Untersuchung alternativer nicht-deterministischer λ-Kalküle zur Implementierung der Montague-Semantik

Diplomarbeit
von Hermine Reichau

Frankfurt, den 28.04.2006

Betreuer: Prof. Dr. Manfred Schmidt-Schauß
Fachbereich 12: Informatik und Mathematik
Institut für Informatik
Lehrstuhl für Künstliche Intelligenz und Softwaretechnologie

Danksagung

Vielen Dank an alle, die mich bei der Entstehung dieser Arbeit durch Geduld, ein offenes Ohr und guten Rat unterstützt haben. Ich danke den Teilnehmern des logisch-semantischen Kolloquiums für ihre kritischen Fragen und hilfreichen Anregungen die mich stets motiviert haben. Insbesondere danke ich Prof. Zimmermann für seine Ausdauer bei der Beantwortung meiner vielen Fragen zur Semantik und seiner kontinuierlichen Unterstützung.

Mein besonderer Dank gilt David Sabel und Prof.Dr. Manfred Schmidt-Schauß für ihre hervoragende Betreuung.

Hermine Reichau

Erklärung nach DPO §11 Abs.11

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt, den 28. April 2006

Hermine Reichau

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

‚Ich habe die symbolische Logik angewendet, unter anderem auch auf dieses Dokument hier. Für meine eigene Person hatte ich es eigentlich nicht nötig, weil ich wußte, auf was es hinausläuft. Aber ich kann es fünf Naturwissenschaftlern vielleicht eher mit Symbolen als mit Worten erklären.‘
Hardin riß ein paar Blätter von dem Block unter seinem Arm und breitete sie aus.‚Übrigens habe ich es nicht selbst gemacht‘, berichtete er. ‚Muller Holk von der Logik-Abteilung hat die Analyse mit seinem Namen unterschrieben. Überzeugen Sie sich.‘
Pirenne beugte sich Über den Tisch, um besser sehen zu können, und Hardin fuhr fort: ‚Die Botschaft von Anakreon stellte natürlich eine einfache Aufgabe dar, weil ihre Verfasser ehr Männer der Tat als des Wortes sind. Der leicht herauszuschälende Kern läßt sich aus der symbolischen Niederschrift ungefähr so übersetzen: ‚Ihr gebt uns innerhalb einer Woche, was wir wollen, oder wir schlagen euch zusammen und nehmen es uns.‘.
Stille trat ein. Die fünf Mitglieder des Kuratoriums gingen die Reihe der Symbole durch, und dann ließ Pierenne sich auf seinen Sitz zurücksinken und hustete verlegen.
Hardin sagte: ‚Es gibt kein Schlupfloch, nicht wahr, Dr. Pirenne? ‘
‚Sieht so aus.‘
‚Gut.‘Hardin legte andere Blätter auf den Tisch. ‚Vor sich sehen Sie jetzt eine Kopie des Vertrages zwischen dem Reich und Anakreon - den übrigens im Namen des Kaisers derselbe Lord Dorwin unterzeichnet hat, der letzte Woche hier war - und dazu eine symbolische Analyse.‘
Der Vertrag bestand aus fünf Seiten Kleingedrucktem, und die Analyse nahm nicht einmal eine halbe Seite ein.
‚Wie sie sehen, Gentlemen, verdampfen ungefähr neunzig Prozent des Vertrages als bedeutungslos, und was übrig bleibt, kann auf folgende interessante Weise beschrieben werden:
Verpflichtungen Anakreons gegenüber dem Reich: *Keine!*
Autorität des Reiches gegenüber Anakreon: *Keine!* ‘
(...)‚Aber wie ‘, fiel Sutt ein, ‚fügt Bürgermeister Hardin die uns von Lord Dorwin zugesicherte Hilfe ins Bild ein? Seine Zusagen...‘Er zuckte die Achseln. ‚Nun, ich fand sie zufriedenstellend.‘Hardin hatte vorsorglicher Weise

> Lord Dorwins fünftägige Rede aufgezeichnet und ebenfalls zur Analyse eingesandt:
>
> ‚Die Analyse war von allen dreien die schwierigste. Als es Holk nach zwei Tagen ununterbrochener Arbeit gelungen war, bedeutungslose Aussagen, vages Geschwätz und nutzlose Erklärungen zu eliminieren - kurz, das ganze Geseiere -, stellte er fest, dass *nichts* mehr übrig war. Es war alles durch das Raster gefallen.
> Gentlemen, Lord Dorwin hat fünf Tage lang geredet und dabei, verdammt noch mal, nichts gesagt, und er hat es so gesagt, daß Sie überhaupt nichts davon merkten. *Das* sind die Zusagen, die Sie von Ihrem hochgepriesenen Reich bekommen haben.‘
>
> (Isaak Asimov, *Foundation*)

1.1 Einleitung

Ende des 19.Jahrhunderts legte Gottlob Frege 1879 mit seiner *Begriffsschrift*[Fre80] den Grundstein für die symbolische Logik mit weitreichenden Folgen. Zu dieser Zeit waren viele Mathematiker damit beschäftigt mathematische Theorien zu axiomatisieren um die Beweisführung zu verbessern. Freges logische Untersuchungen in diesem Bereich unterschieden sich jedoch wesentlich von denen der anderen Mathematiker. Dort wo andere nach Verbesserung der Schlussregeln zwischen Ausdrücken suchten, forderte Frege eine Modifikation der Ausdrücke selbst, die bis dahin weitestgehend natürlich sprachlich waren. Natürlichsprachliche Ausdrücke haben jedoch die Eigenschaft potentiell mehrdeutig zu sein, so dass jegliche Ausdrücke dieser Form, so gut die Schlussregeln auch sein mochten, jeden mathematischen Beweis zu Fall bringen konnten. Seine Forderung nach eine künstlichen Sprache für die mathematische Argumentation brachte die Prädikatenlogik hervor und setzte damit den Grundstein für die formale Semantik in der Linguistik.

1.1.1 Motivation

Die Fähigkeit Sprache zu verwenden ist eine der herausragendsten Eigenschaften des menschlichen Geistes. Sie dient als Verbindung zwischen dem Menschen und seiner Umwelt, indem sie Denken, Sprache und Wirklichkeit in Relation zueinander setzt.
Sprachverstehen lässt sich als eine Abbildung einer sprachlichen Äußerung auf ihre Bedeutung verstehen. Wobei hierdurch noch nicht geklärt ist, woraus diese Bedeutung

besteht oder was sie ist. Ein großer Schritt in diese Richtung wurde von Frege durch seine Unterscheidung von Sinn und Bedeutung gemacht. Anders als in der Kognitionsforschung, wo durch Computersimulation im Rahmen einer kognitven Modellierung nach Formulierung und Präzisierung von Theorien für kognitive Prozesse gesucht wird, ist in der Informatik vor allem die Anwendung bestehender kognitiver/ linguistischer Theorien zur Schaffung einer konfortablen Mensch-Maschine-Schnittstelle von Bedeutung.
Eines der Hauptprobleme bei der Verarbeitung natürlicher Sprachen ist der hohe Grad an Ambiguität der sprachlichen Äußerungen. Neben lexikalischen Ambiguitäten, die auftreten, wenn ein Wort unterschiedliche Objekte in der Welt beschreibt (bspw. „Schloss“) und syntaktischen Ambiguitäten (bspw. „Die Frau kitzelt das Kind mit der roten Feder.“) spielen vor allem auch semantische Ambiguitäten eine Rolle, die durch den Gebrauch quantifizierender Ausdrücke und deren Skopusverhalten zustande kommen (bsw. „Jeder Student liest ein Buch “). Damit die Verarbeitung natürlicher Sprache durch eine Maschine bewältigt werden kann, müssen Lösungen für auftretende Ambiguitäten gefunden werden. Ebebfalls dürfen durch die Verarbeitung keine neuen Ambiguitäten auftreten.

1.1.2 Gegenstand der Untersuchung

Gegenstand dieser interdiziplinären Arbeit zwischen Informatik und Linguistik ist die Untersuchung des Zusammenspiels von natürlichsprachlicher Semantik und der Semantik funktionaler Programmiersprachen. Grundlage für die linguistische Semantik sind hier die Arbeiten von Richard Montague. Kalküle funktionaler Programmiersprachen sind durch eine operationale Semantik gegeben. Beiden gemeinsam ist die Verwendung des λ-Kalküls, eines mathematischen Formalismus zur Beschreibung von Funktionen durch Rechenvorschriften. Ein Interpreter für natürliche Sprache in einer funktionalen Programmiersprache muss beide Semantiken berücksichtigen, damit beide so ineinandergreifen, dass die Auswertung der eingegebenen Ausdrücke korrekt ist. Hierzu werden natürlichsprachliche Ausdrücke unter Verwendung von Montagues modelltheoretischem Ansatz in λ-Ausdrücke umgeformt und durch die Transformationsregeln der λ-Kalküle der Programmiersprachen ausgewertet. Eine besondere Situation entsteht durch die Einführung einer nicht-deterministischen Variable und deren Auswertung in einer bestimmten Umgebung. Der Schwerpunkt dieser Arbeit liegt dabei nicht auf der Implementierung eines Interpreters für natürliche Sprache, sondern auf der Untersuchung der unterschiedlichen Kalküle. Zu Testzwecken wird zunächst ein bereits entworfener Interpreter auf Basis der Montague-Semantik, der in der funktionalen Programmiersprache Haskell implementiert ist, vorgestellt, und um nicht-Determinismus erweitert. Dann wird das genaue Zusammenspiel zwischen der Semantik der Programmiersprache und der Semantik des natürlichsprachlichen Ausdrucks unter Verwendung der nicht-deterministischen Variable daraufhin untersucht, ob die Transformationen für sprachliche Ausdrücke dieser Art korrekt sind.

1.1.3 Überblick

Die folgenden Abschnitte dieses Kapitels sollen eine Einleitung zu den Grundkonzepten des klassischen λ-Kalküls und der funktionalen Programmierung geben, soweit sie in dieser Arbeit relevant sind. Ein letzer Abschnitt führt in die Implementierung des getesteten Interpreters ein.
In *Kapitel 2* wird der modelltheoretische Ansatz Montagues zur Erfassung sprachlicher Ausdrücke und ihrer Bedeutung eingeführt. Darauf folgt eine Analyse der Arbeitsweise des getesteten Interpreters und seiner Nähe zur Montague-Semantik. Dann wird das Sprachfragment, welches durch den Interpreter gegeben ist, in die Montague-Semantik übertragen, so dass sämtliche Ausdrücke, welche für die Eingabe in den Interpreter geeignet sind, in entsprechende λ-Ausdrücke übertragen werden können. Dieser Ansatz ist in Bezug auf die Originalarbeiten Montagues in der Weise modifiziert, als es hier nur um die Extension der sprachlichen Ausdrücke geht. Anhand von Beispielen wird diese Übertragung dann anschaulich gemacht.
In *Kapitel 3* wird zunächst der Choice-Operator eingeführt. Über diesen wird dann die nicht-deterministische Variable definiert. Außerdem wird ein kleiner Einblick in Choice-Funktionen und deren Handhabung in der Linguistik gegeben. Nachfolgend werden zwei unterschiedliche Kalküle auf die Korrektheit ihrer Transformationen unter Verwendung von Ausdrücken, welche die nicht-deterministische Variable enthalten untersucht.
Kapitel 4 gibt eine kurze Zusammenfassung, zeigt noch einmal auf, wie Interpreter-Semantik und die Semantik der Kalküle ineinandergreifen und beschäftigt sich mit der hier eingeführten nicht-deterministischen Variable.

1.2 Grundlagen

1.2.1 Funktion und Argument - Extension und Intension

Funktion und Argument

Frege weist in seinem Aufsatz *Funktion und Begriff* [Fre90] darauf hin, dass man bei einem Ausdruck zwischen der Funktion selbst (bei Frege das Prädikat genannt) und dem darauf angewendeten Argument unterscheiden muss. Die Funktion hat die Eigenschaft für sich genommen ungesättigt zu sein. Die Sättigung ergibt sich dann aus der Anwendung des Argumentes auf die Funktion.

> „Es kommt mir darauf an, zu zeigen, dass das Argument nicht mit zur Funktion gehört, sondern mit der Funktion zusammen ein vollständiges Ganzes

> bildet; denn die Funktion für sich allein ist unvollständig, ergänzungsbedürftig oder ungesättigt zu nennen."
>
> [Fre80], S.21f

Unter einer Funktion kann man nun nicht nur mathematische Funktionen verstehen, sondern auch einen ungesättigten sprachlichen Ausdruck. Frege zeigt dies an dem Beispiel: „Caesar eroberte Gallien". Dieses Beispiel läßt sich „zerlegen in ‚Caesar' und ‚eroberte Gallien'. Der zweite Teil ist ungesättigt, „führt eine leere Stelle mit sich, und erst dadurch, dass diese Stelle von einem Eigennamen ausgefüllt wird oder von einem Ausdrucke, der einen Eigennamen vertritt, kommt ein abgeschlossener Sinn zum Vorschein."(S.29) Auch hier nennt Frege den zweiten Teil des Ausdrucks Funktion und Caesar ist das Argument.

Man kann zwei unterschiedliche Arten dieser ungesättigten Ausdrücke unterscheiden. Nehmen wir einen Satz wie „Peter schläft". Hier kann man unterscheiden zwischen dem Funktionsausdruck „schläft" der eine Ergänzung erfordert, und dem Argument „Peter". Durch die Anwendung eines Argumentes auf einen solchen ungesätigten Ausdruck, ergibt sich also ein vollständiger Satz. Bei Frege ist diese Art des ungesättigten Ausdrucks ein Funktionsausdruck *erster Ordnung*. Ein ebensolcher Ausdruck ist „liebt Maria". „liebt" ist ein Ausdruck welcher zwei Argumente benötigt. Mit dem Argument „Maria" wird dieser zu einer Funktion, welche nur noch ein einziges Argument erfordert und damit gleichwertig ist mit dem Ausdruck „schläft".

Ausdruck 1.Ordnung

Eine andere Art ungesättigter Ausdrücke begegnet uns in „alle Menschen". Der Ausdruck „alle" ist ungesättigt und wird mit dem Argument „Menschen" zu einem gesättigten Ausdruck, der als Argument für einen Funktionsausdruck erster Ordnung dienen kann. Diese Art von ungesättigtem Ausdruck wird als Funktionsausdruck *zweiter Ordung* bezeichnet. Die Ausdrücke zweiter Ordnung kennen wir heute als Quantoren.

Ausdruck 2.Ordnung

Ein weiterer Aspekt der hier Beachtung finden muss, ist Freges Unterscheidung von *Sinn* und *Bedeutung*. Dieser Unterschied ergibt sich aus der Beobachtung, dass ein Satz unabhängig davon, ob er wahr ist oder nicht, in der Lage ist, eine Information zu transportieren. Als Beispiel führt er den Satz „Der Abendstern ist der Morgenstern." an. Sowohl Abendstern als auch Morgenstern beziehen sich auf die selbe Sache, nämlich auf einen Planeten unseres Sonnensystems namens Venus. Dieser Sachbezug wird von Frege als Bedeutung bezeichnet. Später, angefangen bei Russell, wird daraus der Begriff der Extension. Beispielsweise sind die Extensionen von Eigennamen die Namensträger. Die Extensionen von intransitiven Verben sind Mengen von Individuen auf welche die Tätigkeit, die das Verb beschreibt, zutrifft (z.B. *schlafen* hat als Extension die Menge aller Individuen, die schlafen). Die Extension eines Satzes ist sein Wahrheitswert.[1]

Extension

Der Sinn eines Ausdrucks ist die Information, die er transportiert. Später wird hierfür der Begriff der Intension geprägt. Intension und Extension sind nicht identisch mit Sinn

Intension

[1] Dies ergibt sich aus dem Parallelismusargument: der Sachbezug eines bitransitiven Verbes wie ‚schenken' ist ein 3-Tupel der Form $\{(x, y, z)|x \textit{ schenkt } z \textit{ dem } y\}$, der Sachbezug eines transitiven

und Bedeutung, gehen aber daraus hervor. Hier genügt zu verstehen, worin der eigentliche Unterschied zwischen den beiden liegt. Daher zurück zum Beispiel: Eine Person versteht sowohl die Bedeutung des Ausdrucks ‚Abendstern' wie auch die Bedeutung des Ausdrucks ‚Morgenstern'. Wäre die Bedeutung eines Ausdrucks gleich seinem Sinn, so würden zwei Ausdrücke mit derselben Bedeutung auch den selben Sinn teilen. Es ist also nicht möglich, dass jemand den Beispielsatz versteht ohne gleichzeitig auch entscheiden zu können, ob der Satz wahr ist. Daraus ergibt sich aber, dass dieser Satz nicht dazu geeignet ist irgendeine Information zu transportieren, da der Satz ja nur mitteilt, was hinlänglich bekannt ist. Dies ist aber nicht der Fall. Manchmal wird die Intension eines Satzes auch als Urteil, welches dieser Satz ausdrückt, bezeichnet. Die Extension hingegen besagt, ob dieses Urteil wahr oder falsch ist.

1.2.2 Der λ-Kalkül

Anfang der 30er-Jahre entwickelte Alonzo Church den λ-Kalkül, um den Begriff der Berechenbarkeit präzise zu formulieren. Sämtliche Möglichkeiten einen Algorithmus zu präzisieren, d.h. eine lückenlose Beschreibung für eine effektive Berechnung aufzustellen [2], führen alle auf die Klasse der berechenbaren Funktionen zurück, die durch das Konzept der λ-Definierbarkeit ausgedrückt werden kann. Diese Beobachtung führte Church zu der nach ihm benannten These:

Church'sche These

Jede intuitiv berechenbare Funktion ist Turing-berechenbar

Turing[Tur37] zeigt, das Turing-berechenbar äquivalent ist mit λ-Definierbarkeit. Funktionale Programmiersprachen basieren im Kern auf der Grundidee des λ-Kalküls, dazu im entsprechenden Abschnitt mehr. Im folgenden wird hier das klassische λ-Kalkül kurz dargestellt. Für eine intensivere Auseinandersetzung sei hier [Bar81] empfohlen.

λ-Term

Ein λ-Term ist ein Wort über folgendem Alphabet:

$v_0, v_1, ...$	Variablen
λ	Abstraktor
$(,)$	Klammern

Verbs wie ‚lieben'ist ein 2-Tupel der Form $\{(x,y)|x\ liebt\ y\}$, der Sachbezug eines intransitiven Verbs wie schlafen ist ein 1-Tupel der Form $\{x|x\ schlaeft\}$. Das heißt, der Sachbezug der Ausdrücke hängt zusammen mit ihrer Valenz (die Fähigkeit eine gewisse Anzahl von Argumenten aufzunehmen)→ Paralellismusargument. Ein Satz hat die Valenz 0, da er voll gesättigt ist. Nun können wir mit dem Parallelismusargument schließen, dass der Sachbezug eines Satzes „Peter schläft" ein 0-Tupel der Form $\{()|Peter\ schlaeft\}$ ist. Ist der Satz wahr, so befindet sich in der Menge des Sachbezuges gerade das leere Tupel, also $\{\emptyset\}$. Ist der Satz jedoch falsch, ist die Menge die leere Menge, also $\emptyset$. Sagen wir nun zwei Mengen sind dann gleich, wenn sie dieselben Elemente haben, so ergeben sich für alle beliebigen Sätze diese beiden möglichen Sachbezüge.

[2] zum Beispiel anhand von Turing-Maschinen, Register-Maschinen, μ-rekursiven Funktionen und dergleichen

Die Variablen können sowohl in gebundener als auch in freier Form vorkommen.

„Eine Variable x kommt frei in einem λ-Term M vor, wenn x nicht im Skopus eines λx liegt. Sonst kommt x gebunden vor.“[Bar81],S.24

freie/gebundene Variable

Der Skopus ist der Wirkungsbereich des Abstraktors. Bei einem Ausdruck λx.s ist dieser Skopus s und alle darin vorkommenden Variablen x sind durch das λ gebunden. Das gilt allerdings nur, wenn in s nicht noch ein λx vorkommt. In diesem Fall wird die Variable immer von dem Abstraktor gebunden, der ihr am nächsten steht. Zum Beispiel ist λx.(x+λx.x) nicht dasselbe wie λx.(x+x). Das sieht man leicht, wenn man beiden dasselbe Argument gibt: (λx.(x+λx.x))(5) ist mit β-Konversion (siehe unten) 5+λx.x. Hier wird noch ein zweites Argument benötigt, zum Beispiel 3: 5+(λx.x)(3) → 5+3=8. Gibt man dem zweiten Ausdruck das Argument 5 so erhält man: λx.(x+x)(5) → 5+5=10 und es ist kein weiteres Argument mehr nötig.

Applikation/ Abstraktion

Für λ-Terme gilt die Applikation und die Abstraktion:

Applikation:
Sind M und N λ-Terme dann ist auch N angewendet auf M ein λ-Term. Man schreibt M(N) oder auch (MN).

Abstraktion:
Ist x eine Variable und M ein λ-Term, dann ist die Abstraktion $\lambda x.M$ ebenfalls ein λ-Term.

Konversionen

Neben λ-Termen gibt es noch die λ-Konversionen. Diese sind Regeln, welche auf λ-Terme angewendet werden können. Die wichtigsten sind die α- und die β-Konversion . Sie reichen im Prinzip aus, um λ-Terme auszuwerten:

β-Konversion:
$(\lambda x.M)N = M[x := N]$
Erklärung: Die β-Konversion ist die Anwendung einer Funktion auf ein Argument. Dabei werden alle Vorkommen der durch den Abstraktor gebundenen Variable durch das Argument ersetzt.

α-Konversion:
$\lambda x.M = \lambda y.M[x := y]$ und y kommt nicht in M vor
Erklärung: Die α-Koversion ist eine Umbenennung von gebundenen Variablen. Die neue Variable darf weder frei noch gebunden in M vorkommen. Die Umbenennung darf keine neue Bindung zur Folge haben.

Obgleich das λ-Kalkül auf den ersten Blick recht einfach aussieht, ist es sehr mächtig. Wie bereits erwähnt lässt sich zeigen, dass sich durch λ-Terme genau die Klasse der berechenbaren Funktionen darstellen lässt.

1.2.3 funktionale Programmierung

Ein Programm in einer funktionalen Programmiersprache besteht aus einer Menge von Ausdrücken, welche Funktionen definieren. Diese Funktionen werden auf andere Ausdrücke angewendet (Applikation). So berechnet ein Programm die Ausgabe aus einer Eingabe durch Anwendung der in ihm definierten Funktionalgleichungen. Dabei sind einige elementare Ausdrücke vorgegeben mit deren Hilfe komplexere Funktionen definiert werden können. Funktionale Programmiersprachen sind sehr mächtig. Dieser Ansatz ist nach der Church'schen These dazu geeignet, alle algorithmischen Aufgabenstellungen zu bearbeiten, denn Basis aller funktionalen Programmiersprachen ist das λ-Kalkül.
Beispiele für Funktionen sind:

quad(x)=x·x	quadriere x
double(x) = x+x	verdoppele x
qu_do(x) = quad(double(x))	quadriere das doppelte von x

Auswertungsstrategien

Eines der wichtigsten Unterscheidungskriterien funktionaler Programmiersprachen ist ihre Auswertungsstrategie. Hier untertscheidet man zunächst call-by-value und call-by-name.

call-by-value
Die call-by-value Strategie, wertet zunächst das Argument bis zu einer Normalform aus und wendet dies dann auf die Funktion an. In Normalform ist hier ein Ausdruck, der nicht mehr weiter ausgewertet werden kann. Diese Strategie bezeichnet man auch als strikte Auswertung.

Beispiel 1 *Wie wird qu_do(2) mit der call-by-value Strategie ausgewertet?*
qu_do(2) = quad(double(2))
$\rightarrow$ *qu_do(2) = quad(2+2)*
$\rightarrow$ *qu_do(2) = quad(4)*
$\rightarrow$ *qu_do(2) = 4·4*
$\rightarrow$ *qu_do(2) = 16*

Die Auswertung ist beendet, da sie einen Ausdruck, hier eine Zahl, erreicht hat, welche in Normalform ist.

call-by-name
Die call-by-name Strategie lässt das Argument zunächst unausgewertet und wendet es so auf die Funktion an. Erst nach einsetzen wird das Argument dann ausgewertet.Diese Strategie bezeichnet man auch als nicht-strikte Auswertung. Diese Auswertung entspricht im klassischen Lambda-Kalkül mit [Bar84] (Definition 13.2.1) der Reduktion am weitesten links stehenden Redex.

Beispiel 2 *Wie wird qu_ do(2) mit der call-by-name Strategie ausgewertet?*
qu_ do(2) = quad(double(2))
$\rightarrow$ *qu_ do(2) = double(2)·double(2)*
$\rightarrow$ *qudo(2) = (2+2)·(2+2)*
$\rightarrow$ *qu_ do(2) = 4·4*
$\rightarrow$ *qu_ do(2) = 16*

Die call-by-name Methode muss den Ausdruck 2+2 zweimal berechnen und sich die Ergebnisse merken, so dass diese Methode zunächst einmal weniger effizient ist, sowohl vom Platzbedarf, als auch vom Rechenaufwand aus gesehen. Dennoch gibt es Gründe, call-by-name call-by-value vorzuziehen. Betrachten wir zwei andere Funktionen:

eq (x) = eq (x)
konst(x) = 1

Und berechnen wir den Ausdruck *konst(eq(2))*. Bei der Auswertung durch call-by-value kommen wir hier nicht weiter. Bei der Auswertung des Arguments erhalten wir eine nicht endende Berechnung, so dass die Berechnung des gesamten Ausdrucks nicht terminiert. Die call-by-name Methode hingegen liefert bereits nach einem Schritt den Wert 1.
Eine weitere Auswertungsstrategie, vereint die Vorteile der call-by-name Strategie mit den Vorteilen der call-by-value Strategie. Es ist die call-by-need Strategie.

call-by-need
Call-by-need setzt zunächst das unausgewertete Argument in die Funktion ein, muss allerdings den Wert des Arguments nur einmal berechnen. Dies bewerkstelligt sie, indem sie, statt der von call-by-value verwendeten Substitution, Sharing benutzt.

Beispiel 3 *Wie wird qu_ do(2) mit der call-by-need Strategie ausgewertet?*
qu_ do(2) = quad(double(2))
qu_ do(2) = let x = double(2) in quad(x)
qu_ do(2) = let x = 2+2 in quad(x)
qu_ do(2) = let x = 4 in quad(x)
qu_ do(2) = quad(4)
qu_ do(2) = 4·4
qu_ do(2) = 16

Anstatt wie bei der Substitution jedes Vorkommen der Variable x im Rumpf der Funktion (rechte Seite) durch das Argument zu ersetzen, wird ein Ausdruck der Form *let x=t in*

s gebildet. Dies bedeutet, dass im Ausdruck s alle Variablen x den Wert t bekommen, so dass x nur einmal für den ganzen Ausdruck berechnet werden muss, unabhängig davon, wie oft es in s vorkommt.

Variablen

Ein weiteres wichtiges Merkmal funktionaler Programmiersprachen ist in den Beispielen erkennbar: Die Bedeutung der Variable ist hier eine andere, als in imperativen Programmiersprachen. Variablen sind keine Container, denen Werte zugewiesen werden, sondern sie sind Platzhalter für Funktionen und Werte mit denen im mathematischen Sinne gerechnet werden kann.

pure/ impure

Ein letzer Blick fällt auf ein weiteres Merkmal funktionaler Programmiersprachen. Diese können *pure* oder *impure* sein. Pure funktionale Programmiersprachen bewahren die referentielle Transparenz ihrer Ausdrücke:

referentielle Transparenz

Ein Ausdruck t ist referentiell transparent, wenn jeder Subausdruck von t und sein zugehöriger Wert (das Ergebnis der Auswertung des Subausdrucks) ausgetauscht werden können, ohne dass sich dadurch der (mittels einer festen Auswertungsstrategie berechnete) Wert von t ändert.[Kut200],S.5

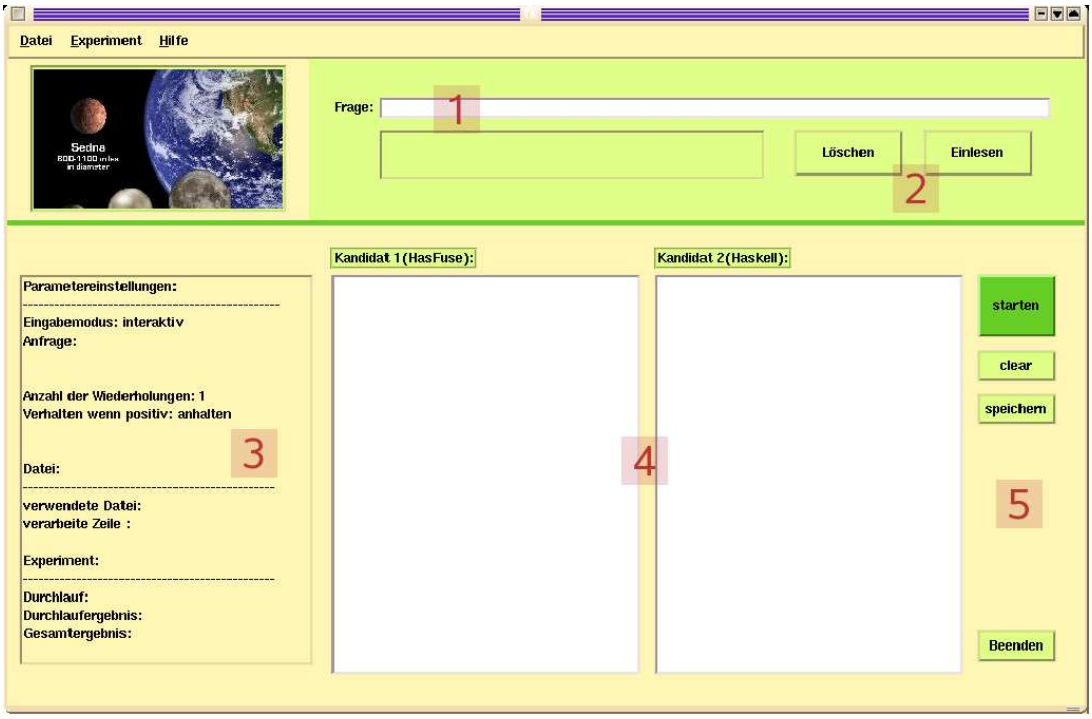
Pure Programmiersprachen erlauben nur solche Erweiterungen, welche mit dem klassischen λ-Kalkül vereinbar sind, während impure Programmiersprachen auch Ausdrücke beinhalten, welche die referentielle Transparenz verletzen[3].

1.3 Der Interpreter

Dieser Abschnitt führt in die allgemeine Implementierung des Interpreters ein und stellt eine erste Vermutung auf.
Es wurde der Interpreter für natürliche Sprache aus [FL89] in der puren funktionalen Programmiersprache Haskell implementiert und um nicht-Determinismus erweitert. Das Programm wurde einmal mit dem ghc[4]-5.04.3 , welcher lazy evaluation für die Auswertung verwendet, und mit dem HasFuse-Compiler kompiliert, welcher auf diesem ghc basiert, jedoch eine call-by-need Strategie verwendet und ein unsafe-Perform IO bietet. Lazy evaluation ist sehr nahe an dem was call-by-need ist, jedoch kann es hier gelegentlich zur Verdoppelung von Ausdrücken kommen, was bei call-by-need nicht geschieht. Bei call-by-name hingegen wird prinzipiell verdoppelt. Diese beiden Interpreterversionen wurden auf die Verarbeitung sprachlicher Ausdrücke untersucht. Die Vermutung war, dass beide Versionen sich durch die unterschiedlichen Auswerungsstrategien in manchen Situationen nicht gleich verhalten.
Um diese beiden Versionen komfortabler zu testen, wurde ein GUI in Python erstellt.

[3] bspw. Zuweisungen oder Exceptions
[4] Glasgow Haskell Compiler

Abbildung 1.1: Screenshot Benutzeroberfläche



Hierbei haben die durch Zahlen gekennzeichneten Bereiche folgende Funktion:

1 Eingabeleiste: Im interaktiven Modus wird hier die Frage eingegeben.
Im Datei Modus wird der Name der Datei angegeben, aus welcher die Fragen ausgelesen werden.

2 Textbuttons: Einlesen liest die Frage in den Interpreter ein.
Sie erscheint dann in der Parameterliste als aktuelle Frage.

Löschen löscht die aktuelle Eingabe (auch aus der Parameterliste).

3 Parameter: Diese Anzeige besteht aus vier Bereichen:

(1) Anzeige des Eingabemodus. Mögliche Werte: interaktiv/aus Datei
Anzeige der aktuellen Anfrage (nur diese wird bearbeitet)

(2) Experimentparameter:
Anzahl der Wiederholungen gibt an, wie oft die Frage den Kandidaten gestellt werden soll.

Mögliche Werte: 1-20
Verhalten wenn positiv gibt an, ob gestoppt werden soll,
wenn das Ergebnis einer Anfrage an die Kandidaten positiv ausfällt.
Mögliche Werte: anhalten/weiter
Die Verhaltensparameter können über die Menüleiste Eintrag
Experiment → *Parameter festlegen* gesetzt werden.

(3) *Datei*: Zeigt im Dateimodus den Namen der verwendeten Datei und die aktuelle Zeilennummer. Es wird immer eine Zeile der Datei eingelesen, so dass die verwendete Textdatei pro Zeile eine Frage enthalten sollte. Die vorgegebene Datei *Fragen.txt* enthält die Fragen aus [FL89].

(4) *Experiment:* zeigt die Nummer des aktuellen Durchlaufs und sein Ergebnis an. Das Ergebnis ist negativ, wenn beide Kandidaten gleich antworten und positiv, wenn beide Kandidaten etwas Unterschiedliches antworten. Das Gesamtergebnis wird positiv, wenn mindestens ein Teilergebnis positiv ausgefallen ist.

4 Kadidaten: Zeigt nach Start der Anfrage die Frage, die Anfragenummer, die Durchlaufnummer der Form *Anfragenummer.Durchlaufnummer*, die Antwort und das Ergebnis des Durchlaufes und des aktuellen Gesamtdurchlaufs.

5 Bedienung: Starten startet die Anfrage. Es wird so oft gefragt, wie Wiederholungen eingestellt sind.

clear löscht die Antwortfelder der Kandidaten.

speichern Speichert den aktuellen Inhalt der Antwortfelder als Protokoll.
Dieses Protokoll kann über die Menüleiste mit
Experiment → *Protokoll anzeigen* abgerufen werden.
Die Bezeichnung des Protokolls ist dabei jeweils Ptk_Datum_Uhrzeit.

Beenden beendet das Programm.

Eine Anfrage an den Interpreter wird folgendermaßen gestellt:
Im interaktiven Modus die Frage eingeben und Einlesen.
Die Wörter müssen auch mit ihrer Groß- und Kleinschreibung wie im Vokabular geschrieben werden(keine Großschreibung am Satzanfang). Die Frage/der Satz endet immer mit ? (auch bei indirekten Fragen).

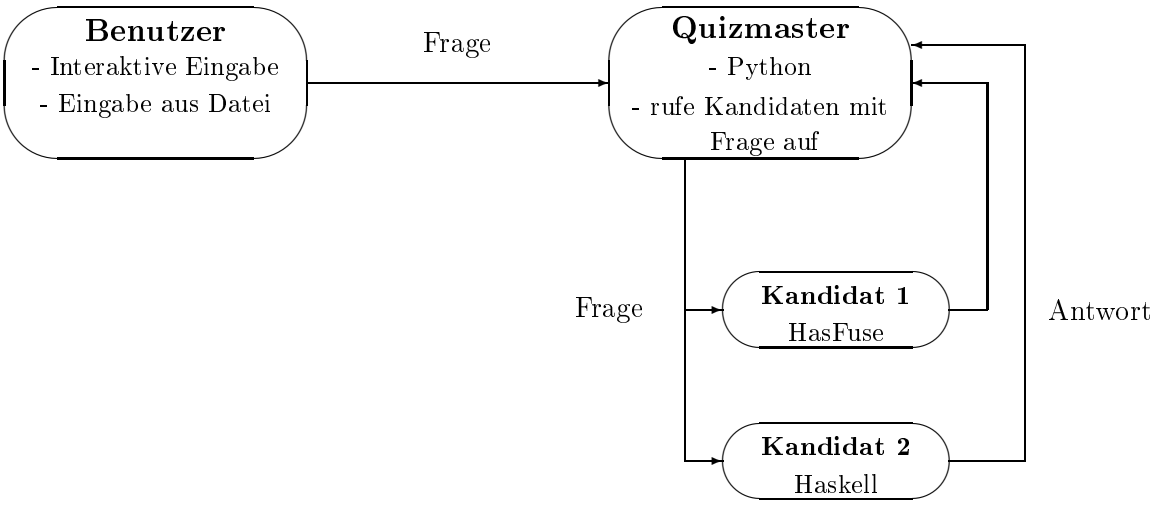
Im Dateimodus den Namen der Datei angeben und Einlesen. Die Datei muss eine Textdatei mit Endung .txt sein und darf pro Zeile eine Frage enthalten, die ebenfalls auf die spezielle Groß- und Kleinschreibung achtet und stets mit ? endet. Ist die Frage in der Parameteranzeige sichtbar, kann Start betätigt werden. Es wird nur die Frage in der Parameteranzeige berücksichtigt. Die Parameteranzeige kann entweder durch erneutes Einlesen einer Frage überschrieben werden oder durch löschen gelöscht werden. Ausführlichere Erklärungen zum Programm finden sich in der Menüleiste unter *Hilfe → Dokumentation*.

Bei einer Anfrage agiert das Python-Programm wie eine Art Quiz-Master. Seine beiden Kandidaten sind:

Kandidat 1 : HasFuse
Kandidat 2 : Haskell

Wie schon erwähnt haben beiden Kandidaten den selben Quellcode[5] und sind lediglich mit zwei unterschiedlichen Compilern übersetzt. Die Frage wird über den Quiz-Master an die beiden Kandidaten weiter gegeben, welche dann ihre Antwort an ihn zurück geben. Dabei wird auf der Standardausgabe die Auswahl sichtbar, welche die Kandidaten während der Beantwortung der Frage machen. Dies geschieht nur dort, wo die Frage nicht-deterministisch ist. Die Vermutung war, dass der mit HasFuse übersetze Interpreter durch seine Auswertungsstrategie jeweils die richtigen Antworten gibt, gerade auch im nicht- deterministischen Fall, wohingegen der mit Haskell übersetzte Interpreter dort unter Umständen einen Fehler macht. Um diesen Fehler zu erkennen, müssen die Fragen so ausgewählt sein, dass es im Fall einer korrekten Auswertung nur eine mögliche Antwort gibt. Ohne zuviel vorzugreifen, soll diese Art von Fangfrage kurz erläutert werden. Nehmen wir einen Satz wie *Peter wohnt in Frankfurt und studiert in Frankfurt* und

Abbildung 1.2: Anfrage an den Interpreter



[5]Listing im Anhang

ersetzen Peter durch die Variable x. Später werden wir sehen das ein Satz wie *x wohnt und studiert in Frankfurt* stark vereinfacht eine Funktion ist, die etwa so aussieht:

λx.wohnt_in_Frankfurt(x) und studiert_in_Frankfurt(x)

Jetzt nehmen wir an, dass Peter zwar in Frankfurt wohnt aber nicht dort studiert und sein Freund Paul zwar in Frankfurt studiert aber nicht dort wohnt. Wird nun für x Peter eingesetzt, erhalten wir eine falsche Aussage, ebenso, wenn für x Paul eingesetzt wird. Da x eine Variable ist, die vom Lambda gebunden wird, dürfen die beiden x im Rumpf des Abstraktors eigentlich nicht zwei verschiedene Werte annehmen. In diesem Fall wäre *Paul wohnt in Frankfurt und Peter studiert in Frankfurt* zwar auch falsch aber *Peter wohn in Frankfurt und Paul studiert in Frankfurt* wäre wahr. Diese Kombination ist aber nicht im Sinne der Aussage *x wohnt und studiert in Frankfurt.* Sehen wir uns die Auswertungsstrategien der Kandidaten an:

1. Kandidat: HasFuse - call-by-need
(λx.wohnt_in_Frankfurt(x) und studiert_in_Frankfurt(x))(Peter oder Paul)
→ let x = (Peter oder Paul) in (wohnt_in_Frankfurt(x) und studiert_in_Frankfurt(x))

Nach Einsetzen wird nun das Argument ausgewertet. Entweder wird nun Peter gewählt:
→ let x = Peter in (wohnt_in_Frankfurt(x) und studiert_in_Frankfurt(x))

oder Paul:
→ let x = Paul in (wohnt_in_Frankfurt(x) und studiert_in_Frankfurt(x))

Ergebnis:
→ Peter wohnt in Frankfurt und Peter studiert in Frankfurt → falsch √
→ Paul wohnt in Frankfurt und Paul studiert in Frankfurt → falsch √

2. Kandidat: Haskell - call-by-name
(λx.wohnt_in_Frankfurt(x) und studiert_in_Frankfurt(x))(Peter oder Paul)
wohnt_in_Frankfurt(Peter oder Paul) und studiert_in_Frankfurt(Peter oder Paul)

Nach Einsetzen wird das Argument nun zweimal separat ausgewertet, jedesmal kann entweder Peter oder Paul gewählt werden:
→ wohnt_in_Frankfurt(Peter) und studiert_in_Frankfurt(Peter)
→ wohnt_in_Frankfurt(Paul) und studiert_in_Frankfurt(Peter)
→ wohnt_in_Frankfurt(Peter) und studiert_in_Frankfurt(Paul)
→ wohnt_in_Frankfurt(Paul) und studiert_in_Frankfurt(Paul)

Ergebnis:
→ Peter wohnt in Frankfurt und Peter studiert in Frankfurt → falsch √
→ Paul wohnt in Frankfurt und Peter studiert in Frankfurt → falsch √
→ Peter wohnt in Frankfurt und Paul studiert in Frankfurt → richtig ★
→ Paul wohnt in Frankfurt und Paul studiert in Frankfurt → falsch √

Im dritten Fall sollte die Haskellversion des Interpreters also „Yes, this is true “antworten, während HasFuse immer Nein sagt. In diesem Fall würde das Ergebnis des Durchlaufs positiv ausfallen. Im zweiten Fall wäre die Auswertung zwar auch nicht korrekt, aber

dies ist dann nicht sichtbar, da sie auf den selben Wert führt wie die korrekte Auswertung. Die einzige Möglichkeit dies zu sehen, ist dann über Anzeige der Auswahl auf der Standardausgabe. Ist die Frage keine Fangfrage, können durch die unterschiedlichen Auswahlmöglichkeiten auch positive Durchläufe entstehen, die aber dann nicht gedeutet werden können, da die Möglichkeiten für korrekte Antworten nicht genug eingegrenzt wurden. Das obige Beispiel kann nicht vom Interpreter ausgewertet werden, da das nötige Vokabular fehlt. Der Interpreter beantwortet Fragen zum Sonnensystem. Das im Interpreter verfügbare Vokabular ist unter $Hilfe \rightarrow Vokabular$ aufgelistet.
Der eben geschilderte Unterschied tritt jedoch nicht auf. Beide Kandidaten verhalten sich bei der Beantwortung der Fragen gleich. Zu einem späteren Zeitpunkt wird diskutiert, warum dies so ist. Zuvor sind jedoch einige Untersuchungen notwendig.

2 Montague-Semantik

„There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians“

R.Montague;UG

Diese Arbeit beschäftigt sich nur mit einem Ausschnitt aus Montagues Arbeit zu einem ganz bestimmten Zweck. Daher ist hier nicht das Ziel, die Semantik Montagues im Ganzen wiederzugeben. Für eine tiefergehende Auseinandersetzung mit der Montague-Semantik sei der Leser auf Dowty, Wall und Peters[DWP81] verwiesen. Es soll hier eine Methode gefunden werden, sprachliche Ausdrücke semantisch korrekt in λ-Ausdrücke umzuformen, die dann weiter verarbeitet werden können.
In seinem Aufsatz ‚Universal Grammar‘(im weiteren UG) entwirft Montague das System einer allgeinen formalen Theorie der Sprache. Im Gegensatz zu der sonst geläufigen Verwendung des Begriffs der „Universalgrammatik“ in der Linguistik als Bezeichnung für angeborene Prinzipien der Sprachfähigkeit, entwirft Montague mit seiner universellen Grammatik eine einzige Theorie, in welcher sowohl natürliche Sprachen, als auch künstliche Sprachen mit ihrer Syntax und Semantik berücksichtigt werden. Genau diese Eigenschaft der Theorie, dass sie auf alle denkbaren natürlichen, künstlichen und logischen Sprachen anwendbar ist, gibt uns ein Werkzeug an die Hand, mit welchem die unterschiedlichsten Sprachen auf einmal formalisierbar und vor allen Dingen auch vergleichbar werden.
Die Theorie von UG beinhaltet grundlegende Konzepte für die Bedeutung von Ausdrücken sowie die Art und Weise, wie diese im Zusammenspiel mehrerer Ausdrücke zustande kommt. Seine Basis ist dabei eine Frege'sche Sicht auf die Sprache. Einzelne Ausdrücke der Sprache, die Basisausdrücke, sind Funktionen, die durch Anwendung von Regeln zu komplexen Ausdrücken zusammengesetzt werden. Die Bedeutung der komplexen Ausdrücke kommt durch die Bedeutung der Basisausdrücke und deren Komposition zustande (Frege'sches Kompositionalitätsprinzip[1]). Komplexe Ausdrücke, welche mehr

Kompositionsprinzip

als eine Bedeutung haben, müssen also auf verschiedene Weisen aus den Regeln herleitbar sein. In UG definiert Montague zunächst eine *disambiguated language* unter Zuhilfenahme einer eindeutigen Klammerung der einzelnen Ausdrücke. Eine Sprache L besteht dann aus der Menge der Ausdrücke A in der disambiguierten Sprache und einer binären Relation R, welche als Funktion auf die Menge angewendet wird, so dass wir sowohl eindeutige als auch ambige Ausdrücke in L haben. Dies geschieht durch das Auflösen der Klammern in der disambiguierten Sprache. Basis dieser disambiguierten Sprache ist eine Algebra.
Die Grammatik von ‚The Proper Treatment of Quanitfication in Ordinary English'(im weiteren PTQ)ist nicht dieselbe wie in UG, sie ist jedoch mit ihr kompatibel. Vor allem sind die *analysis trees* Ausdrücke, denen die disambiguierte Sprache zugrunde liegt. Dabei übernehmen die *structural operation indices* die Aufgabe der Klammerung in der disambiguierten Sprache [Dow79]. Bei Montague ist ein komplexer Ausdruck, wie ein Satz, aufgebaut aus einer Menge von *basic expressions*, welche durch syntaktische Regeln der Sprache zu größeren Ausdrücken zusammengesetzt werden. Einige Basisausdrücke sind Funktionen im Sinne Freges und manche sind Argumente. Wie oben erläutert, werden die Argumente auf die Funktionen angewendet. Diesen Vorgang nennt man funktionale Applikation.

Die Basisausdrücke einer Sprache, welche sich in unterschiedliche syntaktische Kategorien unterteilen lassen, entstehen durch ein Typsystem, welches den extensionalen Gehalt der entsprechenden syntaktischen Kategorie wiederspiegelt. Das folgende Kapitel soll nun diese Dinge näher beleuchten.

[1] obwohl Frege dieses Prinzip niemals explizit formuliert hat, hat seine Theorie der Sprache in welcher Ausdrücke Funktionen und Auswertung Applikationen von Argumenten auf Funktionen sind dieses Prinzip zum ersten mal so angewendet und damit eingeführt

2.1 Aspekte der Montague-Semantik

Ein wesentlicher Teil der Arbeiten UG und PTQ besteht in der Formulierung einer intensionalen Sprache für die Interpretation sprachlicher Ausdrücke. Diese ist ein modelltheoretischer Ansatz, welcher eine Aussage auf ihren intensionalen Gehalt in allen möglichen Welten und mit allen möglichen Variableneinsetzungen untersucht. Für diese Arbeit ist dieser Ansatz jedoch nicht von Belang. Gesucht ist hier nur die extensionale Umsetzung eines natürlichsprachlichen Ausdrucks in einen transformierbaren λ-Ausdruck. Eine vollständige Auswertung mittels PTQ ergibt nicht immer einen λ-Ausdruck. Der entstandene Ausdruck kann jedoch durch eine Abstraktion leicht umgewandelt werden.

2.1.1 Eine disambiguierte Sprache

Die Basis für Montagues Modell bildet eine Algebra. $\langle A, F_\gamma \rangle_{\gamma \in \Gamma}$ wobei A eine nichtleere Menge von Ausdrücken[2] und F_γ mit $\gamma \in \Gamma$ eine Menge von Operatoren auf dieser Menge ist. Für die Menge A soll dadurch sichergestellt werden, dass sie in dem Sinne *disambiguiert* werden kann, dass jeder Ausdruck, der durch Anwendung eines Operators aus F_γ auf irgendeinen Ausdruck des Inputs (unabhängig davon ob er wohlgeformt ist oder nicht) entsteht, von jedem anderen Ausdruck in A unterschieden werden kann. Eine syntaktische Mehrdeutigkeit wird auf diese Weise ausgeschlossen.

disambiguierte Sprache

Definition 1 (Notationen) .....
Folgende Notationen werden im weiteren verwendet:

- α, β, ξ, η, *kleine griechische Buchstaben stehen für Ordinalzahlen*
- A *ist die Menge aller Ausdrücke*
- F_γ *mit* $\gamma \in \Gamma$ *ist die Menge aller strukturellen Operatoren der Sprache welche durch die Menge* Γ *indiziert werden*

[2] Diese Ausdrücke sind alle möglichen Ausdrücke nicht nur die wohlgeformten.

Definition 2 (disambiguierte Sprache) .....
Eine disambiguierte Sprache ist ein 5-Tupel $\langle A, F_\gamma, X_\delta, S, \delta_0 \rangle_{\gamma\in\Gamma,\delta\in\Delta}$, *so dass:*

1. $\langle A, F_\gamma \rangle$ *ist eine Algebra*
2. $\forall\delta \in \Delta$ *ist* X_δ *eine Teilmenge von* A
   die Menge X_δ *ist die Menge der* ***Basisausdrücke*** *in* A, *welche zur Kategorie* δ *aus der Menge aller Kategorien* Δ *gehören.*
3. A *ist die kleinste Menge* $X_\delta \subset A$ *mit* $\delta \in \Delta$ *und* A *ist abgeschlossen unter allen Operationen* F_γ *mit* $\gamma \in \Gamma$
4. X_δ *ist disjunkt vom Wertebereich von* F_γ
5. *jeder Operator in* F_γ *ist nur einmal definiert*
6. S *ist eine Menge von Sequenzen der Form* $\langle F_\gamma, \langle \delta_\xi \rangle_{\xi<\beta}, \varepsilon \rangle$
   mit $\gamma \in \Gamma$ *und* $\delta_\xi, \varepsilon \in \Delta$ *diese Sequenzen bestehen aus einem strukturellen Operator* F_γ *zusammen mit einer syntaktischen Kategorie* δ_ξ *auf die der Operator angewendet wird und einer Kategorie* ε *zu welcher das Ergebnis dieser Operatoranwendung gehört. Diese Sequenzen sind* ***syntaktische Regeln****.*
7. $\delta_0 \in \Delta$, *dies ist die Kategorie der* ***Deklarativsätze***

Eindeutigkeitsklauseln nach welchen die Algebra eine Termalgebra ist:
Ist $\Re =< A, F_\gamma, X_\delta, S, \delta_0 >_{\gamma\in\Gamma,\delta\in\Delta}$, *dann generiert* $\Re$ *die Familie der syntaktischen Kategorien* C *gdw.*

(i) C *ist eine Familie aus Teilmengen von* A *welche durch* Δ *indiziert wird*

(ii) $X_\delta \subseteq C_\delta \forall\delta \in \Delta$

(iii) Ist $\langle F, \langle \delta_\xi \rangle_{\xi<\beta} \rangle \in S$ *(syntaktische Regeln) und* $a_\xi \in C_{\delta\xi}$ *dann ist* $\langle F \langle a_\xi \rangle_{\xi\langle\beta} \rangle \in C_\epsilon$

(iv) Wenn $F_i(x_1...x_n) = F_j(y_1...y_n)$, *dann ist* $i{=}j$ *und es gilt* $x_k = y_k \forall k$

Die Vereinigungsmenge $\bigcup_{\delta\in\Delta} X_\delta$ ist die Menge, welche die Algebra $\langle A, F_\gamma \rangle$ generiert.
Eine **Sprache** ist dann ein Paar, bestehend aus einer disambiguierten Sprache und einer binären Relation R. In UG entwirft Montague hierzu ein Fragement des Englischen

([Mon74],S.237). Auf einer Menge von Basisausdrücken (in UG noch nicht getypt), definiert er eine Anzahl von Klammerregeln. Die Klammerung sorgt für syntaktische Eindeutigkeit. Ein Beispiel[3] für einen Ausdruck in dieser disambiguierten Sprache sieht so aus:

Abbildung 2.1: Disambiguierung durch Klammerung

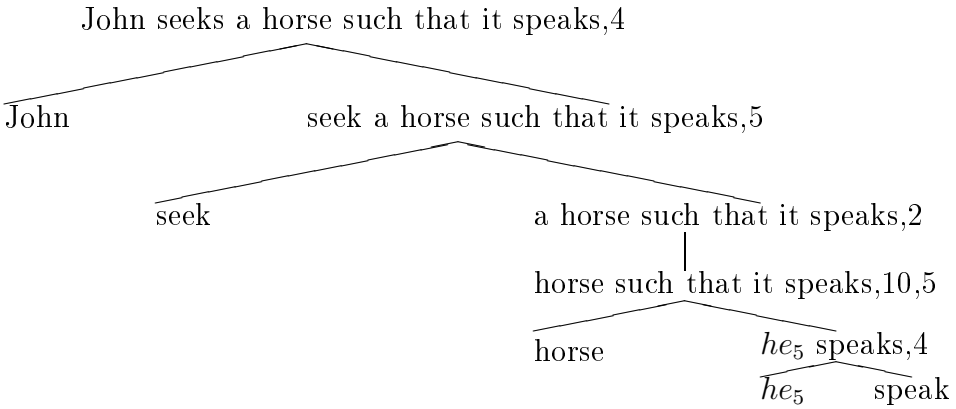
⁅ John ⁀ seeks a ⁅ horse such v_5 that ⁅ it v_5 speaks ⁆ ⁆ ⁀ ⁆

Eine Sprache enthält jedoch sowohl syntaktisch eindeutige, als auch syntaktisch mehrdeutige Konstruktionen. R löscht die Klammerung der disambiguierten Sprache, so dass die Eindeutigkeit unter Umständen verloren gehen kann. Nach Anwendung der ambiguierenden Relation R erhalten wir den Ausdruck: Sprache

John seeks a horse such that it speaks.

Auch in PTQ wird eine Form dieser disambiguierten Sprache verwendet. Dargestellt wird die Disambiguierung jedoch nicht durch Klammerung und deren Auflösung, sondern durch eine Baumstruktur, deren Wurzel den potentiell ambigen Ausdruck enthält und deren Verzweigungen und die Angabe der entsprechenden Regeln den Ausdruck disambiguiert. Das vorherige Beispiel sieht in diesem Fall so aus:

Abbildung 2.2: Disambiguierung durch Baumstruktur



[3] siehe [Dow79]

Die Zahlen hinter den einzelnen Ausdrücken zeigen die Regel an nach welcher eine Verzweigung vorgenommen wird. Löscht man diese Regeln und auch die damit einhergehenden Verzweigungen, so bleibt nur noch der Ausdruck in der Wurzel übrig. Dann erhalten wir wiederum den potentiell ambigen Ausdruck *John seeks a horse such that it speaks.*

2.1.2 Typsystem und syntaktische Kategorien

Typen & syntaktische Kategorien

Durch Einführung eines Typsystems werden in PTQ die disambiguierten Ausdrücke des Englischen kategorisiert. Jeder Kategorie wird ein entsprechender Typ zugewiesen. Das Typsystem baut auf zwei Grundtypen auf, Ausdrücke vom Typ e haben Individuen als Extension, während Ausdrücke vom Typ t Wahrheitswerte als Extension haben. Aus diesen beiden Grundtypen können durch die folgende rekursive Definition beliebige syntaktische Kategorien generiert werden:

Definition 3 (Typen und syntaktische Kategorien) .....
Seien e und t Grundtypen und Cat_α sei die syntaktische Kategorie vom Typ α , dann gilt
1. sind A und B Typen, dann sind auch A/B und A//B Typen[4]*, wobei ein Ausdruck vom Typ A/B (bzw.A//B) ein ungesättigter Ausdruck ist der ein Argument vom Typ B erwartet und durch funktionale Applikation dieses Arguments zu einem Ausdruck vom Typ A wird.*
2. gilt Typ A $\neq$ Typ B, dann gilt auch $Cat_A \neq Cat_B$

In UG sind die Typen durch Aufzählung gegeben. Für den Zusammenhang zwischen Typen und syntaktischen Kategorien in PTQ gilt Typ(B/A) = Typ(B//A) = (a,b), wobei a und b Grundtypen sind. Montague gibt nun die Typen der traditionellen syntaktischen Kategorien an (Tabelle 2.1)

Ein Basisausdruck einer Sprache ist dann ein Element von $\bigcup_{A \in Cat} B_A$. Die Mengen B_e und B_t sind leer. Unter P_A verstehen wir die Menge aller Phrasen vom Typ A. Also zum Beispiel, wie Montague anführt, P_{CN} ist die Menge aller *commonnoun phrases.* Auf diesen Basisausdrücken sind nun syntaktische Regeln definiert. Diese sollen an dieser Stelle nicht aufgezählt werden. Im nächsten Abschnitt werden sie im Rahmen der Funktionsweise eines Interpreters für natürlicher Sprache, welcher auf Montagues Idee aufbaut, in Erscheinung treten und von dort aus näher erläutert.

[4]Semantisch gesehen gibt es keinen Unterschied zwischen A/B und A//B, dieser Unterschied besteht nur auf syntaktischer Ebene.

Tabelle 2.1: syntaktische Kategorien der Basisausdrücke

Bezeichnung	syntaktische Kategorie	Kategorie	Beispiel
IV	intransitive Verbphrase	t/e	run,walk,talk
T	Term	t/IV	John, Mary, ninety
TV	transitive Verben	IV/T	find,lose,eat
IAV	IV-modifizierende Adverbien	IV/IV	rapidly,slowly
CN	Sortale	t//e	man, woman, fish
	Satz-modifizierende Adverbien	t/t	necessarily
	IAV-erzeugende Präpositionen	IAV/T	in, about
	Einstellungsverben	IV/t	believe that
	Infinitiv einbettende Verben	IV//IV	try to

2.2 Frost und Launchburry: ein Fragment

Die Grundidee, welcher [FL89] folgen besteht wie bei Montague in der Verwendung einer syntaktisch nicht ambigen Sprache, so dass das Kompositionsprinzip anwendbar ist, um die Bedeutung komplexer Ausdrücke aus der Bedeutung der Einzelausdrücke abzuleiten. Im Abschnitt 2.2.2 wird nun zunächst auf die syntaktischen Produktionsregeln des Interpreters und ihrem Äquivalent in PTQ (soweit vorhanden) eingegangen. In Abschnitt 2.2.3 werden diesen noch die entsprechenden Transformationsregeln in PTQ hinzugefügt, und damit eine Übersetzungsmöglichkeit gegeben von natürlichsprachlichen Ausdrücken, die im Interpreter möglich sind, in eine für die weitere Arbeit verwendbare Form. Diese Übersetzung erfolgt nach den Regeln in PTQ.

2.2.1 Ein Interpreter für natürliche Sprache

Frost und Lauchburry [FL89] entwerfen einen Interpreter für natürliche Sprache in lazy funktionalen Programmiersprachen. Ihre Strategie ist die Angabe von Produktionsregeln in BNF-Form von denen jede mit einer Transformationsregel verquickt ist. Um in diesem Rahmen sowohl grammatische, als auch semantische Aspekte der Sprache zu unterstützen, entwerfen sie ihren Interpreter nach dem Vorbild Montagues. In diesem Kapitel soll zunächst der Gesamtansatz vorgestellt werden. Dann wird das in [FL89] verwendete Fragment in BNF angegeben. Es wird für die weiteren Untersuchungen eine Implementierung dieses Konzeptes in der Programmiersprache Haskell verwendet, die sich sehr eng an [FL89] hält. Es wurde in dieser Implementierung lediglich das Lexikon erweitert und einige Veränderungen aus programmiertechnischen Gründen vorgenommen. Später

wurde dann noch eine Form von nicht-Determinismus hinzugefügt, welcher jedoch die ursprüngliche Funktionsweise des Programms lediglich dahingehend beeinflusst, dass eine bestimmte Form unvollständiger Ausdrücke ebenfalls ausgewertet werden kann. Verweise auf die Zeilen des Quelltextes sind im weiteren angegeben durch (▶ *Zeilennummer*). Das Listing befindet sich im Anhang.

2.2.2 Basisausdrücke und syntaktische Regeln

Das Lexikon, welches die Wörter bzw. Basisausdrücke der Sprache enthält, wird implementiert als Listen von Paaren und zwar derart, dass jede basis-syntaktische Struktur eine separate Liste erhält. Die in den Listen verwendeten Paare haben die Form (w,t), wobei hier w das Wort und t die zugehörige Übersetzung darstellt. Es ist möglich, dass das selbe Wort in unterschiedlichen syntaktischen Kategorien verwendet wird. Auf diese Weise teilen sich mehrere Wörter ein und dieselbe Übersetzung. Auch möglich, aber hier nicht implementiert, ist, dass ein Wort, welches sich in nur einer syntaktischen Kategorie befindet, auch mehre Übersetzungen bekommen kann. Wörter beschreiben hier Entitäten und Mengen von Entitäten. Diese Entitäten haben nur innerhalb des Interpreters eine Bedeutung, obwohl sie auf bestimmte Weise mit individuellen Objekten in der Welt verknüpft sind. Die hier verwendeten basis-syntatktischen Kategorien und ihre Umsetzung[5]:

Eigennamen

1. Eigennamen (propernouns):
   Die Übersetzung eines Eigennamens ist ein abstraktes Objekt, welches nur innerhalb des Interpreters Bedeutung hat. Dieses Objekt ist eine Funktion, welche Eigenschaften erhält und dann testet, ob der Eigenname über diese Eigenschaft verfügt. Das klingt so etwas merkwürdig. Ein Eigenname hat als Extension eigentlich die Person, die durch den Eigennamen bezeichnet wird. Diese Person verfügt dann über Eigenschaften, die hier getestet werden können. Eigenschaften selbst sind Mengen von Individuen (hier Eigennamen), welche über diese Eigenschaft verfügen. Der Test des abstrakten Objektes, welches als Übersetzung dient, besteht also darin, nachzusehen, ob sich das Individuum, welches getestet werden soll, in der Menge der getesteten Eigenschaft befindet. Darauf wird in Hinblick auf andere syntaktischen Kategorien, die damit in Zusammenhang stehen (bspw. Adjektive), noch einmal eingegangen. Bei Montague ist dies die Menge der Basisausdrücke der Kategorie t/IV. D.h. ein Term ist eine einfach ungesättigte Funktion

[4] Auf die überaus interessante Frage, auf welche Weise sie miteinander verknüpft sind kann hier nicht weiter eingegangen werden.

[5] Die vollständige verwendete Vokabelliste der zugehörigen Implementierung befindet sich im Anhang.

welche als Argument ein intransitives Verb erwartet und dann zu einem Satz auswertet. Die entsprechende syntaktische Regel ist S4 [Mon74,S251], sie wird bei den intransitiven Verben genauer ausgeführt.

Beispiel 4 *(Eigennamen)* .....
Wie wird der Eigenname „mars“ ausgewertet?
„mars “(▶ 54)
(w,t)=(„mars“,test_property_ wrt 12)
test_property_ wrt e ps = elem e ps (▶ 308)
einsetzen: test_property_ wrt 12 ps = elem 12 ps
Bekommt diese Funktion eine Menge ps, testet sie ob 12 ein Element dieser Menge ist.

2. Sortale (commonnouns):
Sortale sind Gemeinnamen. Ihre Extension ist eine Menge von Individuen deren Hyperonym[6] eben das Sortal ist. Beispielsweise beschreibt *Planet* die Menge aller Individuen, die Planeten sind. Im Interpreter ist dies ebenso implementiert. Die Übersetzung eines Sortals ist eine Funktion, welche die Menge aller Entitäten zurück gibt, auf die dieses Sortal zutrifft. Diese Menge kann dann als Input für die Funktion der Übersetzung der Eigennamen verwendet werden. So wird getestet, ob eine Entität, welche durch einen Eigennamen beschrieben ist, als Hyperonym das betreffende Sortal hat. Ein Sortal in PTQ ist ein Ausdruck der Kategorie t//e der *common noun phrases* CN. Sortale

Beispiel 5 *(Sortale)* .....
Auswertung des Ausdrucks „planet “.
„planet “
(w,t)=(„planet“,commonnoun_planet) (▶ 41)
commonnoun_planet=[9..17](▶ 217)

3. Adjektive: Adjektive
Adjektive beschreiben Mengen, welche diejenigen Entitäten enthalten, auf die die Eigenschaft, die das Adjektiv ausdrückt, zutrifft. Beispielsweise sind in der Menge des Adjektivs *rot* alle Entitäten als Elemente enthalten, welche die Eigenschaft haben, rot zu sein.

   In Montagues Fragment in PTQ werden Adjektive nicht verwendet. Allerdings werden Adverbien verwendet, die sich meiner Meinung nach äquivalent verhalten. Adverbien sind Basisausdrücke der Kategorie IV/IV der Adverbien IAV. Wendet man ein intransitives Verb auf ein Adverb an, so erhält man wiederum ein intransitives Verb (wenn auch in modifizierter Form). Es ist möglich auf ein solch modifiziertes intransitives Verb wiederum die Funktion eines Adverbs anzuwenden. Auf diese Weise lassen sich beliebig viele Adverbien an einem Verb verarbeiten. Parallel hierzu lässt sich nun das Adjektiv konstruieren:

[6] Hyperonym = übergeordneter Begriff.

Definition 4 (Adjektiv) .....
Ein Adjektiv ist ein Basisausdruck der syntaktischen Kategorie der CN-modifizierenden Adjektive CTN vom Typ CN/CN.

Das Adjektiv wartet auf einen Ausdruck der Form CN (also ein Sortal) und gibt wiederum ein (modifiziertes) Sortal zurück. Dies läßt sich ebenso wie beim Adverb immer wiederholen. Eine entsprechende Regel zur funktionalen Applikation, ich nenne sie hier S10', da sie parallel zur Regel S10 für Adverbien ist, sieht dann folgendermaßen aus:

Definition 5 (Funktionale Applikation von Adjektiven) .....
S10' If $\delta \in P_{CN/CN}$ and $\beta \in P_{CN}$, then $F_7'(\delta, \beta) \in P_{CN}$, where $F_7'(\delta, \beta) = \delta\beta$.

Ausnahmefälle in denen das Adjektiv auf einen Ausdruck der Kategorie T angewendet wird, wie zum Beispiel *die rote Zora* oder *der eiserne Hans* bei denen Kennzeichnungen zurück gegeben werden und die prädikative Verwendung von Adjektiven wie *Der Planet ist rot.*[7], werden hier außer Acht gelassen.

Beispiel 6 (Auswertung von Adjektiven) .....
Wie wird der Ausdruck „red planet " ausgewertet?
„planet: "
(w,t)=(„planet",commonnoun_planet)(▶ 41)
commonnoun_planet = [9..17](▶ 217)
„red: "
(w,t) =(„red ",adjective_red)(▶ 138)
adjective_red = [12,13,14,22](▶ 240)
Diese beiden Mengen werden später von der Interpretationsfunktion geschnitten, für das Verständnis reicht es hier zunächst, dies folgendermaßen zu schreiben:
$[9..17] \cap [12, 13, 14, 22] = [12, 13, 14]$
Diese Menge kann dann später verwendet werden, um zu testen, ob sich eine bestimmte Entität darin befindet, welche durch einen Eigennamen gegeben ist. Vollständige Auswertungsbeispiele befinden sich am Ende dieses Kapitels.

Determinierer

4. Determinierer:
Determinierer sind Funktionen welche zwei Argumente nehmen, welche Funktionen sind, die eine Menge zurück geben. Determinierer bei Montague gehören nicht mehr zu den Basisausdrücken, wohl aber zu den Basisregeln. Anwendung findet hier die Basisregel S2:

S2 If $\zeta \in P_{CN}$, $then F_0(\zeta), F_1(\zeta), F_2(\zeta) \in P_T$, where
$F_0(\zeta) = every\ \zeta$,
$F_1(\zeta) = the\ \zeta$,

[7] Nach Montague in *English as a formal language*[Mon74],S.212 sollen diese als Ausdrücke der Form ‚Dieser Planet ist ein rotes Objekt' aufgefaßt werden.

> $F_2(\zeta)$ is *a* ζ or *an* ζ according as the first word in ζ takes a or an.
>
> [Mon74] p.251

Beispiel 7 *(Auswertung von Determinierern)*
Wie wird der Ausdruck „a man“ ausgewertet?
„man “:
(w,t) =(„man “, commonnoun_ man)(▶ 35)
commonnoun_ man = [84..96](▶ 223)
„a “:
(w,t)=(„a“, determiner_ a)(▶ 188)
determiner_ a xs ys = length (intersect xs ys) \|=0(▶ 287)
Diese Funkton nimmt zwei Mengen xs und ys und schneidet sie miteinander (intersect), dann überprüft sie, ob die Länge (length) des Schnittes ungleich 0 ist (\|=0). Nun haben wir bereits die Menge [84..96] von „man “. Diese wird eingesetzt:
determiner_ a [84..96] ys = length(intersect [84..96] ys)\|=0
Dies ist nun eine einfach ungesättigte Funktion, die auf ein weiteres Mengenargument wartet. Nehmen wir an, wir geben dieser Funktion die Menge aller Individuen, die schlafen. Dann wird diese Menge geschnitten mit der Menge aller Männer [84..96], ist dieser Schnitt nicht leer, dann gibt es (mindestens) einen Mann der schläft. In diesem Fall gibt die Determiniererfunktion den Wert 1 zurück.

5. Intransitive Verben: intransitive Verben
Intransitive Verben verhalten sich wie Sortale. Ihre Übersetzung ist eine Funktion, welche eine Menge zurück gibt, auf die das Verb zutrifft. Intransitive Verben in PTQ sind Ausdrücke der Kategorie t/e der intransitiven Verbphrasen IV. Sie dienen als Argument für die Funktionen der Kategorien T (den Eigennamen) in Regel S4, IV//IV(den Infinitiv einbettende Verben) in Regel S8, IAV (den Adverbien) in Regel 19 (wird hier nicht verwendet, da sie im Interpreterfragment nicht vorkommen) und für die Konjunktion und Disjunktion von Verbalphrasen in Regel S12 (siehe: 7. Junktoren). Für das Beispiel unten wird die Regel S4 verwendet:

> S4 If $\alpha \in P_{t/IV}$ und$\delta \in P_{IV}$ then $F_4(\alpha, \delta) \in P_t$,where
> $F_4(\alpha, \delta) = \alpha\delta'$ und δ' is the result of replacing
> the first *verb* (i.e. member of $B_{IV}, B_{TV}, B_{IV/t}$ *or* $B_{IV//IV}$) in δ by its third person singular present.

Beispiel 8 *(Auswertung von intransitiven Verben)*
Knüpfen wir an das vorherige Beispiel an, und sehen wir wie der Ausdruck „a man exists“ ausgewertet wird:
„a man “:
wir wissen bereits, dass dies eine einfach ungesättigte Funktion der Form

determiner_ a [84..96] ys = length(intersect [84..96] ys) \=0 ist.
„exists “:
(w,t)=(„exists “, intransverb_ exists)
(▶ 171) intransverb_ exists = entityset(▶ 263)
(also alle Entitäten im Interpreter: entityset=[1..100])
Setzen wir nun ein:
determiner_ a [84..96] [1..100] = length(intersect [84..96] [1..100]) \=0
intersect [84..96] [1..100] = [84..96]
length [84..96] \=0 → true

transitive Verben

6. Transitive Verben:
Ein transitives Verb ist eine Funktion, welche als Argumente zwei Funktionen nimmt, welche auf Mengeninklusion testen, und die positiv getesteten Element in eine Menge schreibt, und dann testet ob eine bestimmte Relation zwischen diesen Mengen gilt. Dies geschieht dadurch, dass die entstandene Menge durch das erste Argument getestet wird. Ist dieser Test erfolgreich dann gilt diese Relation und die Funktion liefert den Wert 1, sonst 0.
Die Kategorie intransitiver Verben ist IV/T und die zugehörige syntaktische Regel bei Montague ist S5:

S5 If $\delta \in P_{IV/T}$ and $\beta \in P_T$, then $F_5(\delta, \beta) \in P_{IV}$, where
$F_5(\delta, \beta) = \delta\beta$ if β does not have the form he_n and
$F_5(\delta, he_n) = \delta\ him_n$

Beispiel 9 *(Auswertung transitiver Verben)*.....
Wie wird der Ausdruck „deimos orbits mars“ ausgewertet?
„deimos“:
(w,t)=(„deimos “, test_property_ wrt 18)(▶ 60)
test_property_ wrt e ps = elem e ps(▶ 308)
diese Funktion testet ob die Entität e ein Element der Menge ps ist. Einsetzen:
test_propertywrt 18 ps = elem 18 ps
„mars“:
(w,t)=(„mars “, test_property_ wrt 12)
test_property_ wrt e ps = elem e ps
test_propertywrt 12 ps = elem 12 ps
„orbits “:
(w,t)=(„orbits “, trans_ verb rel_ orbit)(▶ 173)

```
rel_ orbit = [(x,8)|x<-[9..17]]
             ++ [(67,11),(18,12),(19,12)]
             ++ [(x,13)|x<-[20..35]]
```

++ [(x,14)|x<-[36..52]]
++ [(x,15)|x<-[53..57]]
++ [(x,15)|x<-[68..77]]
++ [(x,16)|x<-[58..65]]
++ [(66,17)]
(▶ 311)

Diese Funktion beschreibt die Relation zwischen zwei Elementen von denen jeweils das eine das andere umkreist. Das erste Element der Paare ist der Umkreisende, das zweite Element der Paare ist der, welcher umkreist wird.

trans_verb rel p = [x|(x,image_x) <- (collect rel), p image_x](▶ 328)

nimmt nun die Relation rel_orbit und testet den hinteren Teil der Paare image_x mit der Funktion für den Mars test_property_wrt 12, dabei merkt er sich das erste Element aller Paare, in denen die 12 als zweites Element auftritt. Das Ergebnis ist in unserem Beispiel die Menge [18,19]. Als nächstes wird durch p image_x die Funktion von Deimos auf die entstandene Menge angewendet, also
test_property_wrt 18 [18,19]
elem 18 [18,19] → true
So erhalten wir als Auswertung den Wahrheitswert 1.

7. Junktoren: Junktoren

Junktoren bilden keine synaktische Kategorie. Sie sind Regeln für funkionale Applikation. Die Junktoren „und“ und „oder“ werden, abhängig von den syntaktischen Kategorien der Ausdrücke übersetzt, die sie miteinander verbinden. In [FL89] heißt es dazu:

> The variety of defintions is a cost associated with the set-based approach. In Montague's method conjunctions are translated to polymorphic functions whose definitions are independent of the syntactic category. (p.112)

[FL89] kennt termphrase_and(▶ 275), verbphrase_and(▶ 280)
und noun_and(▶ 287) und die zugehörigen Disjunktionen.
termphrase_and nimmt zwei Argumente aus der Kategorie der Eigennamen oder aber eine Determiniererphrase bestehend aus einem Determinierer und einem Sortal und als drittes Argument einen Ausdruck, der darauf getestet werden soll. Zunächst wird das erste Argument mit dem dritten Argument getestet, was einen Wahrheitswert ergibt. Dann wird das zweite Argument auf dem dritten Argument getestet, was wiederum einen Wahrheitswert ergibt. Diese beiden Wahrheitswerte werden verundet, so dass sich als Ergebnis wieder ein Wahrheitswert ergibt.
verbphrase_and nimmt als Argumente zwei Mengen, welche aus intransitiven Verben entstanden sind[8] und bildet daraus den Mengenschnitt.

[8] Dazu gehören auch transitive Verben, welche bereits ein Argument bekommen haben, diese sind dann ebenso wie intransitive Verben Funktionen, die nur ein Argument erwarten und können so als intransitive Verben behandelt werden.

Der Name noun_join ist zunächst etwas irreführend. Hier werden nicht zwei Nomen miteinander verbunden, sondern zwei Sätze, denen noch das Subjekt fehlt. Beispielsweise: ... is a red planet and is a blue planet. Die Ausdrücke „is a red planet“ und „is a blue planet“ sind zwei Mengen aus Entitäten, für die gilt, dass sie ein roter Planet sind bzw. dass sie ein blauer Planet sind. Diese beiden Mengen werden durch das noun_and miteinander geschnitten. Diese Menge kann dann wiederum dem Subjekt der syntaktischen Kategorie Eigenname oder aber einer Determiniererphrase als Argument übergeben werden. Die zugehörigen Disjunktionen sind bei verbphrase_or(► 284) und noun_or(► 290) einfache Mengenvereinigungen, bei termphrase_or(► 278) werden die beiden Wahrheitswerte zum Schluß verodert anstatt verundet. Ein sentence_and (_or) wurde hinzugefügt um Regel S11 zu implementieren. Dieses berechnet zunächst den Wahrheitswert der beiden Sätze und verknüpft sie dann entsprechend miteinander. Die Regeln von Montague für die Konjunktion und Disjunktion von Ausdrücken sind die Regeln S11 bis S13 in PTQ. Hier werden die Operatoren F_8 für „und“ und F_9 für „oder“ verwendet:

„S11 If $\Phi, \Psi \in P_t$, then $F_8(\Phi, \Psi), F_9(\Phi, \Psi) \in P_t$,
where $F_8(\Phi, \Psi) = \Phi$ *and* $\Psi, F_9(\Phi, \Psi) = \Phi$ *or* Ψ.“
(Regel für die Konjunktion und Disjunktion zweier Sätze.)
„S12 If $\gamma, \delta \in P_{IV}$, then $F_8(\gamma, \delta), F_9(\gamma, \delta) \in P_{IV}$“
(Regel für die Konjunktion und Disjunktion zweier intransitiver Verben.)
„S13 If $\alpha, \beta \in P_T$, then $F_9(\alpha, \beta) \in P_T$“
(Regel für die Disjunktion zweier *termphrases*.)

[Mon74] p.252

Die beiden Operatoren F_8 und F_9, werden für die syntaktischen Kategorien der Sätze(S11) und der intransitiven Verbphrasen(S12) verwendet. S13 hingegen verwendet nur die Disjunktion für *termphrases*. Montague geht mit dem Weglassen des Operators F_8 für *termphrases* Problemen aus dem Weg, die sich aus dem entstehenden Plural ergeben, wenn man zwei *termphrases* miteinander durch „und“ verbindet. In FL wird diesem Problem begegnet, in dem es zwar ein termphrase_and gibt, die Plurale aber auf die selbe Funktion führen wie Ausdrücke im Singular (siehe Beispiel[9]).

Beispiel 10 *(Junktoren)* .....
Beispiel für die Auswertung eines termphrase_and:
mars and venus are a planet?
„mars “:
(w,t)=(„mars “,test_property_ wrt 12)(► 54)
„and “:
(w,t)=(„and “,termphrase_and)(► 197)
termphrase_and p q x = px && qx (► 275)

[9] Das *Linkingverb* „are“wird ebenso wie „is“ ausgewertet.

„venus “:
(w,t)=(„venus“,test_property_ wrt 10)(▶ 52)
„are a planet “[10]
are a planet wertet aus zu [9..17]:

→ *termphrase_ and test_property_ wrt 12 test_property_ wrt 10 [9..17]=*
test_property_ wrt 12 [9..17] && test_property_ wrt 10 [9..17]
→ *true && true*
→ *true*

8. Indefinitpronomen: Indefinitpronomen
Indefinitpronomen werden in [FL89] paraphrasiert (▶ 152 – 157) und dann ausgewertet. Anstatt die Indefinitpronomen gesondert auszuwerten, übersetzt man sie in die entsprechende Determiniererphrase und wertet sie dann als solche aus (siehe 4. Determinierer).
Als Beispiel soll hier der oben genannte Satz ausgewertet werden:

Beispiel 11 *(everything)* .....
everything spins
→ *every thing spins*
„every“
(w,t)=(„every“,determiner_ every)(▶ 191)
determiner_ every= includes (▶ 293)
„thing “
(w,t)=(„thing“,commonnoun_ thing)(▶ 33)
commonnoun_ thing =
union (union commonnoun_ sun commonnoun_ planet)
(union commonnoun_ moon commonnoun_ person)
=[8..96]
„spins“
(w,t)=(„spins “,intransverb_ spin)(▶ 174)
intransverb_ spin=[8..65](▶ 266)
includes [8..96][8..65] → *false*
denn [8..96] ist nicht vollständig in [8..65] enthalten

An den bisherigen Beispiel war bereits indirekt zu erkennen, wie man von der Auswertung der einzelnen Ausdrücke auf die Auswertung des ganzen Satzes kommt. Die interne Verarbeitung in der Implementierung von FL weicht im technischen Sinne von der Verarbeitung eines Satzes bei Montague ab, auch wenn das Ergebnis letztendlich

[10] Der Einfachheit halber ist hier nicht die ganze Auswertung dieses Ausdrucks über die Funktion *verbphrase*(▶ 453) angegeben, diese Auswertung ergibt jedoch die Menge aller Entitäten für die gilt, dass sie Planeten sind, die Menge [9..17].

dasselbe ist. Dies liegt an dem mengenverarbeitenden Konzept der Implementierung. Hier soll jedoch nur das Fragment extrahiert werden, um die Sätze, die darin möglich sind, durch Montague soweit auszuwerten, dass λ-Ausdrücke entstehen, die dann später durch die Transformationsregeln der zu untersuchenden Kalküle weiter umgewandelt werden können. Daher wird hier nicht weiter darauf eingegangen wie in [FL89] ganze Sätze ausgewertet werden, sondern es werden hierzu die Transformationsregeln aus PTQ verwendet. Zu einem späteren Zeitpunkt kehren wir dann noch einmal zur Verarbeitung von komplexen Ausdrücken zurück, um zu sehen, warum die Implementierung von [FL89] im Falle von nicht-deterministischen Argumenten unabhängig von der verwendeten Programmiersprache keinen Fehler macht, obwohl die Auswertung durch die unterschiedlichen Kalküle sehr wohl Fehler ergeben kann. Die Fragepronomen lassen sich allein durch UG und PTQ nicht einbetten. Dies ist hier auch nicht nötig, da alle direkten Fragen in indirekte Fragen umgewandelt werden können. Siehe hierzu [Kar77]. Für eine Einbindung von Fragepronomen in die Theorie Montagues siehe [GS82].
Abschließend sei hier das Fragment als BNF angegeben:

Q	$\Rightarrow$	S | “does “S | “who “VP_j| “what “VP_j| “which “NC VP_j| “howmany “NC VP_j
S	$\Rightarrow$	NP_jVP_j
VP_j	$\Rightarrow$	$VP\ jv\ VP \mid VP$
NC	$\Rightarrow$	$RNC\ jn\ NC \mid RNC$
NP_j	$\Rightarrow$	$NP\ jt\ NP \mid NP$
VP	$\Rightarrow$	$tVP \mid IV \mid be\ det\ NC$
RNC	$\Rightarrow$	$SNC\ RPN\ VP_j \mid SNC$
jn	$\Rightarrow$	“and“| “or“
NP	$\Rightarrow$	$T \mid DP$
tVP	$\Rightarrow$	$TV\ NP_j \mid be\ ptV\ prp\ NP_j$
IV	$\Rightarrow$	“exist“| “orbit“| “spin“
be	$\Rightarrow$	“is“| “are“| “was“| “were“
det	$\Rightarrow$	“the“| “a“| “some“| “no“| “every“| “all“| “one“
RPN	$\Rightarrow$	“that“| “which“| “who“
SNC	$\Rightarrow$	$CN \mid adje\ CN$
T	$\Rightarrow$	“mars“| “phobos“| “Bernard“|...
DP	$\Rightarrow$	$IPN \mid det\ NC$
TV	$\Rightarrow$	“discover“| “orbit“
ptV	$\Rightarrow$	“discovered“| “orbited“
prp	$\Rightarrow$	“by“
CN	$\Rightarrow$	“thing“| “man“| “woman“| “planet“|...
$adje$	$\Rightarrow$	$adje\ CTN \mid CTN$
CTN	$\Rightarrow$	“atmospheric“| “red“| “gaseous“
IPN	$\Rightarrow$	“someone“| “something“| “somebody“| “everyone“| “everything“
jv	$\Rightarrow$	“and“| “or“
jt	$\Rightarrow$	“and“| “or“

2.3 Transformationsregeln und Übertragung des Fragments

Jede syntaktische Regel in PTQ ist verknüpft mit einer Transformationsregel, die Ausdrücke im Rahmen einer intensionalen Logik auswertet. Hierzu wird ME_a, die Menge der *meaningful expressions* vom Typ a, verwendet, welche die Menge aller wohlgeformten Ausdrücke einer Sprache beschreibt. Die Typen der intensionalen Logik weichen leicht von den bisher verwendeten ab:

meaningful expressions

Die kleinste Menge Y von Typen ist dergestalt, dass
(1) e,t ∈ Y
(2) wenn a,b ∈ Y, dann $\langle a, b\rangle$ ∈Y
(3) wenn a∈Y, dann $\langle s, a\rangle$ ∈Y
s ist der intensionale Typ, der Sinn des Ausdrucks. Für die Zwecke dieser Arbeit wird er nicht benötigt. Nun werden Variablen und Konstanten eingeführt, wobei $v_{n,a}$ die n-te Variable vom Typ a und Con_a die Menge der Konstanten vom Typ a ist. Für die MEs gilt nun folgende Definition:

Definition 6 (Meaningful expressions) .....
Sei ME_a die Menge der meaningful expressions, dann gilt:

1. *Jede Variable und Konstante vom Typ a ist in ME_a.*
2. *Wenn $\alpha \in ME_a$ und u ist eine Variable vom Typ b, dann ist $\lambda u.\alpha \in ME_{b,a}$.*
3. *Wenn $\alpha \in ME_{\langle a,b\rangle}$ und $\beta \in ME_a$, dann gilt $\alpha(\beta) \in ME_b$.*
4. *Wenn $\alpha, \beta \in ME_a$, dann gilt $\alpha = \beta \in ME_t$.*
5. *Wenn $\Phi, \Psi \in ME_t$ und u ist eine Variable, dann gilt $\neg\Phi, [\Phi \wedge \Psi], [\Phi \vee \Psi]$, $[\Phi \rightarrow \Psi], [\Phi \leftrightarrow \Psi], \exists u\Phi, \forall u\Phi \in ME_t$*[11]
6. *Wenn $\alpha \in ME_a$, dann $[^\wedge\alpha] \in ME_{\langle s,a\rangle}$.*
7. *Wenn $\alpha \in \langle s, a\rangle$, dann $[^\vee\alpha] \in ME_a$.*
8. *ME_a ist die Menge aller Objekte α, so dass gilt αRa, wobei R die kleinste Relation ist, so dass 1.-7. gilt.*

[11] die Ausdrücke $\Box\Phi$(=notwendiger Weise),WΦ(= es ist der Fall, dass...) und HΦ(=es war der Fall, dass...), wurden weggelassen, aber auch sie gehören in diesem Fall zu ME_t

Die Ausdrücke in 6. und 7. wollen wir kurz ein wenig näher betrachten. [$^\wedge\alpha$] (=„alpha up“) ist das intensionalisierte α, d.h. dass hier die zu α zugehörige Intension gemeint ist. [$^\vee\alpha$](=„alpha down“) kann man als Umkehrfunktion von [$^\wedge\alpha$] sehen. [$^\vee\alpha$] bewirkt, dass ein intensionalisierter Ausdruck wieder auf die extensionale Ebene zurückkehrt. Da wir hier nur die extensionale Ebene betrachten wollen, legen wir fest, dass kein alpha-up durchgeführt werden soll. Folge davon ist ebenfalls, dass keine Typen auftreten, in welchen der intensionale Typ s auftritt. Die folgenden Regeln werden also an den betreffenden Stellen vereinfacht.

2.3.1 Übertragung der Basisausdrücke

Für die Übertragung beginnnen wir mit den Basisausdrücken. Da die Menge der Ausdrücke in manchen syntaktischen Kategorien groß ist, befindet sich im Anhang ein Vokabular auf welches hier referiert wird. Nach S1 gilt, dass $B_A \subseteq P_A$. Das heißt, dass die Basisausdrücke vom Typ A selbst auch schon als Phrasen der Kategorie A gelten.

Basisausdrücke

Definition 7 (Basisausdrücke des FL-Fragments) .....
Es seinen B_A die Basisausdrücke der syntaktischen Kategorie vom Typ A, dann gilt:

Tabelle 2.2: Basisausdrücke

Bezeichnung	syntaktische Kategorie	Ausdrücke
B_T	Eigennamen	→ Vokabular
B_{CN}	Sortale	→ Vokabular
B_{CTN}	Adjekive	{atmospheric, blue, gaseous, red, ringed, solid, vacuumous}
B_{IV}	intransitive Verben	{ exist, spin}
B_{TV}	transitive Verben	{ discover, orbit}

Aus Gründen der Vereinfachung wurden hier die passiv transitiven Verben weggelassen. Die Indefinitpronomen werden in [FL89] vor der Übersetzung paraphrasiert, daher stellen sie keine eigene syntaktische Kategorie. Da sie wie Sortale sind, können wir sie nach der

Paraphrasierung mit Definition 3(2.) zur syntaktischen Kategorie der Sortale hinzufügen und im gleichen Sinne auswerten. Die Paraphrasierung soll hier äquivalent zu [FL89] vorgenommen werden:

Indefinitpronomen

(1) someone/somebody → „a person“
(2) something → „a thing“
(3) everyone → „every person“
(4) everything → „every thing“

2.3.2 Zusammenstellung der Tranformationsregeln

Wie bereits erwähnt, sollen die Transformationsregeln aus PTQ derart angeglichen werden, dass Umformungen, welche eine Itensionalisierung enthalten abgeändert werden, da wir hier nur die extensionale Ebene verwenden wollen. Jede syntaktische Regel S_n ist verknüpft mit einer zugehörigen Transformationsregel T_n, wobei die Indizes der zusammengehörigen Regeln dieselben sind.

1. Eigennamen

Für die Eigennamen wurde zunächst keine explizite syntaktische Regel angegeben, sondern auf die Regel S4 für intransitive Verben verwiesen. Montague gibt die Transformation von Eigennamen wie folgt an:

Transformation Eigennamen

> T1 (d) John, Mary, Bill, ninety translate into j*, m*, b*, n* respectively. “

Wobei er zuvor definiert, dass wenn $\alpha \in ME_e$ dann gilt, dass α^* mit $P = v_{0,\langle s,\langle\langle s,e\rangle,t\rangle\rangle}$ $\lambda P.P\{^\wedge\alpha\}$ ist. Es gilt die Klammerkonvention $\alpha\{\beta\} = {}^\vee\alpha(\beta)$.
Beobachten wir folgendes in Anlenung an [DWP81]:

(1)	Peter ißt.	E(p)
(2)	Peter friert.	F(p)
(3)	Peter singt.	S(p)

Man kann Sätze der Form Peter P so verstehen, dass das was P ausdrückt, die Eigenschaft hat, für Peter war zu sein, genau dann wenn Peter in der Menge die P denotiert enthalten ist, unabhängig davon was für eine VP P genau ist. So lässt sich definieren, dass Peter den semantischen Wert $\lambda P.P\{^\wedge p\}$) hat. Nach Anwendung der Klammerkonvention ergibt sich $\lambda P.{}^\vee P(^\wedge p)$, $\lambda P.P({}^\vee{}^\wedge p)$ und daraus $\lambda P.P(x)$.

2. nicht-logische Konstanten

Zu den Ausdrücken welche in nicht-logische Konstanten übersetzt werden, gehören die Sortale und Verben. Für intransitive Verben haben wir bereits die syntaktische Regel S4 ausfindig gemacht. Die zugehörige Transformationsregel lautet:

nicht-logische Konstanten

> T4 If $\delta \in P_{t/IV}, \beta \in P_{IV}$, and δ, β translate inot δ', β' respetively, then $F_4(\delta, \beta)$ translates into $\delta'(\beta')$.

Unter Beachtung der Klammerkonvention können wir einen Satz wie „Peter singt“ wie folgt transformieren:

$\lambda P.[P\{p\}](^\wedge singen')$
$\rightarrow {}^\wedge singen'\{p\}$ (β-Konversion)
$\rightarrow {}^{\vee\wedge} singen'(p)$(Klammerkonvention)
α-up und α-down heben sich gegenseitig auf, daher:
$\rightarrow$ singen'(p)

Die Eliminierung von $\vee\wedge$, auch down-up-Cancellation genannt, ist ein Gesetz der intensionalen Typenlogik. Dieses Gesetz gilt aufgrund der modelltheoretischen Semantik bei Montague, welche hier aber nicht gegeben ist. Die down-up-Cancellation ist ein Spezialfall der β-Konversion. Die klassische β-Konversion ist für Montagues Logik nicht allgemein gültig. In den hier betrachteten Fällen jedoch schon, da die Intension außen vor gelassen wird.
Um Ähnliches für Sortale durchzuführen, müssen wir ein wenig vorgreifen und uns der Hilfe des Determinierers ‚a‘ bedienen. In T2 erfahren wir, dass dieser Determinierer mit $\lambda P[\lambda Q.\exists x[P\{x\} \wedge Q\{x\}]]$ transformiert wird. Wir verfolgen die Transformation wieder anhand eines Beispiels, um zu sehen, dass auch Sortale als nicht-logische Konstanten übersetzt werden können. Nehmen wir den Satz „A man walks“:

$\lambda P[\lambda Q.\exists x[P\{x\} \wedge Q\{x\}]](^\wedge walk')(^\wedge man')$
$\rightarrow \lambda Q.\exists x[{}^\wedge walk'\{x\} \wedge Q\{x\}](^\wedge man')$ (β-Konversion)
$\rightarrow \exists x[{}^\wedge walk'\{x\} \wedge {}^\wedge man'\{x\}]$ (β-Konversion)
$\rightarrow \exists x[{}^{\vee\wedge} walk'(x) \wedge {}^{\vee\wedge} man'(x)]$ (Klammerkovention)
eliminiere α-up und α-down:
$\rightarrow \exists x[walk'(x) \wedge man'(x)]$

Im weiteren wird nun die Transformation von Verben V in V' und von Sortalen N in N' verwendet.

3.Adjektive

Da Adjektive nicht in PTQ verwendet werden, folgen wir weiterhin der Strategie, ihre Handhabung denen der Adverbien abzuschauen. Ebenso wie wir aus der syntaktischen Regel S10 für Adverbien die Regel S10' für Adjektive hergeleitet haben, verfahren wir nun mit der Transformationsregel und geben eine von T10 abweichende Definition an:

Definition 8 (Transformationsregel für Adjektive) .....
T10' If $\delta \in P_{CN/CN}, \beta \in P_{CN}$ and δ, β translate into δ', β' respectively, then $F_7'(\delta, \beta)$ translates into $\delta'(\beta')$.

So können wir rot für sich genommen wieder als nicht-logische Konstante übersetzen. Sie beschreibt eine Funktion, welche Eigenschaften von Individuen beschreibt und auch auf

diese angewendet wird (nämlich darauf, ein Planet zu sein). Haben wir beispielsweise die Phrase „roter Planet“ nach T10' also $rot'(planet')$, so entsteht wiederum eine Funktion, welche eine Menge von Individuen beschreibt, auf welche beide Eigenschaften rot zu sein und ein Planet zu sein zutreffen.

Transformation Adjektive

4. Determinierer

Die Transformationsregel, welche zur syntaktischen Regel S2 für Determinierer gehört, lautet in extensionaler Form wie folgt:

T2 If $\zeta \in P_{CN}$ and ζ translates into ζ', then
every ζ translates into $\lambda P.\forall x[\zeta'(x) \rightarrow P(x)]$
the ζ translates into $\lambda P.\exists y[\forall x[\zeta'(x) \leftrightarrow x = y] \wedge P(y)]$
$F_2(\zeta)$[12] translates into $\lambda P \exists x[\zeta'(x) \wedge P(x)]$.

Für den Determinierer ‚a‘ haben wir weiter oben eine Form dieser Transformation verwendet, in welcher das ζ' als Prädikat nach außen gezogen, bzw. noch nicht eingesetzt war. Hier gehen wir jedoch davon aus, dass das ζ schon vorhanden und daher durch eine Anwendung der β-Konversion integriert wurde.

Transformation Determinierer

5. transitive Verben

Die für transitive Verben verwendete syntaktische Regel S5 nimmt einen Ausdruck der Kategorie T und gibt einen Ausdruck der Kategorie IV zurück. Das Resultat dieser Transformation kann dann weiter transformiert werden wie intransitive Verben. Die Transformationsregel für transitive Verben ist die Regel T5:

T5 If $\delta \in P_{IV/T}, \beta \in P_T$ and δ, β translates into δ', β' respectively, then $F_5(\delta, \beta)$ translates into $\delta'(\beta')$.

6. das Verb *be*

Das Verb „be“ nimmt eine Sonderstellung ein. Es wird in PTQ als transitives Verb betrachtet. Dennoch wird es nicht als nicht-logische Konstante übersetzt, sondern gilt als logisches Wort mit der besonderen Transformation(nach[DWP81]):

Transformation Verb be

T1 (b) be translates into $\lambda\wp.\lambda x.\wp\{\hat{}\lambda y.[x = y]\}$

Bei der Transformation eines Ausdrucks *A is B*, wird zunächst *is B* ausgewertet und dies wird dann auf A als Argument angewendet. Im weiteren wird für *be* die extensionale

Form $\lambda \mathrm{P}.\lambda x.\mathrm{P}(\lambda y.[x = y])$ verwendet.

7.Junktoren

Für die Junktoren haben wir wieder jeweils eine mit der syntaktischen Regel verknüpfte Transformationsregel:

T11	If $\phi, \psi \in P_t$ and ϕ, ψ translate into ϕ', ψ' respectively, then ϕ **and** ψ translates into $[\phi \wedge \psi]$, ϕ **or** ψ translates into $[\phi \vee \psi]$.
T12	If $\gamma, \delta \in P_{IV}$ and γ, δ translate into γ', δ' respectively, then γ **and** δ translates into $\lambda x.[\gamma'(x) \wedge \delta'(x)]$, γ **or** δ translates into $\lambda x.[\gamma'(x) \vee \delta'(x)]$.
T13	If $\alpha, \beta \in P_T$ and α, β translate into $\alpha'\beta'$ respectively, then α **or** β translates into $\lambda P.[\alpha'(P) \vee \beta'(P)]$.

2.4 Auswertungsbeispiele

Im folgenden Abschnitt sollen nun diese Regeln anhand von Beispielen anschaulich gemacht werden. Die vollständig ausgewerteten Ausdrücke enthalten jedoch keine λ-Abstraktion mehr. Hier soll nun festgelegt werden, dass Variablen für Individuen aus diesen Ausdrücken herausgezogen werden, indem der Ausdruck derart umgeformt wird, dass die Individuenvariable durch λ-Abstraktion an den Restausdruck gebunden ist, und die Individuenvariable selbst als Argument angehängt wird. Ein Beispiel:
John is a man transformiert zu man'(j). Also kann die Individuenvariable j durch Abstraktion herausgezogen werden und als Argument angehängt werden und wir erhalten $(\lambda x.man'(x))(j)$ was äquivalent ist zu man'(j). Diese Umformung ist nötig, um später mit den Kalkülen darauf zu arbeiten.

Ausdruck: mars spins

Ausdruck	Transformation	Regel	Kategorie
1 mars	$\lambda P.P(m)$	T1(d)	t/IV
2 spins	spin'	nicht-logische Konst.	IV
3 mars spins	$\lambda P.P(m)$(spin')	T4	(t/IV)(IV)
	spin'(m)	fkt. Applikation	t
	$(\lambda x.spin'(x))(m)$	Abstraktion	

Ausdruck: every planet spins

Ausdruck	Transformation	Regel	Kategorie
1 every	$\lambda P.\forall x[\zeta'(x) \rightarrow P(x)]$	T2	
2 planet	planet'	nicht-logische Konst.	CN
3 every planet	$\lambda P.\forall x[planet'(x) \rightarrow P(x)]$	T2	t/IV
4 spins	spin'	nicht-logische Konst.	IV
5 every planet spins	$(\lambda P.\forall x[planet'(x) \rightarrow P(x)]$(spin')	T4	(t/IV)(IV)
6	$\forall x[planet'(x) \rightarrow spin'(x)]$	fkt.Applikation	t
7		Abstraktion???	

Ausdruck: phobos orbits mars

Ausdruck	Transformation	Regel	Kategorie
1 phobos	$\lambda Q.Q(p)$	T1(d)	t/IV
2 mars	$\lambda P.P(m)$	T1(d)	t/IV
3 orbits	orbit'	Basisausdruck	IV/T
4 orbits mars	orbit'$(\lambda P.P(m))$	T5	IV
5 phobos orbits mars	$(\lambda Q.Q(p))(orbit'(\lambda P.P(m)))$	T4	(t/IV)(IV)
	$(orbit'(\lambda P.P(m)))(p)$	fkt.Applikation	t

Weiter lässt sich dieses Beispiel nicht auswerten.

Ausdruck: mars is a red planet

Ausdruck	Transformation	Regel	Kategorie
1 mars	$\lambda P.P(m)$	T1(d)	t/IV
2 is	$\lambda Q.\lambda x.Q(\lambda y.[x = y])$	T1(b)	
3 a	$\lambda R.\exists z.[\zeta'(z) \wedge R(z)]$	T2	
4 planet	planet'	n.-l. Konst.	t//e
5 red	red'(δ')	T10'	CN/CN
6 red planet	red'(planet')		CN
7 a red planet	$(\lambda R.\exists z.[red'(planet')(z) \wedge R(z)])$	T2	T
8 is a red planet	$(\lambda Q.\lambda x.Q(\lambda y.[x = y])$ $(\lambda R.\exists z.[red'(planet')(z) \wedge R(z)]))$	T1(b)	IV
9	$\lambda x.((\lambda R.\exists z.[red'(planet')(z) \wedge R(z)])$ $(\lambda y.[x = y]))$	fkt.Appl.	IV
10	$\lambda x.\exists z[red'(planet')(z)$ $\wedge((\lambda y.[x = y])(z))]$	fkt.Appl.	
11	$\lambda x.\exists z[red'(planet')(z) \wedge [x = z]]$	fkt.Appl.	
12 mars is a red planet	$(\lambda P.P(m))(\lambda x.\exists z[red'(planet')(z)$ $\wedge[x = z]])$	T4	(t/IV)(IV)
13	$(\lambda x.\exists z[red'(planet')(z) \wedge [x = z]])(m)$	fkt.Appl.	t
14	$\exists z[red'(planet')(z) \wedge [m = z]]$	fkt.Appl.	
15	red'(planet')(m)		
16	$(\lambda x.red'(planet'))(m)$	Abstr.	

Ein Vergleich der letzten vier Beispiele führt uns nun zum nächsten Kapitel. Es wird sich zeigen, in welcher Weise es zu Problemen bei der Auswertung der transformierten Ausdrücke durch die unterschiedlichen λ-Kalküle kommen kann.

Ia) phobos spins and exists

Ausdruck	Transformation	Regel	Kategorie
1 phobos	$\lambda P.P(p)$	T1(d)	t/IV
2 spins	spin'	n.-l.Konst.	IV
3 exists	exist'	n.-l.Konst.	IV
4 spins and exists	$\lambda x.[spin'(x) \wedge exist'(x)]$	T12	IV
5 phobos spins and exists	$(\lambda P.P(p))$ $(\lambda x.[spin'(x) \wedge exist'(x)])$	T4	t
6	$(\lambda x.[spin'(x) \wedge exist'(x)])(p)$	fkt.Appl.	

Ib) phobos spins and phobos exists

Ausdruck	Transformation	Regel	Kategorie
1 phobos	$\lambda P.P(p)$	T1(d)	t/IV
2 spins	spin'	n.-l.Konst.	IV
3 exists	exist'	n.-l.Konst.	IV
4 phobos spins	spin'(p)	siehe Bsp.1	t
5 phobos exists	exist'(p)	siehe Bsp.1	t
6 phobos spins and exists	[spin'(p)$\wedge$exist'(p)]	T11	t
7	$(\lambda.[spin'(x) \wedge exist'(x)])(p)$	Abstraktion	

Die Transformation zeigt, dass die beiden Sätze *phobos spins and exists* und *phobos spins and phobos exists* äquivalent sind. Sie führen beide auf den gleichen Ausdruck. Ersetzen wir das Definitum *Phobos* durch das Indefinitum *a planet* und transformieren erneut.

IIa) a planet spins and exists

Ausdruck	Transformation	Regel	Kategorie
1 a	$\lambda P.\exists x.[\zeta'(x) \wedge P(x)]$	T1(d)	
2 planet	planet'	n.-l.Konst.	CN
3 a planet	$\lambda P.\exists x.[planet'(x) \wedge P(x)]$	T1(d)	T
4 spins and exists	$\lambda y.[spin'(y) \wedge exist'(y)]$	siehe Ia)4	IV
5 a planet spins and exists	$(\lambda P.\exists x.[planet'(x) \wedge P(x)])$ $(\lambda y.[spin'(y) \wedge exist'(y)])$	T4	(t/IV)(IV)
6	$\exists x.[planet'(x) \wedge ((\lambda y.[spin'(y)$ $\wedge exist'(y)])(x))])$	fkt.Appl.	t
7	$\exists x.[planet'(x)$ $\wedge[spin'(x) \wedge exist'(x)]]$	fkt.Appl.	

IIb) a planet spins and a planet exists

Ausdruck	Transformation	Regel	Kategorie
1 a	$\lambda P.\exists x.[\zeta'(x) \wedge P(x)]$	T1(d)	
2 planet	planet'	n.-l.Konst.	CN
3 a planet	$\lambda P.\exists x.[planet'(x) \wedge P(x)]$	T1(d)	T
4 spins	spin'	n.-l.Konst.	IV
5 a planet spins	$\lambda P.\exists x.[planet'(x) \wedge P(x)](spin')$	T4	(t/IV)(IV)
6	$\exists x.[planet'(x) \wedge spin'(x)]$	fkt.Appl.	t
7 exists	exist'	n.-l.Konst.	IV
8 a planet exists	$\exists x.[planet'(x) \wedge exist'(x)]$	wie in 3-6	t
9 a planet spins and a planet exists	$[\exists x.[planet'(x) \wedge spin'(x)]$ $\wedge\exists x.[planet'(x) \wedge exist'(x)]]$	T11	t
10	$[\exists x.[planet'(x) \wedge spin'(x)]$ $\wedge\exists y.[planet'(y) \wedge exist'(y)]]$	Umbenen -nung	

Die Sätze in IIa) und IIb) sind nicht mehr äquivalent und dies ist richtig. Mit diesem Beispiel wird im Folgenden gezeigt, dass manche Transformationen unter bestimmten Auswertungsstrategien einen Eigennamen, der durch eine nicht-deterministische Variable ersetzt wurde, dazu bringen, sich wie ein Indefinitum zu verhalten. Was nicht korrekt ist, wie an der Unterscheidung von Ia)b) und IIa)b) sichtbar wird.

3 λ-Kalküle und Transformationen

Nachdem sprachliche Ausdrücke sicher in äquivalente λ-Ausdrücke übertragen werden können, sollen λ-Kalküle mit unterschiedlichen Transformationen und Strategien untersucht werden. Geklärt werden soll, ob die für sprachliche λ-Ausdrücke anwendbaren Transformationen korrekt sind, in dem Sinne, dass sie die Extension der Ausdrücke, insbesondere auch der Teilausdrücke erhalten bleibt. Ein besonderer Aspekt der hierbei näher untersucht werden soll, ist das Verhalten der Transformationen beim Auftreten von nicht-Determinismus.

3.1 Choice

Es wird untersucht, in wieweit die Transformationen der unterschiedlichen Kalküle korrekt sind, wenn das Argument eine nicht-deterministische Auswahlfunktion enthält. Diese Auswahlfunktion hat in den untersuchten Kalkülen die Form eines zweistelligen Auswahloperators der Form $\oplus$. Dieser Auswahloperator wird als erratic-Choice bezeichnet. Sein Verhalten gleicht dem Werfen einer fairen Münze:

$M \oplus N \rightarrow M$
$M \oplus N \rightarrow N$

Gegeben also zwei Argumente, wählt dieser Operator zufällig eines dieser Argumente aus, ohne Rücksicht darauf was diese Argumente sind.

3.1.1 Ausdrücke mit Choice-Operator

Es ergibt sich bei Auswertung eines Ausdrucks, welcher einen $\oplus$-Operator enthält eine Menge möglicher Ergebnisse. So ergibt zum Beispiel der Term $(\lambda x.x)(1 \oplus 2)$ die Ergebnismenge E $=\{1,2\}$ oder $(\lambda x.x + 2)(2 \oplus 4)$ ergibt E$=\{4,6\}$.

Ist M ein Term, so ist E die Menge aller möglichen Ergebnisse nach Auswertung dieses Terms. E_k ist die Menge der korrekten Ergebnisse und E_f die Menge der falschen Ergebnisse, dann sind E_k und E_f disjunkt und E enthält keine anderen Ergebnisse als diese beiden.

Zu dieser Definition sind noch zwei Fragen zu klären. Zum einen die Frage, was unter der Auswertung eines Terms verstanden werden soll, zum anderen muss festgelegt werden, was korrekte bzw. falsche Ergebnisse sind. Die erste Frage ist recht schnell beantwortet. Ein Term wird durch Anwendung von Transformationsregeln ausgewertet. Die Auswertung ist beendet, wenn es keine Transformationsregel mehr gibt, die angewendet werden kann. Wie weit und ob ein Term ausgewertet werden kann und auf welche Weise dies geschieht, hängt dabei von den Transformationsregeln des Kalküls selbst ab.
Um die zweite Frage zu beantworten bedarf es einiger Überlegungen. Ausgangspunkt dieser Überlegungen soll die Feststellung sein, dass die beiden Ausdrücke x+x und 2×x äquivalent sind, und zwar deshalb, weil in der Mathematik die Multiplikation mit n definiert ist als das Addieren n vieler gleicher Summanden. In diesem Fall ist n eben 2. Schreiben wir die beiden Ausdrücke als λ-Ausdruck, ändert dies auch nichts an dieser Äquivalenz. Setzen wir einmal verschiedene Argumente für x ein:

$(\lambda x.x + x)(2) \rightarrow_\beta 2 + 2 = 4$
$(\lambda x.2 \times x)(2) \rightarrow_\beta 2 \times 2 = 4$

$(\lambda x.x + x)(3) \rightarrow_\beta 3 + 3 = 6$
$(\lambda x.2 \times x)(3) \rightarrow_\beta 2 \times 3 = 6$

Die äquivalenten Ausdrücke liefern für das gleiche Argument jeweils das gleiche Ergebnis zurück. Wir erhalten $E = E_k$ für den deterministischen Fall. Es sollen nun aber auch Argumente eingesetzt werden, welche den $\oplus$-Operator enthalten. Nehmen wir als Argument $2 \oplus 3$ und werten die Terme $(\lambda x.x + x)(2 \oplus 3)$ und $(\lambda x.2 \times x)(2 \oplus 3)$ aus. Allerdings gibt es nun zwei Stellen, an denen dieser Term ausgewertet werden könnte und nicht nur eine wie bisher. Einmal kann erst das Argument ausgewertet werden und dann die β-Reduktion durchgeführt werden, oder es wird erst die β-Reduktion ausgeführt und dann das Argument ausgewertet. Die erste Methode ist die strikte Auswertung. Die zweite Methode ist die nicht-strikte Auswertung. Beide Auswertungsmethoden sollen untersucht werden. Wünschenswert ist, dass $E = E_k$ gilt und die Ergebnisse äquivalent zur deterministischen Auswertung sind. Probieren wir zunächst die strikte Auswertung:

$(\lambda x.x + x)(2 \oplus 3) \rightarrow_\oplus (\lambda x.x + x)(2) \rightarrow_\beta 2 + 2 \rightarrow \{4\}$
$(\lambda x.2 \times x)(2 \oplus 3) \rightarrow_\oplus (\lambda x.2 \times x)(2) \rightarrow_\beta 2 \times 2 \rightarrow \{4\}$

$(\lambda x.x + x)(2 \oplus 3) \rightarrow_\oplus (\lambda x.x + x)(3) \rightarrow_\beta 3 + 3 \rightarrow \{6\}$
$(\lambda x.2 \times x)(2 \oplus 3) \rightarrow_\oplus (\lambda x.2 \times x)(3) \rightarrow_\beta 2 \times 3 \rightarrow \{6\}$

Wir erhalten $E = \{4, 6\}$. Bei der nicht-strikten Auswertung erhalten wir folgendes:

$(\lambda x.x + x)(2 \oplus 3) \rightarrow_\beta (2 \oplus 3) + (2 \oplus 3) \rightarrow \{4, 5, 6\}$
$(\lambda x.2 \times x)(2 \oplus 3) \rightarrow_\beta 2 \times (2 \oplus 3) \rightarrow \{4, 6\}$

Das Entscheidende ist hier nicht, dass im ersten Fall eine 5 als Ergebnis auftaucht,

sondern der Umstand, dass die beiden Ausdrücke nicht mehr äquivalent sind in dem Sinne, dass sie für jede Einsetzung gleicher Argumente auch gleiche Ergebnisse liefern. Wird durch den $\oplus$-Operator beides mal 2 ausgewählt, oder beides mal 3, dann erhalten wir als Ergebnis, gerade 4 oder 6. In den Fällen wo die beiden Operatoren etwas Unterschiedliches wählen erhalten wir die 5.

Definition 9 (korrektes Ergebnis)
Korrekte Ergebnisse sind die unter applikativer Auswertung erreichbaren Ergebnisse.

Dies ist erst einmal eine allgmeine Definition. Korrekte Ergebnisse erhalten wir dann, wenn wir zunächst das Argument vollständig auswerten. So ist sichergestellt, dass für alle Vorkommen einer gebundenen Variable derselbe Wert eingesetzt wird. Können wir diese auch verwenden, wenn wir sprachliche Ausdrücke verwenden? Die Extension eines Satzes ist sein Wahrheitswert. Daher sind zwei Sätze dann äquivalent, wenn ihr Wahrheitswert übereinstimmt. Allerdings reicht dies allein hier nicht aus. Die Ergebnismengen ausgewerteter sprachlicher Ausdrücke sind Mengen von Wahrheitswerten.

Fall 1:
Wir erhalten für *titan spins and titan exists* und *tethys spins and tetyhs exists* denselben Wahrheitswert. Die beiden sind äquivalent, denn sie sind beide von der Form *x spins and x exists*. Dies läßt sich leicht umformen zu *x spins and exists*. x ist hier ein Eigenname und muss in beiden Fällen mit dem gleichen Wert belegt werden (siehe hierzu auch noch einmal Beispiel I a) und b) am Ende des Kapitels 2).

Fall 2:
Ebenso wären dann aber auch *titan spins and tethys exists* und *tethys spins and titan exists* äquivalent, denn auch sie haben den gleichen Wahrheitswert wie die Ausdrücke in Fall 1. Das kann man aber so nicht sagen, denn diese haben die Form *x spins and x exists* aber sie lassen sich nicht zusammenfassen. Dies ist aber nicht korrekt, denn x ist hier ebenfalls ein Eigenname, verhält sich aber wie ein Indefinitum, indem x verschiedene Werte annimmt. Die Repräsentation ist nun nicht mehr $\lambda x.[spin'(x) \wedge exist'(x)](p)$ wie es im ersten Fall war, sondern $\exists x.[[planet'(x) \wedge spin'8x)] \wedge \exists y.[planet'(y) \wedge exist'(x)]]$. Die beiden x sind hier voneinander unabhängig geworden (vergleiche hierzu auch noch einmal Beispiel II a) und b) am Ende des Kapitels 2).

Um dies zu vermeiden reicht es also hier nicht aus nur zu fordern, der Wahrheitswert der Ausdrücke müsse übereinstimmen, sondern es muss auch gegeben sein, dass innerhalb eines Ausdrucks dieselben Variablen mit dem selben Wert belegt werden müssen, um Äquivalenz mit den Ergebnissen zu erreichen, welche wir im deterministischen Fall erhalten.

Im folgenden soll eine nicht-deterministische Variable eingeführt werden, durch welche das Fragment aus [FL89] erweitert werden soll. Diese Variable soll Platzhalter für einen Eigennamen sein.

Definition 10 (ω^*) .....
ω^* ist ein Platzhalter für ein Wort mit dem Anfangsbuchstaben ω. Gibt es n Worte $\omega_1 ... \omega_n$ mit dem Anfangsbuchstaben ω so ist
$\omega^* = (\omega_1 \oplus (\omega_2 \oplus (...(\omega_{n-1} \oplus \omega_n)))) = (\omega_1 \oplus \omega_2 ... \oplus \omega_n)$.

ω^* simuliert den Fall, dass eine Eingabe im Interpreter nicht vollständig oder fehlerhaft ist. Eventuell ist auch der Name nicht bekannt, sondern nur sein Anfang. So läßt sich dennoch eine Abfrage durchführen, die auswertbar ist. Im Interpreter ist ω^* wie folgt implemeniert:
Findet der Interpreter ein ω^* so sucht er in der Datenliste entsprechend der syntaktischen Kategorie, in welcher sich ω^* befindet alle Wörter, welche mit ω beginnen und sammelt sie in einer Liste. Dann wird eine Zufallszahl im Intervall 0 bis Länge_der_Liste - 1 mit Hilfe eines unsafePerformIO erzeugt. Diese Zufallszahl bestimmt den Listenindex des Wortes, welches dadurch ausgewählt wird (▶ 615 – 620). Dies geschieht durch eine Erweiterung der Interpretationsfunktion m durch m' und m" (▶ 382 – 401). Es können im Interpreter also nicht nur Eigennamen durch ein ω^* ersetzt werden, sondern jedes beliebige Wort. Im weiteren wird jedoch speziell das Problem der Eigennamen untersucht, die sich unter Umständen wie Indefinita verhalten. Vor dieser Untersuchung soll aber zunächst kurz dargestellt werden wie Indefinita und Choice-Funktionen in der Linguistik zusammenhängen.

3.1.2 Choice-Funktionen und Indefinita

Indefinita, beziehungsweise indefinite Nominalphrasen, haben ein nicht-triviales Skopusverhalten welches in syntaktischen und semantischen Theorien zu Komplikationen führt. Ein Ansatz für diese Problematik ist die Verwendung von Choice-Funktionen. Diese sind nicht zu verwechseln mit dem zuvor vorgestellten Choice-Operator $\oplus$, welcher eines seiner beiden Argumente beliebig wählt. Die Choice-Funktion wählt aus einer Menge ein Element nach einer vorgegebenen Bedingung aus:

Definition 11 (Choice-Bedingung) *Eine Funktion f ist eine Choice Funktion, nur dann wenn für jedes nicht-leere Prädikat P gilt:*
f(P) ist definiert und ist in der Extension von P, d.h. es gilt P(f(P)).

Gilt die Choice-Bedingung nicht, so hat dies einen unangenehmen Nebeneffekt, der durch das Verhalten der Choice-Funktion zustande kommt, falls man die oben erwähnte Bedingung beiseite läßt. Ohne diese Bedingung verhält die Choice-Funktion sich so, dass

sie aus einer Menge ein Element auswählt, falls diese Menge jedoch leer ist, irgendein Element auswählt. Ein Beispiel:

Some woman smiles

Mit [Win97] gilt hier die Äquivalenz:

$\exists f[CH(f) \wedge smile'(f(woman'))]$
$\Leftrightarrow \exists x[woman'(x) \wedge smile'(x)] \vee [\neg\exists woman'(x) \wedge \exists x smile'(x)]$

Das Problem ist nun, dass durch diese Äquivalenz der Satz wahr werden kann, selbst wenn es keine Frau im Diskursuniversum gibt. Die Auswahlfunktion wählt dann einfach irgendein Individuum aus, welches keine Frau ist. Wenn dieses Individuum lächelt, dann wird der Satz wahr, obwohl es gar keine Frau gibt, die lächeln könnte.
Diese Äquivalenz kommt dadurch zustande, dass sich die Choice-Funktion auf Hilbert's ϵ-Operator zurückführen lässt. Von Heusinger deutet diesen als Auswahlfunktion Φ „die jeder nicht-leeren Menge eine Element dieser Menge zuordnet, während sie der leeren Menge ein beliebiges Element des Individuenbereichs zuweist. Damit ist die Auswahlfunktion immer definiert."[vHe97](S.60)
Der ϵ-Operator wird verwendet, um in Beweisen Ausdrücke mit Quantoren durch einen ϵTerm zu ersetzen und damit die Beweisführung zu vereinfachen. Ein ϵ-Term ist ein Ausdruck ohne freie Variable. Kommen doch freie Variablen vor, so spricht man von einem ϵ-Ausdruck.
Modelltheoretisch beschreibt [vHe97] den ϵ-Operator mit einem Modell M und einer Variablenbelegung g durch:
$|[\epsilon x\alpha]|^{M,g} = \Phi(\{d : |[\alpha]|^{M.g^{d/x}} = 1)$.
Die doppelten Klammern sind Extensionsklammern. $\epsilon x\alpha$ ist ein ϵ-Term derart, dass alle x in α durch ϵ gebunden sind. Die Extension eines Ausdrucks, hängt vom Diskursuniversum ab, welches durch das Modell M und die Variablenbelegung g beschrieben wird. Φ als Auswahlfunktion wählt nun aus g genau die Belegung d von x aus, für die α wahr wird. Also $||[\alpha]||$ als Extension von α im Modell M wobei in α alle Vorkommen von x durch d ersetzt werden ($g^{d\backslash x}$). d ist diejenige Belegung die α wahr macht.

Einen Satz wie *a planet spins and exists* hat nach diesen Überlegungen die Form

$\exists f[CH(f)\wedge$spin'(f(planet'))], was äquivalent ist zu
$\exists$ x [planet'(x) $\wedge$ spin'(x)] $\vee$ [$\neg\exists$ x planet'(x) $\wedge\exists$ x spin'(x)].

Nun fehlt noch ein kleiner Schritt. Nehmen wir eine Auswertung im Interpreter an, dann ist die Tatsache, dass die Menge der entsprechenden Planeten leer ist kein Problem. In diesem Fall kann der Interpreter sagen, dass er entweder die Frage nicht versteht, oder dass der Satz falsch ist. Im ersten Fall findet er keinen passenden Eintrag, im zweiten Fall findet er einen Eintrag, welcher diesen Satz jedoch falsch macht. Die Choice-Funktion hingegen muss in dem Fall, dass es kein Individuum gibt welches planet'(x) wahr macht, irgendetwas anderes wählen, damit zumindest der ganze Ausdruck wahr wird.
Der entscheidende Vorteil der Choice-Funktion ist jedoch, dass sie immun gegen Kopien ist. D.h. wann immer f etwas für x auswählt ist es stets dasselbe. Unabhängig von den Reduktionen erhalten wir durch f stets E=E_k. Die Problematik mit der leeren Menge

kann hier aus genannten Gründen ignoriert werden.
Eigennamen bezeichnen von sich aus schon ein fest vorgegebenes Individuum. Das ist der Grund, weshalb die beiden Sätze: *phobos spins and exists* und *phobos spins and phobos exists* äquivalent sind, beide führen auf die Repräsentation $(\lambda x.[spin'(x) \wedge exist'(x)])$ wie bereits gezeigt wurde. Bei einer indefiniten Nominalphrase, wie zum Beispiel *a planet* sind die beiden Sätze *a planet spins and exists* und *a planet spins and a planet exists* nicht äquivalent. Im zweiten Fall kann für das erste *a planet* ein anderes Individuum ausgewählt werden wie für das zweite. Dies sieht man bereits an der Repräsentation $[\exists x.[planet'(x) \wedge spin'(x)] \wedge \exists x.[planet'(x) \wedge exist'(x)]]$. Wenn im weiteren davon die Rede ist, dass sich ein Eigenname wie ein Indefinitum verhält, so ist damit gemeint, dass sich beim Einsetzen eines Argumentes, welches ein Choice enthält, das unter einer Anzahl von Eigennamen wählen soll, eine Repräsentation wie bei einem Indefinitum ergibt. Dies geschieht dadurch, dass an den unterschiedlichen Stellen in denen die gebundene Variable x auftaucht, verschiedene Werte für x eingesetzt werden.

3.2 Untersuchung verschiedener Kalküle

Hier werden nun zwei Kalküle genauer untersucht welche über einen Choice-Operator verfügen. Das erste Kalkül aus [LP95] verwendet den klassichen λ-Kalkül mit den darin enthaltenen Transformationen zusammen mit dem Choice-Operator. Dann wird für eine erste Anäherung an die Lösung der Problematik der Vorschlag einer Substitution aus [Man95] verfolgt und auf Brauchbarkeit untersucht. Das zweite Kalkül verwendet eine vom klassischen Kalkül abweichende β-Reduktion. In diesem Fall läßt sich dann zeigen, dass das Problem, welches im ersten Kalkül auftrat, durch die veränderten Transformationen auch ohne Substitution lösbar ist und der Idee der eben beschriebenen Choice-Funktion in der Semantik der Funktionalität nach, sehr nahe kommt, wenn auch von einer anderen Seite.

3.2.1 Kalkül mit klassischen Transformationsregeln

Ich verwende für diesen Abschnitt einen klassichen Kalkül mit Choice-Erweiterung aus [LP95]. Die Menge aller Terme $\Lambda_{\oplus}$ des nicht-deterministischen Kalküls ist folgendermaßen definiert:

Definition 12
Sei Var eine nichtleere Menge von Variablen, dann sind folgende Ausdrücke Terme:

- $\forall x \in Var : x \in \Lambda_{\oplus}$;
- $M, N \in \Lambda_{\oplus} \Rightarrow (MN) \in \Lambda_{\oplus}$;
- $M \in \Lambda_{\oplus}, x \in Var \Rightarrow (\lambda x.M) \in \Lambda_{\oplus}$
- $M, N \in \Lambda_{\oplus} \Rightarrow M \oplus N \in \Lambda_{\oplus}$

Dies reicht jedoch zur Auswertung von sprachlichen Ausdrücken nicht aus. Wir nehmen an, dass sprachliche Ausdrücke Konstanten und prädikatenlogische Operatoren erlaubt sind.
Die benötigten Transformationsregeln sind:
(β) $(\lambda x.M)N \rightarrow M[N/x]$;
$(\oplus 1)$ $M \oplus N \rightarrow M, M \oplus N \rightarrow N$;

Zunächst soll das Beipiel vom Ende des Kapitels 2 untersucht werden, welches schon berechnet wurde:

phobos spins and exists $\longrightarrow (\lambda x.[spin'(x) \wedge exist'(x)])(phobos)$.

1. Auswertung ohne nicht-Determinismus:
Da phobos nicht weiter ausgewertet werden muss, kann hier nur auf eine Art und Weise ausgewertet werden, nämlich durch Anwendung der β-Regel. Eine Aufteilung in strikte und nicht strikte Auswertung ist hier nicht nötig.

Beispiel 12
$(\lambda x.[spin'(x) \wedge exist'(x)])(phobos)$
$\rightarrow_{\beta} [spin'(phobos) \wedge exist'(phobos)]$

Diese Auswertung ist korrekt. Nun wird phobos ersetzt durch ω^* mit $\omega = n$. Wir erhalten hierdurch $n^* = (naiad \oplus neptune \oplus nereid)$. Dies wird nun dem Ausdruck als Argument gegeben.

2. Auswertung mit nicht-Determinismus:
Da das Argument noch nicht vollständig ausgewertet ist, kann sowohl eine strikte Auswertung angewendet werden, oder aber eine nicht-strikte Auswertung.

Beispiel 13 (n^* spins and exist) .....

$(\lambda x.[spin'(x) \wedge exit'(x)])(n^*)$

a)strikte Auswertung

(1) $\longrightarrow_{\oplus 1} (\lambda x.[spin'(x) \wedge exit'(x)])(naiad)$
$\longrightarrow_{\beta} [spin'(naiad) \wedge exit'(naiad)]$
(2) $\longrightarrow_{\oplus 1} (\lambda x.[spin'(x) \wedge exit'(x)])(neptune)$
$\longrightarrow_{\beta} [spin'(neptune) \wedge exit'(neptune)]$
(3) $\longrightarrow_{\oplus 1} (\lambda x.[spin'(x) \wedge exit'(x)])(nereid)$
$\longrightarrow_{\beta} [spin'(nereid) \wedge exit'(nereid)]$

b)nicht-strikte Auswertung

$\longrightarrow_{\beta} [spin'(n^*) \wedge exit'(n^*)]$
(1) $\longrightarrow_{\oplus 1} [spin'(naiad) \wedge exit'(naiad)]$
(2)* $\longrightarrow_{\oplus 1} [spin'(naiad) \wedge exit'(neptune)]$
(3)* $\longrightarrow_{\oplus 1} [spin'(naiad) \wedge exit'(nereid)]$
(4) $\longrightarrow_{\oplus 1} [spin'(neptune) \wedge exit'(neptune)]$
(5)* $\longrightarrow_{\oplus 1} [spin'(neptune) \wedge exit'(naiad)]$
(6)* $\longrightarrow_{\oplus 1} [spin'(neptune) \wedge exit'(nereid)]$
(7) $\longrightarrow_{\oplus 1} [spin'(nereid) \wedge exit'(nereid)]$
(8)* $\longrightarrow_{\oplus 1} [spin'(nereid) \wedge exit'(neptune)]$
(9)* $\longrightarrow_{\oplus 1} [spin'(nereid) \wedge exit'(naiad)]$

Die strikte Auswertung ist korrekt. Sie wertet den Term für jedes mögliche Individuum welches durch n^* gegeben ist so aus, als ob jedes einzelne Individuum deterministisch ausgewertet worden wäre[1] und die Auswertung des deterministischen Ausdrucks ist korrekt. Ein Problem entsteht jedoch bei der nicht-strikten Auswertung. Wir können beobachten, dass wir viel mehr Ergebnisse bekommen. Die mit $\star$ markierten Ergebnisse sind falsch. Wenn ich sage, für ein bestimmtes Individuum (Eigenname!) gilt, dass es sich dreht und dass es existiert, so meine ich damit auf keinen Fall, dass sich ein bestimmtes Individuum dreht und ein anderes exisitert. Dies ist jedoch der Fall, wenn ich sage *a planet spins and a planet exists*($\rightarrow$ Indefinitum).
Zunächst einmal ist hier festzuhalten, dass die call-by-name Auswertung unter nicht-deterministischen Bedingungen nicht korrekt ist.
Ist k die Anzahl der durch λ gebundenen Variablen in einem Term und n die Anzahl der Alternativen in ω^*, dann ergibt die Auswertung im nicht-strikten Fall n^k verschiedene

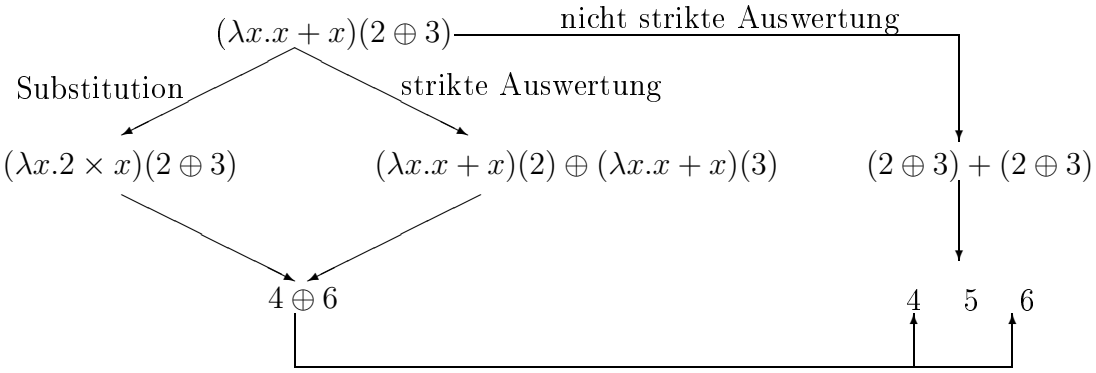
[1] Um das zu sehen braucht man im deterministischen Beispiel nur phobos durch jeweils einen der drei anderen Namen zu ersetzten.

Ergebnisse [2]. Genau n viele sind davon richtig, nämlich die, in welcher alle gebundenen Variablen dieselbe Alternative auswählen. Im strikten und im deterministischen Fall erhalten wir genau n viele Ergebnisse, die alle korrekt sind. Die korrekten Ergebnisse in allen Fällen sind identisch.

[2]Kombinatorisch gesehen ist dies ein geordnetes Ziehen mit Zurücklegen.

3.2.2 Substitution

Einen ersten Ansatz für dieses Problem finden wir in [Man95] in Kapitel 6.4.3. Mandel schlägt die Lösung des Problems durch eine Substitution vor. Er zeigt dies an dem Beispiel $(\lambda x.x + x)(2 \oplus 3)$ wie folgt:



In Mandels Beispiel ist die nicht-strikte Auswertung ohne Substitution nicht angegeben, wir sehen aber, dass diese auch zum oben bereits erkannten Problem führt. Die Frage ist also, ob wir auch für unsere sprachlichen Ausdrücke geeignete Substitutionen finden können. Was Mandel hier vorschlägt ist eine Spracherweiterung, die an das Lambek-Kalkül erinnert [vBe975]S.40ff, in welchem nur ein einziges Vorkommen einer Variable in einem Term erlaubt ist.

Definition 13 (geeignete Substitution) *Eine Substitution ist eine geeignete Substitution, wenn sie einen sprachlichen Ausdruck mit mehreren Vorkommen einer gebundenen Variable im Rumpf ersetzten kann, durch einen Ausdruck mit nur einem Vorkommen einer gebundenen Variable im Rumpf.*

Nehmen wir den Ausdruck n^* *spins and exists* also wie bisher
$(\lambda x.[spin'(x) \wedge exit'(x)])(n^*)$ und versuchen wir eine geeignete Substitution zu finden. Hierzu testen wir diesen Ausdruck im Interpreter aus [FL89]. Kandidat2 (Haskell) verdoppelt manchmal Ausdrücke. Dennoch trifft er nur an einer einzigen Stelle eine Auswahl. Die formale Repräsentation dieses Satzes nach Montague hat jedoch zwei gebundene Variablen. Was also passiert im Interpreter mit dem Ausdruck n^* *spins and exists*?

n^* wird ausgewertet zu einer dieser drei Möglichkeiten:
naiad: test_property_wrt 60 (▶ 102)
neptune: test_property_wrt 16 (▶ 58)
nereid: test_property_wrt 59 (▶ 101)

spins: (w,t)=("spins ",intransverb_spin)(▶ 174)
intransverb_spin = [8..65](▶ 266)
⟶ spin = [8..65]

and: (w,t)=("and ",verbphrase_and)(▶ 200)
verbphrase_and = intersect (▶ 281)

exists: (w,t)=("exists ", intransverb_exists) (▶ 170)
intransverb_exists = entityset (▶ 263)
entityset = [1..100] (▶ 211) ⟶ exists = [1..100]

Nun werden die beiden Mengen aus spins und exists durch das *and* geschnitten:
[8..65]∩[1..100]=[8..65]

Diese Menge dient nun der Funktion test_property_wrt x als Argument und wir erhalten folgende möglichen Ergebnisse:
test_property_wrt 60 [8..65] → true
test_property_wrt 16 [8..65] → true
test_property_wrt 59 [8..65] → true

Innerhalb des Interpreters ist die Repräsentation des Ausdrucks n^* *spins and exist* nicht von der Form $(\lambda x.[spin'(x) \wedge exit'(x)])(n^*)$. Sondern ist S die Menge aller Individuen die sich drehen und E die Menge aller Individuen welche existieren, dann ist die Repräsentation von der Form $(\lambda x.x \in S \cap E)(n^*)$.
Gehen wir noch einmal zurück zu dem Fall, in dem wir ein einfaches Individuum einsetzen, also *phobos spins and exists*. Die Repräsentation hier wäre also $(\lambda x.x \in S \cap E)(phobos)$. Am Ende von Kapitel 2 haben wir gesehen, dass *phobos spins and exists* äquivalent ist mit *phobos spins and phobos exists*. Gilt dies auch für die Repräsentation durch Mengen? Vom Interpreter würde *phobos spins and phobos exists* so ausgewertet werden: phobos: test_propertywrt 19

test_property_wrt 19 [6...65]⟶ true
test_property_wrt 19 [1...100]⟶ true
sentence_and = true && true

Die Repräsentation von *phobos spins and phobos exists* im Interpreter ist also $(\lambda x.x \in S \wedge x \in E)(phobos)$.
Für den Schnitt zweier Mengen gilt die Definition:

$A \cap B := \{x | x \in A \wedge x \in B\}$.

Diese Repräsentation ist also äquivalent zu der Repräsentation von *phobos spins and exists*.

Wollen wir nun den Satz *phobos spins and exists* auswerten substituieren wir $[spin'(x) \wedge exist'(x)]$ mit $x \in S \cap E$:

Beispiel 14 (n^* spins and exists) .....
$(\lambda x.[spin'(x) \wedge exit'(x)])(n^*)$
Sei S die Menge aller Individuen x für die gilt, dass spin'(x) und E die Menge aller Individuen x für die gilt, dass exist'(x). Dann läßt sich $[spin'(x) \wedge exit'(x)]$ *substituieren durch* $x \in S \cap E$. *Dabei sei sub die Substitution und* sub^{-1} *die Aufhebung der Substitution.*

a)strikte Auswertung

$(\lambda x.[spin'(x) \wedge exit'(x)])(n^*)$
$\longrightarrow_{sub} (\lambda x.x \in S \cap E)(n^*)$

$\longrightarrow_{\oplus 1} (\lambda x.x \in S \cap E)(nereid)$
$\longrightarrow_{\beta} nereid \in S \cap E$
$\longrightarrow_{sub^{-1}} [spin'(nereid) \wedge exist'(nereid)]$

$\longrightarrow_{\oplus 1} (\lambda x.x \in S \cap E)(neptune)$
$\longrightarrow_{\beta} neptune \in S \cap E$
$\longrightarrow_{sub^{-1}} [spin'(neptune) \wedge exist'(neptune)]$

$\longrightarrow_{\oplus 1} (\lambda x.x \in S \cap E)(naiad)$
$\longrightarrow_{\beta} naiad \in S \cap E$
$\longrightarrow_{sub^{-1}} [spin'(naiad) \wedge exist'(naiad)]$

b)nicht-strikte Auswwertung

$(\lambda x.[spin'(x) \wedge exit'(x)])(n^*)$
$\longrightarrow_{sub} (\lambda x.x \in S \cap E)(n^*)$
$\longrightarrow_{\beta} n^* \in S \cap E$

$\longrightarrow_{\oplus 1} naiad \in S \cap E$
$\longrightarrow_{sub^{-1}} [spin'(naiad) \wedge exit'(naiad)$

$\longrightarrow_{\oplus 1} neptune \in S \cap E$
$\longrightarrow_{sub^{-1}} [spin'(neptune) \wedge exit'(neptune)$

$\longrightarrow_{\oplus 1} nereid \in S \cap E$
$\longrightarrow_{sub^{-1}} [spin'(nereid) \wedge exit'(nereid)$

Zuvor wurde gesagt, dass $x \in S \cap E$ und $x \in S \wedge x \in E$ schon allein wegen der grundlegenden Definition äquivalent sind. Zweiteres ist jedoch per Definition keine geeignete

Substitution. Wir haben also eine geeignete Substitution gefunden und die nicht-strikte Auswertung arbeitet darauf korrekt. Aber läßt sich immer eine geeignete Substitution finden? Nein. Betrachten wir hierzu ein Gegenbeispiel:
rhea or rosalind orbit saturn and rhea orbits uranus.
Berechnen wir hierzu zunächst die λ-Repräsentation. s^+ und u^+ sind hier nur eine verkürzte Schreibweise. Sie werden nicht weiter ausgewertet und bleiben fest. Ebenso sind $orbit'(s^+)$ und $orbit'(u^+)$ feste Ausdrücke die nicht mehr getrennt werden. Zusammen mit einem Argument drücken sie die Relation aus, dass dieses Argument s^+ oder u^+ umkreist.

rhea or rosalind orbit saturn and rhea orbits uranus

Ausdruck	Transformation	Regel
1 rhea	$\lambda P.P(r_1)$	T1(d)
2 rosalind	$\lambda Q.Q(r_2)$	T1(d)
3 or_T	$\lambda R.[\alpha'(R) \vee \beta'(R)]$	T13
4 rhea or rosalind	$\lambda R.[(\lambda P.P(r_1))(R) \vee (\lambda Q.Q(r_2))(R)$	T13
	$\rightarrow \lambda R.[R(r_1) \vee R(r_2)]$	fkt.Appl.
5 orbit	orbit'	Basisa.
6 saturn	$\lambda S.S(s) \rightarrow s^+$	T1(d)
7 orbit saturn	$orbit'(s^+)$	T5
8 rhea or rosalind orbit saturn	$(\lambda R.[(R(r_1) \vee R(r_2)])(orbit'(s^+))$ $\rightarrow [orbit'(s^+)(r_1) \vee orbit'(s^+)(r_2)]$	T4 fkt.Appl.
9 uranus	$\lambda U.U(u) \rightarrow u^+$	T1(d)
10 orbit uranus	$orbit'(u^+)$	T5
11 rhea orbits uranus	$(\lambda P.P(r_1))(orbit'(u^+)$ $\rightarrow orbit'(u^+)(r_1)$	T4 fkt.Appl.
12 and_t	$[\Phi \wedge \Psi]$	T11
13 rhea or rosalind orbit saturn and rhea orbits uranus	$[(orbit'(s^+)(r_1) \vee (orbit'(s^+)(r_2))$ $\wedge orbit(u^+)(r_1)]$	T11
	$\lambda x.[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2))$ $\wedge orbit(u^+)(x)](r_1)$	Abstr.

Wir ersetzen in dem berechneten Ausdruck
$\lambda x.[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(x)](r_1)$
rhea durch ein nicht-deterministisches $r^* = (rhea \oplus rosalind)$ und versuchen dann zu

einer geeigneten Substitution zu gelangen:
$\lambda x.[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(x)](r^*)$
Sei O_s die Menge aller Individuen x für die gilt, dass sie Saturn umkreisen, also $x \in O_s$ und O_u die Menge aller Individuen x für die gilt, dass sie Uranus umkreisen, also $x \in O_u$. Dann ist:
$[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(x)]$
$\longrightarrow [(x \in O_s \vee r_2 \in O_s) \wedge x \in O_u]$
$\longrightarrow [(x \in O_s \wedge x \in O_u) \vee (r_2 \in O_s \wedge x \in O_u)]$
$\longrightarrow [(x \in O_s \cap O_u) \vee (r_2 \in O_s \wedge x \in O_u)]$
Dies läßt sich nicht weiter in dem Sinn vereinfachen, dass sich noch ein x tilgen läßt, um zu einer geeigneten Substitution zu gelangen. Es lässt sich auf diese Weise keine geeignete Substitution finden. Bei der Asuwertung bleibt das Problem erhalten. Bei der nicht-strikten Auswertung erhalten wir $2^2 = 4$ Ergebnisse von denen zwei korrekt sind und den Ergebnissen der strikten Auswertung entsprechen.

Beispiel 15 (rhea or rosalind orbit saturn and rhea orbits uranus) .....

$\lambda x.[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(x)](r^*)$

a) *strikte Auswertung*
$\longrightarrow_{\oplus 1} \lambda x.[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(x)](r_1)$
$\longrightarrow_{\beta} [(orbit'(s^+)(r_1) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r_1)]$

$\longrightarrow_{\oplus 1} \lambda x.[(orbit'(s^+)(x) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(x)](r_2)$
$\longrightarrow_{\beta} [(orbit'(s^+)(r_2) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r_2)]$

b) *nicht-strikte Auswertung*
$\longrightarrow_{\beta} [(orbit'(s^+)(r^*) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r^*)]$
(1) $\longrightarrow_{\oplus 1} [(orbit'(s^+)(r_2) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r_2)]$
(2) $\longrightarrow_{\oplus 1} [(orbit'(s^+)(r_1) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r_1)]$
(3)* $\longrightarrow_{\oplus 1} [(orbit'(s^+)(r_2) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r_1)]$
(4)* $\longrightarrow_{\oplus 1} [(orbit'(s^+)(r_1) \vee (orbit'(s^+)(r_2)) \wedge orbit(u^+)(r_2)]$

Die bisherige Lösung zur Tilgung nicht korrekter Ergebnisse ist nicht sehr zufriedenstellend. Die Methode der Substitution ist zwar in der Lage falsche Ergebnisse zu eliminieren, jedoch nur dann, wenn es eine geeignete Substitution gibt. Wir haben an einem Gegenbeispiel gesehen, dass es diese nicht immer gibt. Das eigentliche Problem wird mit der Substitution auch nicht gelöst sondern bestenfalls umgangen. Wir erhalten immer dann nicht korrekte Ergebnisse, wenn im Rumpf der Abstraktion die gebundene Variable mehr als einmal vorkommt und die β-Reduktion ein Argument, welches ein Choice enthält, in

diese Variable hineinkopiert. Eine geeignete Substitution umgeht dieses Problem, indem sie dafür sorgt, dass die gebundene Variable nur einmal auftritt. Im folgenden nähern wir uns diesem Problem von einer anderen Seite. Anstatt wie zuvor zu versuchen, Repräsentationen zu finden, welche nur eine gebundene Variable enthalten, betrachten wir ein Kalkül mit einer modifizierten Beta-Reduktion.

3.2.3 Kalkül mit modifizierter β-Reduktion

Im folgenden soll das Kalkül aus [SSS04] untersucht werden. Dieses Kalkül verwendet eine modifizierte β-Reduktion mit Variablen-Sharing. Die Menge aller Ausdrücke E in diesem Kalkül ist durch folgende Syntax gegeben:

E ::=V | (c $E_1 ... E_{ar(c)}$ | (seq $E_1 E_2$) | ($case_T E Alt_1 ... Alt_{|T|}$ | ($E_1 E_2$)
($E_1 \oplus E_2$) |($\lambda V.E$) | (letrec $V_1 = E_1, ..., V_n = E_n$ in E)
Alt::=(Pat $\rightarrow$ E)
Pat::= (c $V_1 ... V_n$)

Zur Auswertung sprachlicher Ausdrücke nehmen wir an, dass Eigennamen Konstanten sind und dass Ausdrücke wie Verben u.ä., prädikatenlogische Operatoren und Mengenoperatoren Konstruktoren der Form (c $V_1 ... V_{ar(c)}$) sind.
Die für die Transformation sprachlicher Ausdrücke hier verwendeten Transformationen sind dann:

(choice-l)	$(s \oplus t) \rightarrow s$
(choice-r)	$(s \oplus t) \rightarrow t$
(lbeta)	$((\lambda x.s)r) \rightarrow$ (letrec x = r in s)
(cpcx-in)	(letrec x=y, Env in C[x]) $\rightarrow$ (letrec x=y, Env in C[y])
	mit y ist Variable und x $\neq$ y, Env ist Abkürzung für $x_1 = E_1, ..., x_n = E_n$
(gc2)	letrec $x_1 = s_1, ..., x_n = s_n$ in t) $\rightarrow$ t
	für alle i: x_i kommt nicht in t vor

Sehen wir uns die Auswertung des ersten Beispiels an, also n^* *spins and exists*. Wir sehen, dass sich hier sowohl bei der strikten als auch bei der nicht-strikten Auswertung $E = E_k$ ergibt, und das auch ohne Substitution:

Beispiel 16 .....
$\lambda x.[spin'(x) \wedge exits'(x)](n^*)$

a)strikte Auswertung

$\rightarrow_{choice-l}$ $(\lambda x.[spin'(x) \wedge exist'(x)](nereid)$
$\rightarrow_{l\beta}$ $letrec\ x = nereid\ in\ [spin'(x) \wedge exist'(x)]$
$\rightarrow_{cpcx-in^2}$ $letrec\ x = nereid\ in\ [spin'(nereid) \wedge exist'(nereid)]$
$\rightarrow_{gc2}$ $[spin'(nereid) \wedge exist'(nereid)]$

$\rightarrow_{choice-r(l)}$ $(\lambda x.[spin'(x) \wedge exist'(x)](neptune)$
$\rightarrow_{l\beta}$ $letrec\ x = neptune\ in\ [spin'(x) \wedge exist'(x)]$
$\rightarrow_{cpcx-in^2}$ $letrec\ x = neptune\ in\ [spin'(neptune) \wedge exist'(neptune)]$
$\rightarrow_{gc2}$ $[spin'(neptune) \wedge exist'(neptune)]$

$\rightarrow_{choice-r(r)}$ $(\lambda x.[spin'(x) \wedge exist'(x)](naiad)$
$\rightarrow_{l\beta}$ $letrec\ x = naiad\ in\ [spin'(x) \wedge exist'(x)]$
$\rightarrow_{cpcx-in^2}$ $letrec\ x = naiad\ in\ [spin'(naiad) \wedge exist'(naiad)]$
$\rightarrow_{gc2}$ $[spin'(naiad) \wedge exist'(naiad)]$

b)nicht-strikte Auswertung

$\rightarrow_{l\beta}$ $letrec\ x = n^*\ in\ [exist'(x) \wedge spin'(x)]$

$\rightarrow_{choice-l}$ $letrec\ x = nereid\ in\ [exist'(x) \wedge spin'(x)]$
$\rightarrow_{cpcx-in^2}$ $letrec\ x = nereid\ in\ [spin'(nereid) \wedge exist'(nereid)]$
$\rightarrow_{gc2}$ $[spin'(nereid) \wedge exist'(nereid)]$

$\rightarrow_{choice-r(l)}$ $letrec\ x = neptune\ in\ [exist'(x) \wedge spin'(x)]$
$\rightarrow_{cpcx-in^2}$ $letrec\ x = neptune\ in\ [spin'(neptune) \wedge exist'(neptune)]$
$\rightarrow_{gc2}$ $[spin'(neptune) \wedge exist'(neptune)]$

$\rightarrow_{choice-r(r)}$ $letrec\ x = naiad\ in\ [exist'(x) \wedge spin'(x)]$
$\rightarrow_{cpcx-in^2}$ $letrec\ x = naida\ in\ [spin'(naiad) \wedge exist'(naiad)]$
$\rightarrow_{gc2}$ $[spin'(naiad) \wedge exist'(naiad)]$

Das letrec verhält sich hier im Prinzip wie die Choice-Funktion f. Es bindet das Argument außerhalb des Ausdrucks, so dass bei Auswertung des Arguments der entstandene Wert immer im Skopus des gesamten Ausdrucks bleibt. Bei den Strategien zuvor hat sich

durch die Kopie des Choice-Argumentes der Skopus mitkopiert, so dass zwei voneinander unabhängige Skopi entstanden sind, in denen das Argument für sich ausgewertet wird. Der Unterschied zur Choice-Funktion besteht darin, dass es dem letrec egal ist, ob das Argument, welches von ihm gebunden wird, so auswertet, dass der Ausdruck über den es Skopus besitzt wahr wird oder nicht. Das Kriterium für die Korrektheit der Auswertung ist hier nicht die Wahrheit oder Falschheit des auszuwertenden Ausdrucks, sondern, dass für die gebundene Variable sichergestellt ist, dass sie im gesamten Skopus dem selben Wert über ihre Bindung erhält. Und dies ist hier gegeben. Betrachten wir das Beispiel, welches sich durch Substitution nicht lösen ließ:

Beispiel 17 .....
$\lambda x.[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)](r^*)$

a)strikte Auswertung

$\rightarrow_{choice-l} \lambda x.[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)](r_1)$
$\rightarrow_{l\beta}$ *letrec* $x = r_1$ *in* $[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)]$
$\rightarrow_{cpcx-in^2}$ *letrec* $x = r_1$ *in* $[([orbit'(s^+)](r_1) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_1)]$
$\rightarrow_{gc2} [([orbit'(s^+)](r_1) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_1)]$

$\rightarrow_{choice-r} \lambda x.[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)](r_2)$
$\rightarrow_{l\beta}$ *letrec* $x = r_2$ *in* $[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)]$
$\rightarrow_{cpcx-in^2}$ *letrec* $x = r_2$ *in* $[([orbit'(s^+)](r_2) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_2)]$
$\rightarrow_{gc2} [([orbit'(s^+)](r_2) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_2)]$

b)nicht-strikte Auswertung

$\rightarrow_{l\beta}$ *letrec* $x = r^*$ *in* $[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)]$

$\rightarrow_{choice-l}$ *letrec* $x = r_1*$ *in* $[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)]$
$\rightarrow_{cpcx-in^2}$ *letrec* $x = r_1$ *in* $[([orbit'(s^+)](r_1) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_1)]$
$\rightarrow_{gc2} [([orbit'(s^+)](r_1) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_1)]$

$\rightarrow_{choice-r}$ *letrec* $x = r_2*$ *in* $[([orbit'(s^+)](x) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](x)]$
$\rightarrow_{cpcx-in^2}$ *letrec* $x = r_2$ *in* $[([orbit'(s^+)](r_2) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_2)]$
$\rightarrow_{gc2} [([orbit'(s^+)](r_2) \vee [orbit'(s^+)](r_2)) \wedge [orbit'(u^+)](r_2)]$

Das Problem der Mehrfachbelegung einer Variable x in einem Term, wird von diesem Kalkül gelöst, indem die β-Reduktion modifiziert wird. Das Choice hingegen bleibt wie

es ist. Im Fall der Choice-Funktion wird gerade das Choice anders definiert und Problemlösungen werden hier auch in der Modifikation der Choice-Funktion gesucht (siehe [Win97]). Das klassische λ-Kalkül mit seiner herkömmlichen β-Reduktion wird hier unverändert gelassen.

4 Zusammenfassung und Diskussion

Nach der Einführung in die allgemeine Thematik in Kapitel 1, wurde in Kapitel 2 ein direkter Einblick gegeben in die Funktionsweise eines natürlichsprachlichen Interpreters und darauf aufbauend, nach PTQ, das Sprachfragment aus diesem Interpreter extrahiert. Dadurch wurde es möglich, sprachliche Ausdrücke in eine Form umzuwandeln, welche von unterschiedlichen λ-Kalkülen ausgewertet werden können. Dadurch konnte die Auswertung sprachlicher Ausdrücke in unterschiedlichen Kalkülen verglichen werden. In Kapitel 3 wurde zunächst der Begriff des Choice-Operators eingeführt. Es wurde festgestellt, dass Ausdrücke mit Choice-Operatoren bei der Auswertung Ergebnismengen zurück geben. Hierzu wurde der Begriff der Ergebnismenge und des korrekten Ergebnisses definiert. Es wurde eine nicht-deterministische Variable eingeführt, um das in dieser Arbeit verwendete Fragment um nicht-Determinismus zu erweitern. Sodann wurden zwei unterschiedliche Kalküle auf die Transformation sprachlicher Ausdrücke hin untersucht. Ein besonderer Aspekt war hierbei das Verhalten von Argumenten, welche eine Auswahl aus einer Menge von Eigennamen treffen sollten. Im klassischen Kalkül verhalten sich diese wie Indefinita. Es wurde eine Lösung des Problems durch Substitution vorgeschlagen. Hierzu wurde zunächst definiert, was unter einer geeigneten Substitution zu verstehen ist. Es wurde jedoch gezeigt, dass es nicht immer geeignete Substitutionen für alle Ausdrücke gibt. Ein weiterer Ansatz verwendete zur Lösung des Problems ein λ-Kalkül mit modifizierter β-Reduktion. Hier stellte sich heraus, dass sich das Problem der Eigennamen, die sich wie Indefinita verhalten auflöst und die Auswertung mit der Auswertung über die klassische β-Reduktion in Verbindung mit der Choice-Funktion übereinstimmt.

4.1 Interpreter und Kalküle

Wie anfangs beschrieben, unterscheiden sich funktionale Programmiersprachen wesentlich durch die Art ihrer Auswertung. Wesentlich ist hier auch, ob Sharing verwendet wird oder nicht. Für die Auswertung sprachlicher Ausdrücke ergibt sich in dieser Untersuchung mit nicht-deterministischen Ausdrücken folgendes:

Tabelle 4.1: Auswertung der Untersuchung

	ohne Sharing	**mit Sharing**
strikte Auswertung	√	√
nicht-strikte Auswertung	★	√

Die Situation bei nicht-strikter Auswertung ohne Sharing ist problematisch bei Anwesenheit von nicht-Determinismus, da sie eine Anzahl von nicht korrekten Ergebnissen liefert. Betrachten wir nun noch einmal den Interpreter in [FL89]. Das bei der Auswertung durch die Kalküle entstandene Problem wird hier nicht sichtbar, selbst wenn die verwendete Programmiersprache nicht-strikte Auswertung ohne Sharing verwendet. Dies liegt in der Art und Weise, wie die Ausdrücke vom Interpreter verarbeitet werden. Bei der Substitution haben wir uns diese Arbeitsweise abgeschaut, um zu einer geeigneten Substitution zu kommen. Die Arbeitsweise des Interpreters hält sich sehr eng an Montagues Vorgaben bei der Verarbeitung und zeigt, welche Bedeutung ein Ausdruck $(\lambda x.[spin'(x) \wedge exist'(x)])(p*)$ in der Linguistik hat. Es gibt hier insofern keine Kopie des Arguments in jedes Vorkommen der Variable. Wenn eine solche Einsetzung vorgenommen werden soll, so kann dies nur unter der Verwendung der Choice-Funktion, welche auf dem ϵ-Operator basiert geschehen, bzw. es steht fest, dass alle Vorkommen von x unter einer gemeinsamen Bindung, immer nur denselben Wert annehmen dürfen. Durch die Mengeverarbeitung des Interpreters ist dies von vorneherein gesichert. Kommt ω^* tatsächlich mehrmals in einem Satz vor, so ist es in diesem Fall korrekt, wenn die unterschiedlichen Vorkommen eine unterschiedliche Auswahl treffen, dies liegt in der Natur von ω^* wie der nächste Abschnitt zeigt. Der Interpreter führt dies auch so durch. Im Anhang befinden sich drei Anfrageprotokolle in denen dies sichtbar wird. Am Beispiel *r* or rosalind orbit saturn and r* orbits uranus* läßt sich erkennen, dass beide Kandidaten sich gleich verhalten. Die Frage ist eine Fangfrage und darf, sofern die beiden r* denselben Wert bekommen, immer nur mit „No, this is false.“ beantwortet werden, da sich die Bedingungen des Satzes für gleich eingesetzte r* wiedersprechen. Beide Kandidaten sind jedoch in der Lage, diese Fangfrage mit „Yes, this is true.“ zu beantworten, was zeigt, dass hier unterschiedlich für jedes r* gewählt wurde. Es wäre noch möglich ω^* so zu implementieren, dass es sich beim ersten Auftreten die Auswahl merkt, und bei jedem weiteren Auftreten nachgesehen wird, ob es dieses ω^* schon gibt. Wenn ja, wird der bereits gewählte Wert verwendet. Auf diese Weise hätte man dann ein Sharing für mehrere Auftreten simuliert. Meint man zwei unterschiedliche ω^* mit demselben Anfangsbuchstaben, so könnte man hierfür eine andere Bezeichnung einführen, bspw. ω°. Tritt dieses auf, wird jedes mal neu gewählt.
Die Montague-Semantik des Interpreters, wie sie durch [FL89] umgesetzt wurde und die Semantik der funktionalen Programmiersprache in welcher dieser Interpreter implementiert ist, gerade in Bezug auf die Auswertungsstrategie, sind voneinander relativ unabhängig.

4.2 ω^* und nicht-strikte Auswertung

Bei der Auswertung von sprachlichen Ausdrücken die ω^* enthalten, ist aufgefallen, dass sie sich, obwohl sie Platzhalter für einen Eigennamen sind, bei nicht-strikter Auswertung ohne Sharing wie ein Indefinitum verhalten. Bei strikter Auswertung hingegen, ist dies nicht der Fall. Wenn ω^* kopiert wird, so werden die beiden Kopien voneinander unabhängig. Es entsteht in diesem Fall gerade die Repräsentation in Beispiel IIb). ω^* bei nicht-strikter Auswertung ohne Sharing ist ein Existenzquantor der Form:
$\exists$x : x beginnt mit ω.

Anhang

Anhang 1: Literaturverzeichnis

Sig	Ref
[Bar81]	Barendregt,H.P.: *The Lambda Calculus - Its Syntax and Semantics*, Studies in Logic and the Foundation of Mathmatics, Vol 103,Elsevier, North-Holland, 1981
[Chu41]	Church, Alonzo: *The Calculi of Lambda Conversion*,Princeton University Press, Princeton, 1941
[Dow79]	Dowty, David: *Word Meaning and Montague Grammar*,Dordrecht, Reidel Publishing Company,1979
[DWP81]	Dowty, David / Wall, Robert E. / Peters, Stanley: *Intoduction to Montague Semantics*, Dordrecht,Kluver Academic Publishers Group, 1981
[FL89]	Frost,R./Launchburry,J.: *Construction Natural Language Interpreters in a Lazy Functional Language*, In: The Computer Journal, Vol.32, No.2,1989
[Fre80]	Frege,Gottlob: *Funktion,Begriff,Bedeutung*,Hrsg.:Günther Patzig,8.Aufl., Vandenhoeck&Ruprecht,1980
[GS82]	Groenendijk,J./Stokhof,M.: *Semantic Analysis of WH-Complements*, In: Linguistics and Philosophy 5, 1982,S.175-233
[HL2000]	Hügli, Anton / Lübcke, Poul(Hrsg.): *Philosophie im 20.Jahrhundert*,Hamburg,Rowohlt Taschenbuch,2000
[Kar]	Karttunen,Lauri: *Syntax and Semantics of Questions*, In:Linguistics and Philosophy 1,1977,S.3-44
[Kut2000]	Kutzner,Arne: *Ein nichtdeterministischer call-by-need Lambda Kalkül mit erratic Choice: operationale Semantik, Programmtransformationen und Anwendungen*, Diss. FB Informatik, J.W.v.Goethe Universität Frankfurt,2000

[LP95] De'Ligurio,U. / Piperno,A.:
Nondeterministic Extensions of Untyped λ-Calculus,Rome,
Academic Press 1995

[Man95] Mandel, Luis:
Constraint Lambda Calculus, Diss. FB Mathematik,
Ludwig-Maximilian-Universität München, 1995

[Mon74] Montague, Richard:
Formal Philosophy,Hrsg.:Richmond H. Thomason,New Haven,London,
Yale Univerity Press,1974

[Qui69] Quine W.V.O.:
Grundzüge der Logik, Übers.:Dirk Siefkes,suhrkamp taschenbuch wissenschaft 65,
Suhrkamp Verlag, Frankfurt, 1969

[Sab03] Sabel, David:
*Realisierung der Ein-/Ausgabe in einem Compiler für Haskell
bei Verwendung einer nichtdeterministischen Semantik*, DiplA., FB Informatik,
J.W.v.Goethe Universität,Frankfurt,2003

[SSS04] Schmidt-Schauß, M./ Schütz, M. / Sabel D.:
On the safety of Nöcker's strictness analysis, Frank Report 19, Institut für
Informatik, J.W. Goethe-Universität Frankfurt a.M, 2004,
www.ki.cs.uni-frankfurt.de/papers/frank/

[Tur37] Turing, Alain:
Computability and λ-definability, Journal of Symbolic Logic 2, S.153-163, 1937

[vBe95] van Benthem, Johan:
Language in action,MIT Press, Amsterdam,1995

[vHe97] von Heusinger,Klaus:
*Salienz und Referenz - der Epsilonoperator in der Semantik der Nominalphrase
und anaphorischer Pronomen*, studia grammatica 43, Akademie Verlag, Berlin,1997

[Win97] Winter,Yoad:
Choice Functions and the Scopal Semantics of Indefinites,
In:Linguistics and Philosophy 20, 399-467, 1997

Anhang 2: Listing Interpreter

```
module Main where

import Data.List
import Prelude hiding(fail)
import Random
import System.IO.Unsafe


infixr 8 ///
infixl 6 >>>
infixr 4 |||

main = do
        c <- readFile "Input.txt"
        writeFile "Output2.txt" (interpret c)

-- Mengenverarbeitende Funktionen ------------------------

{-union :: [Int] -> [Int] -> [Int]
union as bs = as ++(bs\\as)

intersect :: [Int] -> [Int] -> [Int]
intersect as bs = as \\ (as\\bs)-}

includes as bs = (as\\bs) == []

intersct :: ([Int],[Int]) -> [Int]
intersct (a,b) = intersect a b
-- Syntaktische Kategorien -------------------------------

commonnoun :: [(String,[Int])]
commonnoun =
        [("thing", commonnoun_thing),
         ("things", commonnoun_thing),
         ("man", commonnoun_man),
         ("men", commonnoun_man),
         ("woman", commonnoun_woman),
         ("women", commonnoun_woman),
         ("person", commonnoun_person),
         ("persons", commonnoun_person),
         ("planet", commonnoun_planet),
         ("planets", commonnoun_planet),
         ("moon", commonnoun_moon),
         ("moons", commonnoun_moon),
         ("sun", commonnoun_sun),
         ("celestial-body",commonnoun_celbody)]

propernoun :: [(String, [Int] -> Bool)]
propernoun =
```

```
50          [("sun", test_property_wrt 8),
             ("mercurio", test_property_wrt 9),
             ("venus", test_property_wrt 10),
             ("earth", test_property_wrt 11),
             ("mars", test_property_wrt 12),
55           ("jupiter", test_property_wrt 13),
             ("saturn", test_property_wrt 14),
             ("uranus", test_property_wrt 15),
             ("neptune", test_property_wrt 16),
             ("pluto", test_property_wrt 17),
60           ("deimos", test_property_wrt 18),
             ("phobos", test_property_wrt 19),
             ("io", test_property_wrt 20),
             ("europa", test_property_wrt 21),
             ("ganymed", test_property_wrt 22),
65           ("kallisto", test_property_wrt 23),
             ("amalthea", test_property_wrt 24),
             ("himalia", test_property_wrt 25),
             ("elara", test_property_wrt 26),
             ("pasiphae", test_property_wrt 27),
70           ("sinope", test_property_wrt 28),
             ("lysithea", test_property_wrt 29),
             ("carme", test_property_wrt 30),
             ("ananke", test_property_wrt 31),
             ("leda", test_property_wrt 32),
75           ("adrastea", test_property_wrt 33),
             ("thebe", test_property_wrt 34),
             ("metis", test_property_wrt 35),
             ("mimas", test_property_wrt 36),
             ("enceladus", test_property_wrt 37),
80           ("tethys", test_property_wrt 38),
             ("dione", test_property_wrt 39),
             ("rhea", test_property_wrt 40),
             ("titan", test_property_wrt 41),
             ("hyperion", test_property_wrt 42),
85           ("japetus", test_property_wrt 43),
             ("phoebe", test_property_wrt 44),
             ("janus", test_property_wrt 45),
             ("epimetheus", test_property_wrt 46),
             ("1980S6", test_property_wrt 47),
90           ("telesto", test_property_wrt 48),
             ("calypso", test_property_wrt 49),
             ("atlas", test_property_wrt 50),
             ("prometheus", test_property_wrt 51),
             ("pandora", test_property_wrt 52),
95           ("ariel", test_property_wrt 53),
             ("umbriel", test_property_wrt 54),
             ("titania", test_property_wrt 55),
             ("oberon", test_property_wrt 56),
             ("miranda", test_property_wrt 57),
100          ("triton", test_property_wrt 58),
             ("nereid", test_property_wrt 59),
```

```
            ("naiad", test_property_wrt 60),
            ("thalassa", test_property_wrt 61),
            ("galathea", test_property_wrt 62),
            ("despina", test_property_wrt 63),
            ("larissa", test_property_wrt 64),
            ("proteus", test_property_wrt 65),
            ("charon", test_property_wrt 66),
            ("luna", test_property_wrt 67),
            ("cordelia", test_property_wrt 68),
            ("ophelia", test_property_wrt 69),
            ("bianca", test_property_wrt 70),
            ("cressida", test_property_wrt 71),
            ("desdemona", test_property_wrt 72),
            ("juliet", test_property_wrt 73),
            ("portia", test_property_wrt 74),
            ("rosalind", test_property_wrt 75),
            ("belinda", test_property_wrt 76),
            ("puck", test_property_wrt 77),
            ("Hall", test_property_wrt 84),
            ("Bernad", test_property_wrt 85),
            ("Galilei", test_property_wrt 86),
            ("Bernad", test_property_wrt 87),
            ("Perine", test_property_wrt 88),
            ("Melotte", test_property_wrt 89),
            ("Nicholson", test_property_wrt 90),
            ("Kowal", test_property_wrt 91),
            ("Marius", test_property_wrt 92),
            ("Huygens", test_property_wrt 93),
            ("Kuiper", test_property_wrt 94),
            ("Cassini", test_property_wrt 95),
            ("Tombough", test_property_wrt 96)
           ]

adjective :: [(String,[Int])]
adjective =
        [("atmospheric", adjective_atmospheric),
         ("red", adjective_red),
         ("gaseous", adjective_gaseous),
         ("solid", adjective_solid),
         ("blue",adjective_blue),
         ("vacuumous",adjective_vacuumous),
         ("ringed",adjective_ringed)]

relpronoun :: [(String, [Int] -> [Int] -> [Int])]
relpronoun =
        [("who", relpronoun_who),
         ("that", relpronoun_that),
         ("which", relpronoun_that)]

indefinitepronoun :: [( String, [Int] -> Bool)]
indefinitepronoun =
        [("someone", meaning_of detphrase "a person."),
```

```
           ("something", meaning_of detphrase "a thing."),
           ("somebody", meaning_of detphrase "a person."),
           ("everyone", meaning_of detphrase "every person."),
           ("everything", meaning_of detphrase "every thing.")]

transverb :: [(String, ([Int] -> Bool) -> [Int])]
transverb =
        [("discover", trans_verb rel_discover),
         ("discovered", trans_verb rel_discover),
         ("orbit", trans_verb rel_orbit),
         ("orbited", trans_verb rel_orbit),
         ("orbits", trans_verb rel_orbit)]

intransverb :: [(String,[Int])]
intransverb =
        [("exist", intransverb_exists),
         ("exists", intransverb_exists),
         ("orbit", meaning_of verbphrase "orbit something"),
         ("orbits", meaning_of verbphrase "orbit something"),
         ("spin", intransverb_spin),
         ("spins", intransverb_spin)]

passtrvb :: [(String, ([Int] -> Bool) -> [Int])]
passtrvb =
        [("discovered", passtr_verb rel_discover),
         ("orbited", passtr_verb rel_orbit)]

linkingverb :: [(String, a -> a)]
linkingverb =
        [("is", id), ("was", id), ("are", id), ("were", id)]

determiner :: [(String,[Int] -> [Int] -> Bool)]
determiner =
        [("the", determiner_a),
         ("a", determiner_a),
         ("some", determiner_a),
         ("no", determiner_none),
         ("every", determiner_every),
         ("all", determiner_every),
         ("one", determiner_one),
         ("two", determiner_two)]

termphrasejoin :: [(String, (a-> Bool) ->(a-> Bool) -> a -> Bool)]
termphrasejoin = [("and", termphrase_and), ("or", termphrase_or)]

verbphrasejoin :: [(String, [Int] -> [Int] -> [Int])]
verbphrasejoin = [("and", verbphrase_and), ("or", verbphrase_or)]

nounjoin :: [(String, [Int] -> [Int] -> [Int])]
nounjoin = [("and", noun_and), ("or", noun_or)]

preposition :: [(String, a -> a)]
```

```
preposition = [("by", id)]

-- "Denotatlisten" -------------------------------------

entityset :: [Int]
entityset = [1..100]

commonnoun_sun :: [Int]
commonnoun_sun = [8]

commonnoun_planet :: [Int]
commonnoun_planet = [9..17]

commonnoun_moon :: [Int]
commonnoun_moon = [18..77]

commonnoun_man :: [Int]
commonnoun_man = [84..96]

commonnoun_woman :: [Int]
commonnoun_woman = []

commonnoun_person :: [Int]
commonnoun_person = union commonnoun_woman commonnoun_man

commonnoun_thing :: [Int]
commonnoun_thing = union (union commonnoun_sun commonnoun_planet)
                         (union commonnoun_moon commonnoun_person)

commonnoun_celbody:: [Int]
commonnoun_celbody = union commonnoun_sun (union commonnoun_planet
                           commonnoun_moon)

adjective_red :: [Int]
adjective_red = [12,13,14,22]

adjective_blue :: [Int]
adjective_blue = [11,14,15,16]

adjective_ringed :: [Int]
adjective_ringed = [13,14,15,16]

adjective_gaseous :: [Int]
adjective_gaseous = [13,14,15,16]

adjective_solid :: [Int]
adjective_solid = (commonnoun_planet ++ commonnoun_moon)
                                     \\ adjective_gaseous

adjective_atmospheric :: [Int]
adjective_atmospheric = [10,11,12,22,41]
```

```
adjective_vacuumous :: [Int]
adjective_vacuumous = (commonnoun_planet ++ commonnoun_moon)
                       \\ adjective_atmospheric

intransverb_exists :: [Int]
intransverb_exists = entityset

intransverb_spin :: [Int]
intransverb_spin = [8..65]

relpronoun_that :: [Int] -> [Int] -> [Int]
relpronoun_that = intersect

relpronoun_who :: [Int] -> [Int] -> [Int]
relpronoun_who = intersect

termphrase_and :: (a -> Bool) -> (a -> Bool) -> a -> Bool
termphrase_and p q x = p x && q x

termphrase_or :: (a -> Bool) -> (a -> Bool) -> a -> Bool
termphrase_or p q x = p x || q x

verbphrase_and :: [Int] -> [Int] -> [Int]
verbphrase_and = intersect

verbphrase_or :: [Int] -> [Int] -> [Int]
verbphrase_or = union

noun_and :: [Int] -> [Int] -> [Int]
noun_and = intersect

noun_or :: [Int] -> [Int] -> [Int]
noun_or = union

determiner_every :: [Int] -> [Int] -> Bool
determiner_every = includes

determiner_a :: [Int] -> [Int] -> Bool
determiner_a xs ys = (length (intersect xs ys)) /=0

determiner_none :: [Int] -> [Int] -> Bool
determiner_none xs ys = (length (intersect xs ys)) == 0

determiner_one :: [Int] -> [Int] -> Bool
determiner_one xs ys = (length (intersect xs ys)) == 1

determiner_two :: [Int] -> [Int] -> Bool
determiner_two xs ys = (length (intersect xs ys )) == 2

test_property_wrt :: Eq a => a -> [a] -> Bool
test_property_wrt e ps = elem e ps
```

```
rel_orbit :: [(Int,Int)]
rel_orbit = [(x,8)|x<-[9..17]]
            ++ [(67,11),(18,12),(19,12)]
            ++ [(x,13)|x<-[20..35]]
            ++ [(x,14)|x<-[36..52]]
            ++ [(x,15)|x<-[53..57]]
            ++ [(x,15)|x<-[68..77]]
            ++ [(x,16)|x<-[58..65]]
            ++ [(66,17)]

rel_discover :: [(Int,Int)]
rel_discover = [(84,19),(84,18)]
               ++ [(y,x)|x<-[20..23],y<-[86,92]]
               ++ [(85,24),(86,25),(86,26),(87,27)]
               ++ [(88,x)| x<-[28..31]]
               ++ [(89,32),(93,41),(94,57),(94,59),(96,17)]

trans_verb :: [(Int,Int)] -> ([Int] -> Bool) -> [Int]
trans_verb rel p = [x|(x,image_x) <- (collect rel), p image_x]

passtr_verb :: [(Int,Int)] -> ([Int] -> Bool) -> [Int]
passtr_verb rel = trans_verb (invert rel)

collect :: [(Int,Int)] -> [(Int,[Int])]
collect [] = []
collect ((x,y):t) = (x,y:[e2|(e1,e2)<-t, e1==x])
                    :collect [(e1,e2)|(e1,e2)<-t, e1/=x]

invert :: [(a,a)] -> [(a,a)]
invert = map swap where swap (x,y) = (y,x)

-- Interpretationsfunktionen ----------------------------

succeed :: a -> b -> [(a,b)]
succeed v inp = [(v,inp)]

fail :: a -> [b]
fail inp = []

p ||| q = \inp -> (p inp ++ q inp)

p1 /// p2 = \inp -> [((v1,v2),inp2)|(v1,inp1)<- p1 inp,
                                    (v2,inp2)<- p2 inp1]

(>>>) :: (a -> [(b,c)]) -> (b -> d) -> a -> [(d,c)]
p >>> fn = \inp -> [(fn v, inp')|(v,inp') <- p inp]

item :: Eq a => (a,b) -> [a] -> [(b,[a])]
item (word,val) [] = fail []
item (word,val) (wd:wds)
   |wd == word = succeed val wds
   |otherwise = fail wds
```

```
always :: (a,b)->[a] ->[(b,[a])]
always (word,val) []= fail []
always (word,val) (wd:wds) = succeed val wds

epsilon :: [String] -> [([Int],[String])]
epsilon a = [ (entityset,a)]

vepsilon x = [(id,x)]

tepsilon x = [(id,x)]

epsilonn x = [(id,x)]

-------------------------------------------------------

m :: [(String,b)] -> [String] -> [(b,[String])]
m ps qs     = case (m'' ps qs) of
                []  ->  m' ps qs
                x   ->  x
m'' [] = fail

m'' (p:ps)= (item p)
            ||| m'' ps  --
            ||| seq (unsafePerformIO (print (map fst liste) >> print zahl))
                    (item (liste!!zahl))

m' ps  (q@([wd]:wds))= let
                        liste=(collection ps wd)
                        zahl=max (getInt 0 (length(liste)-1)) 0
                        ele = (liste!!zahl)
                        (a,_) = ele
                        element = case liste of
                           [] ->  fail
                           other -> seq (output) (always (ele))
                        output = unsafePerformIO (do
                                                    print (a)
                                                    print (zahl))
                       in (element q)
m' ps as = fail as
-------------------------------------------------------


qmark :: [String] -> [(String,[String])]
qmark = item ("?","?")

dot :: [String] -> [(String,[String])]
dot = item (".",".")

meaning_of :: ([String] -> [(a,[String])]) -> String -> a
meaning_of = \interp -> \x ->
               the_value ((( interp /// dot ) >>> fst )( wordss x ))
```

```
the_value :: [(a,b)] -> a
the_value [(v,inp)] = v

-- Satzkonstruktionen

simplenounclause :: [String] -> [([Int],[[Char]])]
simplenounclause = m commonnoun
                   ||| adjectives /// m commonnoun >>> intersct

relnounclause :: [String] -> [([Int],[String])]
relnounclause = epsilon
                ||| m relpronoun /// verbphrase /// joinvbphrase >>> applysnd

adjectives :: [[Char]] -> [([Int],[[Char]])]
adjectives = m adjective
             ||| m adjective /// adjectives >>> intersct

nounclause =
     epsilonn
     ||| m nounjoin /// simplenounclause /// relnounclause /// nounclause
                         >>> (\ (a,(b,(c,d))) -> \x -> a (intersect b c)(d x))

transverbphrase :: [String] -> [([Int],[String])]
transverbphrase =
  m transverb /// termphrase /// jointermphrase >>> applysp
   ||| m linkingverb /// m passtrvb /// m preposition
       /// termphrase /// jointermphrase >>> drop3rdsp

detphrase :: [String] -> [([Int] -> Bool,[String])]
detphrase = m indefinitepronoun
            ||| m determiner
                 /// simplenounclause /// relnounclause /// nounclause
                       >>> (\(a,(b,(c,d))) -> (\x -> a (intersect b c)(d x)))

termphrase :: [String] -> [([Int] -> Bool,[String])]
termphrase = m propernoun ||| detphrase

verbphrase :: [String] -> [([Int],[String])]
verbphrase =
    transverbphrase
    ||| m intransverb
    ||| m linkingverb /// m determiner
        /// simplenounclause /// relnounclause /// nounclause
         >>> (\(a,(b,(c,(d,e)))) -> (a (e (intersect c d ))))

jointermphrase =
   tepsilon
   ||| m termphrasejoin /// termphrase /// jointermphrase >>> applysp

joinvbphrase :: [String] -> [([Int] ->[Int],[String])]
joinvbphrase =
```

```
  vepsilon
  ||| m verbphrasejoin /// verbphrase /// joinvbphrase >>> special2

sentence :: [String] -> [(Bool,[String])]
sentence = termphrase /// jointermphrase
           /// verbphrase /// joinvbphrase >>> reorder4


-- Zusatzfunktionen -----------------------------------

apply2 (x,y) = x y
apply3 (x,(y,z)) = x y z
applysp (x,(y,z)) = x (z y)
applysnd (x,(y,z)) = z y
applyfstsnd (x,(y,z)) = x (z y)
reorder (x,(y,z)) = y x z
drop2nd (x,(y,z)) = x z
drop3rd (w,(x,(y,z))) = w x z
drop3rdsp (w,(x,(y,(u,v)))) = w x (v u)
apply3rdtofrd (w,(x,(y,z))) = w x (z y)
reorder4 (x,(y,(z,w))) = (y x (w z))

spec (x,(y,z))=y x z

special :: ([Int],([Int],([Int] -> [Int] -> [Int],[Int]))) ->[Int]
special (a,(b,(c,d))) = c (intersect a b) d
special2 (x,(y,z)) = \u -> z (x u y)

-- Interpretationsfunktion ----------------------------

interpret :: String -> String
interpret x = disambiguate (map fst ((question /// qmark >>> fst)(wordss x)))

-- Word-Processing ------------------------------------

disambiguate :: [String] -> String
disambiguate [] = "I am sorry. I cannot understand your query!Try again. "
disambiguate [ans] = ans
disambiguate answers
 = "Your query is ambiguous. There are more than one interpretations of it: "
   ++ concat (map ( \x ->(newans x)++"\n")answers)
       where newans a = "\n * " ++ a

-- Fragen-Processing ----------------------------------

question :: [String] -> [(String,[String])]
question = sentence >>> truefalse
           ||| sentence ///m sentence_and///sentence >>>spec
           ||| sentence ///m sentence_or ///sentence >>>spec
           ||| m doesq /// sentence >>> apply2
           ||| m didq /// sentence >>> apply2
           ||| m quest1 /// verbphrase /// joinvbphrase >>> applyfstsnd
```

```
                ||| m quest2 /// simplenounclause /// relnounclause /// nounclause
                       /// verbphrase /// joinvbphrase
                            >>> (\(a,(b,(c,(d,(e,f))))) -> a(d(intersect b c))(f e))
                ||| howmany /// simplenounclause /// relnounclause /// nounclause
                    /// verbphrase /// joinvbphrase
                            >>> (\(a,(b,(c,(d,(e,f))))) -> a(d(intersect b c))(f e))

howmany :: [String] -> [([Int] ->[Int] -> String,[String])]
howmany = m howq /// m manyq >>> fst

truefalse :: Bool -> String
truefalse True = "Yes, this is true.\n"
truefalse False = "No, this is false.\n"

sentence_and :: [(String,Bool->Bool ->String)]
sentence_and = [("and",truefalse2)]

truefalse2:: Bool->Bool->String
truefalse2 x y= case (x&&y) of
                  True -> "Yes, this is true.\n"
                  False ->"No, this is false.\n"

sentence_or :: [(String,Bool->Bool ->String)]
sentence_or = [("or",truefalse3)]

truefalse3:: Bool->Bool->String
truefalse3 x y= case (x||y) of
                  True -> "Yes, this is true.\n"
                  False ->"No, this is false.\n"

doesq :: [(String,Bool -> String)]
doesq = [("does",yesno)]

didq :: [(String,Bool -> String)]
didq = [("did",yesno)]

quest1 :: [(String,[Int] -> String)]
quest1 = [("who", function_whoq)]

quest2 :: [(String,[Int] ->[Int] -> String)]
quest2 = [("which", function_whichq)]

howq :: [(String,[Int] -> [Int] -> String)]
howq = [("how", function_howmanyq)]

manyq :: [(String, a -> a)]
manyq = [("many",id)]

function_whoq :: [Int] -> String
function_whoq xs = check "nobody"
                     [name_of e | e <- xs,
                        elem e (union commonnoun_man commonnoun_woman)]
```

```
570
    function_whatq :: [Int] -> String
    function_whatq xs = check "nothing" [name_of e| e <- xs]

    function_whichq :: [Int] -> [Int] -> String
575 function_whichq  xs ys = check "none" [name_of e| e <- intersect xs ys]

    function_howmanyq :: [Int] -> [Int] -> String
    function_howmanyq xs ys = number (length (intersect xs ys))

580 yesno :: Bool -> String
    yesno True = "Yes!"
    yesno False = "No!"

    number :: Int -> String
585 number n
       |(n<14) = head (drop n ["none.","one.","two.","three.","four.","five.",
                               "six.","seven.","eight.","nine.","ten.","elven.",
                               "twelwe","thirteen."])
       |otherwise = show n
590
    name_of :: Int -> String
    name_of e = head [name|(name,f) <-propernoun, f[e]]

    check :: String -> [String] -> String
595 check str [] = str
    check str wds = unwordss (nub wds)

    wordss :: String -> [String]
    wordss [] = []
600 wordss (' ':' ':cs) = wordss (' ':cs)
    wordss (' ':'?':cs) = []:['?']:wordss cs
    wordss (' ':'.':cs) = []:['.']:wordss cs
    wordss (' ':cs) = []:wordss cs
    wordss ('.':cs) = []:['.']:wordss cs
605 wordss ('?':cs) = []:['?']:wordss cs
    wordss (c:cs) = (c:ln):lns
                          where (ln:lns) = wordss cs

    unwordss :: [String] -> String
610 unwordss [x] = x ++ "."
    unwordss [x,y] = x ++ ", and " ++y++"."
    unwordss (x:xs) = x++ ", " ++ unwordss xs

    ----------------nD-Erweiterung ----------------------------------------
615
    collection :: Eq a => [([a],b)] ->a ->[([a],b)]
    collection [] _ = []
    collection (((p:ps),b):xs) i = if (i==p)
                   then ((p:ps),b):(collection xs i)
620                else collection xs i
```

```
getInt :: Int -> Int -> Int
getInt x y = unsafePerformIO(getStdRandom (randomR (x,y)))
```

Anhang 3: Vokabular der Basisausdrücke

1. Eigennamen:
   adrastea, amalthea, ananke , ariel, atlas

   belinda, Bernard, bianca

   calypso, carme, Cassini, charon, cordelia, cressida

   deimos, desdemona, despina, dione

   earth, elara, enceladus, epimetheus, europa

   galathea, Galilei, ganymed

   Hall, himalia, Huygens, hyperion

   io

   janus, japetus, juliet, jupiter

   kallisto, Kowal, Kuiper

   larissa, leda, luna, lysithea

   Marius, mars, Melotte, mercurio, metis, mimas, miranda

   naiad, neptune, nereid, Nicholson

   oberon, ophelia

   pandora, pasiphae, Perine, phoebe, phobos, pluto, portia, prometheus
   proteus, puck

   rhea, rosalind

   saturn, sinope, sun

   telesto, tethys, thalassa, thebe, titan, titania, Tombough, triton

   umbriel, uranus

   venus

   1980S6

2. Sortale:
   celestial-body, man, men, moon, moons, person, persons, planet, planets, sun
   thing, things, woman, women

3. Adjektive:
   atmospheric, blue, gaseous, red, ringed, solid, vacuumous

4. intransitive Verben:
   exist, exists, spin, spins

5. transitive Verben:
   discover, discovers, discovered, orbit, orbits, orbited

Anhang 4: Protokolle

1. Experiment-Protokoll Sonne,Mond und Sterne.
   Erstellt am: Sun Apr 2 10:43:27 2006

   Durchlaufergebnisse:

   Anfrage: r or rosalind orbit saturn and r orbits uranus?
   Gesamtergebnis der Anfrage: positiv

   Durchlauf 1: positiv

   Einzelheiten:
   Antworten Kandidat 1:
   Anfrage 1:
   r or rosalind orbit saturn and r orbits uranus?
   ****
   Durchlauf: 1.1
   Yes, this is true.

   Durchlaufergebnis: positiv
   Gesamtergebnis: positiv
   Choice Kandidat 1:

   rhea
   0
   rhea
   0
   rhea
   0
   rosalind
   1
   rosalind
   1
   rhea
   0
   rhea
   0

   Antworten Kandidat 2:
   Anfrage1:
   r or rosalind orbit saturn and r orbits uranus?
   ****
   Durchlauf: 1.1
   No, this is false.

   Durchlaufergebnis: positiv

Gesamtergebnis: positiv

Choice Kandidat 2:
rhea
0
rhea
0
rosalind
1
rhea
0
rosalind
1
rhea
0
rhea
0

2. Experiment-Protokoll Sonne,Mond und Sterne.
Erstellt am: Sun Apr 2 10:49:44 2006

Durchlaufergebnisse:

Anfrage: r or rosalind orbit saturn and r orbits uranus?
Gesamtergebnis der Anfrage: negativ

Durchlauf 1: negativ

Einzelheiten:
Antworten Kandidat 1:
Anfrage 1:
r or rosalind orbit saturn and r orbits uranus?
****

Durchlauf: 1.1
Yes, this is true.

Durchlaufergebnis: negativ
Gesamtergebnis: negativ

Choice Kandidat 1:
rosalind
1
rosalind
1
rhea
0
rosalind
1
rhea
0

rhea
0
rosalind
1
Antworten Kandidat 2:
Anfrage1:
r or rosalind orbit saturn and r orbits uranus?
****
Durchlauf: 1.1
Yes, this is true.
Durchlaufergebnis: negativ
Gesamtergebnis: negativ
Choice Kandidat 2:
rosalind
1
rosalind
1
rhea
0
rosalind
1
rhea
0
rhea
0
rosalind
1

3. Experiment-Protokoll Sonne,Mond und Sterne.
   Erstellt am: Sun Apr 2 10:52:08 2006
   Durchlaufergebnisse:
   Anfrage: r or rosalind orbit saturn and r orbits uranus?
   Gesamtergebnis der Anfrage: positiv
   Durchlauf 1: positiv
   Einzelheiten:
   Antworten Kandidat 1:
   Anfrage 1:
   r or rosalind orbit saturn and r orbits uranus?
   ****
   Durchlauf: 1.1
   No, this is false.
   Durchlaufergebnis: positiv

Gesamtergebnis: positiv
Choice Kandidat 1:
rosalind
1
rhea
0
rhea
0
rhea
0
rosalind
1
rosalind
1
rosalind
1
Antworten Kandidat 2:
Anfrage1:
r or rosalind orbit saturn and r orbits uranus?
****
Durchlauf: 1.1
Yes, this is true.
Durchlaufergebnis: positiv
Gesamtergebnis: positiv
Choice Kandidat 2:
rhea
0
rosalind
1
rhea
0
rosalind
1
rosalind
1
rosalind
1
rhea
0

Anhang 4: Anleitung zur CD

1. Um den Interpreter auf der CD zu testen, müssen folgende Compiler auf dem System installiert sein:

   1.) **Python 2** (frei verfügbar unter www.python.org)
   2.) **ghc 5.04.3** (frei verfügbar unter www.haskell.org)
   3.) **HasFuse** (zu erfragen am Lehrstuhl für Künstliche Intelligenz und Softwaretechnologie an der J.W.v.Goethe Univeristät Frankfurt a.M., www.ki.cs.uni-frankfurt.de)

2. Auf der CD befindet sich der Ordner SonneMondundSterne. Dieser sollte auf die Festplatte kopiert werden. Von dort kann das Programm SonneMondundSterne.py aufgerufen werden.

3. Wichtig ist, dass für die Dateien Input.txt, Input2.txt, Output.txt und Output2.txt *read* und *write* Rechte gesetzt sind.