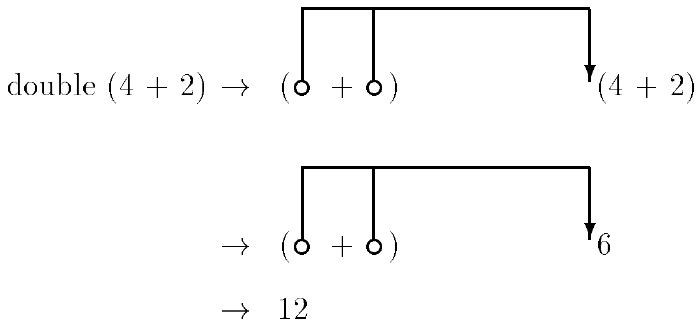
Die Anzahl der Reduktionsschritte bei einer äußereren Reduktion ist im obigen Beispiel viel geringer als bei innereren Reduktion, da der aktuelle Parameter double 4 überhaupt nicht ausgewertet wird. Im folgenden Beispiel [HR92] ist die Differenz der Anzahl der Reduktionsschritte, die beide Verfahrensweisen bei der Auswertung eines Ausdrucks benötigen, noch erheblich drastischer.

fst (5,infinity)	fst (5,infinity)
→ fst (5,1+infinity)	→ 5
→ fst (5,1 + (1+infinity))	
⋮	

Wird beim ersten Beispiel im Falle der äußeren Reduktion der Nachteil der Mehrfachauswertung eines identischen Terms $(4 + 2)$ eliminiert, benötigen innere und äußere Auswertung dieselbe Anzahl an Reduktionsschritten zur Auswertung des Ausdrucks. Das Problem der Mehrfachauswertung kann gelöst werden, indem man Terme durch

Graphen repräsentiert, in denen identische Terme miteinander verknüpft sind. Das folgende Bild gibt ein Beispiel für einen solchen Graph:

Jedes Vorkommen des Subterms (4 + 2) wird durch einen entsprechenden Pfeil auf den Term (4 + 2) angezeigt. Bei Verwendung von Graphen-Reduktion ergibt sich eine Reduktionsfolge, die nur aus drei Schritten besteht.



Die zweite Idee der ,,lazy evaluation" 2. (siehe Siete 14) beruht auf der zuletzt gezeigten Graphen-Reduktion. In einer Graphrepräsentation kann ein Teilausdruck mehrere ,,Vorgänger" besitzen. Wenn dieser Teilausdruck vereinfacht wird, erfahren alle Ausdrücke, die auf den Teilausdruck verweisen, von der Vereinfachung. *Äußere* Reduktion angewendet auf eine Graphrepräsentation bezeichnet man als *äußere*-Graphreduktion (lazy evaluation). Die ,,lazy evaluation" benötigt zur Auswertung eines Ausdrucks höchstens so viele Reduktionsschritte wie die innere Reduktion.

2.5 Variablendeklaration

Eine Variabledeklaration ist die Deklaration einer Konstantenfunktion und besteht meistens aus einer einzigen Gleichung der Form [JM93]:

$$var = right\_hand\_side$$

z. B.: `val` $= sum[1 \cdot\cdot 100]$
`longList` $= [1 \cdot\cdot 10000]$

Durch Evaluierung dieser Variablen in lazy evaluation, können wir zwei wichtige Merkmale betrachten.

Wenn der Wert der Variable `val` erst nach dem Laden der Datei, die die Variable `val` beinhaltet, benötigt wird, wird die Summe der ganzen Zahlen von 1 bis 100 evaluiert und das Ergebnis wird auf die Definition von `val` überschrieben. Diese Berechnung findet nicht mehr statt, solange die Datei nicht neu geladen wird. Ab dem zweiten Gebrauch des Wertes der Variablen `val` wird dadurch die Reduktion für die Berechnung gespart:

? val
5050
(809 reduction, 1120 cells)

? val
5050
(1 reduction, 7 cells)

Zudem können wir an die Überschreibung des evaluierten Wertes der Variablen im Fall von `longList` denken, bei dem der Wert viel Speicherplatz benötigt. Falls das Auswerten der Definition `longList` verlangt wird, wird die Definition durch die vollständige Liste der ganzen Zahlen von 1 bis 10000 nach dem Evaluieren besetzt. Dieser besetzte Speicherplatz kann nicht mehr für andere Berechnungen benutzt werden, ohne die Datei neu zu laden.

Kapitel 3

Terminologische Repräsentation

Eine Wissensbasis, die in einer angemessenen Repräsentationssprache formuliertes Wissen aufnimmt, wird *Wissensrepräsentationssystem* (*engl. knowledge representation system*) genannt. Eine Zielsetzung im Bereich Wissensrepräsentation ist, ein System zu entwickeln, mit dessen Hilfe die Repräsentation und Verarbeitung von Wissen möglich ist.

Eine Art von Wissensrepräsentationssystemen sind die *terminologischen Repräsentationssysteme*. Terminolgische Repräsentationssysteme sind eine Weiterentwicklung des Wissensrepräsentationssystems KL-ONE [BS85]. Die Entwicklung von KL-ONE geht auf R. J. Brachmann und seine Mitarbeitern zurück. Diese haben vor ca. 15 Jahren mit der Entwicklung von KL-ONE begonnen.
Allen auf KL-ONE basierenden Wissensrepräsentationssystemen ist gemeinsam, daß sie aus zwei Komponeneten zusammengesetzt sind. Diese zwei Komponenten sind eine **TBox** und eine **ABox**. Jede dieser beiden Komponeneten hat ihre eigene Repräsentationssprache und ihr eigenes Folgerungssystem (*engl. inference system*) :
Die TBox dient zur terminologischen Repräsentation, sie enthält in einer *terminologischen Sprache* beschriebene Informationen. Eine terminologische Sprache ist ein Repräsentationsschema für ein terminologisches Repräsentationssystem; sie ist eine formale Sprache und besteht aus zwei Elementen. Diese Elemete sind die *syntaktischen Primitive* und die *Operatoren*. Die syntaktischen Primitive setzen sich aus *Konzepten* und *Rollen* zusammen. Ein Operator ist *rollen-konstruierend* oder *konzept-konstruierend*, je nach der Art der durch einen Operator realisierten Verknüpfung.

Die **TBox** enthält die Wissen über Konzepte und Rollen (Klassen von Individuen) sowie über Beziehungen zwischen Konzepten (bzw. Rollen). Die ABox enthält *die behauptenden Repräsentationen* (*engl. assertional representations*), d.h. sie enthält Informationen über die einzelnen Individuen. Durch diese Informationen wird eine konkrete Welt beschrieben (Die Individuen sind hier die Instanzen von Konzepten, d.h. Elemente von Mengen; die Instanzen von Rollen sind Paare von Individuen).

Die verschiedenen verwendeten terminologischen Repräsentationen unterscheiden sich bezüglich der syntaktischen Operatoren. In diesem Kapitel definiere ich eine Terminologische Sprache $\mathcal{ALC}$, die von Schmidt-Schauß und Smolka [SS88] eingeführt wird. Im zweiten Abschnitt dieses Kapitels werde ich die Schreibweise von $\mathcal{ALC}$ unter Zuhilfenahme von Beispielen einführen.

3.1 Terminologische Sprachen

Das *Vokabular* von $\mathcal{ALC}$ besteht aus drei *disjunkten*(*engl. disjoint*) Mengen von Symbolen. Diese Mengen sind:

- Eine Menge von Zeichen für primitive Konzepte

- Eine Menge von Zeichen für primitive Rollen

- Eine Menge von strukturellen Symbolen

Die Menge der Zeichen, die zur Identifikation primitiver Konzepte benutzt werden, enthält stets zwei besonders ausgezeichnete Symbole. Diese beiden Symbole sind '$\top$' und '$\bot$' und werden Konzeptsymbole genannt. Das Symbol '$\top$' wird zur Identifikation des *Topkonzepts*, das Symbol '$\bot$' zur Identifikation des *Bottomkonzepts* (leeres Konzept) benutzt. Die Menge der strukturellen Symbole enthält die 5 Symbole '**and**', '**or**', '**not**', '**some**' und '**all**'. Diese Symbole heißen Operatoren und haben die folgenden Bedeutungen:

- '**and**' : Dieser Operator dient zur Spezifikation der *Konjunktion* zweier Konzepte.

- '**or**' : Dieser Operator dient zur Spezifikation der *Disjunktion* zweier Konzepte.

- '**not**' : Dieser Operator dient zur Spezifikation der *Negation* eines Konzepts.

- '**some**' : Dieser Operator dient zur Spezifikation der *existentiellen Beschränkung* auf einer Rolle und einem Konzept.

- '**all**' : Dieser Operator dient zur Spezifikation *der universellen Beschränkung* auf einer Rolle und einem Konzept.

Die Operatoren '**and**', '**or**', '**not**' sind konzept-konstruierend auf Konzepten und die Operatoren '**some**', '**all**' konzept-konstruierend auf Rollen und Konzepten.

Die Konzeptbeschreibung ist wie folgt definiert [SR91]:
Wenn '**A**' ein beliebiges primitives Konzeptsymbol (Konzeptname) ist, können zwei durch '**C**' und '**D**' bezeichnete Konzeptbeschreibungen (Konzeptterme) durch andere Konzepte nach der folgenden Regel(in Backus Naur Form) konstruiert werden:

$$\mathbf{C}, \mathbf{D} \longrightarrow \mathbf{A}|(\mathbf{and\ C\ D})|(\mathbf{or\ C\ D})|(\mathbf{not\ C}) \tag{3.1}$$

Sei '**Q**' ein primitives Rollensymbol. Erweitert wird die Regel (3.1) durch das Hinzufügen von der existenziellen und universellen Beschränkungen (*engl. existential and engl. universal restriction constructs*):

$$\mathbf{C}, \mathbf{D} \longrightarrow \cdots |(\mathbf{some\ Q\ C})|(\mathbf{all\ Q\ C}). \tag{3.2}$$

Alle möglichen Konzeptbeschreibungen **C** oder **D** werden rekursiv durch die Regeln (3.1) und (3.2) definiert.

Die modell-theoretische Semantik von Konzeptbeschreibungen in $\mathcal{ALC}$ ist durch eine *Interpretation* $\mathcal{I}$ gegeben, die als ein Paar $(\mathcal{D}^{\mathcal{I}}, \cdot^{\mathcal{I}})$ definiert ist. $\mathcal{D}^{\mathcal{I}}$ ist *die Domäene* (*engl. domain*) der Interpretation, und $\cdot^{\mathcal{I}}$ ist *eine Interpretationsfunktion.* Eine Interpretationsfunktion ordnet jeder Konzeptbeschreibung **C** eine Untermenge $\mathbf{C}^{\mathcal{I}}$ von $\mathcal{D}^{\mathcal{I}}$

und jeder Rolle $\mathbf{Q}$ eine binäre Relation $\mathbf{Q}^{\mathcal{I}}$ über der Menge $\mathcal{D}^{\mathcal{I}}$ (d.h.$\mathbf{Q}^{\mathcal{I}} \subseteq \mathcal{D}^{\mathcal{I}} \times \mathcal{D}^{\mathcal{I}}$) zu. Die Interpretationsfunktion spezifiziert die Bedeutung der gekennzeichneten und zusammengesetzten Konzepte wie folgt:

$$\begin{aligned}
\top^{\mathcal{I}} &= \mathcal{D}^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(\textbf{and C D})^{\mathcal{I}} &= \mathbf{C}^{\mathcal{I}} \cap \mathbf{D}^{\mathcal{I}} \\
(\textbf{or C D})^{\mathcal{I}} &= \mathbf{C}^{\mathcal{I}} \cup \mathbf{D}^{\mathcal{I}} \\
(\textbf{not C})^{\mathcal{I}} &= (\mathbf{C}^{\mathcal{I}})' \qquad (= \mathcal{D}^{\mathcal{I}} - \mathbf{C}^{\mathcal{I}}) \\
(\textbf{some Q C})^{\mathcal{I}} &= \{x \mid (\exists y)[(x,y) \in \mathbf{Q}^{\mathcal{I}} \;\&\; y \in \mathbf{C}^{\mathcal{I}}]\} \\
(\textbf{all Q C})^{\mathcal{I}} &= \{x \mid (\forall y)[(x,y) \in \mathbf{Q}^{\mathcal{I}} \Rightarrow y \in \mathbf{C}^{\mathcal{I}}]\}.
\end{aligned} \tag{3.3}$$

Außer diesen Operatorsymbolen gibt es in der Menge der strukturellen Symbole noch die zwei weiteren Symbole '$\sqsubseteq$' und '$\doteq$'. '$\sqsubseteq$' spezifiziert eine *Spezialisation* und '$\doteq$' eine *Äquivalenz*. Die beiden Symbole werden verwendet, um die Beziehungen zwischen Konzeptbeschreibungen zu repräsentieren. Durch die Spezialisation $\sqsubseteq$ wird beschrieben, daß ein Konzept in einem anderen Konzept enthalten ist. Der Ausdruck $\mathbf{C} \sqsubseteq \mathbf{D}$ bezeichnet beispielsweise, daß das Konzept $\mathbf{C}$ dem Konzept $\mathbf{D}$ enthalten ist.
Die Äquivalenz '$\doteq$' wird benutzt, um die wechselseitige Unterordnung zweier Konzepte zu spezifizieren. Der Ausdruck $\mathbf{C} \doteq \mathbf{D}$ beschreibt beispielsweise, daß $\mathbf{C}$ und $\mathbf{D}$ ineinander enthalten sind. Spezialisation- und Äquivalenzbeziehungen (durch $\doteq$ und $\sqsubseteq$ hergestellte Beziehungen zwischen Konzepten) repräsentieren die explizit in einer Wissensbasis(*engl. knowledge base*) enthaltenen Informationen und werden terminologische Axiome genannt. Der Begriff 'terminologische Axiome' wurde gewählt, weil Spezialisation- und Äquivalenzbeziehungen wie Axiome verwendet werden, wenn abgeleitete Informationen ermittelt werden.
Im folgenden definiere ich die Syntax der terminologischen Axiome. Seien 'σ' und 'τ' zwei beliebige terminologische Axiome. Ihr Syntax ist formal wie folgt definiert:

$$\sigma, \tau \longrightarrow \mathbf{C} \sqsubseteq \mathbf{D} \mid \mathbf{C} \doteq \mathbf{D}. \tag{3.4}$$

Eine endliche Menge von terminologischen Axiomen bildet eine *Terminologie* (TBox) und wird mit $\mathcal{T}$ gekennzeichnet.
Es wird davon gesprochen, daß eine Interpretation $\mathcal{I}$ von $\mathcal{ALC}$ ein terminologisches

Axiom σ erfüllt (es wird durch $\models_{\mathcal{I}} \sigma$ bezeichnet), gdw für die Interpretation der Konzepte folgendes gilt:

$$\begin{aligned} &\models_{\mathcal{I}} \mathbf{C} \sqsubseteq \mathbf{D} \quad gdw \quad \mathbf{C}^{\mathcal{I}} \subseteq \mathbf{D}^{\mathcal{I}} \\ &\models_{\mathcal{I}} \mathbf{C} \doteq \mathbf{D} \quad gdw \quad \mathbf{C}^{\mathcal{I}} = \mathbf{D}^{\mathcal{I}}. \end{aligned} \tag{3.5}$$

Eine Interpretation $\mathcal{I}$ ist ein *Modell* für eine Terminologie T ($\models_{\mathcal{I}} T$) , gdw alle terminologische Axiome in T durch $\mathcal{I}$ erfüllt werden. Ein terminologisches Axiom σ ist *Folge* (*engl. σ is the consequence of* T ($T \models \sigma$)) einer Terminologie T , gdw σ durch *alle* Modelle von T erfüllt wird. Ist T eine leere Menge, schreibt man $\models \sigma$ und sagt, daß das Axiom σ *zulässig* (*engl. valid*) ist.

Im folgenden definiere ich die Begriffe *Subsumtion* und *Äquivalenz* bezüglich einer Terminologie T wie folgt:

$$\begin{aligned} &\mathbf{C} \preceq_T \mathbf{D} \quad gdw \quad T \models \mathbf{C} \sqsubseteq \mathbf{D} \\ &\mathbf{C} \approx_T \mathbf{D} \quad gdw \quad T \models \mathbf{C} \doteq \mathbf{D}. \end{aligned} \tag{3.6}$$

Der Ausdruck $\mathbf{C} \preceq_T \mathbf{D}$ bedeutet, daß das Konzept $\mathbf{C}$ von dem Konzept $\mathbf{D}$ in der Terminologie T *subsumiert* ist. Durch den Ausdruck $\mathbf{C} \approx_T \mathbf{D}$ wird beschrieben, daß die Konzepte $\mathbf{C}$ und $\mathbf{D}$ bezüglich der Terminologie T *semantisch äquivalent* sind. Ist die Terminologie leer, so kann man $\mathbf{C} \preceq \mathbf{D}$ und $\mathbf{C} \approx \mathbf{D}$ statt $\mathbf{C} \preceq_{\emptyset} \mathbf{D}$ und $\mathbf{C} \approx_{\emptyset} \mathbf{D}$ schreiben. Ferner gelten folgende Schreibweisen:

$$\begin{aligned} &\mathbf{C} \preceq \mathbf{D} \quad gdw \quad \mathbf{C} \preceq_T \mathbf{D} \ \textit{für alle Terminologie } T \\ &\mathbf{C} \approx \mathbf{D} \quad gdw \quad \mathbf{C} \approx_T \mathbf{D} \ \textit{für alle Terminologie } T. \end{aligned} \tag{3.7}$$

Eine Konzeptbeschreibung $\mathbf{C}$ heißt *inkonsistent* (*engl. inconsistent or incoherent*) bezüglich einer Terminologie T, gdw $\mathbf{C} \approx_T \perp$ gilt; ansonsten heißt Konzeptbeschreibung $\mathbf{C}$ *konsistent* (*consistent or coherent*).

Lemma 3.1 Für eine beliebige Terminologie T ist die Subsumtionsrelation $\preceq_T$ eine reflexive und transitive Relation.

Beweis. Für ein beliebiges Konzept $\mathbf{C}$ bezüglich T gilt:

$\mathbf{C}^{\mathcal{I}} \subseteq \mathbf{C}^{\mathcal{I}}$ für alle gültigen Modelle $\mathcal{I}$ von T.

$\Longrightarrow \models_{\mathcal{I}} \mathbf{C} \sqsubseteq \mathbf{C}$ für alle Modelle $\mathcal{I}$.

$\Longrightarrow T \models \mathbf{C} \sqsubseteq \mathbf{C}$. $\Longrightarrow \mathbf{C} \preceq_T \mathbf{C}$. ($\preceq_T$ ist reflexiv.)

Angenommen, es gilt:

$\mathbf{C} \preceq_T \mathbf{D}$ und $\mathbf{D} \preceq_T \mathbf{E}$ für drei beliebige Konzepte **C**, **D** und **E**.

$\Longrightarrow T \models \mathbf{C} \sqsubseteq \mathbf{D}$ und $T \models \mathbf{D} \sqsubseteq \mathbf{E}$.

$\Longrightarrow \models_\mathcal{I} \mathbf{C} \sqsubseteq \mathbf{D}$ und $\models_\mathcal{I} \mathbf{D} \sqsubseteq \mathbf{E}$ für alle Modelle $\mathcal{I}$ von T.

$\Longrightarrow \mathbf{C}^\mathcal{I} \subseteq \mathbf{D}^\mathcal{I}$ und $\mathbf{D}^\mathcal{I} \subseteq \mathbf{E}^\mathcal{I}$ für alle Modelle $\mathcal{I}$.

$\Longrightarrow \mathbf{C}^\mathcal{I} \subseteq \mathbf{E}^\mathcal{I}$ für alle Modelle, denn $\subseteq$ ist transitiv.

$\Longrightarrow \models_\mathcal{I} \mathbf{C} \sqsubseteq \mathbf{E}$ für alle Modelle $\mathcal{I}$.

$\Longrightarrow T \models \mathbf{C} \sqsubseteq \mathbf{E}$

$\Longrightarrow \mathbf{C} \preceq_T \mathbf{E}$. ($\preceq_T$ ist transitiv.) qed.

Wenn sich zwei Ausdrücke **C** und **D** einander subsumieren ($\mathbf{C} \preceq_T \mathbf{D}$ und $\mathbf{D} \preceq_T \mathbf{C}$), sind sie äquivalent ($\mathbf{C} \approx_T \mathbf{D}$)(Dies kann ähnlich wie obiges Lemma bewiesen werden). Die Terme **C** und **D** müssen nicht notwendigerweise identisch sein. Daher ist $\preceq_T$ keine *antisymmetrische* Relation. Deshalb ist die Relation $\preceq_T$ keine partielle Ordnung (*engl. partial order*). Wie gewöhnlich erhält man durch Bildung des Quotienten von $\preceq_T$ nach $\approx_T$ eine partielle Ordnung ($\preceq_T / \approx_T$). Mit dieser Ordnung erhält man eine *partiell geordnete Menge* ($\mathsf{C}/\approx_T, \preceq_T / \approx_T$) bestehend aus *Äquvalenzklassen* von Konzepten, wobei C die Menge aller Konzepte bezeichnet. In dieser partiell geordneten Menge sind die äquivalenten Konzepte nicht unterscheidbar, weil sie mit einer bestimmten Äquivalenzklasse verbunden sind.

Sei beispielsweise (**and A B**) $\approx_T \perp$ für zwei beliebige Konzepte **A** und **B**. (Das heißt '(**and A B**) $\doteq \perp$' wird durch alle Modelle von T erfüllt.) Dann gilt (**and A B**) $\preceq_T$ (**not** $\top$) und (**not** $\top$) $\preceq_T$ (**and A B**). Die beiden Terme (**and A B**) und (**not** $\top$) sind nicht identisch, in der Ordnung ($\preceq_T / \approx_T$) jedoch äquivalent, d. h. sie sind in der gleichen Äquivalenzklasse enthalten.

Nebel und Smolka [NS89] haben die im vorletzten Absatz aufgeführte partiell geordnete Menge, die eine hierarchische Struktur bildet, mit dem Namen **Konzepttaxonomie** in der Terminologie T versehen. Damit ist die terminologische Sprache $\mathcal{ALC}$ vollständig definiert.

3.2 Beispiele für terminologische Repräsentationen

In diesem Abschnitt zeige ich anhand von Beispielen, wie die Operatoren in $\mathcal{ALC}$ verwendet werden, um zusammengesetzte Konzepte zu beschreiben.

Im folgenden wird mittels der terminologischen Sprache $\mathcal{ALC}$ Wissen über eine Umwelt von Personen repräsentiert.
'person', 'weiblich', 'männlich', 'student', 'frauenparteimitglied' und 'vegetarier' seien primitive Konzepte und 'hat_Kind' und 'ist_Kusin(e)' seien primitive Rollen. 'person' sei das Top-Konzept. Konzepte werden als Menge interpretiert und Rollen als binäre Relationen.

Wie ich bei der Definition von $\mathcal{ALC}$ aufgeführt habe, gibt es folgende zwei Arten von Operatoren: *konzept-konstruierende auf Konzepten* und *konzept-konstruierende auf Rollen* und *Konzepten*. Die konzept-konstruierenden Operatoren auf Konzepten sind '**and**', '**or**' und '**not**'. Mit diesen Operatoren kann man die Menge der vegetarischen Studenten durch den Ausdruck

(**and student vegetarier**)

erzeugen (ein neues Konzept (**and student vegetarier**) wird erzeugt). Die Menge aller männlichen oder weiblichen Personen (d.h. die Vereinigung) kann durch

(**or männlich weiblich**),

die Menge aller nicht vegetarischen Individuen durch

(**not vegetarier**)

repräsentiert werden. Die konzept-konstruierenden Operatoren auf Rollen und Konzepten sind '**some**' und '**all**'. Der folgende Ausdruck

(**some ist_Kusin(e) student**)

wird als die Menge aller Personen, deren Kusine oder Kusin ein Student ist, interpretiert. Genauer gesagt haben alle Personen in dieser Menge mindestens eine Kusine (einen Kusin), die (der) Studentin (Student) ist. Die Menge aller Personen, deren Kinder sämtlich Studenten sind, wird als

(all hat_Kind Student)

repräsentiert.

Neue Konzepte werden mittels der in den Beispielen aufgeführten Art und Weise erzeugt. Die bestehenden Möglichkeiten zur Bildung eines neuen Konzepts sind:

- die Vereinigung bzw. Durchschnittsbildung zweier Konzepte,
- die Bildung des Komplements eines Konzepts,
- die Anwendung der Operatoren '**some**' und '**all**' auf eine Rolle und ein Konzept.

Die Subsumtionsbeziehungen (*engl. subsumption relationships*) zwischen Konzepten werden, wie im vorherigen Abschnitt gesagt, durch das Symbol '$\sqsubseteq$' repräsentiert. Die gegenseitige Unterordnung wird Äquivalenz genannt; für sie wird das Symbol '$\doteq$' benutzt. Im folgenden stelle ich verschiedene explizite Informationen über Konzepte mit Hilfe von Subsumtions- und Äquivalenzbeziehungen dar, die in der terminologischen Sprache $\mathcal{ALC}$ formuliert sind:

$$\textbf{männlich} \sqsubseteq \textbf{person} \tag{3.8}$$

$$\textbf{weiblich} \sqsubseteq \textbf{person} \tag{3.9}$$

$$\textbf{person} \doteq (\textbf{or männlich weiblich}) \tag{3.10}$$

$$\textbf{student} \sqsubseteq \textbf{person} \tag{3.11}$$

$$\textbf{vegetarier} \sqsubseteq \textbf{person} \tag{3.12}$$

$$\textbf{frauenparteimitglied} \sqsubseteq \textbf{weiblich} \tag{3.13}$$

$$(\textbf{and männlich weiblich}) \doteq \bot \tag{3.14}$$

$$\vdots$$

Die oben aufgeführten Ausdrücke sind terminologische Axiome und enthalten explizite

Informationen. Diese Menge von terminologischen Axiomen ist ein Beispiel für eine Terminologie T (TBox). Diese expliziten Informationen kann man als Grundlage zur Herleitung impliziter Informationen benutzen.

Das Symbol für die Äquivalenz kann dafür benutzt werden, ein neues benanntes Konzept zu erzeugen. Soll beispielsweise das Bottom-konzept den Namen '**Niemand**' erhalten, so kann dies mit dem Ausdruck

$$\textbf{Niemand} \doteq (\textbf{not person}) \tag{3.15}$$

erzielt werden.

Die Relationen $\preceq_T$ und $\approx_T$ werden, wie im letzten Abschnitt gesagt, zwischen zwei Konzepten im Bezug auf Modelle der Terminologie T definiert. Beispielsweise gilt die folgende Beziehung :
frauenparteimitglied $\preceq_T$ **person** Denn es gilt: **frauenparteimitglied** $\preceq_T$ **person** gdw

$$T \models \textbf{frauenparteimitglied} \sqsubseteq \textbf{person}.$$

(**frauenparteimitglied** $\sqsubseteq$ **person** wird durch alle Modelle von der Terminologie T erfüllt.)

Ferner gelten die folgenden Beziehungen:
(**and Vegetarier Weiblich**) $\preceq$ **Weiblich**
(Begründung: Die Beziehung **(and Vegetarier Weiblich)** $\sqsubseteq$ **Weiblich** gilt für alle Modelle und für alle Terminologien, in denen die Konzepte 'Vegetarier' und 'Weiblich' enthalten sind).
(not (not Student)) $\approx$ **Student**
(Begründung: Die Beziehung **(not (not Student))** $\doteq$ **Student** gilt für alle Modelle und für alle Terminologien, in denen das Konzept 'Student' enthalten ist).

Kapitel 4

Algebraische Terminologische Repräsentation

In diesem Kapitel gebe ich einen algebraischen Überblick über Mengen und Relationen. Ich betrachte die algebraischen Strukturen *Boolesche Algebren, Relationenalgebren, Boolesche Module* und *Peirce-Algebren*. Weiterhin wird die terminologische Sprache $\mathcal{ALC}$ zur terminologischen Sprache $\mathcal{ALCX}$ erweitert. Ich zeige, daß die modelltheoretische Semantik der definierten terminologischen Sprache $\mathcal{ALCX}$ in einen algebraischen Rahmen paßt, d. h. jeder terminologischer Operator in $\mathcal{ALCX}$ algebraisch ausgedrückt werden kann. Damit erhält man eine algebraische Semantik für die Sprache $\mathcal{ALCX}$. Dies ermöglicht, daß die in den oben erwähnten Algebren erfüllten Gleichungsaxiome und daraus geschlußfolgerten Gleichungen als Simplifikationsmittel zum Optimieren von Anfragen an eine Wissensbasis in der terminologischen Sprache $\mathcal{ALCX}$ verwendet werden können.

4.1 Algebren von Mengen und Relationen

Terminologische Repräsentationssprachen können algebraisch interpretiert werden [SR91, BC92, BBS92]. Die Algebren, die benutzt werden, um terminologischen Sprachen eine algebraische Semantik zu verleihen, sind Boolesche Algebren, Relationenalgebren [TA41], Boolesche Module [BC81] und Peirce-Algebren [BK88]. Diese Algebren werden durch unterschiedliche Operationen auf Mengen und Relationen beschrieben, wobei Gleichungsaxiome und daraus geschlußfolgerte Gleichungen erfüllt werden. Jede in diesen Algebren erfüllte Gleichung besteht aus zwei äquivalenten Termen, wobei ein Term durch einen anderen, äquivalenten Term ersetzt werden kann.

(4.1)**Definition** Eine (*homogene*) *Algebra* $\mathcal{A}$ ist ein geordnetes Paar $(A,\ F)$ mit einer beliebigen Menge $A \neq \emptyset$ und einer Menge F von *endlichstelligen* Operationen auf A. Eine n-stellige (oder endlichstellige) Operation auf A ist eine beliebige Funktion f von A^n nach A. Wenn die Menge F endlich ist, also $F = \{f_0, \cdots, f_{m-1}\}$, dann wird die Algebra $(A,\ F)$ durch $(A,\ f_0, \cdots, f_{m-1})$ bezeichnet. A wird die *Trägermenge* der Algebra genannt und es wird vorausgesetzt, daß A unter jeder Operation in F abgeschlossen ist. Im allgemeinen braucht F nicht notwendig endlich zu sein und kann auch leer sein. Die Operationen in F werden die *Fundamentaloperationen* der Algebra (*engl. fundamental operations of the algebra*) genannt. Die Menge F der Operationen und ihre Stellenanzahl bestimmen den *Typ* (oder die *Ähnlichkeit*) der Algebra.

Im folgenden beschränke ich mich auf Algebren mit höchstens zweistelligen Operationen. Nullstellige Operationen bezeichnet man auch als *Konstanten*.

4.1.1 Boolesche Algebren

In diesem Abschnitt wird der Kalkül des Schlußfolgerns algebraisch auf Mengen beschrieben. Die klassische Darstellung der Gleichungsaxiome und daraus gefolgerten Gleichungen über Mengen gibt den Begriff der *Booleschen Algebra* an.

(4.2) **Definition** Eine *Boolesche Algebra* ist eine Algebra $\mathcal{B} = (B, +, \cdot, ', 0, 1)$ mit der Trägermenge B, in der für jedes $a, b, c \in B$ die folgenden Gleichungsaxiome gelten:
B1 $a + a = a, \qquad a \cdot a = a$
B2 $a + b = b + a, \qquad a \cdot b = b \cdot a$
B3 $a + (b + c) = (a + b) + c, \qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c$
B4 $a \cdot (a + b) = a, \qquad a + a \cdot b = a$
B5 $a + b \cdot c = (a + b) \cdot (a + c), \qquad a \cdot (b + c) = a \cdot b + a \cdot c$
B6 $a + 0 = a, \qquad a \cdot 1 = a$
B7 $a + a' = 1, \qquad a \cdot a' = 0.$

Die zwei binären Operationen $+$ und $\cdot$ werden als *Vereinigung* (*engl. join*) und *Durchschnitt* (*engl. meet*), die einstellige Operation $'$ als *Komplement* (*engl. complement*) und die Konstanten 0 und 1 als *Null* (*engl. zero*) und *Einheit* (*engl. unit*) bezeichnet. Die Reihenfolge der Operationen bzgl. ihrer Bindung ist $'$, $\cdot$ und $+$.

Diese Axiomatisierung der Booleschen Algebra ist weder unabhängig noch minimal. Es existieren auch andere Axiomatisierungen zur Beschreibung der Booleschen Algebren (Huntington [HE04, HE33]). Daß die obige Beschreibung der Booleschen Algebra nicht minimal ist, kann daran abgelesen werden, daß beispielsweise die Axiome B2, B3 und B4 das Axiom B1 implizieren. Eine Boolesche Algebra kann alternativ als ein komplementärer und distributiver Verband (*a complemented and distributive lattice*) $(B, \leq)$ definiert werden, wobei $\leq$ eine auf B definierte partiell geordnete Ordnung ist. (Ein Verband ist eine partiell geordnete Menge, wobei jedes Paar von Elementen eine kleinste obere Schranke (eine Vereinigung) und eine größte untere Schranke (ein Durchschnitt) besitzt.) Dies wird im folgenden gezeigt:

(4.3) **Definition** Ein Verband ist eine Algebra $\mathcal{G} = (G, +, \cdot)$ mit der Trägermenge G, in der für jedes $a, b, c \in G$ die folgenden Gleichungsaxiome gelten:

G1 $\quad a + a = a, \qquad a \cdot a = a$

G2 $\quad a + b = b + a, \qquad a \cdot b = b \cdot a$

G3 $\quad a + (b + c) = (a + b) + c, \qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c$

G4 $\quad a \cdot (a + b) = a, \qquad a + a \cdot b = a$

(4.4) **Definition** Definiere $a \leq b$ *gdw* $a + b = b$ (oder $a \cdot b = a$).

(4.5) **Satz** Die Relation $a \leq b$ ist in einem beliebigen Verband reflexiv, antisymmetrisch und transitiv.

Beweis i) Reflexivität

Es gilt $a \leq a$, da $a \cdot a = a$

ii) Antisymmetritát

Wenn $a \leq b$ und $b \leq a$,

dann gilt $a = a \cdot b = b \cdot a = b$

iii) Transitivität

Wenn $a \leq b$ und $b \leq c$,

dann gilt $a = a \cdot b = a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot c$

$\Rightarrow a \leq c$

qed.

(4.6) **Definition** Wenn ein Verband das Gleichungsaxiom B5 erfüllt, wird es ein *distributiver Verband* genannt.

(4.7) **Definition** *Ein komplementärer Verband* ist ein Verband mit 0 und 1, in dem jedes Element a wenigstens ein Komplement besitzt.

(4.8) **Satz** In einem beliebigen distributiven und komplementären Verband sind die Komplemente aller Elemente *eindeutig* bestimmt.

Beweis a' und a'' seien zwei Komplemente zu für a.

$a' = a' \cdot 1 = a' \cdot (a + a'') = a' \cdot a + a' \cdot a'' = a' \cdot a''$

also $a' = a' \cdot a''$

$a'' = a'' \cdot 1 = a'' \cdot (a + a') = a'' \cdot a + a'' \cdot a' = a'' \cdot a'$

also $a'' = a'' \cdot a'$

$a'' = a'' \cdot a' = a' \cdot a'' = a'$

qed.

Es gibt elementare arithmetische Eigenschaften von Booleschen Algebren, die aus den Axiomen folgen. Im folgenden führe ich einige davon auf, die ich später zum Simplifizieren verwenden werde.

(4.9) **Satz** In einer beliebigen Booleschen Algebra gelten die folgenden Eigenschaften:

B8 $a + 1 = 1, \qquad a \cdot 0 = 0$
B9 $a'' = a$
B10 $(a + b)' = a' \cdot b', \qquad (a \cdot b)' = a' + b'$
B11 $0' = 1, \qquad 1' = 0.$

Beweis B8: $a + 1 = a + a + a' = a + a' = 1$

$a \cdot 0 = a \cdot a \cdot a' = a \cdot a' = 0$ qed.

Beweis B9: $a = a \cdot 1 = a \cdot (a' + a'') = a \cdot a' + a \cdot a'' = a \cdot a''$

$a'' = a'' \cdot 1 = a'' \cdot (a' + a) = a'' \cdot a' + a'' \cdot a = a'' \cdot a$

Daher $a'' = a$ qed.

Beweis B10: i) $a + b + (a+b)' = 1$

$a' \cdot b' = a' \cdot b' \cdot (a + b + (a+b)') = 0 + 0 + a' \cdot b' \cdot (a+b)'$

$\Rightarrow a' \cdot b' \leq (a+b)'$

ii) $(a + b + a' \cdot b') = (a + b + a' \cdot b') \cdot (a + a')$

$= a + a \cdot b + 0 + 0 + b \cdot a' + b' \cdot a'$

$= a + (b + b') \cdot a' = a + a' = 1$

also $a + b + a' \cdot b' = 1$

iii) $(\mathrm{a+b})' = (a+b)' \cdot 1 = (a+b)' \cdot (a + b + a' \cdot b')$

$= (a+b)' \cdot (a + b) + (a+b)' \cdot a' \cdot b'$

$\Rightarrow (a+b)' \leq a' \cdot b'$

Daher $(a+b)' = a' \cdot b'$

Aus $(a+b)' = a' \cdot b'$ folgt bereits $(a \cdot b)' = a' + b'$ (Dualitätsprinzip)

qed.

Beweis B11: $0' = (a \cdot a')' = a' + a'' = a' + a = 1$

$1' = (a + a')' = a' \cdot a'' = a' \cdot a = 0$ qed.

Das Musterbeispiel einer Booleschen Algebra ist eine *volle Boolesche Algebra* (*engl. a full Boolean Algebra*) über eine nicht-leere Menge U $\mathcal{B}(U) = (\mathbf{2}^U, \cup, \cap, ', \emptyset, U)$. Die Menge U heißt das *Universum* der Boolschen Algebra. $\mathbf{2}^U$ bezeichnet die Menge aller Untermengen von U (Potenzmenge von U) und ist durch $\subseteq$ partiell geordnet.

4.1.2 Relationenalgebren

In diesem Abschnitt betrachte ich die Algebra *binärer Relationen*. Eine binäre Relation über einer beliebigen nicht-leeren Menge U ist eine Untermenge des Kartesischen Produktes U^2 ($U^2 = U \times U = \{(x,y) \mid x \in U \;\&\; y \in U\}$). Die Menge U heißt das Universum der binären Relation. Binäre Relationen werden im folgenden durch *R, S, T, ...* bezeichnet. Neue Relationen können durch Benutzung der mengentheoretischen (oder Booleschen) Operationen Vereinigung, Durchschnitt und Komplement gebildet werden. Dazu werden neue Relationen mit *relationalen Operationen* und *Konstanten* wie folgt konstruiert:

(4.10) *Komposition*: $R \,;\, S = \{\,(x,y) \mid (\exists z)[(x,z) \in R \,\&\, (z,y) \in S]\}$
(4.11) *Konversion*: $R^{\smile} = \{\,(x,y) \mid (y,x) \in R\}$
(4.12) *Identität*: $Id = \{\,(x,x) \mid x \in U\}$.

Die Komposition wird als das *relationale Produkt* und die Identitätsrelation als die *diagonale Relation* bezeichnet. Wenn beispielsweise R die Relation '**hat_Kind**' und S die Relation '**ist_verheiratet**' sind, ist $R \,;\, S$ die Relation '**ist-Schwiegereltern**. $R^{\smile}$ ist die Relation '**ist-Kind-von**'. Genauso wie die Booleschen Operationen durch Gleichungsgesetze dargestellt werden, werden auch die relationalen Operationen und Konstanten durch Gleichungsgesetze dargestellt. Zum Beispiel:

(4.13) Die Komposition ist assoziativ: $(R \,;\, S) \,;\, T = R \,;\, (S \,;\, T)$
(4.14) Die Konversion ist eine Involution: $R^{\smile\smile} = R$
(4.15) Die Konversion ist distributiv über ';' und kehrt die Ordnung um:

$$(R;S)^{\smile} = S^{\smile} \,;\, R^{\smile}$$

(4.16) Id ist die Identität von U^2 bezüglich Komposition: $R \,;\, Id = R = Id \,;\, R$.

Tarski [TA41] trennte den Kalkül der Relationen von der elementaren Theorie der Relationen und hat eine Menge von Gleichungen als Axiome für die Beschreibung des Kalküls der Relationen vorgeschlagen. Im folgenden wird die Definition von Chin und Tarski [CT51] aufgeführt.

(4.17) **Definition** Eine *Relationenalgebra* ist eine Algebra $\mathcal{R} = (R, +, \cdot, ', 0, 1, ;, \smile, e)$, in der für jede Relation $r, s, t \in R$ die folgenden Axiome erfüllt sind:

R1 $(R, +, \cdot, ', 0, 1)$ ist eine Boolesche Algebra
R2 $r;(s;t) = (r;s);t$
R3 $r;e = r = e;r$
R4 $r^{\smile\smile} = r$

R5 $(r+s);t = r;t + s;t$
R6 $(r+s)^{\smile} = r^{\smile} + s^{\smile}$
R7 $(r;s)^{\smile} = s^{\smile};r^{\smile}$
R8 $r^{\smile};(r;s)' + s' = s'$.

Die Reihenfolge der Operationen bezüglich ihrer Bindung ist $\smile, \prime, ;, \cdot$, und $+$. Wie in Booleschen Algebren bezeichnet man $+, \cdot, \prime, 0, 1$ als Vereinigung, Durchschnitt, Komplement, Null und Einheit. Die Operationen ; und $\smile$ sind als *relationales Produkt* (oder *Komposition*) und *Konversion* bekannt. Das gekennzeichnete Element e in R wird das *Identitätselement* genannt. (Sowohl eine binäre Relation als auch die Trägermenge einer Relationenalgebra $\mathcal{R}$ wird mit R bezeichnet. Jedoch kann man beide im Kontext deutlich unterscheiden.)

Ein Beispiel einer Relationenalgebra ist die *volle* Relationenalgebra. Die volle Relationenalgebra über einem beliebigen nicht-leeren Universum U wird durch $(2^{U^2}, \cup, \cap, \prime, \emptyset, U^2, ;, \smile, Id)$ definiert. Die Operationen ; und $\smile$ sind die relationentheoretischen Operationen *Komposition* und *Konversion*, die in (4.10) und (4.11) definiert sind. Diese Algebra wird $\mathcal{R}(U)$ bezeichnet.

Chin und Tarski [CT51] haben sich bisher am umfassendsten mit der Arithmetik von Relationenalgebren beschäftigt. (Gleichungsmäßige Schlußfolgerung in Algebren ist als Arithmetik von Algebren bekannt, um es von der universellen-algebraischen Untersuchung von Algebren zu unterscheiden.) Im folgenden Satz führe ich einige Eigenschaften der Arithmetik von Relationenalgebren auf, die ich später zur Simplifikation verwenden werde.

(4.18) **Satz** (Chin und Tarski [CT51]) In einer beliebigen Relationenalgebra $\mathcal{R}$ sind für jede Relation $r, s, t \in R$ die folgenden Eigenschaften erfüllt:

R9 $(r \cdot s)^{\smile} = r^{\smile} \cdot s^{\smile}$
R10 $r;(s+t) = r;s + r;t$

Um diesen Satz zu beweisen, wird zuvor folgender Satz bewiesen:

(4.19) **Satz** In einer beliebigen Relationenalgebra $\mathcal{R}$ ist für beliebige Relationen

$r, s \in R$ die folgende Eigenschaft erfüllt (Die Bezeichnung $\leq$ ist die auf Seite 29 definierte Bezeichnung):

$$r \leq s \text{ und } r^{\smile} \leq s^{\smile} \text{ sind äquivalent.}$$

Beweis.
$$
\begin{aligned}
r \leq s \quad &\Rightarrow \quad r + s = s \\
&\Rightarrow \quad (r+s)^{\smile} = s^{\smile} \\
&\Rightarrow \quad r^{\smile} + s^{\smile} = s^{\smile} \text{ (durch Anwendung von R6)} \\
&\Rightarrow \quad r^{\smile} \leq s^{\smile}. \\
r^{\smile} \leq s^{\smile} &\Rightarrow \quad r^{\smile} + s^{\smile} = s^{\smile} \\
&\Rightarrow \quad (r^{\smile} + s^{\smile})^{\smile} = (s^{\smile})^{\smile} \\
&\Rightarrow \quad r + s = s \text{ (durch Anwendung von R6 und R4)} \\
&\Rightarrow \quad r \leq s.
\end{aligned}
$$

qed.

Beweis R9: Aus Satz (4.19) folgt, daß

$r \cdot s \leq r, \qquad r \cdot s \leq s \Rightarrow (r \cdot s)^{\smile} \leq r^{\smile}$ und $(r \cdot s)^{\smile} \leq s^{\smile}$
$\Rightarrow \quad$ i) $(r \cdot s)^{\smile} \leq r^{\smile} \cdot s^{\smile}$

gilt. Wegen Satz (4.19) und R4 gilt desweiteren $(r^{\smile} \cdot s^{\smile})^{\smile} \leq r \cdot s$. Durch wiederholte Anwendung von 4.19 auf letztere Ungleichung erhält man

$$\text{ii) } r^{\smile} \cdot s^{\smile} \leq (r \cdot s)^{\smile}.$$

Aus i) und ii) folgt $(r \cdot s)^{\smile} = r^{\smile} \cdot s^{\smile}$.

qed.

Beweis R10: Es gilt folgende Form von R5

$$((r^{\smile} + s^{\smile}); t^{\smile})^{\smile} = ((r^{\smile}; t^{\smile}) + (s^{\smile}; t^{\smile}))^{\smile}.$$

$\Rightarrow t \; ; (r^{\smile} + s^{\smile})^{\smile} = (r^{\smile}; t^{\smile})^{\smile} + (s^{\smile} + t^{\smile})^{\smile}$ (durch Anwendung von R7 und R6)
$\Rightarrow t \; ; (r + s) = (t \; ; r) + (t \; ; s)$ (durch Anwendung von R6 und R4)

qed.

4.1.3 Boolesche Module

Um Gleichungsgesetze mit Konzepten und Rollen, die sich gegenseitig beeinflussen, zu modellieren, betrachte ich nun eine algebraische Beschreibung von Mengen, auf denen Relationen wirken. Außer den mengen-konstruierenden Operationen auf Mengen, wie Vereinigung, Durchschnitt und Komplement (im Abschnitt 4.1.1), gibt es mengen-konstruierende Operationen auf Relationen. Die folgenden Definitionen sind Beispiele solcher Operationen:

(4.20) *Domain*: $\mathrm{dom}(R) = \{x \mid (\exists y)[(x, y) \in R]\}$
(4.21) *Range*: $\mathrm{ran}(R) = \{y \mid (\exists x)[(x, y) \in R]\}$

Die anderen Operationen verbinden eine Relation und eine Menge, um eine Menge zu erzeugen. Zum Beispiel:

(4.22) *Peirce Produkt*: $R : A = \{x \mid (\exists y)[(x, y) \in R \;\&\; y \in A]\}$
(4.23) *Image*: $\mathrm{ima}(R) = \{y \mid (\exists x)[(x, y) \in R \;\&\; x \in A]\}$.

Für meine Anwendung ist das Peirce-Produkt die am besten geeignete Operation. Der terminologische Operator **some** stimmt mit dem Peirce-Produkt überein. Die anderen obigen Operationen lassen sich durch das Peirce-Produkt ausdrücken:

(4.24) $\mathrm{dom}(R) = R : U$
(4.25) $\mathrm{ran}(R) = R^{\smile} : U$
(4.26) $\mathrm{ima}(R) = R^{\smile} : A$

Das Peirce-Produkt $R : A$ ist die Menge aller Elemente, die in R mit einem Element aus A in Beziehung stehen. Wenn R beispielsweise die Relation '**hat_Kind**' ist und A die Menge '**Student**' ist, dann ist $R : A$ die Menge aller Eltern, die mindestens ein Kind haben, das Student ist.

Gegenstand dieses Abschnitts sind die Gleichungsgesetze, die durch das Peirce Produkt erfüllt werden:

(4.27) *Id* ist eine Identität des Peirce Produktes: $Id : A = A$
(4.28) Das Peirce-Produkt ist distributiv über die Vereinigung:
$$R : (A \cup B) = R : A \cup R : B$$
(4.29) Das Peirce-Produkt ist schwach assoziativ: $R : (S : A) = (R \,;\, S) : A$.

Boolesche Algebren wurden eingeführt, um Kalküle über Mengen zu formalisieren; Relationenalgebren wurden eingeführt, um Kalküle über Relationen zu formalisieren. Brink [BC88] versuchte die Kalküle mit der Operation Peirce Produkt zu formalisieren, indem er *Boolesche Module* einführte. Folglich ist ein Boolescher Modul als eine zwei-sortige Algebra definiert. Eine zwei-sortige Algebra hat zwei Trägermengen mit den fundamentalen Operationen, die auf jeder Trägermenge jeweils definiert sind, und zusätzlichen fundamentalen Operationen, die auf den Elementen in beiden Trägermengen definiert sind. Für eine formale Definition von mehr-sortigen oder heterogenen Algebren siehe Ehrig und Mahr [EM85] sowie Manes und Arbib [MA86].

(4.30) **Definition** (Brink [BC88]) Ein *Boolescher Modul* ist eine zwei-sortige Algebra $\mathcal{M} = (\mathcal{B}, \mathcal{R}, :)$, wobei $\mathcal{B} = (B, +, \cdot, ', 0, 1)$ eine Boolesche Algebra ist, $\mathcal{R} = (R, +, \cdot, ',$ $0, 1, ;, \smile, e)$ eine Relationenalgebra ist und : eine Abbildung $\mathcal{R} \times \mathcal{B} \longrightarrow \mathcal{B}$, nämlich das Peirce-Produkt, ist, so daß für beliebige $r, s \in R$ und $a, b \in B$ gilt :

M1 $\quad r : (a + b) = r : a + r : b$

M2 $\quad (r + s) : a = r : a + s : a$

M3 $\quad r : (s : a) = (r; s) : a$

M4 $\quad e : a = a$

M5 $\quad 0 : a = 0$

M6 $\quad r^{\smile} : (r : a)' \leq a'$.

Die Reihenfolge der Operationen bzgl. ihrer Bindung ist $'$, $\smile$, :, ;, $\cdot$ und $+$. Die Operationen ($+$, $\cdot$ und $'$) und die Konstanten (0 und 1) in der Booleschen Algebra $\mathcal{B}$ werden genauso bezeichnet wie die Operationen der Booleschen Algebra, die der Relationenalgebra zugrundeliegt. Jedoch kann man sie im Kontext unterscheiden.

Wenn $\mathcal{B}(U)$ eine volle Boolesche Algebra und $\mathcal{R}(U)$ eine volle Relationenalgebra über irgendeiner nicht-leeren Menge U ist, dann ist $\mathcal{M}(U) = (\mathcal{B}(U), \mathcal{R}(U), :)$ ein Beispiel eines Booleschen Modules. Man nennt $\mathcal{M}(U)$ das *volle Boolesche Modul* über U.

Das Peirce-Produkt ist, wie in (4.22) bereits gezeigt, eine natürliche algebraische Fassung des terminologischen Operators **some**. Um eine natürliche algebraische Darstellung des terminologischen Operators **all** zu finden, wendet man das Peirce-Produkt zusammen mit dem Komplement an $(r : a')'$. In einem vollen Booleschen Modul $\mathcal{M}(U)$

wird dies wie folgt interpretiert:

(4.31) **Satz** (Schmidt [SR91]) Gegeben sei eine beliebige binäre Relation R über eine Menge U mit $U \neq \emptyset$ und eine beliebige Teilmenge A ($A \subseteq U$). Es gilt:

$$(R : A')' = \{x \mid (\forall y)[(x, y) \in R \Rightarrow y \in A]\}$$

Beweis. Wegen der Definition des Peirce Produktes (4.20) und der standardmäßigen Gesetze der Prädikatenlogik erster Stufe gilt:
$x \in (R : A')'$ gdw $\neg$ $(\exists\ y)$ $[(x,\ y) \in R\ \&\ y \notin A]$ gdw $(\forall y)[(x,\ y) \notin R$ or $y \in A]$ gdw $(\forall y)[(x,\ y) \in R \Rightarrow y \in A]$.

Wenn die Domäne von R das ganze Universum U umfaßt, dann gilt:
$x \in (R : A')'$ genau dann wenn alle Elemente y, zu denen x in der Beziehung R steht, aus A sind. D. h. x steht nur zu Elementen aus A in der Beziehung R.
Angenommen, R ist die Relation '**hat_Kind**' und A ist die Menge aller Studenten. Dann ist $(R : A')'$ die Menge aller Eltern, deren Kinder sämtlich Studenten sind.

4.1.4 Peirce-Algebren

Im vorherigen Abschnitt wurden Boolesche Module als Algebren dargestellt, die einen Kalkül auf Mengen bzw. Relationen beschreiben. Bei dem Kalkül wurden Mengen mit Relationen verknüpft, um neue Mengen zu bilden. Es gibt terminologische Operatoren, die auf Rollen und Konzepten gebildet werden, um Rollen zu konstruieren. '**restrict**' ist ein Beispiel für einen solchen terminologischen Operator. Wegen derartiger Operatoren sind wir an einer Algebra interessiert, in der Mengen und Relationen aufeinander wirken, um neue Relationen zu bilden. Eine solche Algebra soll *relation-konstruierende Operationen* auf Relationen und Mengen formalisieren. Zum Beispiel:

(4.32) *Bereichsbeschränkung*: $R \lceil A = \{(x, y) \mid (x, y) \in R\ \&\ x \in A\}$
(4.33) *Zielbereichsbeschränkung*: $R \rfloor A = \{(x, y) \mid (x, y) \in R\ \&\ y \in A\}$
(4.34) *Kartesisches Produkt*: $A \times B = \{(x, y) \mid x \in A\ \&\ y \in B\}$
(4.35) *Rechte Zylindrifikation*: ${}^{c}A = \{(x, y) \mid y \in A\}$
(4.36) *Linke Zylindrifikation*: $A^{c} = \{(x, y) \mid x \in A\}$.

Diese Operationen sind zyklisch definierbar. Man kann zum Beispiel die Operationen von (4.32) bis (4.35) mit linker Zylindrifikation wie folgt definieren:

(4.37) $R \lceil A = R \cap A^c, \qquad A^c = U^2 \lceil A$
(4.38) $R \rfloor A = R \cap A^{c\smile}, \qquad A^c = (U^2 \rfloor A)^{\smile}$
(4.39) $A \times B = A^c \cap B^{c\smile}, \qquad A^c = A \times U$
(4.40) ${}^cA = A^{c\smile}, \qquad A^c = ({}^cA)^{\smile}$.

In diesem Abschnitt werden die Booleschen Module erweitert, um Relationen-konstruierende Operationen auf Relationen und Mengen zu formalisieren. Die resultierenden Algebren heißen *Peirce Algebren.* Weil die Operationen (4.32) bis (4.36) zyklisch definierbar sind, genügt es eine der Operationen in Peirce-Algebren zu formalisieren.
Eine Peirce-Algebra ist notwendigerweise ein Boolescher Modul $(\mathcal{B}, \mathcal{R}, :)$. Letzterem wird eine zusätzliche Operation hinzugefügt. Diese Operation ist das algebraische Gegenstück für die linke Zylindrifikation.

(4.41) **Definition**(Britz [BK88]) Sei $\mathcal{B}$ eine Boolesche Algebra und $\mathcal{R}$ eine Relationenalgebra. Eine Peirce-Algebra ist eine zwei-sortige Algebra $\mathcal{P} = (\mathcal{B}, \mathcal{R}, :, {}^c)$ mit einem Booleschen Modul $(\mathcal{B}, \mathcal{R}, :)$ und einer Abbildung ${}^c : B \longrightarrow R$, so daß für alle $a \in B$ und $r \in R$ gilt:
P1 $\quad a^c : 1 = a$
P2 $\quad (r : 1)^c = r ; 1$.

Man bezeichnet ^{c} als die Zylindrifikationsoperation. Die Priorität der Ordnung innerhalb der Operationen (in absteigender Ordnung) ist ^{c}, $\prime$, $\smile$, $:$, $;$, $\cdot$ und $+$.

Ein Beispiel für eine Peirce-Algebra ist die *volle* Peirce-Algebra $\mathcal{P}(U) = (\mathcal{B}(U), \mathcal{R}(U), :, {}^c)$ über einer Menge $U \neq \emptyset$ mit einem vollen Booleschen Modul $(\mathcal{B}(U), \mathcal{R}(U), :)$ über U sowie der in (4.36) definierten linken Zylindrifikationsoperation auf Mengen ^{c}.

Zur Vorbereitung für den nächsten Abschnitt führe ich eine Beschränkungsoperation $\rfloor$ ein, die wie folgt definiert wird:

P3 $\quad r \rfloor a = r \cdot a^{c\smile}$.

Wegen (4.38) ist die Operation $\rfloor$ das algebraische Gegenstück zur Operation der Zielbereichsbeschränkung (*engl. the range restriction operation*).

4.2 Algebraische Semantik

$\mathcal{ALC}$ ist eine konzept-beschreibende Sprache. Konzepte und Rollen können kombiniert werden, um neue Konzepte zu bilden. Konzepte werden in $\mathcal{ALC}$ durch die in Abschnitt 3.1 definierte Subsumtionsordnung miteinander in Beziehung gesetzt. Um mächtigere Sprachen zu erhalten, wurde von verschiedenen Personen versucht, relationen-konstruierende Operatoren und die Subsumtionsordnung auf Rollen in $\mathcal{ALC}$ einzuführen. Beispielsweise führen Hollunder [HB89], Hollunder *et al* [HNS90] und Donini *et al* [DLNHN90] die Rollenkonjunktion und die Rollensubsumtion in $\mathcal{ALC}$ ein. KL-ONE (Woods und Schmolze [WS91]) und NIKL (Schmolze [SJ89]) verfügen über konzept- und rollen-konstruierenden Operatoren und sind mit der Konzept- und Rollensubsumtion ausgestattet. Patel-Schneider [PS87] hat eine mächtige terminologische Sprache eingeführt, die $\mathcal{U}$ genannt wird. Diese Sprache $\mathcal{U}$ hat die terminologischen Sprachen vieler anderer Systeme als Untersprachen.
Im folgenden Abschnitt definiere ich eine terminologische Sprache $\mathcal{ALCX}$ unter Zuhilfenahme der modell-theoretischen Semantik (Prädikatenlogik erster Stufe). $\mathcal{ALCX}$ ist eine Erweiterung von $\mathcal{ALC}$ und eine Untersprache von $\mathcal{U}$. Weiterhin verfügt $\mathcal{ALCX}$ über 3 Operatoren mehr als $\mathcal{ALC}$ und bildet eine getrennte *Rollentaxonomie*. Die hinzugefügten Operatoren sind '**inverse**' (Rolleninversion), '**compose**' (Rollenkomposition) und '**restrict**' (Rollenbeschränkung). Zusätzlich besteht in $\mathcal{ALCX}$ die Möglichkeit, die Operatoren '**and**' und '**or**' auf beliebige Rollenbeschreibungen anzuwenden. Dabei sind die 4 Operatoren '**and**', '**or**', '**inverse**' und '**compose**' Rollen-konstruierend auf Rollen; der Operator '**restrict**' ist Rollen-konstruierend auf Konzepten und Rollen.
Im zweiten Abschnitt wird eine algebraische Repräsentation der Semantik der Sprache $\mathcal{ALCX}$ gegeben. Die dabei benutzten algebraischen Strukturen sind Boolesche Algebren, Relationenalgebren, Boolesche Module und Peirce-Algebren, wie sie in Abschnitt 4.1 dargestellt wurden. Dort wurden diese Algebren als Beschreibungen unterschiedlicher Operationen auf Mengen und Relationen vorgestellt, wobei Gleichungsaxiome und daraus geschlußfolgerte Gleichungen erfüllt werden. Mittels der algebraischen Semantik der Sprache $\mathcal{ALCX}$ kann jeder in $\mathcal{ALCX}$ definierte terminologische Ausdruck mit einem algebraischen Term assoziiert werden. Ferner kann jede in den oben erwähnten Algebren erfüllte Gleichung in eine semantisch äquivalente Relation zwischen zwei terminologischen Ausdrücken überführt werden.

4.2.1 Definition der Sprache $\mathcal{ALCX}$

Ich beginne die Definition der Sprache $\mathcal{ALCX}$ mit der Vorstellung der Syntax. Das Vokabular von $\mathcal{ALCX}$ besteht aus drei disjunkten Mengen von Symbolen. Diese Mengen sind:
Eine Menge von Zeichen für primitive Konzepte, eine Menge von Zeichen für primitive Rollen und eine Menge von strukturellen Symbolen. Die Menge der strukturellen Symbole enthält die Operatoren '**and**', '**or**', '**not**', '**some**', '**all**', '**inverse**', '**compose**' und '**restrict**' sowie zusätzlich die Symbole '$\sqsubseteq$' und '$\doteq$'. Wie $\mathcal{ALC}$ hat $\mathcal{ALCX}$ zwei ausgezeichnete primitive Konzepte:
Das Topkonzept '$\top$' und das Bottomkonzept '$\bot$'. $\mathcal{ALCX}$ hat eine ausgezeichnete primitive Rolle '**self**', die *Identitätsrolle* genannt wird. Wie in Abschnitt 3.1 bezeichnet '**A**' ein beliebiges primitives Konzeptsymbol, '**C**' '**D**' beliebige Konzeptbeschreibungen und '**Q**' eine beliebige primitive Rolle. Desweitern bezeichnen '**R**' und '**S**' beliebige Rollenbeschreibungen. Konzeptbeschreibungen können nach der folgenden Regel gebildet werden, die Konjunktionen, Disjunktionen und Negationen spezifiziert:

$(4.42)\mathbf{C},\ \mathbf{D} \longrightarrow \mathbf{A}|(\mathbf{and\ C\ D})|(\mathbf{or\ C\ D})|(\mathbf{not\ C}).$

Andere konzept-konstruierende Operatoren werden zur Repräsentation der existentiellen und universellen Beschränkungen benutzt. Dabei sind die verwendeten Rollen nicht nur primitive Rollen, sondern auch *zusammengesetzte* Rollen (beliebige Rollenbeschreibungen). Die Regel (4.42) wird dazu wie folgt erweitert:

$(4.43)\mathbf{C},\ \mathbf{D} \longrightarrow \cdots\ |(\mathbf{some\ R\ C})|(\mathbf{all\ R\ C}).$

Rollenbeschreibungen werden mit Hilfe der drei folgenden Regeln (4.44) bis (4.46) ermöglicht. Zur Spezifikation von Rollenkonjunktionen und Rollendisjunktionen wird die Regel

$(4.44)\mathbf{R},\ \mathbf{S} \longrightarrow \mathbf{Q}|(\mathbf{and\ R\ S})|(\mathbf{or\ R\ S}).$

verwendet. Zur Spezifikation von Rolleninversionen und Rollenkompositionen wird die Regel

$(4.45)\mathbf{R},\ \mathbf{S} \longrightarrow \cdots\ |(\mathbf{inverse\ R})|(\mathbf{compose\ R\ S}).$

verwendet. Rollenbeschränkungen (*engl. role restriction*) werden unter Verwendung der folgenden Regel spezifiziert:

(4.46)**R, S** $\longrightarrow \cdots \mid$(**restrict R C**).

Im folgenden erläutere ich die Semantik der Sprache $\mathcal{ALCX}$. Wie in $\mathcal{ALC}$ wird eine Interpretation in der Sprache $\mathcal{ALCX}$ durch eine Domäne $\mathcal{D}^{\mathcal{I}}$ und eine Interpretationsfunktion $\cdot^{\mathcal{I}}$ definiert. Eine Interpretationsfunktion ordnet jeder Konzeptbeschreibung **C** eine Untermenge $\mathbf{C}^{\mathcal{I}}$ von $\mathcal{D}^{\mathcal{I}}$ und jeder Rolle **R** eine binäre Relation $\mathbf{R}^{\mathcal{I}}$ über der Menge $\mathcal{D}^{\mathcal{I}}$ (d.h. $\mathbf{R}^{\mathcal{I}} \subseteq \mathcal{D}^{\mathcal{I}} \times \mathcal{D}^{\mathcal{I}}$) zu. Die Semantik der ausgezeichneten Konzepte und der konzept-konstruierenden Operatoren, beide wurden im letzten Kapitel für die Sprache $\mathcal{ALC}$ definiert, wird in der Sprache $\mathcal{ALCX}$ ebenfalls durch (3.3) definiert. Dabei ist in (3.3) das primitive Rollensymbol '**Q**' in der Definition von **some** und **all** durch das allgemeine Rollensymbol '**R**' zu ersetzen. Die Semantik der ausgezeichneten Rolle und der Rollen-konstruierenden Operatoren wird wie folgt vereinbart:

(4.47)

$$
\begin{aligned}
\mathbf{self}^{\mathcal{I}} &= \{(x,x) \mid x \in \mathcal{D}^{\mathcal{I}}\} \\
(\mathbf{and\ R\ S})^{\mathcal{I}} &= \mathbf{R}^{\mathcal{I}} \cap \mathbf{S}^{\mathcal{I}} \\
(\mathbf{or\ R\ S})^{\mathcal{I}} &= \mathbf{R}^{\mathcal{I}} \cup \mathbf{S}^{\mathcal{I}} \\
(\mathbf{inverse\ R})^{\mathcal{I}} &= \{(x,y) \mid (y,x) \in \mathbf{R}^{\mathcal{I}}\} \\
(\mathbf{compose\ R\ S})^{\mathcal{I}} &= \{(x,y) \mid (\exists z)[(x,z) \in \mathbf{R}^{\mathcal{I}} \ \&\ (z,y) \in \mathbf{S}^{\mathcal{I}}]\} \\
(\mathbf{restrict\ R\ C})^{\mathcal{I}} &= \{(x,y) \mid [(x,y) \in \mathbf{R}^{\mathcal{I}} \ \&\ y \in \mathbf{C}^{\mathcal{I}}]\}.
\end{aligned}
$$

Somit sind die Interpretationen der Rollen-konstruierenden Operatoren die gewöhnlichen mengen-theoretischen Operationen Identität, Durchschnitt, Vereinigung, Konversion, Komposition und Zielbereichsbeschränkung (*range restriction*). Beispiele für Konzeptbeschreibungen, die mit **and, or, not, some** und **all** repräsentiert werden, sind im vorherigen Kapitel angegeben. Wie bereits erwähnt, kann die Relation '**ist_Kind_von**' als (**inverse hat_Kind**) und die Relation '**ist_Schwiegereltern**' als (**compose hat_Kind ist_verheiratet**) repräsentiert werden. Im folgenden gebe ich einige Beispiele zur Erläuterung aller übrigen Operatoren:

(4.48) (**and hat_Kind ist_Lehrer**) repräsentiert die Relation, die gleichzeitig die Eltern - Kind und die Lehrer - Kind Beziehung beschreibt.

(4.49) (**or hat_Kind ist_verheiratet**) repräsentiert die Relation, die die Beziehung zwischen Eltern und ihren Kindern oder zwischen einem Ehepaar beschreibt.

(4.50) (**restrict hat_Kind Student**) repräsentiert die Relation, die die Beziehung zwischen Eltern und ihren Kindern beschreibt, wobei die Kinder Studenten sind.

Im folgenden wird die Definition von Subsumtion und Äquivalenz für Konzepte in $\mathcal{ALC}$ hin zur Subsumtion und Äquivalenz für Rollen erweitert:

(4.51)

$$\sigma, \tau \longrightarrow \cdots \mid \mathbf{R} \sqsubseteq \mathbf{S} \mid \mathbf{R} \doteq \mathbf{S}.$$

Die obige Regel erweitert die Definition (3.4) von terminologischen Axiomen um Rollenspezialisation und Rollenäquivalenz. Ihre Semantik wird, ähnlich der Semantik für die terminologischen Axiome (3.5) in Abschnitt 3.1, wie folgt definiert:

(4.52)

$$\begin{aligned} \models_{\mathcal{I}} \mathbf{R} \sqsubseteq \mathbf{S} \quad & gdw \quad \mathbf{R}^{\mathcal{I}} \subseteq \mathbf{S}^{\mathcal{I}} \\ \models_{\mathcal{I}} \mathbf{R} \doteq \mathbf{S} \quad & gdw \quad \mathbf{R}^{\mathcal{I}} = \mathbf{S}^{\mathcal{I}}. \end{aligned}$$

Auf die Definition der Begriffe *Modell* für eine Terminologie, *Folge einer Terminologie*, *zulässige* terminologische Axiome, *Rollensubsumtion* und *Rollenäquivalenz* bezüglich einer Terminologie der Sprache $\mathcal{ALCX}$ verzichte ich. Diese sind direkte Verallgemeinerungen der entsprechenden Begriffe aus $\mathcal{ALC}$.
Eine partiell geordnete Menge von Konzeptäquivalenzklassen bezeichnet man als Konzepttaxonomie. Entsprechend bezeichnet man als Rollentaxonomie eine partiell geordnete Menge $(\mathbf{R}/\approx_T, \preceq_T / \approx_T)$ von Rollenäquivalenzklassen, die bezüglich dem Quotient von Rollensubsumtion nach Rollenäquivalenz geordnet ist. $\mathbf{R}$ bezeichnet dabei im letzten Ausdruck die Menge aller Rollenbeschreibungen in der Sprache $\mathcal{ALCX}$.

Das Symbol '∇', wie es R. A. Schmidt [SR91] eingeführt hat, bezeichnet den Ausdruck (**or self (not self)**) und das Symbol '$\wedge$' den Ausdruck (**not** ∇). Es gelten die folgenden Gleichungen:

(4.53)

$$\begin{aligned} \nabla^{\mathcal{I}} &= \mathcal{D}^{\mathcal{I}} \times \mathcal{D}^{\mathcal{I}} \\ \wedge^{\mathcal{I}} &= \emptyset. \end{aligned}$$

Das Symbol ∇ wird als *Toprolle* und das Symbol $\wedge$ als *Bottomrolle* bezeichnet.

4.2.2 Algebraische Semantik der Sprache $\mathcal{ALCX}$

Eine Interpretation $\mathcal{I}$ von $\mathcal{ALCX}$ ist ein Paar $(U, \cdot^{\mathcal{I}})$, wobei U $(= \mathcal{D}^{\mathcal{I}})$ das Universum der Interpretation und $\cdot^{\mathcal{I}}$ die Interpretationsfunktion der Interpretation ist. Ein Konzept **C** wird als eine Menge $\mathbf{C}^{\mathcal{I}}$ $(\subseteq U)$ interpretiert. Eine Rolle **R** wird als eine binäre Relation $\mathbf{R}^{\mathcal{I}}$ über der Menge U interpretiert. Die Interpretationsfunktion $\cdot^{\mathcal{I}}$ wird im algebraischen Kontext definiert, statt sie (wie bei bisherigen Vereinbarungen der Semantik üblich) modell-theoretisch zu definieren. Um die algebraische Interpretation zu betonen, wird auf die in Abschnitt 4.1 eingeführte Notation zurückgegriffen. Die Symbole $\mathbf{C}^{\mathcal{I}}$, $\mathbf{D}^{\mathcal{I}}$, $\cdots$ sowie $\mathbf{R}^{\mathcal{I}}$, $\mathbf{S}^{\mathcal{I}}$, $\cdots$ werden durch die Symbole C, D, $\cdots$ und R, S, $\cdots$ abgekürzt.

Die in Abschnitt 3.1 definierte Interpretation der Konzeptbeschreibungen kann wie folgt umgeschrieben werden (dabei wird '**Q**' durch '**R**' ersetzt):

(4.54)

$$
\begin{aligned}
\top^{\mathcal{I}} &= U \\
\bot^{\mathcal{I}} &= \emptyset \\
(\mathbf{and\ C\ D})^{\mathcal{I}} &= C \cap D \\
(\mathbf{or\ C\ D})^{\mathcal{I}} &= C \cup D \\
(\mathbf{not\ C})^{\mathcal{I}} &= C' \\
(\mathbf{some\ R\ C})^{\mathcal{I}} &= R : C \\
(\mathbf{all\ R\ C})^{\mathcal{I}} &= (R : C')'.
\end{aligned}
$$

Die ausgezeichneten Konzepte Topkonzept ($\top$) und Bottomkonzept ($\bot$) sowie die Booleschen Operatoren (**and**, **or**, **not**) werden wie in Abschnitt 3.1 definiert. Der Operator **some** ist das Peirce-Produkt (in (4.22) definiert) und der Operator **all** ist die in Satz (4.31) betrachtete Konstruktion aus dem Peirce-Produkt und dem Komplement.

Die Interpretation der Rollenbeschreibungen wird wie folgt verändert. Die modell-theoretische Semantik der Identitätsrolle **self** stimmt mit der in (4.12) gegebenen Definition der Identitätsrelation *Id* überein.
Die Booleschen Operatoren **and** und **or** (diesmal für Rollen angewendet) werden wie in (4.47) definiert. Wie aus (4.11) und (4.10) ersichtlich ist, kann der **inverse** und

compose Operator äquivalent zur Konversion ($\smile$) und der relationalen Komposition (;) definiert werden. Daher gilt :

(4.55)

$$\begin{aligned}
\mathbf{self}^{\mathcal{I}} &= Id \\
(\mathbf{and}\ \mathbf{R}\ \mathbf{S})^{\mathcal{I}} &= R \cap S \\
(\mathbf{or}\ \mathbf{R}\ \mathbf{S})^{\mathcal{I}} &= R \cup S \\
(\mathbf{inverse}\ \mathbf{R})^{\mathcal{I}} &= R^{\smile} \\
(\mathbf{compose}\ \mathbf{R}\ \mathbf{S})^{\mathcal{I}} &= R; S.
\end{aligned}$$

Der **restrict** Operator kann unter Benutzung von linker Zylindrifikation oder als Zielbereichsbeschränkung formuliert werden:

(4.56)

$$(\mathbf{restrict}\ \mathbf{R}\ \mathbf{C})^{\mathcal{I}} = R \cap C^{c\smile} \qquad (= R\rfloor C).$$

Im folgenden wird die Semantik der in (4.53) definierten Top- und Bottomrollen (∇ und $\wedge$) angebeben:

(4.57)

$$\begin{aligned}
\nabla^{\mathcal{I}} &= U^2 \\
\wedge^{\mathcal{I}} &= \emptyset
\end{aligned}$$

Die Interpretation der terminologischen Axiome, die die Spezialisation- und Äquivalenzrelationen zwischen Konzepten und Rollen spezifizieren, wird wie in Abschnitt 3.1 definiert:

(4.58)

$$\begin{aligned}
&\models_{\mathcal{I}} \mathbf{C} \sqsubseteq \mathbf{D} \quad gdw \quad C \subseteq D \\
&\models_{\mathcal{I}} \mathbf{C} \doteq \mathbf{D} \quad gdw \quad C = D.
\end{aligned}$$

Ebenso wird die Interpretation für Spezialisation und Äquivalenz zwischen Rollen definiert:

(4.59)

$$\models_{\mathcal{I}} \mathbf{R} \sqsubseteq \mathbf{S} \quad gdw \quad R \subseteq S$$
$$\models_{\mathcal{I}} \mathbf{R} \doteq \mathbf{S} \quad gdw \quad R = S$$

Die algebraischen Gegenstücke der terminologischen Konstanten Top- bzw. Bottomkonzept ($\top$, $\bot$) sind das Top- bzw. Bottomelement 1 bzw. 0 der partiellen Ordnung einer Booleschen Algebra. Auf ähnlicher Weise wird mit der Top- bzw.Bottomrolle (∇, $\wedge$) verfahren, diese werden mit der 0 bzw. 1 einer Relationenalgebra verbunden. Die der ausgezeichneten Rolle **self** entsprechende Konstante ist das Identitätselement e einer Relationenalgebra. Die dem Symbol $\sqsubseteq$ entsprechende Beziehung in den jeweiligen Algebren ist die Inklusion. Dem Symbol $\doteq$ entspricht in den jeweiligen Algebren die Gleichung.

Mit Hilfe der oben aufgeführten Übertragung auf einen algebraischen Kontext kann die modell-theoretische Semantik terminologischer Ausdrücke in $\mathcal{ALCX}$ durch Konstanten und Operationen im Kalkül der Peirce-Algebren formuliert werden. Jeder terminologische Ausdruck kann direkt mit einem algebraischen Term assoziiert werden. Welcher terminologische Ausdruck mit welchem algebraischen Term korrespondiert, kann Abbildung (4.1) entnommen werden. In dieser Abbildung werden die unterschiedlichen terminologischen Ausdrücke, ihre jeweiligen Interpretationen sowie ihre assoziierten algebraischen Beschreibungen gegenübergestellt.

Am Ende dieses Abschnittes wird anhand des Axioms (B4) gezeigt, wie die algebraischen Eigenschaften in Verbindung zu der Semantiktheorie von $\mathcal{ALCX}$ stehen. Das Axiom B4 lautet:

(4.60) $a \cdot (a + b) = a$ (a, b sind beliebige Elemente einer Booleschen Algebra).

Die Identität B4 wird im Kalkül von Mengen erfüllt. Dies impliziert, daß für beliebige Konzeptbeschreibungen A, B das assoziierte terminologische Argument

(4.61)$(\mathbf{and}\ \ \mathsf{A}\ \ (\mathbf{or}\ \ \mathsf{A}\ \ \mathsf{B})) \doteq \mathsf{A}$

in allen Interpretationen $\mathcal{I}$ für $\mathcal{ALCX}$ erfüllt wird. (Begründung: $(\mathbf{and}\ \mathsf{A}\ (\mathbf{or}\ \mathsf{A}\ \mathsf{B}))^{\mathcal{I}} = \mathsf{A}^{\mathcal{I}} \cap (\mathbf{or}\ \mathsf{A}\ \mathsf{B})^{\mathcal{I}} = \mathsf{A}^{\mathcal{I}} \cap (\mathsf{A}^{\mathcal{I}} \cup \mathsf{B}^{\mathcal{I}}) = \mathsf{A}^{\mathcal{I}}$) Dies impliziert wiederum

(4.62) $T \models (\mathbf{and}\ \ \mathsf{A}\ \ (\mathbf{or}\ \ \mathsf{A}\ \ \mathsf{B})) \doteq \mathsf{A}$.

Daher gilt

Abbildung 4.1: Algebraische Semantik von $\mathcal{ALCX}$

Terminologischer Ausdruck	Interpretation	Algebraischer Term
$\top$	U	1
$\bot$	ϕ	0
(**and C D**)	$C \cap D$	$a \cdot b$
(**or C D**)	$C \cup D$	$a + b$
(**not C**)	C'	a'
∇	U^2	1
Λ	ϕ	0
self	Id	e
(**and R S**)	$R \cap S$	$r \cdot s$
(**or R S**)	$R \cup S$	$r + s$
(**inverse R**)	$R^{\smile}$	$r^{\smile}$
(**compose R S**)	$R \,;\, S$	$r \,;\, s$
(**some R C**)	$R : C$	$r : a$
(**all R C**)	$(R : C')'$	$(r : a')'$
(**restrict R C**)	$R \cap C^{c\prime} = R \rfloor C$	$r \cdot a^{c\smile} = r \rfloor a$

(4.63) (**and** A (**or** A B)) $\approx$ A für alle Konzeptbeschreibungen A, B.

Die universelle Identität $a \cdot (a + b) = a$ kann somit in die semantisch äquivalente Relation (**and** A (**or** A B)) $\approx$ A überführt werden. Das heißt, die beiden Konzepte (**and** A (**or** A B)) und A werden durch die Interpretationsfunktion der gleichen Menge zugeordnet.(Die Gleichungsaxiome und die gleichungsmäßigen Eigenschaften in Algebren in Abschnitt 4.1 hat Schmidt R. A. [SR91] als *universelle Identitäten* bezeichnet.) Für alle übrigen in Abschnitt 4.1 aufgeführten universellen Identitäten kann eine vergleichbare Überführung gezeigt werden.

Jede universelle Identität (z. B. $a \cdot (a + b) = a$) kann in eine semantisch äquivalente Relation (z. B. (**and** A (**or** A B)) $\approx$ A) überführt werden. Diese terminologischen, semantisch äquivalenten Relationen nenne ich *terminologische Identitäten.*

Kapitel 5

Algebraische Simplifikation von Anfragen an eine KL-ONE-Wissensbasis

In den letzten Jahren wurden viele auf KL-ONE [BS85] basierende Wissensrepräsentationssysteme entwickelt, zum Beispiel BACK [NvL88, Neb88], CLASSIC [BBMR89], KANDOR [PS84], KL-TWO [VM85], KRYPTON [BPL85], LOOM [MB87], NIKL [KBR86], SB-ONE [KA89], KRIS [BH90]. Allen diesen Systemen ist gemeinsam, daß das Wissen in zwei Teilen getrennt repräsentiert wird. Diese Teile sind ein terminologischer Teil (**TBox**) und ein assertionaler Teil (**ABox**). Wie im dritten Kapitel gesehen, enthält die TBox in einer terminologischen Sprache formulierte terminologische Axiome. Diese beschreiben Konzepte und Rollen sowie Beziehungen zwischen Konzepten und Beziehungen zwischen Rollen. Die ABox enthält Informationen über einzelne Individuen; in ihr werden Individuen bzw. Paare von Individuen spezifiziert, die Instanzen von Konzepten bzw. Instanzen von Rollen sind.
Die Folgerungsmechanismen in diesen Systemen ermöglichen die Repräsentation expliziter Informationen (Wissen), wie auch impliziter Informationen bezüglich einer (**TBox**) und einer (**ABox**). Folgerungsmechanismen sind Algorithmen, mit deren Hilfe sich beispielsweise Schlußfolgerungen für die folgenden Probleme spezifizieren lassen:

- Ist das repräsentierte Wissen konsistent ? (Konsistenzproblem)

- Ist ein Konzept allgemeiner als ein anderes ? (Subsumtionsproblem)

- Welche Tatsachen sind aus dem repräsentierten Wissen deduzierbar ? (Instanziationsproblem)

- Wo genau liegt ein Konzept in einer Konzepthierarchie ? (Klassifikationsproblem)

- Gegeben sei ein Objekt a aus einer ABox. Instanz welcher Konzepte ist das Objekt a ? (Erkennungsproblem)

- Was sind die Instanzen eines gegebenen Konzeptes ?
  Was sind die Instanzen einer gegebenen Rolle ? (Retrieval-Problem)

Das von mir entwickelte Programm, das in der funktionalen Programmiersprache **Gofer** erstellt ist, bietet eine Lösung des Retrieval-Problems. Das Wissen über eine Umwelt von Personen, das in den letzten Kapiteln mittels der terminologischen Sprache $\mathcal{ALCX}$ repräsentiert wurde, wird in meinem Programm mit Hilfe der in **Gofer** gegebenen Möglichkeiten formuliert. Dabei verwende ich die primitiven Konzepte 'person', 'student', 'maennlich', 'weiblich', 'vegetarier' sowie die primitiven Rollen 'hat_Kind', 'ist_verheiratet', 'ist_Kusin', 'ist_Lehrer'. Das ausgezeichnete Top-Konzept ist 'person', da Wissen über eine Umwelt von Personen behandelt wird. Zusammengesetzte Konzepte und Rollen, die durch Verwendung der terminologischen Operatoren in $\mathcal{ALCX}$ konstruiert werden, kommen in meiner Wissensbasis vor. An mein Programm können Anfragen gestellt werden, die in meiner Wissensbasis vorkommende Konzepte bzw. Rollen darstellen. Diese Anfragen werden interaktiv ausgewertet; die Instanzen eines Konzeptes bzw. einer Rolle werden direkt ausgegeben. Im Mittelpunkt meiner Arbeit steht die Optimierung solcher Anfragen.

Im letzten Kapitel habe ich die algebraischen Strukturen, Boolesche Algebren, Relationenalgebren, Boolesche Module und Peirce-Algebren betrachtet und damit eine algebraische Semantik für die Sprache $\mathcal{ALCX}$ erhalten. Die algebraische Semantik für die Sprache $\mathcal{ALCX}$ ermöglicht, daß die in den zuletzt erwähnten Algebren erfüllten Gleichungsaxiome und die daraus schlußgefolgerten Gleichungen als Simplifikationsmittel verwendet werden können. Die im letzten Absatz erwähnten Anfrageausdrücke werden durch fortgesetzte Simplifikation optimiert.

Im folgenden erkläre ich, wie in meinem Programm bei der Evaluation einer Anfrage vorgegangen wird, und wie die Optimierung einer Anfrage erfolgt.

5.1 Instanzmengen primitiver Konzepte bzw. primitiver Rollen

Im ersten Teil meines Programms werden primitive Konzepte (Konzeptnamen), primitive Rollen (Rollennamen) sowie ihre Instanzen eingeführt. Ein Konzept wird als eine Menge interpretiert, eine Rolle als eine Relation. Konzepte und Rollen werden durch die Deklaration von Konstantefunktionen als Mengen und Relationen wie folgt realisiert:

- Die Konstantefunktionen mit den Namen **person, student, maennlich, weiblich, vegetarier** und **basisbottom** bezeichnen Konzepte. Alle diese Funktionen haben den Typ **[String]** (eine Liste von String). Die Funktionsergebnisse bezeichnen Mengen von Instanzen der entsprechenden Konzepte.

- Die Konstantenfunktionen mit den Namen **hat_Kind, ist_verheiratet, ist_Kusin, ist_Lehrer** und **r_bottom** bezeichnen Rollen. Alle diese Funktionen haben den Typ **[(String, String)]** (eine Liste von Paaren). Die Funktionsergebnisse sind aus Tupeln bestehende Relationen, wobei die Tupel die Instanzen der Rollen sind.

Den aus den oben aufgeführten Definitionen bestehenden Teil bezeichne ich als *terminologische* Datenbank.

5.2 Syntax der Anfragen (Die Datentypen Konzeptterm und Relationenterm)

Im folgenden stelle ich die zwei Datentypen **Konzeptterm** und **Relationenterm** vor. Mit Hilfe dieser beiden Datentypen werden Anfragen beschrieben, die Konzeptterme und Rollenterme aus $\mathcal{ALCX}$ darstellen. Ein Ziel dieser Arbeit ist, wie bereits erwähnt, die Simplifikation von Anfragen. Die Simplifikation von Anfragen wird in meinem Programm unter Zuhilfenahme von 'pattern matching' realisiert. Den primitiven Konzepten und primitiven Rollen im Programm sind entsprechende Mengen und Relationen direkt zugeordnet. Durch die Einführung der beiden Datentypen Konzeptterm und Relationenterm wird ein zeitaufwendiger direkter Vergleich von Mengen durch „pattern matching" vermieden.

In den meisten terminologischen Sprachen, die von verschiedenen Autoren definiert wurden, sind die terminologischen Operatoren Konjunktion, Disjunktion und Rollenkomposition als n-stellige Operatoren definiert. In diesem Fall spezifiziert die Interpretationsfunktion die Bedeutung der zusammengesetzten Konzepte $\mathbf{C}_i$ und Rollen $\mathbf{R}_i$ wie folgt:

$$
\begin{aligned}
(\mathbf{and}\,\mathbf{C_1},\,\cdots,\mathbf{C_n})^{\mathcal{I}} &= \mathbf{C_1^{\mathcal{I}}} \cap \,\cdots\, \cap \mathbf{C_n^{\mathcal{I}}} \\
(\mathbf{or}\,\mathbf{C_1},\,\cdots,\mathbf{C_n})^{\mathcal{I}} &= \mathbf{C_1^{\mathcal{I}}} \cup \,\cdots\, \cup \mathbf{C_n^{\mathcal{I}}} \\
(\mathbf{and}\,\mathbf{R_1},\,\cdots,\mathbf{R_n})^{\mathcal{I}} &= \mathbf{R_1^{\mathcal{I}}} \cap \,\cdots\, \cap \mathbf{R_n^{\mathcal{I}}} \\
(\mathbf{or}\,\mathbf{R_1},\,\cdots,\mathbf{R_n})^{\mathcal{I}} &= \mathbf{R_1^{\mathcal{I}}} \cup \,\cdots\, \cup \mathbf{R_n^{\mathcal{I}}} \\
(\mathbf{compose}\,\mathbf{R_1},\,\cdots,\mathbf{R_n})^{\mathcal{I}} &= \mathbf{R_1^{\mathcal{I}}};\,\cdots\,;\mathbf{R_n^{\mathcal{I}}}
\end{aligned}
$$

Ohne Beschränkung der Allgemeinheit wurden Konjunktion, Disjunktion und Rollenkomposition in der Sprache $\mathcal{ALCX}$ in Abschnitt 4.2.1 als binäre Operatoren definiert. Im Programm habe ich diese Operatoren als n-stellige Operatoren vereinbart.

- *Datentyp* **Konzeptterm**

Der Datentyp **Konzeptterm** beschreibt die Ausdrücke zur Darstellung der Konzepte in meiner Wissensbasis. Konzepte in einer terminologischen Wissensbasis sind entweder primitive Konzepte oder durch konzept-konstruierende Operatoren zusammengesetzte Konzepte. Die Konstruktoren *Person, Student, Maennlich, Weiblich* und *Vegetarier* in **Konzeptterm** stellen die jeweiligen primitiven Konzepte *person, student, maennlich, weiblich* und *vegetarier* dar. Der Konstruktor *Basisbottom* stellt das leere Konzept dar. Die Konstruktorfunktionen *And, Or, Not, Some* und *All* stellen die terminologischen Operatoren **and, or, not, some** und **all** dar. Die Operatoren **and**, **or** und **not** sind konzept-konstruierend auf Konzepten und haben Konzepte als Argumente. Daher werden die Konstruktorfunktionen **And, Or** und **Not** auf den Datentyp **Konzeptterm** angewendet. Die Ergebnisse dieser Konstruktorfunktionen haben wieder den Typ **Konzeptterm** und stellen zusammengesetzte Konzepte dar. Zum Beispiel hat die Konstruktorfunktion **And** folgenden Typ und folgende Bedeutung:

`And :: [Konzeptterm]` $\longrightarrow$ `Konzeptterm`

> Falls die Konstruktorfunktion *And* auf *Student* und *Vegetarier* gebildet wird, wird der Ausdruck (*And* [*Student*, *Vegetarier*]) erzeugt, der das Konzept „studentische Vegetarier“ (*and student vegetarier*) darstellt.

Die Operatoren *some* und *all* sind konzept-konstruierend auf Rollen und Konzepten. Daher werden die Konstruktorfunktionen *Some* und *All* auf dem Datentyp **Relationenterm** (Dieser beschreibt Ausdrücke, die Rollen darstellen.) und auf dem Datentyp **Konzeptterm** gebildet.
Sämtliche Konzepte in meiner Wissensbasis können durch den Datentyp **Konzeptterm** dargestellt werden. Die folgende Tabelle enthält einige Beispielausdrücke aus Konzeptterm und deren zugeordnete Konzepte:

Darstellungen	Konzeptterme
Person	person
(And [Student, Vegetarier])	(and student vegetarier)
(Not (And [Student, Vegetarier]))	(not (and student vegetarier))
(Some HatKind Student)	(some hat_Kind student)
(All HatKind (And [Student, Vegetarier]))	(all hat_Kind (and student vegetarier))

Durch Hinzufügung der Konstruktorfunktion *Menge* (`Menge :: [String]` $\longrightarrow$ `Konzeptterm`) können Ausdrücke gebildet werden, die unbenannte, variable Mengen bestimmter Personen repräsentierende Konzepte darstellen. Beispielsweise wird ein Ausdruck (**Menge ["Stefan"]**) durch Anwendung der Konstruktorfunktion *Menge* auf eine beliebige Menge ["Stefan"] erzeugt. (Dieser Ausdruck stellt ein atomares (einelementiges) Konzept dar.) Der Ausdruck (**Some HatKind (Menge ["Stefan"])**) stellt das Konzept dar, dessen Instanzen die Eltern von Stefan sind.

- *Datentyp* **Relationenterm**

  Der Datentyp **Relationenterm** beschreibt Ausdrücke, die Rollen meiner Wissensbasis darstellen. Rollen in einer terminologischen Wissensbasis sind primitive Rollen oder durch rollen-konstruierende Operatoren zusammengesetzte Rollen. Die Konstruktoren *HatKind, IstKusin, IstLehrer* und *IstVerheiratet* des Datentyps **Relationenterm** stellen die jeweiligen Rollen *hat_Kind, ist_Kusin, ist_Lehrer* und *ist_verheiratet* dar. Der Konstruktor *Relationenbottom* stellt die leere Rolle *Bottomrolle* dar. Die Konstruktorfunktionen *RAnd, ROr, Compose, Inverse* und *Restrict* stellen die terminologischen Operatoren *and, or, compose, inverse* und *restrict* dar. Die Operatoren *and, or, compose* und *inverse* sind rollen-konstruierend auf Rollen und haben somit Rollen als Argumente. Aus diesem Grund werden Ausdrücke, die Rollen darstellen, durch die Anwendung der Konstruktorfunktionen *RAnd, ROr, Compose* und *Inverse* auf Argumente vom Datentyp **Relationenterm** konstruiert. Diese konstruierten Ausdrücke stellen zusammengesetzte Rollen dar. So hat z. B. die Konstruktorfunktion *Inverse* folgenden Typ und folgende Bedeutung:

  $$\texttt{Inverse} :: \texttt{Relationenterm} \longrightarrow \texttt{Relationenterm}.$$

  > Der Ausdruck (*Inverse HatKind*) vom Datentyp **Relationenterm**, der die Rolle (*inverse hat_Kind*) darstellt, wird durch Anwendung der Konstruktorfunktion *Inverse* auf das Argument *HatKind* vom Datentyp **Relationenterm** konstruiert.

  Der Operator *restrict* ist rollen-konstruierend auf Rollen und Konzepten. Durch Anwendung der Konstruktorfunktion *Restrict* auf ein Argument vom Datentyp Relationenterm sowie auf ein Argument vom Datentyp Konzeptterm wird ein rollen-darstellender Ausdruck konstruiert. Die folgende Tabelle enthält einige Beispielausdrücke aus Relationenterm und deren zugeordnete Rollen:

Darstellungen	Rollenterme
HatKind	hat_Kind
(ROr [HatKind, IstVerheiratet])	(or hat_Kind ist_verheiratet)
(Inverse HatKind)	(inverse hat_Kind)
(Restrict HatKind (And [Student, Vegetarier]))	(restrict hat_Kind (and student vegetarier))

Die Konstruktorfunktion *Relation* (`Relation :: [(String, String)]` $\longrightarrow$ `Relationenterm`) kann vergleichbar der Konstruktorfunktion 'Menge' beim Datentyp Konzeptterm erklärt werden. Diesmal werden Ausdrücke zur Darstellung beliebiger unbenannter Relationen konstruiert.
Sämtliche Rollen können durch Ausdrücke vom Datentyp **Relationenterm** dargestellt werden.

Die Eingabe einer Anfrage hat folgendes Aussehen:

finde_alle (*Ausdruck*) bzw. *finde_Relation* (*Ausdruck*).

finde_alle und finde_Relation sind Funktionen, die den als Argument übergebenen Ausdruck berechnen. Berechnen bedeutet dabei, daß die Instanzen eines Konzepts bzw. einer Rolle ermittelt werden. Daher werden diese Funktionen im folgenden als Evaluatoren bezeichnet. Ein als Argument an die Funktion finde_alle übergebener Ausdruck muß vom Datentyp Konzeptterm sein. Entsprechend muß ein an die Funktion finde_Relation übergebener Ausdruck vom Datentyp Relationenterm sein. Im folgenden nenne ich derartige als Argumente übergebene Ausdrücke *Anfrageausdrücke*.

5.3 Evaluator

Die zwei Funktionen *evaluiere_K* und *evaluiere_R* werden definiert, um Anfrageausdrücke auswerten zu können. Der Evaluator evaluiere_K wertet Ausdrücke vom Datentyp **Konzeptterm** aus. Auswertung bedeutet dabei, daß die Instanzen des durch das Funktionsargument dargestellten Konzepts am Bildschirm ausgegeben werden. Mit dem Evaluator evaluiere_R werden Ausdrücke mit dem Datentyp **Relationenterm** ausgewertet. Auswertung bedeutet dabei, daß die Instanzen der durch das Funktionsargument dargestellten Rollen am Bildschirm ausgegeben werden. Im folgenden wird die Typdeklaration der Evaluatoren aufgeführt:

```
evaluiere_K :: Konzeptterm ⟶ [String]

evaluiere_R :: Relationenterm ⟶ [(String, String)]
```

Die Evaluatoren sind rekursiv definiert. Die Definition der Evaluatoren kann wie folgt erklärt werden:
Die Evaluatoren bilden einen Konstruktor (Konstruktor ohne Parameter) des Datentyps Konzeptterm oder Relationenterm auf eine Variable ab, die durch eine Variablendeklaration (Konstantenfunktion) definiert ist. Bei einer derartigen Variablendeklaration wird eine sortierte Liste konstruiert. Die Sortierung einer Liste wird dabei erzielt, indem ein sortiertes Abbild unter Zuhilfenahme der Funktion *sort* aus dem „standard prelude" deklariert wird. Die Sortierung erfolgt, um die später aufgeführten Funktionen 'und', 'oder', ⋯ effizienter (das Wort 'effizient' hat hier dieselbe Bedeutung wie in Abschnitt 5.4.) definieren zu können. Dies wird im folgenden an einem Beispiel gezeigt:

evaluiere_K (Person) = s_person

s_person = sort person.

Die folgenden Tabellen geben die Zuordnung zwischen den Konstruktoren und ihren entsprechenden Variablen wieder:

Funktion evaluiere_K	
Konstruktoren	Variablen
Basisbottom	bottom
Person	s_person
Student	s_student
Maennlich	s_maennlich
Weiblich	s_weiblich
Vegetarier	s_vegetarier

Funktion evaluiere_R	
Konstruktoren	Variablen
Relationenbottom	r_bottom
HatKind	s_hat_Kind
IstKusin	s_ist_Kusin
IstVerheiratet	s_ist_verheiratet
IstLehrer	s_ist_Lehrer

Die Werte einer Liste entsprechen den Instanzen eines Konzepts oder einer Rolle. Beispielsweise ist der Wert der Funktion s_person die Menge der Instanzen (Interpretation) des Konzeptes 'person'.

Die Evaluatoren bilden jeweils eine Konstruktorfunktion aus den Deklarationen von Konzeptterm und Relationenterm auf eine Funktion ab. Beispielsweise bildet der Evaluator *evaluiere_K* die Konstruktorfunktion *And* auf die Funktion *und* ab. Die folgenden Tabellen geben die Zuordnung zwischen den Konstruktorfunktionen und ihren zugehörigen Funktionen wieder:

Funktion evaluiere_K	
Konstruktorfunktion	Funktion
And	und
Or	oder
Not	nicht
Some	some
All	f_all

Funktion evaluiere_R	
Konstruktorfunktion	Funktion
ROr	oder_r
RAnd	und_r
Inverse	inverse
Compose	compose
Restrict	range_restrict

Die oben aufgeführten zugehörigen Funktionen realisieren die Interpretation der terminologischen Operatoren wie folgt:

- Die Funktion 'und' (und :: [[String]] $\longrightarrow$ [String]) realisiert die Interpretation des terminologischen Operators **and**. Diese Funktion bildet den Schnitt zwischen den Elementen der als Argument übergebenen Liste von Mengen.

- Die Funktion 'oder' (oder :: [[String]] $\longrightarrow$ [String]) realisiert die Interpretation des terminologischen Operators **or**. Diese Funktion bildet die Vereinigung zwischen den Elementen der als Argument übergebenen Liste von Mengen.

- Die Funktion 'nicht' (nicht :: [String] $\longrightarrow$ [String]) realisiert die Interpretation des terminologischen Operators **not**. Diese Funktion bildet das Komplement der Argumentmenge.

- Die Funktion 'some' (some :: [(String, String)] $\longrightarrow$ [String] $\longrightarrow$ [String]) realisiert die Interpretation des terminologischen Operators **some**. Diese Funktion bildet die existentiell beschränkte Menge ihrer Argumente (siehe dazu Abschnitt 4.2.1).

- Die Funktion 'f_all' (f_all :: [(String, String)] $\longrightarrow$ [[String] $\longrightarrow$ [String]) realisiert die Interpretation des terminologischen Operators **all**. Diese Funktion bildet eine universell beschränkte Menge ihrer Argumente (siehe dazu Abschnitt 4.2.1).

- Die Funktion 'und_r' (und_r :: [[(String, String)]] $\longrightarrow$ [(String, String)]) realisiert die Interpretation des terminologischen Operators **and**. Diese Funktion bildet den Schnitt zwischen den Elementen der als Argument übergebenen Liste von Relationen.

- Die Funktion 'oder_r' (oder_r :: [[(String, String)]] $\longrightarrow$ [(String, String)]) realisiert die Interpretation des terminologischen Operators **or**. Diese Funktion bildet die Vereinigung zwischen den Elementen der als Argument übergebenen Liste von Relationen.

- Die Funktion 'inverse' (inverse :: [(String, String)] $\longrightarrow$ [(String, String)]) realisiert die Interpretation des terminologischen Operators **inverse**. Diese Funktion spiegelt jedes Tupel der als Argument übergebenen Relation.

- Die Funktion 'compose' (compose :: [[(String, String)]] ⟶ [(String, String)]) realisiert die Interpretation des terminologischen Operators **compose**. Diese Funktion bildet das relationale Produkt zwischen den Elementen der als Argument übergebenen Liste von Relationen.

- Die Funktion 'range_restrict' (range_restrict :: [(String, String)] ⟶ [String] ⟶ [(String, String)]) realisiert die Interpretation des terminologischen Operators **restrict**. Diese Funktion bildet die zielbereichbeschränkte Relation der als erstes Argument übergebenen Relation und der als zweites Argument übergebenen Menge.

Einige Funktionen sind als Kompositionsfunktion definiert. Beispielsweise ist die Funktion 'und' als die Komposition der zwei Funktionen 'foldr1' aus dem „standard prelude“ und 'merge_und' definiert.

Die beiden Evaluatoren evaluiere_K und evaluiere_R sind auch unter den Namen '*finde_alle*' und '*finde_Relation*' verfügbar. Im folgenden werden die Typdeklarationen und die Definitionen der Funktionen 'finde_alle' und 'finde_Relation' aufgeführt:

```
findealle :: Konzeptterm ⟶ [String]
findealle a = evaluiere_K a

finde_Relation :: Relationenterm ⟶ [(String,String)]
finde_Relation a = evaluiere_R a
```

Im folgenden zeige ich an einem Beispiel, wie die Evaluatoren in meinem Programm arbeiten. Gestellt sei die Anfrage 'finde_alle (Some HatKind And [Student, Vegetarier, Weiblich])'. Bei der Auswertung dieser Anfrage wird wie folgt vorgegangen:

1. Der Evaluator 'evaluiere_K' wird auf das Argument '(Some HatKind And [Student, Vegetarier, Weiblich])' angewendet.

2. Der Ausschnitt 'evaluiere_K (Some b1 b2) = some (evaluiere_R b1) (evaluiere_K b2)' der Deklaration der Funktion 'evaluiere_K' erzeugt den Ausdruck *some* (*evaluiere_R* (*HatKind*)) (*evaluiere_K* (*And* [*Student, Vegetarier, Weiblich*])).

3. Der Ausdruck 'evaluiere_R (HatKind)' wird berechnet, und das Ergebnis in der sortierten Liste *s_hat_Kind* festgehalten.

4. Der Ausdruck 'evaluiere_K (And [Student, Vegetarier, Weiblich])' wird durch rekursive Anwendung der Funktion 'evaluiere_K' zunächst in *und* [*s_student*, *s_vegetarier*, *s_weiblich*] transformiert. Im Anschluß daran bildet die Funktion 'und' die sortierten Listen 's_student', 's_vegetarier' sowie 's_weiblich' auf die sortierte Schnittmenge der Konzeptmengen 'student', 'vegetarier' sowie 'weiblich' ab.

5. Die Funktion 'some' wird auf die Relation 's_hat_Kind' und die konstruierte Schnittmenge angewendet.

6. Als Ergebnis werden alle Instanzen des Konzepts '(some hat_Kind (and (student, vegetarier, weiblich)))' am Bildschirm ausgegeben.

5.4 Algebraische Simplifikationen

In diesem Abschnitt zeige ich, wie Anfrageausdrücke vom Datentyp Konzeptterm bzw. Relationenterm, die in $\mathcal{ALCX}$ definierte Konzeptterme und Rollenterme darstellen, mittels Algebren simplifiziert werden können. Der Prozess der Simplifikation baut auf den in Abschnitt 4.2.2 präsentierten algebraischen Beschreibungen auf. Dort wurden die folgenden Sachverhalte gezeigt:

- Die in Abschnitt 4.2.1 definierten Konzeptbeschreibungen können als eine Boolesche Algebra betrachtet werden. (Eine Menge von Mengen, die unter den Booleschen Operationen abgeschlossen ist, bildet eine Boolesche Algebra.)

- Die in Abschnitt 4.2.1 definierten Rollenbeschreibungen können als eine Relationenalgebra angesehen werden. (Eine Menge binärer Relationen, die unter den Booleschen und Relationen-theoretischen Operationen abgeschlossen ist, bildet eine Relationenalgebra.)

- Die in Abschnitt 4.2.1 definierten, sich wechselseitig beeinflussenden Konzepte und Rollen können als ein Boolescher Modul angesehen werden. (Ein Boolescher Modul setzt sich aus einer Booleschen Algebra und einer Relationenalgebra zusammen, wobei zwischen den Trägermengen der Algebren die Operation *Peirce Produkt* : definiert ist.)

- Die in Abschnitt 4.2.1 definierten, sich wechselseitig beeinflussenden Konzepte und Rollen können als eine Peirce-Algebra betrachtet werden. (Eine Peirce-Algebra setzt sich aus einer Booleschen Algebra und einer Relationenalgebra zusammen, wobei zwischen den Trägermengen der Algebren die Operationen *Peirce-Produkt* : und *Linke Zylindrifikation* ^{c} definiert sind.)

Aufgrund der oben genannten Sachverhalte kann jedes in $\mathcal{ALCX}$ definierte Konzept (bzw. Rolle) mit einem algebraischen Term assoziiert werden. Ferner kann, wie in Abschnitt 4.2.2 gezeigt, jede im Abschnitt 4.1 gegebene universelle Identität in eine terminologische Identität überführt werden.
Im Rahmen dieser Arbeit liegt dem Prozeß der Simplifikation die Idee zugrunde, daß eine universelle Identität aus zwei äquivalenten Termen besteht, für die unterschiedliche Reduktionsschritte zur Evaluierung benötigt werden. (Ein Term kann durch einen anderen Term ersetzt werden.) Dies wird im folgenden anhand einer Beispielsimplifikation erklärt:

Die universelle Identität

$$(a \cdot (a + b) = a)$$

wird beispielsweise in die assoziierte terminologische Identität

$$((\textbf{and}\ \mathsf{A}\ (\textbf{or}\ \mathsf{A}\ \mathsf{B})) \approx \mathsf{A})$$

überführt. Angenommen man sucht alle Instanzen des Konzeptterms (**and** student (**or** student vegetarier)), so wird, wie aus dem letzten Abschnitt bekannt, dieser Konzeptterm als Anfrageausdruck (**And** Student (**Or** Student Vegetarier)) vom Datentyp Konzeptterm dargestellt. Die Anfrage dafür sieht daher wie folgt aus:

finde_alle (**And** Student (**Or** Student Vegetarier)).

Gemäß der oben aufgeführten terminologischen Identität ((**and** A (**or** A B)) $\approx$ A) sind die beiden Ausdrücke, (**and** student (**or** student vegetarier)) und student semantisch äquivalent. Dies bedeutet, daß die beiden Konzeptterme identische Instanzen besitzen. Somit müßte die unten aufgeführte Gleichung gelten:

finde_alle (**And** Student (**Or** Student Vegetarier)) = *finde_alle* (Student).

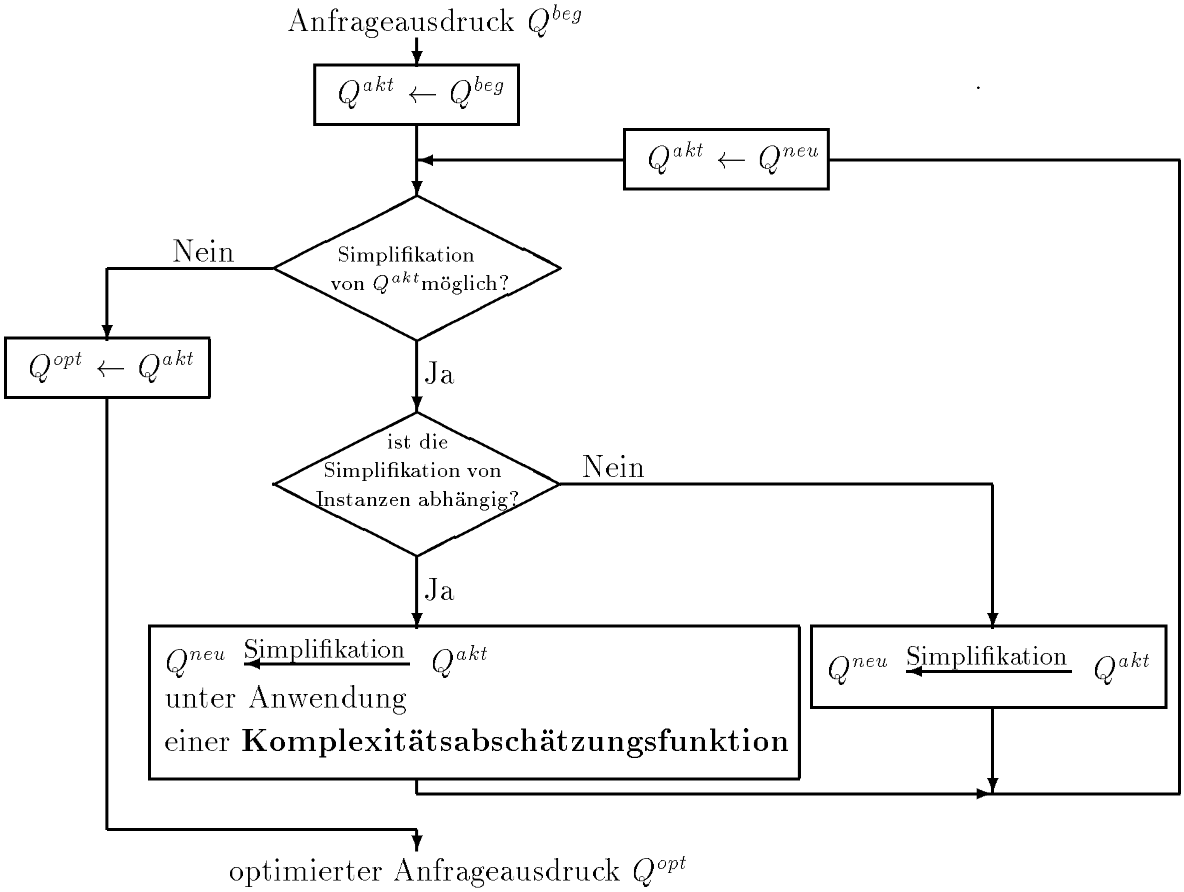
Die Berechnung der Komplexität der Anfrageausdrücke auf der linken bzw. rechten Seite des Gleichheitszeichens zeigt, daß die Anzahl der zur Evaluierung des Ausdrucks (**And** Student (**Or** Student Vegetarier)) erforderlichen Rechenschritte die Anzahl der zur Evaluierung des Ausdrucks (Student) erforderlichen Rechenschritte bei weitem übersteigt. Die Komplexitätsberechnung sieht im einzelnen wie folgt aus (Für die Komplexitätsberechnung verwende ich die Terme der entsprechenden universellen Identität):

> α, β, a, b seien beliebige Elemente (Mengen) aus der Trägermenge einer Booleschen Algebra.
> $k(t)$ sei die zur Evaluierung des Terms t erforderliche Anzahl an Reduktionsschritten.
> $k(\,*(\alpha,\,\beta))$ sei die Anzahl erforderlicher Reduktionsschritte bei der Anwendung der Operation $*$ auf die Mengen α und β.
>
> Es gilt:
> $$\begin{aligned} k(a \cdot (a+b)) &= k(a) + k(a) + k(b) + k(\,+(a,\,b)) + k(\,\cdot(a,\,(a+b))) \\ &> k(a) \end{aligned}$$

Durch Anwendung der universellen Identität $a \cdot (a+b) = a$ kann ein Anfrageausdruck der Form $a \cdot (a+b)$ in einen Anfrageausdruck der Form a simplifiziert werden.

$$a \cdot (a+b) \xrightarrow{\text{Simplifikation}} a$$

In meinem Programm sind Funktionen definiert, die derartige Simplifikationen leisten. Diese Funktionen werden im weiteren Simplifizierer genannt. Indem ein Simplifizierer auf einen Anfrageausdruck angewendet wird, kann ermittelt werden, welche Möglichkeiten zur Simplifikation des Anfrageausdrucks bestehen. Nach welchem Algorithmus dabei eine Simplifikation erfolgt, zeigt das folgende Bild schematisch:



In den folgenden Abschnitten wird genauer erklärt, wie mein Programm bei der Simplifikation eines Anfrageausdrucks vorgeht.

5.4.1 Die Ausprägungen Eq Konzeptterm und Eq Relationenterm der Typklasse Eq

In meinem Programm sind die Simplifizierer mit Hilfe von „pattern matching“ definiert. Die beiden Datentypen Konzeptterm und Relationenterm werden als Ausprägungen der Typklasse **Eq** vereinbart, damit die Anfrageausdrücke vom Datentyp Konzeptterm bzw. Relationenterm auf Gleichheit getestet werden können. (Wie eine Ausprägung in die Klasse Eq aufgenommen wird, habe ich bereits in Abschnitt 2.3 berichtet.)

5.4.2 Komplexitätsberechnungen

Wie bereits erwähnt, können universelle Identitäten als Simplifikationsmittel benutzt werden. Dabei existieren universelle Identitäten, bei denen unabhängig von den Instanzen der Argumentterme bestimmt werden kann, welcher der beiden an der universellen Identität beteiligten Terme effizienter ist. Ein Beispiel hierfür ist die im letzten Abschnitt aufgeführte universelle Identität $a \cdot (a + b) = a$. In solchen Fällen kann eine Simplifikation von einem Term zu einem anderen Term immer unproblematisch realisiert werden, da ein Term unabhängig von den Instanztermen immer eine geringere Komplexität als ein anderer Term besitzt.
Es gibt jedoch auch Fälle, in denen diese Unabhängigkeit nicht gegeben ist. Die Entscheidung, welcher der beiden an der universellen Identität beteiligten Terme für die Berechung effizienter ist, hängt in diesen Fällen von den Instanzen der Argumentterme ab.
Im folgenden vergleiche ich für verschiedene zur Simplifikation einsetzbare universelle Indentitäten die Komplexitäten der jeweils (an der universellen Identität) beteiligten Terme. Anhand dieser Komplexitäts-Gegenüberstellungen ist zugleich erkennbar, bei welchen universellen Identiäten die oben aufgeführte Unabhängigkeit gegeben ist, und bei welchen nicht.

a, b seien im folgenden beliebige Elemente (Mengen) aus der Trägermenge einer Booleschen Algebra; weiterhin seien r, s, t beliebige Relationen aus der Trägermenge einer Relationenalgebra.
$k(\,*(a,\,b))$ (bzw. $k(\,*\,(a)))$ sei die Anzahl erforderlicher Reduktionsschritte bei der Anwendung der Operation $*$ auf die Mengen a und b (bzw. a).
$k(t)$ sei die zur Evaluierung des Terms t erforderliche Anzahl an Reduktionsschritten.
A und B seien zwei beliebige Konzeptterme aus meiner Wissensbasis;
R und S seien zwei beliebige Relationenterme aus meiner Wissensbasis.

universelle Identität $0 \cdot a = a \cdot 0 = 0$

Komplexität $k(0 \cdot a) = k(a \cdot 0) = k(a) + k(0) + k(\,\cdot(a,\,0))$
$\geq k(0)$
unabhängig von Instanztermen

universelle Identität $a \cdot 1 = 1 \cdot a = a$

Komplexität $k(a \cdot 1) = k(1 \cdot a) = k(a) + k(1) + k(\ \cdot(a,\ 1))$
$\geq k(a)$
unabhängig von Instanztermen

universelle Identität $a \cdot (a + b) \quad = a \cdot (b + a) = (a + b) \cdot a$
$= (b + a) \cdot a = a$

Komplexität
$$k(a \cdot (a + b)) = k(a) + k(a) + k(b) + k(\ +(a, b)) + k(\ \cdot(a, a + b)) \geq k(a)$$
unabhängig von Instanztermen

universelle Identität $(a + b) \cdot (a + c) \quad = (b + a) \cdot (a + c) = (a + b) \cdot (c + a)$
$= (b + a) \cdot (c + a)$
$= a + (b \cdot c)$

Komplexität
$$k((a + b) \cdot (a + c)) = k(a) + k(b) + k(a) + k(c) + k(\ +(a,\ b)) + k(\ +(a,\ c)) + k(\ \cdot((a + b),\ (a + c)))$$
$$k(a + (b \cdot c)) = k(a) + k(b) + k(c) + k(\ \cdot(b,\ c)) + k(\ +(a,\ (b \cdot c)))$$

$k((a + b) \cdot (a + c)) \geq k(a + (b \cdot c))$

Begründung: $k(\ \cdot((a + b),\ (a + c))) \geq k(\ \cdot(b,\ c))$
$k(\ +(a,\ b)) + k(\ +(a,\ c)) \geq k(\ +(a,\ (b \cdot c))$
unabhängig von Instanztermen

universelle Identität $r^{\smile\smile} = r$

Komplexität $k(r^{\smile\smile}) = k(r) + k(\smile(r)) + k(\smile(r^{\smile}))$
$\geq k(r)$

unabhängig von Instanztermen

Für die im folgenden aufgeführten universellen Identitäten kann man auf ähnliche Art und Weise zeigen, daß die Entscheidung, welcher der Terme effizienter ist, unabhängig von den Instanztermen ist.

universelle Identitäten: $a \cdot a' = 0$

$$
\begin{aligned}
&a \cdot a = a\\
&a + 1 = 1 + a = 1\\
&a + (a \cdot b) = a + (b \cdot a) = (a \cdot b) + a = (b \cdot a) + a = a\\
&a + a = a\\
&0' = 1 \qquad 1' = 0\\
&a \cdot a' = 0\\
&r \cdot 0 = 0 \cdot r = 0\\
&r \cdot (r + s) = r \cdot (s + r) = (r + s) \cdot r = (s + r) \cdot r = r\\
&(r + s) \cdot (r + t) = (s + r) \cdot (r + t) = (r + s) \cdot (t + r)\\
&= (s + r) \cdot (t + r) = r + (s \cdot t)\\
&r \cdot r = r\\
&r^{\smile} \cdot s^{\smile} \quad = (r \cdot s)^{\smile}\\
&r + (r \cdot s) = r\\
&r + r = r
\end{aligned}
$$

Im folgenden zeige ich für eine Reihe von universellen Identitäten, daß die Entscheidung, welcher der beiden (an der universellen Identität beteiligten) Terme effizienter auswertbar ist, von den Instanztermen abhängig ist.

universelle Identität $(a \cdot b) + (a \cdot c) = (b \cdot a) + (a \cdot c) = (b \cdot a) + (c \cdot a) = (a \cdot b) + (c \cdot a) = a \cdot (b + c)$

Komplexität

$$
\begin{aligned}
k((a \cdot b) + (a \cdot c)) &= k(a) + k(b) + k(a) + k(c) + k(\ \cdot(a,\ b))\\
&\quad + k(\ \cdot(a,\ c)) + k(\ +((a \cdot b),\ (a \cdot c)))\\
k(a \cdot (b + c)) &= k(a) + k(b) + k(c) + k(\ +(b,\ c))\\
&\quad + k(\ \cdot(a,\ (b + c)))
\end{aligned}
$$

abhängig von Instanztermen

universelle Identität

$$
\begin{aligned}
r : a + s : a &= (r + s) : a\\
r : a + r : b &= r : (a + b)
\end{aligned}
$$

Komplexität

$$
\begin{aligned}
k(r : a + s : a) &= k(r) + k(a) + k(s) + k(a) + k(\ :(r,\ a))\\
&\quad + k(\ :(s,\ a)) + k(\ +((r : a),\ (s : a)))\\
k((r + s) : a) &= k(r) + k(s) + k(a) + k(\ +(r,\ s))
\end{aligned}
$$

$$+\ k(\ :((r+s),\ a))$$

abhängig von Instanztermen

$$\begin{aligned} k(r:a+r:b) &= k(r)+k(a)+k(r)+k(b)+k(\ :(r,\ a)) \\ &\quad + k(\ :(r,\ b))+k(\ +((r:a),(r:b))) \\ k(r:(a+b)) &= k(r)+k(a)+k(b)+k(\ +(a,\ b)) \\ &\quad + k(\ :(\mathrm{r},\ (\mathrm{a}+\mathrm{b}))) \end{aligned}$$

abhängig von Instanztermen

universelle Identität $r^{\smile}+s^{\smile} = (r+s)^{\smile}$

Komplexität

$$\begin{aligned} k(r^{\smile}+s^{\smile}) &= k(r)+k(s)+k(\ \smile(r))+k(\ \smile(s)) \\ &\quad + k(\ +(r^{\smile},\ s^{\smile})) \\ k((r+s)^{\smile}) &= k(r)+k(s)+k(\ +(r,\ s))+k(\ \smile(r+s)) \end{aligned}$$

abhängig von Instanztermen

universelle Identität $(r;t)+(s;t) = (r+s);t$

Komplexität

$$\begin{aligned} k(r;t+s;t) &= k(r)+k(s)+k(t)+k(t)+k(\ ;(r,\ t)) \\ &\quad + k(\ ;(s,\ t))+k(\ +(r;t,\ r;t)) \\ k((r+s);t) &= k(r)+k(s)+k(t)+k(\ +(r,\ s)) \\ &\quad + k(\ ;(r+s,\ t)) \end{aligned}$$

abhängig von Instanztermen

universelle Identität $(r;s)+(r;t) = r;(s+t)$

Komplexität

$$\begin{aligned} k(r;s+r;t) &= k(r)+k(s)+k(r)+k(t)+k(\ ;(r,\ s)) \\ &\quad + k(\ ;(r,\ t))+k(\ +(r;s,\ r;t)) \\ k(r;(s+t)) &= k(r)+k(s)+k(t)+k(\ +(s,\ t)) \\ &\quad + k(\ ;(r,\ s+t)) \end{aligned}$$

abhängig von Instanztermen

universelle Identität $s^{\smile};r^{\smile} = (r;s)^{\smile}$

Komplexität

$$\begin{aligned} k(s^{\smile};r^{\smile}) &= k(s)+k(r)+k(\ \smile(s))+k(\ \smile(r)) \\ &\quad + k(\ ;(s^{\smile},\ r^{\smile})) \\ k((r;s)^{\smile}) &= k(r)+k(s)+k(\ ;(r,\ s))+k(\ \smile(r;s)) \end{aligned}$$

abhängig von Instanztermen

Im weiteren veranschauliche ich für die erste und dritte universelle Identität, daß die Entscheidung, welcher der beiden an einer universellen Identität beteiligten Terme effizienter auswertbar ist, von den Instanztermen abhängig sein kann. Die Abhängigkeit wird jeweils mittels zweier Beispiele gezeigt. Für alle übrigen oben aufgeführten universellen Identitäten kann diese Abhängigkeit auf die gleiche Art bewiesen werden.

Die erste der oben aufgeführten universellen Identitäten lautet

$$(a \cdot b) + (a \cdot c) = a \cdot (b + c)$$

und ist in die assoziierte terminologische Identität

$$(\textbf{or}\ (\textbf{and}\ \mathsf{A}\ \mathsf{B})\ (\textbf{and}\ \mathsf{A}\ \mathsf{C})) \approx (\textbf{and}\ \mathsf{A}\ (\textbf{or}\ \mathsf{B}\ \mathsf{C}))$$

überführbar. Daß der Term auf der linken wie auf der rechten Seite der effizienter auswertbare sein kann, zeige ich anhand von Anfragen an meine Wissensbasis. Diese Anfragen werten die Instanzen von Konzepten aus, die an der oben genannten terminologischen Identität beteiligt sind.

Beispiel 1 (*Der rechte Term ist effizienter auswertbar*):

Term	Seite	Instanz
$(a \cdot b) + (a \cdot c)$	links	(Or [And [Weiblich, Student], And [Weiblich, Vegetarier]])
$a \cdot (b + c)$	rechts	(And [Weiblich, Or [Student, Vegetarier]])

Es gelten folgende Komplexitäten:

? finde_alle (Or [And [Weiblich, Student], And [Weiblich, Vegetarier]])
[“Gani”, “Sima”, “Siville”]
(287 reductions, 535 cells)

? finde_alle (And [Weiblich, Or [Student, Vegetarier]])
[“Gani”, “Sima”, “Siville”]
(247 reductions, 460 cells)

Für die Instanzterme '(Or [And [Weiblich, Student], And [Weiblich, Vegetarier]])' und '(And [Weiblich, Or [Student, Vegetarier]])' der universellen Identität $(a \cdot b) + (a \cdot c) = a \cdot (b + c)$ gilt somit $k((a \cdot b) + (a \cdot c)) > k(a \cdot (b + c))$. Dies ist gleichbedeutend damit, daß der rechte Term effizienter auswertbar ist.

Beispiel 2 (*Der linke Term ist effizienter auswertbar*):

Term	Seite	Instanz
$(a \cdot b) + (a \cdot c)$	links	(Or [And [Student, Vegetarier], And [Student, Weiblich]])
$a \cdot (b + c)$	rechts	(And [Student, Or [Vegetarier, Weiblich]])

Es gelten folgende Komplexitäten:

? finde_alle (Or [And [Student, Vegetarier], And [Student, Weiblich]])
["Gani"]
(240 reductions, 408 cells)

? finde_alle (And [Student, Or [Vegetarier, Weiblich]])
["Gani"]
(311 reductions, 532 cells)

Für die Instanzterme '(Or [And [Student, Vegetarier], And [Student, Weiblich]])' und '(And [Student, Or [Vegetarier, Weiblich]])' der universellen Identität $(a \cdot b) + (a \cdot c) = a \cdot (b + c)$ gilt somit $k((a \cdot b) + (a \cdot c)) < k(a \cdot (b + c))$. Dies ist gleichbedeutend damit, daß der linke Term effizienter auswertbar ist.

Die nächsten beiden Beispiele beziehen sich auf die dritte universelle Identität. Die dritte universelle Identitäte lautet

$$r^{\smile} + s^{\smile} = (r + s)^{\smile}$$

und ist in die assoziierte terminologische Identität

(or (inverse R) **(inverse** S)) $\approx$ **(inverse (or** R S)) überführbar. Daß der Term auf der linken wie auf der rechten Seite der effizienter auswertbare sein kann, wird wie bei der zuletzt behandelten universellen Identität anhand von Anfragen an meine Wissensbasis gezeigt. (Aus Platzgründen habe ich im Gegensatz zur zuletzt behandelten universellen Identität darauf verzichtet, die Instanzmenge aufzuführen.):

Beispiel 1 (*Der rechte Term ist effizienter auswertbar*):

Term	Seite	Instanz
$r^{\smile} + s^{\smile}$	links	(ROr [(Inverse HatKind), (Inverse HatKind)])
$(r + s)^{\smile}$	rechts	(Inverse (ROr [HatKind, HatKind]))

Es gelten folgende Komplexitäten:

? finde_Relation (ROr [(Inverse HatKind), (Inverse HatKind)])
(5693 reductions, 10444 cells)

? finde_Relation (Inverse (ROr [HatKind, HatKind]))
(3359 reductions, 6332 cells)

Für die Instanzterme '(ROr [(Inverse HatKind), (Inverse HatKind)])' und '(Inverse (ROr [HatKind, HatKind]))' der universellen Identität $r^{\smile} + s^{\smile} = (r + s)^{\smile}$ gilt somit $k(r^{\smile} + s^{\smile}) > k((r + s)^{\smile})$. Dies ist gleichbedeutend damit, daß der rechte Term effizienter auswertbar ist.

Beispiel 2 (*Der linke Term ist effizienter auswertbar*):

Term	Seite	Instanz
$r^{\smile} + s^{\smile}$	links	(ROr [(Inverse HatKind), (Inverse IstVerheiratet)])
$(r + s)^{\smile}$	rechts	(Inverse (ROr [HatKind, IstVerheiratet]))

Es gelten folgende Komplexitäten:

? finde_Relation (ROr [(Inverse HatKind), (Inverse IstVerheiratet)])
(5103 reductions, 9717 cells)

? finde_Relation (Inverse (ROr [HatKind, IstVerheiratet]))
(7870 reductions, 14417 cells)

Für die Instanzterme '(ROr [(Inverse HatKind), (Inverse IstVerheiratet)])' und '(Inverse (ROr [HatKind, IstVerheiratet]))' der universellen Identität $r^{\smile} + s^{\smile} = (r+s)^{\smile}$ gilt somit $k(r^{\smile} + s^{\smile}) < k((r+s)^{\smile})$. Dies ist gleichbedeutend damit, daß der linke Term effizienter auswertbar ist.

5.4.3 Komplexitätsabschätzungen

Wie im letzten Abschnitt gesehen, gibt es universelle Identitäten, bei denen die Entscheidung, welcher der beiden an der universellen Identität beteiligten Terme effizienter auswertbar ist, von den Instanzen der Argumentterme abhängt. In solchen Fällen kann die Entscheidung, welcher Term effizienter ist, mittels einer Komplexitätsabschätzung getroffen werden. Das von mir entwickelte Programm leistet derartige Komplexitätsabschätzungen und simplifiziert einen Ausdruck nur dann, wenn dies effizient ist. Die Funktion, die zur Abschätzung der Komplexität bei Anfragen vom Datentyp 'Konzeptterm' benutzt wird, ist *komplexität_K*. Die entsprechende Funktion, die zur Abschätzung der Komplexität bei Anfragen vom Datentyp Relationenterm benutzt wird, ist *komplexität_R*. Die Typen der beiden Funktionen sind wie folgt vereinbart:

```
komplexität_K :: Konzeptterm ⟶ (Float, Float)

komplexität_R :: Relationenterm ⟶ (Float, Float)
```

Die Komponenten der als Funktionsergebnis zurückgegebenen Tupel besitzen folgende Bedeutung:
Die erste Komponente gibt die Anzahl bzw. abgeschätzte Anzahl der Instanzen des durch den Anfrageausdruck beschriebenen Konzepts (bzw. Rolle) wieder. Die zweite Komponente gibt eine Abschätzung dafür, wieviele Reduktionsschritte zur Berechnung der Instanzen des durch den Anfrageausdruck beschriebenen Konzepts (bzw. Rolle) im Zusammenhang mit den im Programm definierten Evaluationsfunktionen benötigt werden.

5.4.3.1 Arbeitsweise der Funktionen 'komplexität_K' und 'komplexität_R'

Im folgenden erkläre ich, wie bei der Auswertung der Funktion 'komplexität_K' bzw. 'komplexität_R' vorgegangen wird, sofern das übergebene Argument ein *primitives* Konzept bzw. eine *primitive* Rolle darstellt:
Die erste Komponente gibt, wie bereits gesagt, die Anzahl der Instanzen des als Argument übergebenen Konzepts (bzw. Rolle) wieder. Für ein primitives Konzept bzw. eine primitive Rolle gilt, daß die Anzahl der Instanzen durch die Länge der Liste gegeben wird, die das Konzept bzw. die Rolle repräsentiert. Zur Berechnung der Länge einer Liste ist Funktion 'length_Float' (`length_Float :: [a] ⟶ Float`) definiert. Dementsprechend wird beispielsweise die Anzahl der Instanzen des Konzepts 'person' durch die folgende *Variablendeklaration* ermittelt:

```
l_person = length_Float person
```

Die Berechnung der Länge einer Liste in Gofer geschieht wegen der Existenz von lazy evaluation nur einmal (vorausgesetzt, die Programmdatei wird nicht erneut geladen). Aus diesem Grund habe ich die Anzahl der benötigten Rechenschritte mit 0 abgeschätzt. Die folgenden beiden Tabellen geben, getrennt nach den Funktionen 'komplexität_K' und 'komplexität_R', wieder, für welche Argumente welche Ergebniswerte zurückgegeben werden:

Funktion komplexität_K	
Argument	Ergebnis-Tupel
Basisbottom	(0.0, 0.0)
Personen	(l_person, 0.0)
Student	(l_student, 0.0)
Weiblich	(l_maennlich, 0.0)
Maennlich	(l_weiblich, 0.0)
Vegetarier	(l_vegetarier, 0.0)

Funktion komplexität_R	
Argument	Ergebnis-Tupel
Relationenbottom	(0.0, 0.0)
HatKind	(l_hat_Kind, 0.0)
IstKusin	(l_ist_Kusin, 0.0)
IstVerheiratet	(l_ist_verheiratet, 0.0)
IstLehrer	(l_ist_Lehrer, 0.0)

Im weiteren erkläre ich, wie bei der Auswertung der Funktionen 'komplexität_K' und 'komplexität_R' vorgegangen wird, sofern die Argumente zusammengesetzte Konzepte bzw. Rollen darstellen. Die Ermittlung der Komplexität erfolgt in diesem Fall grundsätzlich rekursiv:
Zur Beschreibung zusammengesetzter Konzepte bzw. Rollen werden terminologische Operatoren verwendet. Die terminologischen Operatoren **and**, **or**, $\cdots$ werden durch die Konstruktorfunktionen 'And', 'Or', $\cdots$ aus den Deklarationen der Datentypen 'Konzeptterm' und 'Relationenterm' dargestellt. Zur Realisierung der Interpretation der terminologischen Operatoren sind, wie in Abschnitt 5.3 gezeigt, die Evaluationsfunktionen 'und', 'oder', $\cdots$ definiert. Zur Abschätzung der Komplexität der Evaluationsfunktionen sind entsprechende Funktionen definiert. Beispielsweise ist die Funktion '*k_und*' zur Abschätzung der Komplexität der Evaluationsfunktion '*und*' definiert. Die Funktionen 'komplexität_K' und 'komplexität_R' bilden jeweils eine Konstruktorfunktion aus den Deklarationen der Datentypen 'Konzeptterm' und 'Relationenterm' ('And', 'Or', $\cdots$) auf eine Funktion ab, die zur Abschätzung der Komplexität der zur Konstruktorfunktion gehörigen Evaluationsfunktion dient. Beispielsweise bildet die Funktion 'komplexität_K' die Konstruktorfunktion '*And*' auf die Funktion '*k_und*' ab, die die Komplexität der Evaluationsfunktion '*und*' abschätzt. Die folgenden Tabellen geben wieder, welche Komplexitätsabschätzungsfunktion bei welcher Konstruktorfunktion aufgerufen wird:

Funktion komplexität_K	
Konstruktorfunktion	Abschätzungsfunktion
And	k_und
Or	k_oder
Not	k_nicht
Some	k_some
All	k_all

Funktion komplexität_R	
Konstruktorfunktion	Abschätzungsfunktion
RAnd	k_und
ROr	k_oder
Inverse	k_inverse
Compose	k_compose
Restrict	k_range_restrict

Im folgenden wird anhand des Beispielausdrucks

And [Student, Or [Vegetarier, Weiblich]]

gezeigt, wie die Komplexitätsabschätzungsfunktion 'komplexität_K' arbeitet:

1. Die Komplexitätsabschätzungsfunktion 'komplexität_K' wird auf das Argument (And [Student, Or [Vegetarier, Weiblich]]) angewendet.

2. Der Ausschnitt `komplexität_K (And (a:as)) = k_und (komplexität_K a) : (komplexität_K as)` der Definition der Funktion 'komplexität_K' erzeugt den Ausdruck 'k_und [komplexität_K (Student), komplexität_K (Or [Vegetarier, Weiblich])]'.

3. Der Ausdruck 'komplexität_K (Student)' wird berechnet und das Ergebnis (l_student, 0) festgehalten.

4. Der Ausschnitt `komplexität_K (Or (a:as)) = k_oder (komplexität_K a) : (komplexität_K as)` der Definition der Funktion 'komplexität_K' transformiert zunächst den Ausdruck 'komplexität_K (Or [Vegetarier, Weiblich])' in

den Ausdruck 'k_oder [komplexität_K (Vegetarier), komplexität_K (Weiblich)]'. Der Ausdruck 'komplexität_K (Vegetarier)' wird berechnet und das Ergebnis (l_vegetarier, 0) festgehalten. Der Ausdruck 'komplexität_K (Weiblich)' wird berechnet und das Ergebnis (l_weiblich, 0) festgehalten.

5. Die Funktion 'k_oder' wird auf die Liste von Tupeln '[(l_vegetarier, 0), (l_weiblich, 0)]' angewendet. Sie gibt ein Tupel vom Typ `(Float, Float)` zurück. Dieses Tupel sei im folgenden mit (k_1, k_2) bezeichnet.

6. Die Funktion 'k_und' wird auf die Liste von Tupeln [(l_student, 0), (k_1, k_2)] angewendet. Das zurückgegebene Tupel enthält das Ergebnis der Komplexitätsabschätzung.

5.4.3.2 Definition der Funktionen k_oder, k_und, k_nicht, $\cdots$

Im folgenden begründe ich die Gestalt der Definitionen der Abschätzungsfunktionen 'k_oder', 'k_und', 'k_nicht', $\cdots$. Dabei beschränke ich mich auf eine Auswahl; die Gestalt aller übrigen im Programm erscheinenden Funktionen kann auf ähnliche Weise begründet werden.

- *Komplexitätsabschätzung der Funktion '***oder***'*

Bevor ich eine Komplexitätsabschätzung für die Funktion 'oder' vorstelle (auf dieser Abschätzung basiert die Gestalt der in Abschnitt 5.4.3.1 erwähnten Funktion 'k_oder') erläutere ich zunächst, wie die Funktion 'oder' den Schnitt zwischen den Elementen einer als Argument übergebenen Liste von aufsteigend *sortierten* Mengen $[A_1, A_2, \cdots, A_n]$ bildet.
Um die Menge $A_1 \cup A_2 \cup \cdots \cup A_n$ zu ermitteln, wird zunächst die Vereinigung der beiden Mengen A_1 und A_2 gebildet. Der letztere Vereinigung werde mit V_1
$(V_1 = A_1 \cup A_2)$ bezeichnet. Im Anschluß daran wird für alle Mengen A_i mit $3 \leq i \leq n$ die Vereinigung V_{i-1} gemäß der Regel $V_{i-1} = V_{i-2} \cup A_i$ gebildet. Die so erzeugte Menge V_{n-1} ist identisch der Vereinigung $A_1 \cup A_2 \cup \cdots \cup A_n$.

Für die im folgenden aufgeführte Komplexitätsabschätzung der Funktion 'oder' ist es erforderlich, für jedes Element A_i $(1 \leq i \leq n)$ einer als Argument an die Funktion 'oder' übergebenen Liste von Mengen $[A_1, A_2, \cdots, A_n]$ ein Schätzmaß für die Größe der Menge A_i sowie ein Schätzmaß für die Anzahl erforderlicher Rechenschritte, die zur Ermittlung der Menge A_i erforderlich sind, als ermittelt vorauszusetzen. (Im Programm leistet die Ermittlung dieser Werte die Funktion 'komplexitaet_K'.) Die benötigten Schätzmaße seien durch eine Liste $[(l_1, k_1), \cdots, (l_n, k_n)]$ gegeben, wobei für jedes Tupel gelte, daß

l_i die abgeschätzte Größe der Menge A_i und

k_i die abgeschätzte Anzahl der zur Ermittlung der Menge A_i
erforderlichen Rechenschritte

sei.

1. **Komplexitätsabschätzung für die Berechnung bis zu $A_1 \cup A_2$**

Bevor ich auf den allgemeinen Fall eingehe, beschränke ich mich auf den Fall, daß die Argumentliste nur aus zwei Mengen A_1 und A_2 besteht.
Grundgedanke der Komplexitätsabschätzung ist die Durchschnittsbildung zwischen den Komplexitäten im worst-case (Situation, in der die *maximale Vergleichsanzahl* benötigt wird) und im best-case (Situation, in der die *minimale Vergleichsanzahl* benötigt wird).
Die benötigten Schätzmaße für die Mengen A_1 und A_2 werden, wie bereits oben erwähnt, durch die Tupel (l_1, k_1) und (l_2, k_2) gegeben. Die sortierte Liste der Elemente der Menge A_1 sei $[x_1, x_2, ..., x_{l_1}]$. Die sortierte Liste der Elemente der Menge A_2 sei $[y_1, y_2, ..., y_{l_2}]$.

1.1 Komplexität im worst-case für $A_1 \cup A_2$

Den Kern der Funktion 'oder' bildet die Funktion 'merge_oder', die die Vereinigung *genau* zweier Mengen berechnet. Die Funktion 'merge_oder' ist wie folgt definiert:

```
merge_oder :: [String] → [String] → [String]
merge_oder [] ys = ys
merge_oder xs [] = xs
merge_oder (x:xs) (y:ys)
                    | x == y = x: merge_oder xs ys
                    | x > y = y: merge_oder (x:xs) ys
                    | otherwise = x: merge_oder xs (y:ys)
```

Die Funktion 'merge_oder' besitzt eine rekursive Definition und *verarbeitet* bei jedem Selbst-Aufruf mindestens ein Element der beiden an den Rekursionsauslösenden Aufruf als Argument übergebenen Listen. Im Rahmen der folgenden Abschätzung werden als Komplexitätsmaß zunächst die Vergleiche gezählt, die ein Aufruf der Funktion 'merge_oder' zur Folge hat.

Lemma 5.1 *Gilt $l_1 = 0$ oder $l_2 = 0$, beträgt die Anzahl notwendiger Vergleiche zur Bildung der Vereinigungsmenge $A_1 \cup A_2$ im worst-case 0.*

Beweis: Die mit 5) bis 7) nummerierten Schritte der Definition der Funktion merge_oder werden im Falle von $l_1 = 0$ oder $l_2 = 0$ nicht erreicht. In beiden Fällen werden somit keine Vergleiche durchgeführt. qed.

Gilt $l_1 \geq 1$ und $l_2 \geq 1$, kann die Komplexität im worst-case wie folgt abgeschätzt werden:

Lemma 5.2 *Es gelte: $l_1 \geq 1$ und $l_2 \geq 1$. Die Abarbeitung der beiden Listen xs und ys sei bis zu den beiden Elementen x_i $(1 \leq i \leq l_1)$ und y_j $(1 \leq j \leq l_2)$ erfolgt. (d. h. beim nächsten Aufruf, bzw. rekursiven Selbstaufruf, der Funktion 'merge_oder' werden die beiden Elemente x_i und y_j in dem bei Zeile 4) beginnenden Definitionsabschnitt gegenübergestellt). Folgende Fälle können unterschieden werden:*

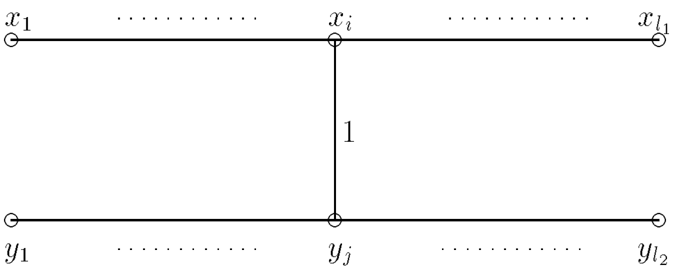
(i) Gilt $x_i = y_j$, erfordert die Gegenüberstellung beider Elemente einen *Vergleich; die beiden Elemente x_i und y_j werden verarbeitet.*

(ii) Gilt $x_i > y_j$, erfordert die Gegenüberstellung beider Elemente zwei *Vergleiche; das Element y_j wird verarbeitet.*

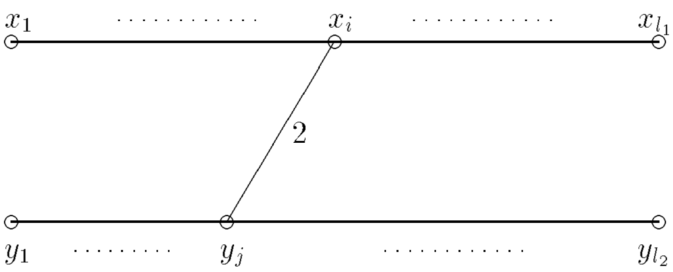
(iii) Gilt $x_i < y_j$, erfordert die Gegenüberstellung beider Elemente zwei *Vergleiche; das Element x_i wird verarbeitet.*

Beweis: Die Aussagen für alle drei Fälle können direkt an der Funktionsdefinition abgelesen werden. Jeder der drei Fälle wird im folgenden mittels einer Abbildung illustriert:

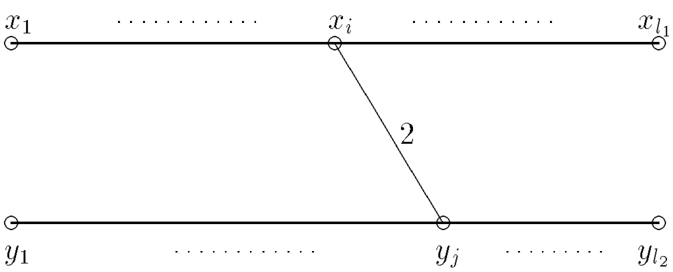
Fall 1: $(x_i = y_j)$



Fall 2: $(x_i > y_j)$



Fall 3: $(x_i < y_j)$



Satz 5.1 *Für zwei beliebige nicht leere Mengen A_1 und A_2 ($l_1 \geq 1$ und $l_2 \geq 1$) mit k vielen gemeinsamen Elementen gilt:*
Die zur Bildung der Vereinigung beider Mengen erforderliche Zahl von Vergleichen beträgt `maximal`

(a) $(k + (l_1 - k) \cdot 2 + (l_2 - k) \cdot 2)$ *Vergleiche, falls* $x_{l_1} = y_{l_2}$ *bzw.*

(b) $(k + (l_1 - k) \cdot 2 + (l_2 - k) \cdot 2 - 2)$ *Vergleiche sonst.*

Beweis: Der von der Funktion 'merge_oder' zur Bildung der Vereinigung zweier Mengen benötigte Aufwand an Vergleichen setzt sich aus der Summe der folgenden drei Größen zusammen:

(a) Die Verarbeitung der k identischen Elemente aus A_1 und A_2 erfordert *genau* k viele Vergleiche. (siehe Lemma 5.2 Teil i))

(b) Für die Verarbeitung der verbleibenden $(l_1 - k)$ vielen Elemente aus A_1 sind *maximal* $(l_1 - k) \cdot 2$ viele Vergleiche erforderlich. (siehe Lemma 5.2 Teil iii))

(c) Für die Verarbeitung der verbleibenden $(l_2 - k)$ vielen Elemente aus A_2 sind *maximal* $(l_2 - k) \cdot 2$ viele Vergleiche erforderlich. (siehe Lemma 5.2 Teil ii))

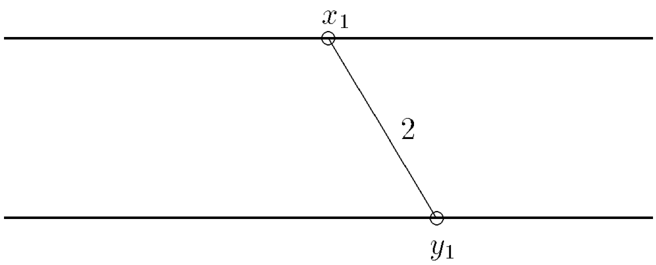
Eine Besonderheit gilt hinsichtlich des Rekursionsabbruchs. Wird dieser durch die Zeilen 2) oder 3) der Funktionsdefinition von 'merge_oder' verursacht, wird das letzte Element aus xs oder ys nicht verarbeitet, was der Einsparung von zwei Vergleichen entspricht. Dieser Fall ist *immer* dann gegeben, wenn die beiden letzten Elemente x_{l_1} und y_{l_2} *verschieden* sind und wird im zweiten der obigen Formelausdrücke berücksichtigt. qed.

Gemäß Satz 5.1 sind für gegebenes l_1 und l_2 maximal $(l_1 \cdot 2 + l_2 \cdot 2 - 2)$ viele Vergleiche erforderlich, um die Vereinigung zweier Mengen zu bilden. Dieses Maximum wird erreicht, wenn $k = 0$ gilt (die beiden Mengen A_1 und A_2 disjunkt sind). Im weiteren beweise ich, daß die durch $(l_1 \cdot 2 + l_2 \cdot 2 - 2)$ gegebene obere Schranke scharf ist.

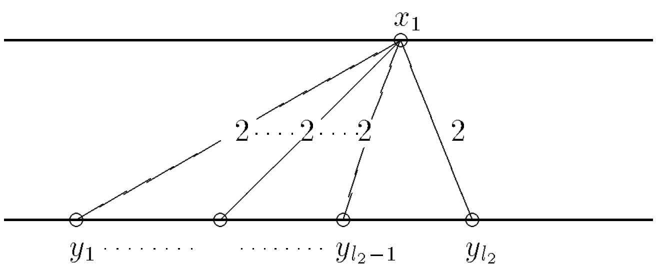
Satz 5.2 *Für beliebiges l_1 und l_2 ($l_1 \geq 1$ und $l_2 \geq 1$) existieren zwei Mengen A_1 und A_2 mit $\mid A_1 \mid = l_1$ und $\mid A_2 \mid = l_2$, so daß mindestens $(l_1 \cdot 2 + l_2 \cdot 2 - 2)$ viele Vergleiche benötigt werden, um die Vereinigung beider Mengen mittels der Funktion 'merge_oder' zu bilden. Die beiden Mengen A_1 und A_2 seien dabei durch zwei aufsteigend sortierte Listen xs und ys repräsentiert.*

Beweis:

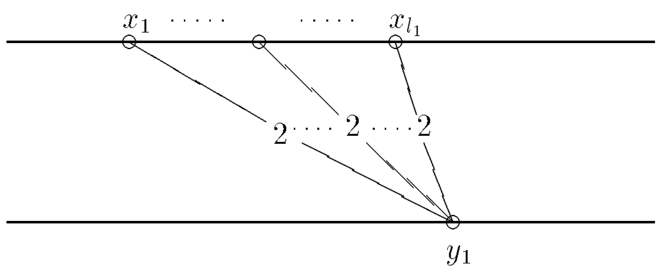
(i) $l_1 = 1$ und $l_2 = 1$
Für zwei beliebige einelementige Mengen mit $x_1 > y_1$ gilt, daß $2(= 1 \cdot 2 + 1 \cdot 2 - 2)$ viele Vergleiche zur Bildung der Vereinigung benötigt werden. Das folgende Bild verdeutlicht dies:



(ii) $l_1 = 1$ und $l_2 \geq 1$
Zu einer beliebigen einelementigen Menge A_1 werde eine Menge A_2 mit $\mid A_2 \mid = l_2$ konstruiert, so daß $y_{l_2-1} < x_1$ und $x_1 > y_{l_2}$ gilt. An der Definition der Funktion 'merge_oder' kann abgelesen werden, daß genau $l_1 \cdot 2 + l_2 \cdot 2 - 2$ viele Vergleiche benötigt, um die Vereinigung beider Mengen zu bilden. Das folgende Bild verdeutlicht dies:



(iii) $l_1 \geq 1$ und $l_2 = 1$
Zu einer beliebigen einelementigen Menge A_2 werde eine Menge A_1 mit $\mid A_1 \mid = l_1$ konstruiert, so daß $y_1 > x_{l_1}$ gilt. An der Definition der Funktion 'merge_oder' kann abgelesen werden, daß genau $l_1 \cdot 2 + l_2 \cdot 2 - 2$ viele Vergleiche benötigt, um die Vereinigung beider Mengen zu bilden. Das folgende Bild verdeutlicht dies:



(iv) $l_1 \geq 2$ und $l_2 \geq 2$

Erfüllen zwei beliebige Mengen A_1 und A_2 mit $\mid A_1 \mid = l_1$ und $\mid A_2 \mid = l_2$ die drei folgenden Bedingungen, werden von der Funktion 'merge_oder' genau $l_1 \cdot 2 + l_2 \cdot 2 - 2$ viele Vergleiche benötigt, um die Vereinigung beider Mengen zu bilden:

i. $\{x_1, x_2, ..., x_{l_1}\} \cap \{y_1, y_2, ..., y_{l_2}\} = \phi$.

ii. $x_{l_1} < y_{l_2}$

iii. $y_{l_2-1} < x_{l_1}$

Die Aussage kann anhand der Definition der Funktion 'merge_oder' überprüft werden. qed.

Um bei der Berechnung der Komplexität den Aspekt der Zählung der Reduktionsschritte zu berücksichtigen, habe ich die abgeschätzte Vergleichszahl mit einem Scalar gewichtet.

Die Elemente der Konzepte und Rollen meiner terminologischen Datenbank werden mittels Zeichenketten dargestellt. In Gofer hängt die Anzahl der Reduktionsschritte, die zum Vergleich zweier Zeichenketten benötigt werden, von der Position des ersten differenten Zeichens und der durchgeführten Operation ('<', '>', '==') ab. Unter Kenntnis dieser Zusammmenhänge habe ich jeden Vergleich mit dem abgeschätzten Scalar 8.8 gewichtet. Den Scalar habe ich ermittelt, indem ich den Durchschnitt der folgenden zwei Werte gebildet habe (In der Funktion 'merge_oder' werden nur die beiden Operationen '==' und '>' benötigt.):

(a) Durchschnittliche Anzahl von Reduktionsschritten zur einmaligen Durchführung der Operation '==': 4.9

(b) Durchschnittliche Anzahl von Reduktionsschritten zur einmaligen Durchführung der Operation '>': 12.6

Alle zwei Werte sind im Bezug auf meine Wissensbasis gebildet und müssen bei einer Veränderung der Wissensbasis adaptiert werden.

1.2 Komplexität im best-case für $A_1 \cup A_2$

Lemma 5.3 *Gilt für zwei beliebigen Mengen A_1 und A_2 mit $\mid A_1 \mid = l_1$ und $\mid A_2 \mid = l_2$ $l_1 = 0$ oder $l_2 = 0$, beträgt die Anzahl notwendiger Vergleiche zur Bildung der Vereinigungsmenge $A_1 \cup A_2$ im best-case 0.*

Beweis: Die mit 5) bis 7) nummerierten Schritte der Definition der Funktion 'merge_oder' werden im Falle von $l_1 = 0$ oder $l_2 = 0$ nicht erreicht. In beiden Fällen werden somit keine Vergleiche durchgeführt. qed.

Satz 5.3 *Zur Bildung der Vereinigung zweier nicht leeren Mengen A_1 und A_2 sind im best-case min $(\mid A_1 \mid, \mid A_2 \mid)$ viele Vergleiche erforderlich.*

Beweis: Anhand von Lemma 5.2 kann abgelesen werden, daß die geringste Vergleichsanzahl zur Bildung der Vereinigung zweier Mengen A_1 und A_2 benötigt wird, sofern $A_1 \subseteq A_2$ oder $A_1 \supseteq A_2$ gilt. qed.

Die Komplexität im best-case wird wie die Komplexität im worst-case mit dem Scalar 8.8 gewichtet.

1.3 Komplexitätsabschätzung für $A_1 \cup A_2$

Als Komplexität für die Vereinigungsbildung wird, wie bereits gesagt, der Durchschnitt zwischen den Komplexitäten im worst-case und im best-case genommen. Somit ergibt sich als Komplexität für die Berechung von $A_1 \cup A_2$:

$$\begin{cases} 0 & : \; falls \quad l_1 = 0 \quad oder \quad l_2 = 0 \\ \left(\frac{((l_1 \cdot 2 + l_2 \cdot 2 - 2) + min(l_1, l_2))}{2}\right) \cdot 8.8 & : \; sonst \end{cases}$$

1.4 Komplexitätsabschätzung bis zur Berechnung von $A_1 \cup A_2$

Bis zu diesem Zeitpunkt wurde in die Komplexitätsabschätzung nicht der zur Ermittlung der Mengen A_1 und A_2 erforderliche Rechenaufwand mit einbezogen. k_1 sei der zur Berechnung der Menge A_1 notwendige Rechenaufwand; entsprechend sei k_2 der zur Berechnung der Menge A_2 notwendige Rechenaufwand. Die Werte k_1 und k_2 werden in meinem Programm durch einen (rekursiven) Aufruf der Funktion 'komplexität_K' ermittelt und können somit als gegeben angesehen werden. Die Werte k_1 und k_2 werden zu dem bestehenden Komplexitätsmaß hinzuaddiert, womit sich als Komplexität bis zur Berechnung von $A_1 \cup A_2$ ergibt:

$$\begin{cases} k_1 + k_2 & : \ falls \quad l_1 = 0 \quad oder \quad l_2 = 0 \\ k_1 + k_2 + \left(\frac{((l_1 \cdot 2 + l_2 \cdot 2 - 2) + min(l_1, l_2))}{2}\right) \cdot 8.8 & : \ sonst \end{cases}$$

Die Funktion 'k_oder' gibt als Ergebnis ein Tupel zurück, dessen erste Komponente ein Schätzmaß für die Größe der Menge $A_1 \cup A_2$ angibt. Die Menge $A_1 \cup A_2$ kann über $max(l_1, l_2)$ bis $(l_1 + l_2)$ viele Elemente verfügen. Als Schätzmaß für die Größe der Vereinigungsmenge nehme ich den folgenden Wert:

$$max(l_1, l_2) + \frac{min(l_1, l_2)}{2}$$

Damit die Genauigkeit des Komplexitätsmaßes erhöht wird, habe ich weitere im Bezug auf die Anzahl der Reduktionsschritte bedeutsame Faktoren in das Komplexitätsmaß mit einbezogen. Als sehr kostenintensiv hinsichtlich der Reduktionsschritte hat sich das Hinzufügen eines Elementes zu einer Liste mittels des Operators ':' erwiesen. Die Elemente der Konzepte und Rollen meiner terminologischen Datenbank werden, wie bereits gesagt, mittels Zeichenketten dargestellt. Um eine Zeichenkette zu einer Liste hinzuzufügen, werden so viele Reduktionsschritte benötigt, wie das hinzuzufügende Element Zeichen besitzt. Deshalb habe ich das Hinzufügen einer Zeichenkette zu einer Liste mit dem Scalar 6.3 gewichtet. Den Scalar habe ich gebildet, indem ich die Summe der Länge aller Zeichenketten meiner terminologischen Datenbank durch die Anzahl der Zeichenketten dividiert habe. Der Scalar muß, wie der für die Vergleichsgewichtung, bei einer Veränderung der Wissensbasis adaptiert werden.
Als die voraussichtliche Anzahl an Anwendungen des Operators ':' habe ich die vorausichtliche Mächtigkeit der zu berechnenden Vereinigungsmenge $A_1 \cup A_2$

verwendet. Die voraussichtliche Anzahl an Anwendungen des Operators ':' wird deshalb mit

$$max(l_1, l_2) + \frac{min(l_1, l_2)}{2}$$

abgeschätzt.

Insgesamt hat damit das von der Funktion 'merge_oder' zurückgegebene Tupel folgendes Aussehen:

$$\begin{cases} \big(l1, \quad k_1 + k_2\big) & : \quad falls \quad l_2 = 0 \\ \big(l2, \quad k_1 + k_2\big) & : \quad falls \quad l_1 = 0 \\ \Big(max(l_1, l_2) + \frac{min(l_1,l_2)}{2}, \quad k_1 + k_2 & \\ \quad +(\frac{((l_1 \cdot 2 + l_2 \cdot 2 - 2) + min(l_1,l_2))}{2}) \cdot 8.8 & \\ \quad +(max(l_1, l_2) + \frac{min(l_1,l_2)}{2}) \cdot 6.3\Big) & : \quad sonst \end{cases}$$

2. **Komplexitätsabschätzung bis zur Berechnung von $A_1 \cup A_2 \cup \cdots \cup A_n$**

Die Vorgehensweise bei der Abschätzung der Berechnungskomplexität der Menge $A_1 \cup A_2 \cup \cdots \cup A_n$ ist an der Definition der Funktion 'merge_oder' orientiert. Zunächst wird gemäß des zuletzt beschriebenen Verfahrens ein Tupel t_1 ermittelt, dessen erste Komponente die Größe der Vereinigungsmenge $V_1 = A_1 \cup A_2$ und dessen zweite Komponente den zur Bildung der Vereinigungsmenge $V_1 = A_1 \cup A_2$ notwendigen Rechenaufwand angibt. Im Anschluß daran wird für alle Mengen A_i mit $3 \leq i \leq n$ gemäß des zuletzt beschriebenen Verfahrens ein Tupel t_{i-1} ermittelt, dessen erste Komponente die Größe der Vereinigungsmenge $V_{i-1} \cup A_i$ $(= A_1 \cup \cdots \cup A_i)$ und dessen zweite Komponente den zur Bildung der Vereinigungsmenge $V_{i-1} \cup A_i$ notwendigen Rechenaufwand angibt. Die zweite Komponente des so berechneten Tupels t_{n-1} gibt das gesuchte Schätzmaß für den zur Bildung der Menge $A_1 \cup A_2 \cup \cdots \cup A_n$ erforderlichen Rechenaufwand an.

Die im obigen Absatz beschriebene Komplexitätsabschätzung wird von der Funktion 'k_oder' geleistet, deren Definition im folgenden aufgeführt ist:

```
k_oder :: [(Float, Float)] → (Float, Float)

k_oder [a] = a
k_oder [(l1,k1), (l2,k2)]
        | l1 == 0.0 = (l2, k1 + k2)
```

```
| l2 == 0.0 = (l1, k1 + k2)
| otherwise = (val2 + (val1/2.0), k1 + k2
        + ((2.0 · (l1+l2 -1.0) + val1)/2.0) · g_vgl_Konzept
        + (val2 + (val1/2.0)) · g_add_Konzept)
            where val1 = minimum [l1, l2]
                  val2 = maximum [l1, l2]

k_oder (a:b:as) = k_oder (k_oder [a, b]:as)

g_vgl_Konzept = 8.8
g_add_Konzept = 6.3
```

5.4.3.3 Bewertung der Komplexitätsabschätzungen

In diesem Abschnitt verdeutliche ich anhand von Beispielen, mit welcher Genauigkeit das den Funktionen 'komplexität_K' und 'komplexität_R' zugrundeliegende Komplexitätsabschätzungsverfahren arbeitet. Für den Anfrageausdruck '(Or [Student, Weiblich])' beispielsweise schätzt die Funktion 'komplexität_K' die Anzahl der Instanzen wie die Anzahl erforderlicher Reduktionsschritte mit einem Fehler von unter 20%. Dies kann anhand des folgenden Ausschnitts einer Gofer-Sitzung überprüft werden:

? finde_alle (Or [Student, Weiblich])
["Alexandra", "Carina", "Cosima", "Fifia", "Frank", "Gani", "Madonna", "Maria", "Michael", "Sima", "Siille", "Steffi", "Tannja", "Zeideh"]
(274 reductions, 613 cells)

? komplexitaet_K (Or [Student, Weiblich])
(13.5,221.45)
(110 reductions, 236 cells)

In den folgenden Tabellen werden für weitere Anfrageausdrücke die tatsächlich benötigte Anzahl an Reduktionsschritten sowie die tatsächliche Anzahl an Instanzen den von der Funktion 'komplexität_K (bzw. komplexität_R)' ermittelten Schätzwerten gegenübergestellt.

Anfrageausdruck: (And [Weiblich, Student, Vegetarier])

	Größe	Reduktionsschritte
finde_alle	1	154 reductions
komplexität_K	0.75	149.108 reductions

Anfrageausdruck: (And [Weiblich, Or [Student, Vegetarier]])

	Größe	Reduktionsschritte
finde_alle	3	247 reductions
komplexität_K	2.75	247.042 reductions

Anfrageausdruck: (Restrict HatKind Student)

	Größe	Reduktionsschritte
finde_Relation	6	1267 reductions
komplexität_R	4.82759	914.71 reductions

Im Falle beider oben aufgeführter Anfrageausdrücke werden die tatsächlichen Werte mit mindestens 75%iger Genauigkeit abgeschätzt.

In meinem Programm werden die Komplexitätsabschätzungsfunktionen 'komplexität_K' und 'komplexität_R' eingesetzt, um zu ermitteln, *welcher der beiden an einer universellen Identität beteiligten Terme* (Anfragenausdrücke) *effizienter* auswertbar ist. Im nächsten Abschnitt zeige ich unter anderem, wie die beiden Komplexitätsabschätzungsfunktionen 'komplexität_K' und 'komplexität_R' im Rahmen einer Simplifikation dazu eingesetzt werden, den effizienter auswertbaren Term zu bestimmen.

5.4.4 Simplifizierer

In meinem Programm sind zwei Funktionen definiert, die zum Simplifizieren von Anfrageausdrücken benutzt werden. Diese beiden Funktionen tragen die Namen '*simple_term*' und '*simple_r_term*'. Die Funktion 'simple_term' simplifiziert Ausdrücke vom Datentyp 'Konzeptterm'. Zur Simplifizierung von Ausdrücken vom Datentyp 'Relationenterm' dient die Funktion 'simple_r_term'. Die Funktionen 'simple_term' und 'simple_r_term' simplifizieren rekursiv, solange dies möglich ist. Die Typdeklarationen der beiden Funktionen sehen wie folgt aus:

`simple_term :: Konzeptterm ⟶ Konzeptterm`

`simple_r_term :: Relationenterm ⟶ Relationenterm`

Die beiden Datentypen 'Konzeptterm' und 'Relationenterm' sind, wie bereits gesagt, Ausprägungen der Typklasse 'Eq'. Dies ist von besonderer Bedeutung im Zusammenhang mit der Vorgehensweise bei der Simplifikation von Anfrageausdrücken in meinem Programm, da die Simplifikation mittels „pattern matching" verwirklicht wird. Im folgenden werde ich dies anhand einiger Beispiele verdeutlichen. Die Beispiele erfassen dabei den Fall, daß eine Komplexitätsabschätzung benötigt wird, wie auch den Fall, daß keine Komplexitätsabschätzung benötigt wird.

Im folgenden seien a und b zwei beliebige Elemente aus der Trägermenge einer Booleschen Algebra B sowie r und s zwei beliebige Relationen aus der Trägermenge einer Relationenalgebra R.
Weiterhin seien A und B zwei beliebige Konzeptterme aus meiner Wissensbasis sowie R und S zwei beliebige Relationenterme aus meiner Wissensbasis.
$k(t)$ bezeichne die Anzahl benötigter Reduktionsschritte, die zur Evaluierung des Terms t erforderlich sind.

5.4.4.1 Anwendung der Simplifikationsfunktion simple_term

Beispiel 1
Dieses Beispiel bezieht sich auf die in meinem Programm mit der Nummer 3 gekennzeichnete Simplifikation. Ausdrücke der Form $a \cdot (a + b)$ vereinfacht die Simplifkationsfunktion 'simple_term' ohne Beachtung der Instanzen der Argumentterme in die Form a. (Begründung siehe Abschnitt 5.4.2) Die benutzte universelle Identität sowie assoziierte terminologische Identität sind zusammen mit einer Komplexitätsberechung unten aufgeführt.

$$\textit{universelle Identität} : (a \cdot (a + b) = a)$$

$$\textit{terminologische Identität} : ((\textbf{and}\ \mathsf{A}\ (\textbf{or}\ \mathsf{A}\ \mathsf{B})) \approx \mathsf{A})$$

$$\textit{Komplexität} : k(a \cdot (a + b)) > k(a)$$

Ersetzt man die Symbole a und b durch in meinem Programm vereinbarte Konzepte so kann man folgende Beispiel-Simplifikationen formulieren:

1. (**And** [Student, **Or** [Student, Weiblich]]) $\xrightarrow{\text{Simplifikation}}$ Student

2. (**And** [**And** [Student, Weiblich], **Or** [**And** [Student, Weiblich], Vegetarier]]) $\xrightarrow{\text{Simplifikation}}$ (**And** [Student, Weiblich])

Der unten aufgeführte Ausschnitt meines Programms zeigt einen Teil der Definition der Simplifikationsfunktion 'simple_term' und läßt erkennen, wie bei der Auswertung der in den Beispielen aufgeführten Ausdrücke vorgegangen wird.

```
simple_term (And [a, Or bs])
          |any (a==)bs = simple_term a
```

Die oben aufgeführten Beispiele können mit Hilfe meines Programms praktisch nachvollzogen werden. Eingaben und Ausgaben haben dabei folgendes Aussehen:

Simplifikationsanfrage:
? simple_term (And [Student, Or [Student, Weiblich]])
Student
(65 reductions, 108 cells)

? simple_term (And [And [Student, Weiblich], Or [And [Student, Weiblich], Vegetarier]])
And [Student, Weiblich]
(213 reductions, 351 cells)

Im folgenden wird gezeigt, daß die beiden zuletzt aufgeführten Simplifikationen sinnvoll sind, indem die Evaluationsfunktion 'finde_alle' jeweils auf den simplifizierten wie nicht simplifizierten Ausdruck angewendet wird.

Anfrage für den nicht simplifizierten Ausdruck:
? finde_alle (And [Student, Or[Student, Weiblich]])
["Frank", "Gani", "Michael"]
(354 reductions, 617 cells)

Anfrage für den simplifizierten Ausdruck:
? finde_alle (Student)
["Frank", "Gani", "Michael"]
(25 reductions, 69 cells)

Anfrage für den nicht simplifizierten Ausdruck:
? finde_alle (And [And [Student, Weiblich], Or[And [Student, Weiblich], Vegetarier]])
["Gani"]
(304 reductions, 511 cells)

Anfrage für den simplifizierten Ausdruck:
? finde_alle (And [Student, Weiblich])
["Gani"]
(169 reductions, 282 cells)

Beispiel2

In einem weiteren Beispiel stelle ich die mit der Nummer 9 gekennzeichnete Simplifikation vor. Bei dieser Simplifikationen ist die Entscheidung, welcher der an der universellen Identität beteiligten Terme effizienter ist, von den Instanzen der Argumentterme abhängig. In diesen Fällen wird die Entscheidung, welcher Term effizienter ist, mittels einer Komplexitätsabschätzungsfunktion getroffen. Die von der 9ten Simplifkation benutzte universelle Identität sowie assoziierte terminologische Identität sind unten aufgeführt.

$$\begin{aligned} \textit{universelle Identität}: a \cdot b + a \cdot c &= b \cdot a + a \cdot c \\ &= b \cdot a + c \cdot a = a \cdot b + c \cdot a \\ &= a \cdot (b + c) \end{aligned}$$

$$\begin{aligned} \textit{terminologische Identität}: &(\textbf{or}\ (\textbf{and}\ \mathsf{A}\ \mathsf{B})\ (\textbf{and}\ \mathsf{A}\ \mathsf{C})) \\ &= (\textbf{or}\ (\textbf{and}\ \mathsf{B}\ \mathsf{A})\ (\textbf{and}\ \mathsf{A}\ \mathsf{C})) \\ &= (\textbf{or}\ (\textbf{and}\ \mathsf{B}\ \mathsf{A})\ (\textbf{and}\ \mathsf{C}\ \mathsf{A})) \\ &= (\textbf{or}\ (\textbf{and}\ \mathsf{A}\ \mathsf{B})\ (\textbf{and}\ \mathsf{C}\ \mathsf{A})) \\ &= (\textbf{and}\ \mathsf{A}\ (\textbf{or}\ \mathsf{B}\ \mathsf{C})) \end{aligned}$$

Die folgende Beispielsimplifikation

$$(\textbf{Or}\ [\textbf{And}\ [\textsf{Weiblich, Student}], \textbf{And}\ [\textsf{Weiblich, Vegetarier}]]) \xrightarrow{\text{Simplifikation}}$$
$$(\textbf{And}\ [\textsf{Weiblich}, \textbf{Or}\ [\textsf{Student, Vegetarier}]])$$

könnte im Rahmen einer umfassenderen Simplifikation auftreten. Bevor ich darauf eingehe, in welchem Ausschnitt meines Programms die obige Beispielsimplifikation bewertet würde, zeige ich zunächst, welches Ergebnis die getrennte Komplexitätsauswertung der beiden an der obigen Beispielsimplifikation beteiligten Ausdrücke liefert.

```
? komplexitaet_K (Or [And [Weiblich, Student], And [Weiblich, Vegetari-
er]])
(2.75,279.175)
(114 reductions, 319 cells)

? komplexitaet_K (And [Weiblich, Or [Student, Vegetarier]])
(2.75,247.042)
(78 reductions, 222 cells)
```

Würde mein Programm im Rahmen einer umfassenderen Simplifikation vor der Entscheidung stehen, ob die obige Beispielsimplifikation sinnvoll ist, so würde die Entscheidung in folgendem Ausschnitt der Funktion ‘simple_term’ getroffen werden:

```
simple_term (Or [And [a, b], And [c, d]])
   | a == c && snd(komplexitaet_K
                     (Or [And [a, b], And [c, d]])) ≥
               snd(komplexitaet_K (And [a, Or[b, d]]))
           = simple_term (And [a, Or[b, d]])
```

Als Funktionsergebnis würde, da die obige Beispielsimplifikation sinnvoll ist, das Ergebnis des rekursiven Aufrufs von *simple_term* (*And* [*a*, *Or*[*b*, *d*]]) zurückgegeben werden.

Im folgenden stelle ich mittels einer Reihe von Ausschnitten aus einer Gofer-Sitzung

- die zum letzten Beispiel gehöhrige Simplifikationsanfrage,

- die Anfrage zur Evaluierung des nicht simplifizierten Ausdrucks,
- sowie die Anfrage zur Evaluierung des simplifizierten Ausdrucks

vor:

> *Simplifikationsanfrage*:
> ? simple_term (Or [And [Weiblich, Student], And [Weiblich, Vegetarier]])
> And [Weiblich, Or [Student, Vegetarier]]
> (453 reductions, 1085 cells)
>
> *Anfrage für den nicht simplifizierten Ausdruck*:
> ? finde_alle (Or [And [Weiblich, Student], And [Weiblich, Vegetarier]])
> ["Gani", "Sima", "Siville"]
> (287 reductions, 535 cells)
>
> *Anfrage für den simplifizierten Ausdruck*:
> ? finde_alle (And [Weiblich, Or [Student, Vegetarier]])
> ["Gani", "Sima", "Siville"]
> (247 reductions, 460 cells)

In dem zuletzt aufgeführten Beispiel war die Differenz zwischen der Anzahl notwendiger Reduktionsschritte zur Evaluierung des simplifizierten und nicht simplifizierten Ausdrucks gering. Addiert man im letzten Beispiel den zur Evaluierung des simplifizierten Ausdrucks notwendigen Aufwand und die von der Simplifikationsanfrage verursachten Kosten, so erhält man einen insgesamten Berechnungsaufwand von 700 Reduktionsschritten. Da dies einem Berechnungsaufwand von 287 Reduktionsschritten zur Evaluierung des nicht simplifizierten Terms gegenübersteht, erscheint die Simplifikation insgesamt als nicht sinnvoll. Daß dies nicht immer so sein muß, wenn eine Komplexitätsabschätzungsfunktion angewendet wird, kann anhand von in meinem Programm eingebetteten Beispielen gesehen werden. (In diesem Zusammenhang weise ich besonders auf Simplifikation 10 in meinem Programm hin.)

5.4.4.2 Anwendung der Simplifikationsfunktion simple_r_term

Beispiel 1
Dieses Beispiel bezieht sich auf die mit der Nummer 17 gekennzeichnete Simplifikation

meines Programms. Die Simplifkationsfunktion 'simple_r_term' überführt Ausdrücke der Form $r^{\smile} \cdot s^{\smile}$ ohne Beachtung der Instanzen der Argumentterme in die Form $(r \cdot s)^{\smile}$. (Begründung siehe Abschnitt 5.4.2) Die benutzte universelle Identität sowie assoziierte terminologische Identität sind zusammen mit einer Komplexitätsberechung im folgenden aufgeführt.

universelle Identität : $(r^{\smile} \cdot s^{\smile} = (r \cdot s)^{\smile})$

terminologische Identität : (**and** (**inverse** R) (**inverse** S)) $\approx$ (**inverse** (**and** R S))

Komplexität : $k(r^{\smile} \cdot s^{\smile}) > k((r \cdot s)^{\smile})$

Werden die Symbole r und s durch in meinem Programm vereinbarte Relationen ersetzt, so kann folgende Beispielsimplifikation formuliert werden.

(**RAnd** [(**Inverse** HatKind) (**Inverse** IstLehrer)]) $\xrightarrow{\text{Simplifikation}}$ (**Inverse** (**RAnd** [HatKind IstLehrer]))

Im weitern stelle ich mittels einer Reihe von Ausschnitten aus einer Gofer-Sitzung

- die zum letzten Beispiel gehöhrige Simplifikationsanfrage,
- die Anfrage zur Evaluierung des nicht simplifizierten Ausdrucks,
- sowie die Anfrage zur Evaluierung des simplifizierten Ausdrucks

vor:

Simplifikationsanfrage:
? simple_r_term (RAnd [(Inverse HatKind), (Inverse IstLehrer)])
Inverse (RAnd [HatKind, IstLehrer])
(33 reductions, 97 cells)

Anfrage für den nicht simplifizierten Ausdruck:
? finde_Relation (RAnd [(Inverse HatKind), (Inverse IstLehrer)])
[("Maria", "Georg"), ("Ralf", "Gani")]
(2763 reductions, 4908 cells)

Anfrage für den simplifizierten Ausdruck:
? finde_Relation (Inverse (RAnd [HatKind, IstLehrer]))
[("Maria", "Georg"), ("Ralf", "Gani")]
(442 reductions, 757 cells)

Beispiel 2
Wie bereits erwähnt, gibt es universelle Identitäten für Relationen, bei denen die Entscheidung, welcher Term effizienter auswertbar ist, von den Instanzen der Argumentterme abhängt. Ein Beispiel für eine derartige universelle Identität ist

$$s^{\smile}; r^{\smile} = (r; s)^{\smile}$$

Die assoziierte terminologische Identität der zuletzt aufgeführten universellen Identität lautet

(**compose** (**inverse** S) (**inverse** R)) = (**inverse** (**compose** R S))

Ersetzt man den Term S durch die Relation 'IstKusin' sowie den Term R durch die Relation 'HatKind' und übergibt den so entstehenden linkseitigen Ausdruck an die Funktion 'simple_r_term', so wird der Ausdruck auf der rechten Seite als simplifizierter Ausdruck zurückgegeben. Die Korrektheit der Rückgabe des rechtseitigen Ausdrucks belegen die folgenden zwei Ausschnitte einer Gofer-Sitzung:

Anfrage für den nicht simplifizierten Ausdruck
? finde_Relation (Compose [Inverse IstKusin, Inverse HatKind])
[("Cosima","Siville"), ("Cosima","Stefan"), ("Frank","Rainer"),
("Frank","Steffi"), ("Gani","Rainer"), ("Gani","Steffi"),
("Michael","Siville"), ("Michael","Stefan")]
(4847 reductions, 8143 cells)

Anfrage für den simplifizierten Ausdruck
? finde_Relation (Inverse (Compose [HatKind, IstKusin]))
[("Cosima","Siville"), ("Cosima","Stefan"), ("Frank","Rainer"),
("Frank","Steffi"), ("Gani","Rainer"), ("Gani","Steffi"),
("Michael","Siville"), ("Michael","Stefan")]
(2670 reductions, 4296 cells)

5.4.5 Erweiterungsmöglichkeit von Simplifikationen

Wie auf Seite (50) berichtet, habe ich im Programm die Operatoren 'Konjunktion', 'Disjunktion' und 'Composition' als n-stellige Operatoren vereinbart. (Die Operationen 'Durchschnitt', 'Vereinigung' und 'Composition' der universellen Identitäten können auf n-stellige Operationen ausgedehnt werden.) Die Vereinbarung der obigen Operatoren als n-stellig ermöglicht die Erweiterung einiger im Programm gegebener Simplifikationen. In einigen Simplifikationen habe ich diese Erweiterungsmöglichkeiten bereits eingesetzt. Ein Beispiel dafür stellt die dritte Simplifikation dar. Dort wird die Operation 'Vereinigung' des linken Terms der entsprechenden universellen Identität $(a \cdot (a + b) = a)$ wird auf n-stellige Operation ausgedehnt. Praktisch kann man dies daran ablesen, daß beim Aufruf des Simplifizierers *simple_term* als Argument (pattern) And [a, Or **bs**] übergeben wird.
Im folgenden stelle ich am Beispiel der vierten Simplifikation eine andere Erweiterungsmöglichkeit von Simplifikationen vor:
Die in der vierten Simplifikation verwendete universelle Identität lautet:

$$(a + b) \cdot (a + c) = a + (b \cdot c)$$

Die in dieser universellen Identität verwendete Operation 'Disjunktion' ist als binärer Operation definiert und kann wie folgt auf eine n-stellige $(n,\ m \geq 2)$ Operation ausgedehnt werden:

$$(a_1 + a_2 + \cdots + a_n) \cdot (b_1 + b_2 + \cdots + b_m)$$

Die Simplifikation dieses Terms kann nach dem folgenden Schema realisiert werden:

- Die Argumente $a_1, \cdots, a_n$ sowie $b_1, \cdots, b_n$ werden zunächst sortiert. (Eine Sortierung der Argumente ist möglich, da die Operation 'Disjunktion' kommutativ und assoziativ ist.) Duplikate jeder Argumentmenge werden dabei im Rahmen der Sortierung entfernt.

- Die Schnittmenge $\{s_1,\ s_2,\ \cdots,\ s_l\}$ beider Argumentmengen wird gebildet.
  Aus beiden Argumentmengen wird jeweils jedes in der Schnittmenge enthaltene Element entfernt.

$$\{a_1, a_2, \cdots a_{n'}\} - \{s_1, \cdots s_l\} := \{d_1, \cdots d_a\}$$
$$\{b_1, b_2, \cdots b_{m'}\} - \{s_1, \cdots s_l\} := \{e_1, \cdots e_b\}$$

- Der folgende Term stellt einen simplifizierten Ausdruck dar:

$$(s_1 + s_2 + \cdots + s_l) + ((d_1 + \cdots + d_a) \cdot (e_1 + e_2 + \cdots + e_b))$$

5.5 Anfragen an die Wissensbasis

Soll eine Anfrage an meine Wissensbasis gestellt werden, so ist nach folgendem Schema vorzugehen:

- Zunächst ist zu überprüfen, ob ein Anfrageausdruck simplifiziert werden kann. Dies geschieht, indem der Simplifizierer *simple_term* (simple_term :: Konzeptterm $\longrightarrow$ Konzeptterm) bzw. *simple_r_term* (simple_r_term :: Relationenterm $\longrightarrow$ Relationenterm) auf den Anfrageausdruck angewendet wird. Der optimierte Ausdruck wird am Bildschirm ausgegeben.

- Im Anschluß hieran können alle Instanzen eines Konzepts bzw. einer Rolle mittels einer optimierten Anfrage ermittelt werden. Dazu sind die Funktion 'finde_alle' bzw. 'finde_Relation' zu verwenden, je nachdem ob der Anfrageausdruck ein Konzept oder eine Rolle darstellt.

Kapitel 6

Zusammenfassung und Ausblick

In Abschnitt 4.1 wurden Algebren auf Mengen und Relationen vorgestellt. Desweitern wurde in Abschnitt 4.2 eine zu $\mathcal{ALC}$ erweiterte terminologische Wissensrepräsentationssprache $\mathcal{ALCX}$ definiert und gezeigt, daß die in Abschnitt 4.1 vorgestellten Algebren zur Festlegung der modell-theoretischen Semantik der Sprache $\mathcal{ALCX}$ benutzt werden können. Als Ergebnis einer derartigen Festlegung kann jeder terminologische Ausdruck direkt mit einem algebraischen Term assoziiert werden. Desweitern können alle in den Algebren aufgeführten universellen Identitäten in semantisch äquivalente terminologische Identitäten überführt werden.

Eine mittels $\mathcal{ALCX}$ repräsentierte Wissensbasis ist in meinem Programm mit Hilfe der in der funktionalen Programmiersprache Gofer (Version 2.28) gegebenen Möglichkeiten formuliert.

Mein Programm bietet durch Simplifizierung von Anfrageausdrücken eine optimierte Lösung des aus dem Bereich der Wissensrepräsentation bekannten Retrieval-Problems.

Die Simplifizierung von Anfrageausdrücken wird in meinem Programm erzielt, indem in den oben genannten Algebren erfüllte universelle Identitäten als Simplifikationshilfsmittel benutzt werden. Die in den Algebren aufgeführten universellen Identitäten bestehen stets aus zwei äquivalenten Termen, für die eine unterschiedliche Anzahl von Reduktionsschritten zur Evaluierung benötigt werden.
Es existieren universelle Identitäten, bei denen unabhängig von den Instanzen der Argumentterme ein Term gegenüber einem anderen Term effizienter auswertbar ist. Bei diesen universellen Identitäten kann eine Simplifikation von einem Term zu einem

anderen Term immer unproblematisch realisiert werden.
Es gibt jedoch auch universelle Identitäten, bei denen die Unabhängigkeit von den Instanzen der Argumentterme nicht gegeben ist. In diesen Fällen wird ein von der Wissensbasis abhängiges Komplexitätsabschätzungsverfahren eingesetzt, um den effizienter auswertbaren Term von zwei an einer universellen Identität beteiligten Termen zu ermitteln.

Daß die Simplifikation eines Anfrageausdrucks große Einsparungen bei dessen Auswertung nach sich ziehen kann, wurde an verschiedenen Beispielen in Abschnitt 5.4.4 gezeigt.

Über eine weitere Möglichkeit, die zum Simplifizieren von Anfrageausdrucken verwendet werden kann, wurde in Abschnitt 5.4.5 berichtet.

Die Genauigkeit der Komplexitätsabschätzung kann erhöht werden, indem das Komplexitätsabschätzungsverfahren erweitert wird. So kann beispielsweise die Komplexität der Funktion 'oder' genauer abgeschätzt werden, indem für zu vereinigende Mengen überprüft wird, ob Teilmengenbeziehungen vorhanden sind. Durch eine genauere Abschätzung der Mächtigkeit entstehender Vereinigungsmengen wird die Komplexitätsabschätzung insgesamt in ihrer Genauigkeit gesteigert.

Eine Änderung der Instanzen meiner terminologischen Datenbank erfordert eine Änderung des Skripts meines Programms. Indem die Instanzen-Daten in einer veränderlichen Datenbank festgehalten würden, könnte hier Abhilfe geschaffen werden. Dies könnte im Rahmen der Erstellung einer benutzterfreundlichen Ein- und Ausgabe-Schnittstelle realisiert werden.

Literaturverzeichnis

[BBMR89] Borgida, A., Brachman, R. J., McGuinness, D. L., Resnick, L. A. "CLASSIC: A Structural Data Model for Objects." In *Proceedings of the International Conference on Management of Data*, Portland, Oregon, 1989.

[BBS92] Brink, C., Britz, K. and Schmidt, R. A. Peirce algebras, *Technical Report MPI-I-92-229*, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1992.

[BC81] Brink, C. Boolean modules, *Journal of Algebra* **71**(2), 291-313, 1981.

[BC88] Brink, C. On the Application of Relations. *S. Afr. J. Philos.* **7** (2), Special Interdisciplinary Edition Devoted to Logic, 105-112, 1988.

[BC92] Brink, C. and Schmidt, R. A. Subsumption computed algebraically, *Computers and Mathematics with Applications* **23**(2-9), 329-342. Special Issue on Semantic Networks in Artificial Intelligence, 1992.

[BH90] Baader, F., Hollunder, B. Knowledge Representation and Inference System. -System Description- *Deutsches Forschungszentrum für Künstliche Intelligenz*, Projektgruppe WINO, Postfach 2080, D-6750 Kaiserslautern, West Germany, 1990.

[BK88] Britz, K. Relations and Programs. M.Sc. Thesis, Department of Computer Science, University of Stellenbosch, South Africa, 1988.

[BPL85] Brachmann, R. J. Pigman Gilbert, V. Levesque, H. J. An essential hybrid reasoning system: knowledge and symbol level accounts in KRYPTON. In *Proceedings of the 9th IJCAI,* pp. 532-539, Los Angeles, Cal., 1985.

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

[BW92] Bird, R., und Wadler, R. *Einführung in die funktionale Programmierung.* Carl Hanser Verlag, München, 1992.

[CA32] Church, A. A set of postulates for the foundation of logic, *Annals of Mathematics*, 2(33): 346-366, 1932.

[CA41] Church, A. The Calculi of Lambda-Conversion, *Annals of Mathematics Studies* No. 6, Princeton University Press, 1941.

[CT51] Chin, L.H., and Tarski, A. Distributive and Modular Laws in the Arithmetic of Relation Algebra. *Univ. Calif. Publ. Math.* **1** (9), 341-384, 1951.

[DLNHN90] Donini, F.M., Lenzerini, M., Nardi, D., and Nutt, W. The Complexity of Concept Languages. *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, San Mateo, California, pp. 151-162, 1991.

[EM85] Ehrig, H., and Mahr, B. *Fundamentals of Algebraic Specification* 1: *Equations and Initial Semantics.* Springer-Verlag, Berlin, 16, 1985.

[HB89] Hollunder, B. Subsumtion Algorithms for Some Attributive Concept Description Languages. SEKI Report SR-89-16, Fachbereich Informatik, Universität Kaiserslautern, Germany, 1989.

[HE04] Huntington, E.V. Sets of Independent Postulates for the Algebra of Logic. *Trans. A.M.S.* **5**, 288-309, 1904.

[HE33] Huntington, E.V. New Sets of Independent Postulates for the Algebra of Logik, with Special Reference to Whitehead and Russell's *Principia Mathematica. Trans. A.M.S.* **35**, 274-304, 1933.

[HNS90] Hollunder, B., Nutt, W., and Schmidt-Schauß, M. Subsumption Algorithms for Concept Description Languages. In *Proceedings of the 9th European Conference on AI*(ECAI 90). A shortened version of Hollunder, B. and Nutt, W. Subsumption Algorithms for Concept Languages. Research Report RR-90-04, DFKI, Kaiserslautern, Germany, 1990.

[HR92] Hinze R. *Einführung in die funktionale Programmierung mit Miranda.* B. G. Teubner-Verlag, Stuttgart, 1992.

[JM93] Jones, M. P. Gofer Functional programming environment, Version 2.28, 1993.

[KA89] Kobsa, A. The SB-ONE knowledge representation workbench. In *Preprints of the Workshop on Formal Aspects of Semantic Networks*, Two Harbors, Cal., February 1989.

[KBR86] Kaczmarek, T. S. Bates, R. Robins, G. Recent developments in NIKL. In *Proceedings of the 5th National Conference of the AAAI*, pp. 578-587, Philadelphia, Pa., 1986.

[MA86] Manes, E.G., and Arbib, M.A. *Algebraic Approaches to Program Semantics.* Springer-Verlag, Berlin, 322, 1986.

[MB87] MacGregor, R. Bates, R. The Loom Knowledge Representation Language. Technical Report ISI/RS-87-188, University of Southern California, Information Science Institute, Marina del Rey, Cal., 1987.

[NS89] Nebel, B., and Smolka, G. Representation and Reasoning with Attributive Descriptions. IWBS Report 81, IWBS, IBM Deutschland, Stuttgart, Germany. To appear in Bläsius, K.-H., Hedtstück, U., and Rollinger, C.-R. (Eds.), *Sorts and Types in Artificial Intelligence.* Springer-Verlag, Berlin, 1989.

[NvL88] B. Nebel, K. von Luck. "Hybrid Reasoning in BACK." In Z. W. Ras, L. Saitta (editors), *Methodologies for Intelligent Systems*, pp. 260-269, North Holland, Amsterdam, Netherland, 1988.

[Neb88] B. Nebel. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371-383, 1988.

[PS84] Patel-Schneider, P. F. Small can be beautiful in knowledge representation. *In Proceedings of the IEEE Workshop on Principles of Knowledge-Based System*, pp. 11-16, Denver, Colo., 1984.

[PS87] Patel-Schneider, P. F. Decidable, Logic-Based Knowledge Representation. PhD Dissertation, University of Toronto, 1987.

[SJ89] Schmolze, J.G. The Language and Semantics of NIKL. Techinical Report 89-4, Department of Computer Science, Tufts University, Medford, MA. To appear in *Computational Intelligence*, 1989.

[SR91] Schmidt, R. A. Algebraic Terminological Representation. M. Sc. Thesis Reprint TR 011, *Department of Mathematics, University of Cape Town, Cape Town, South Africa.* Also available as Technical Report MPI-I-91-216, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1991.

[SS88] Schmidt-Schauß, M., and Smolka, G. Attributive Concept Description with Complements. To appear in *Artificial Intelligence*, 1988.

[TA41] Tarski, A. On the Calculus of Relations. *Journal of Symbolic Logic* **6**, 73-89, 1941.

[VM85] Vilain, M. B. The restricted langeage architecture of a hybrid representation system. In Brachmann, R. J. Levesque, H. J. Reiter, R. (editors), *Proceedings of the 9th IJCAI,* pp. 547-551, Los Angeles, Cal., 1985.

[WS91] Woods, W.A., and Schmolze, J.G. The KLONE Family. *Computers and Mathematics with Applications* **23**, 133-137, 1992.