Friedrich-Alexander-Universität Erlangen-Nürnberg

Institut für Mathematische Maschinen
und Datenverarbeitung VIII

Ein System für dialogische Logik

Diplomarbeit
im Fach Informatik

vorgelegt von
Jürgen Ehrensberger
geb. am 03.01.1968 in Roth

Betreuer: Prof. Dr. G. Görz
Dipl.-Inf. C. Zinn

Beginn der Arbeit: 01.12.95
Abgabe der Arbeit: 03.06.96

Versicherung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 03.06.96

.....

Zusammenfassung

Die dialogische Sichtweise der Logik begründet einen Kalkül, in dem die Gültigkeit logischer Aussagen mittels schematisierter Dialoge überprüft wird. In dieser Arbeit wird die Implementation eines Systems vorgestellt, das es erlaubt, derartige Dialogspiele zu führen und Beweise für Aussagen zu erzeugen. Beweise können dabei sowohl unter der Kontrolle des Benutzers als auch selbständig durch das System gefunden werden. Nach einer Einführung in die dialogische Logik in den ersten beiden Kapiteln skizziert Kapitel 3 den Aufbau des Systems. Die Kapitel 4 bis 8 geben eine detaillierte Beschreibung der Komponenten unter Berücksichtigung ihrer Implementation. Die Anwendung des Systems zum Führen von Dialogspielen ist in Kapitel 9 dargestellt. Das abschließende Kapitel 10 faßt die wichtigsten Eigenschaften des Systems zusammen und nennt Erweiterungsmöglichkeiten.

Inhaltsverzeichnis

1 Einleitung **1**

2 Dialogische Logik **2**

3 Aufbau des Systems **10**

4 Sortenverwaltung **13**

5 Symbolverwaltung **17**

5.1 Variablensymbole 18

5.2 Konstantensymbole 18

5.3 Funktionssymbole 20

5.4 Prädikatsymbole 21

5.5 Junktorsymbole 21

5.6 Quantorsymbole 22

6 Formelaufbau **23**

6.1 Erzeugung von Formelobjekten 23

6.2 Methoden von Term- und Formelobjekten 25

6.2.1 Klassifikation von Objekten 25

6.2.2 Selektormethoden 25

6.2.3 Modifikatormethoden 27

6.2.4 Ausgabe von Objekten 27

6.2.5 Unifikation 28

6.2.6 Ersetzen von Teiltermen 28

7 Dialogspiel **30**

7.1 Sprechakte 30

7.2 Dialogliste 32

7.3 Partikelregeln 36

7.4 Rahmenregeln 40

7.5 Heuristik 44

7.6 Thesen- und Hypothesenverwaltung 45

7.7 Strategiebildung . . . 46
7.7.1 Baumaufbau . . . 46
7.7.2 Termwahl und Termsymbole . . . 49
7.7.3 Primformelübernahme . . . 50
7.7.4 Unifikation und Bindungsliste . . . 51
7.7.5 Failure-Continuations . . . 53
7.8 Benutzerspiel . . . 56
7.9 Strategieverwaltung . . . 57

8 Benutzeroberfläche 59
8.1 Textoberfläche . . . 59
8.1.1 Kommandoeingabe . . . 59
8.1.2 Kommandoausführer . . . 60
8.1.3 Parser . . . 60
8.1.4 Ausgabesteuerung . . . 68
8.1.5 Eingabemenüs . . . 70
8.2 Graphische Oberfläche . . . 71
8.2.1 Textfeld . . . 71
8.2.2 Dialogausgabe . . . 72
8.2.3 Eingabemenüs . . . 73
8.2.4 Strategiegrapher . . . 74

9 Bedienung und Beispielbeweise 76
9.1 Installation . . . 76
9.2 Starten des Systems . . . 76
9.3 Anweisungen . . . 77
9.3.1 Deklarationen . . . 77
9.3.2 Thesen und Hypothesen . . . 79
9.3.3 Dialogspiel . . . 79
9.4 Menüsteuerung . . . 81
9.5 Beispielbeweise . . . 82

10 Resümee und Ausblick 87

A Partikel und Partikelregeln 93

B Funktionen und Methoden 97

C Beispiel einer Kommandodatei 99

Literaturverzeichnis 101

Kapitel 1

Einleitung

Die dialogische Sichtweise der Logik wurde von Paul Lorenzen erstmals in den Vorträgen „Logik und Agon“ [13] 1958 und „Ein dialogisches Konstruktivitätskriterium“ [14] 1959 formuliert. Dieser neue Ansatz entstand im Zusammenhang mit der Präzisierung des Begriffs der Konstruktivität in einer konstruktiven Logik. Statt der von den Vertretern der konstruktiven Logik als einschränkend abgelehnten Präzisierung über den Rekursionsbegriff, schlägt Lorenzen vor, eine Aussage als konstruktiv zulässig zu betrachten, wenn sie *dialogisch-definit* ist. Über die logische Gültigkeit einer Aussage soll in einem Dialog zweier Spieler entschieden werden, der nach festgelegten Regeln geführt wird. Damit eine Aussage dialogisch-definit ist, muß jeder Dialog um sie nach endlich vielen Schritten enden, und es muß feststellbar sein, welcher der Spieler den Dialog gewonnen hat. Lorenzen beschreibt in diesem frühen Entwurf zwar, in welcher Weise zusammengesetzte Aussagen im Dialog behandelt werden sollen, gibt jedoch nicht an, welchen weiteren Regeln ein Dialog folgen muß.

Kuno Lorenz untersucht Dialogspiele 1961 in seiner Dissertation [12] und präzisiert die Dialogregeln. Er unterscheidet dabei strenge, effektive und klassische Dialogspiele. Die weitere Entwicklung der dialogischen Logik ist geprägt von dem Bemühen, ‘vernünftige’ Dialogregeln zu finden und eine Rechtfertigung dieser Regeln zu geben. In *Dialogische Logik* [15] von Paul Lorenzen und Kuno Lorenz sind mehrere Vorträge und Arbeiten wiedergegeben, die die Entwicklung der dialogischen Logik bis 1973 dokumentieren.

Von Seiten der Informatik steht nicht das Problem der Rechtfertigung der Regeln im Vordergrund. In der vorliegenden Arbeit wird ein Programm beschrieben, das es erlaubt, die Gültigkeit komplexer Aussagen im Dialogspiel zu untersuchen. Bereits um 1970 wurde an der Universität Austin, Texas ein Programm zur dialogischen Logik entwickelt, das Lorenzen während eines Gastaufenthalts an dieser Universität kennenlernte. Mit diesem Programm war es möglich, Dialogspiele der effektiven und klassischen Logik sowie modallogische Dialoge zu führen. Eine weitere Implementation aus dem Jahr 1970 stammt von Gerrit Haas [8]. Dieses Programm erzeugte selbständig Gewinnstrategien für effektive Dialoge, war jedoch auf die Aussagenlogik beschränkt.

Kapitel 2

Dialogische Logik

In der dialogischen Logik wird die Allgemeingültigkeit einer logischen Aussage in einem schematisierten Dialog zweier Spieler, dem *Proponenten* und dem *Opponenten*, untersucht. In einem solchen Dialog können die Dialogpartner Aussagen behaupten, Behauptungen des anderen Spielers angreifen und sich gegen Angriffe verteidigen.

Aussagen der dialogischen Logik werden aus Primformeln mit Hilfe der Junktoren *Konjunktion* '$\wedge$', *Disjunktion* '$\vee$', *Subjunktion* '$\rightarrow$', *Negation* '$\neg$' sowie dem *Allquantor* '$\forall$' und dem *Existenzquantor* '$\exists$' zusammengesetzt. Primformeln sind atomare Aussagen, die im Rahmen der formalen Logik nicht weiter untersucht werden.

Die Dialoge folgen festgelegten Dialogregeln, welche in *Partikelregeln* und *Rahmenregeln* unterschieden werden. Die Partikelregeln beschreiben, in welcher Weise eine logisch zusammengesetzte Aussage eines Spielers angegriffen werden kann, und wie sich der Spieler gegen einen solchen Angriff verteidigen kann. Dabei wird stets die Hauptverknüpfung der logischen Aussage betrachtet. Wenn im folgenden beispielsweise die Aussage „$A \vee B$“ betrachtet wird, so ist „$\vee$“ die Hauptverknüpfung, „A“ und „B“ sind Teilformeln, die ihrerseits zusammengesetzt oder atomar sein können.

Die Partikelregeln lauten:

Disjunktion : $A \vee B$

Wenn ein Spieler eine Disjunktion erfolgreich behaupten will, so muß er in der Lage sein, zumindest eine der beiden Teilaussagen erfolgreich behaupten zu können. Ein Angriff auf eine Disjunktion wird mit einem *schlichten Zweifel* ausgeführt, bei dem der angreifende Spieler selbst keine Aussage behauptet. Dieser Zweifel wird im Dialog durch „?“ symbolisiert. Der angegriffene Spieler kann sich verteidigen, indem er eine der beiden Teilaussagen wählt und diese behauptet.

Konjunktion : $A \wedge B$

Ein Spieler, der eine Konjunktion behauptet, muß beide Teilformeln erfolgreich behaup-

ten können. Der angreifende Spieler hat somit das Recht, eine der beiden Teilformeln auszuwählen, die der andere Spieler im weiteren Dialog verteidigen muß. Er verlangt das Setzen der linken Teilformel durch den Zweifel „$L?$“, das Setzen der rechten Teilformel durch „$R?$“. Zur Verteidigung auf einen solchen Angriff muß die geforderte Teilformel gesetzt werden.

Subjunktion : $A \to B$

Ein Spieler, der eine Subjunktion „$A \to B$“ behauptet, muß auf einen Angriff hin bereit sein, das Sukzedens „B“ zu verteidigen, wenn der Angreifer in der Lage ist, das Antezedens „A“ erfolgreich zu behaupten. Um einen Angriff auf die Subjunktion „$A \to B$“ durchzuführen, muß der Angreifer das Antezedens setzen: „$?, A$“. Der angegriffene Spieler verteidigt sich, indem er „B“ behauptet.

Negation : $\neg A$

Eine Negation wird angegriffen, indem die nicht negierte Teilformel behauptet wird: „$?, A$“. Auf einen Zweifel an einer Negation existiert keine Verteidigungsmöglichkeit. Der angegriffene Spieler kann stattdessen selbst eine Aussage des anderen Spielers, auch die beim Zweifel gesetzte Aussage „A“ angreifen.

Existenzaussage : $\exists x A(x)$

Wenn ein Spieler eine Existenzaussage „$\exists x A(x)$“ behauptet, so muß er in der Lage sein, ein Element n aus dem Variabilitätsbereich des Quantors zu wählen, für das er „$A(n)$“ erfolgreich verteidigen kann. Der Angriff auf eine Existenzaussage erfolgt durch einen schlichten Zweifel „$?$“. Der angegriffene Spieler wählt zur Verteidigung ein Element n und behauptet „$A(n)$“.

Allaussage : $\forall x A(x)$

Das Behaupten einer Allaussage erfordert, für jedes Element n des Variabilitätsbereichs die Aussage „$A(n)$“ behaupten zu können. Beim Zweifel an einer Allaussage wählt der Angreifer ein Element n des Variabilitätsbereichs: „$n?$“. Als Verteidigung auf diesen Angriff wird die Aussage „$A(n)$“ gesetzt.

In Abbildung 2.1 ist eine Übersicht über die Partikelregeln dargestellt.

Behauptung	**Angriff**		**Verteidigung**	
$A \vee B$	$?$		A	B
$A \wedge B$	$L?$	$R?$	A	B
$A \to B$	$?, A$		B	
$\neg A$	$?, A$			
$\exists x A(x)$	$?$		$A(n)$	
$\forall x A(x)$	$n?$		$A(n)$	

Abbildung 2.1: Übersicht über die Partikelregeln

Während die Partikelregeln mögliche Reaktionen auf eine Äußerung eines Spielers nennen, normieren die Rahmenregeln den Spielverlauf. Sie legen fest, welcher Spieler jeweils am Zug ist, welche der möglichen Äußerungen in einer bestimmten Dialogsituation erlaubt sind und wann ein Dialog gewonnen ist.

Eine Formulierung der Rahmenregeln lautet in Anlehnung an THIEL [18]:

1. Zu Beginn des Dialogs setzt der Opponent beliebig viele (evtl. keine) Hypothesen. Anschließend setzt der Proponent eine These. Nach dem Setzen der These ziehen beide Spieler abwechselnd, wobei der Opponent beginnt.

2. Jede Äußerung eines Spielers nach dem Setzen der These ist entweder ein Angriff auf eine zuvor gesetzte Behauptung des Dialogpartners oder eine Verteidigung auf einen Angriff des Dialogpartners.

3. Der Opponent darf stets nur auf die unmittelbar vorhergehende Äußerung des Proponenten reagieren: er kann eine im vorhergehenden Zug des Proponenten behauptete, zusammengesetzte Formel angreifen oder sich gegen einen im vorhergehenden Zug gesetzten Angriff verteidigen.

4. Jeder Angriff auf eine Behauptung darf vom Proponenten höchstens einmal beantwortet werden.

5. Der Proponent muß einen Angriff auf eine von ihm gesetzte Behauptung nicht sofort beantworten. Angriffe, zu denen noch keine Verteidigung vorgebracht wurde, heißen *offene Angriffe*. Möchte sich der Proponent gegen einen offenen Angriff verteidigen, so muß er den zuletzt gesetzten Angriff zuerst beantworten.

6. Primformeln dürfen nicht angegriffen werden. Der Opponent darf Primformeln stets behaupten. Der Proponent darf eine Primformel nur behaupten, wenn der Opponent sie zuvor behauptet hat. Der Proponent übernimmt also die Behauptung des Opponenten.

7. Ein Spieler hat einen Dialog gewonnen, wenn sein Dialogpartner keine Zugmöglichkeit mehr hat.

Bemerkungen zu den Rahmenregeln:

- Wenn der Opponent eine vom Proponenten gesetzte Negation „$\neg A$“ durch den Zweifel „$?, A$“ angreift, so müßte der Proponent diesen Angriff zuerst beantworten, bevor er sich gegen frühere, noch offene Angriffe verteidigt. Da es gemäß den Partikelregeln keine Verteidigung auf den Zweifel an einer Negation gibt, bewirkt die Rahmenregel 5, daß der Proponent im weiteren Dialog keinen dieser noch offenen Angriffe beantworten darf.

- Wenn der Proponent eine Subjunktion „$A \to B$“ des Opponenten mit „$?, A$“ bezweifelt, so hat der Opponent die Wahl, entweder diesen Zweifel durch Setzen von „B“ zu beantworten oder seinerseits die Aussage „A“, soweit diese zusammengesetzt ist, anzugreifen. Gemäß der

Regel 3 kann er nur eine dieser beiden Möglichkeiten wahrnehmen. Die andere Zugmöglichkeit verfällt.

Die Äußerungen oder Sprechakte der Dialogpartner werden in einem Dialogtableau notiert. Dabei stehen rechts die Sprechakte des Proponenten, links die Sprechakte des Opponenten. Zwischen diesen beiden Feldern werden die Zeilennummern vermerkt. Durch diese Nummern kann in Angriffs- oder Verteidigungssprechakten Bezug auf frühere Sprechakte des Dialogs genommen werden.

In Abbildung 2.2 ist ein Dialog um die These $\neg\neg(a \vee \neg a)$ dargestellt. In diesem Dialog greift der

Opponent		**Proponent**
	1	$\neg\neg(a \vee \neg a)$
$?, \neg(a \vee \neg a)$	2	$?, a \vee \neg a$
$?$	3	$\neg a$
$?, a$	4	$?2, a \vee \neg a$
$?$	5	a
$\%$	6	

Abbildung 2.2: Beispieldialog 1

Proponent die Behauptung $\neg(a \vee \neg a)$ des Opponenten aus Zeile 2 zum erstenmal unmittelbar nach dem Setzen der Behauptung und zum zweitenmal in Zeile 4 an. Allerdings hat sich die Dialogstellung zwischen diesen beiden Angriffen verändert, da der Opponent in Zeile 4 die Primformel a behauptet hat. Diese Primformel kann der Proponent in Zeile 5 übernehmen, um sich gegen den Zweifel an der Behauptung $a \vee \neg a$ zu verteidigen. Da der Opponent im Schritt 6 gemäß den Rahmenregel nur die Aussage des Proponenten in Zeile 5 angreifen darf, diese als Primformel aber nicht angegriffen werden kann, hat er keine Zugmöglichkeit mehr und verliert den Dialog.

In Abbildung 2.3 wird der Anfang eines Dialogs um die These $(a \vee b) \rightarrow \neg\neg(a \vee b)$ dargestellt. Nach

Opponent		**Proponent**
	1	$(a \vee b) \rightarrow \neg\neg(a \vee b)$
$?, a \vee b$	2	

Abbildung 2.3: Beispieldialog 2

dem Angriff der These durch den Opponent hat der Proponent die Wahl, entweder die Behauptung $a \vee b$ anzugreifen oder sich durch Setzen der Aussage $\neg\neg(a \vee b)$ zu verteidigen. Der Dialog spaltet sich also an dieser Stelle in zwei Teildialoge auf. Im weiteren Verlauf unterteilen sich diese weiter. Insgesamt sind die in den Abbildungen 2.4, 2.5 und 2.6 dargestellten Dialoge möglich.

In diesem Beispiel werden die letzten beiden Teildialoge durch den Proponenten gewonnen, den ersten Teildialog gewinnt der Opponent. Daraus ergibt sich, daß ein einzelner Dialog keine Entscheidung über die Allgemeingültigkeit einer Aussage liefert. Vielmehr müssen alle möglichen Dialogverläufe betrachtet werden. Übersichtlicher als in Tableaus können die möglichen Dialogverläufe

Opponent		**Proponent**
	1	$(a \vee b) \rightarrow \neg\neg(a \vee b)$
$?, a \vee b$	2	$\neg\neg(a \vee b)$
$?, \neg(a \vee b)$	3	$?, a \vee b$
$?$	4	%

Abbildung 2.4: Teildialog 1

Opponent		**Proponent**
	1	$(a \vee b) \rightarrow \neg\neg(a \vee b)$
$?, a \vee b$	2	$?$
a	3	$\neg\neg(a \vee b)$
$?, \neg(a \vee b)$	4	$?, a \vee b$
$?$	4	a
%	5	

Abbildung 2.5: Teildialog 2

Opponent		**Proponent**
	1	$(a \vee b) \rightarrow \neg\neg(a \vee b)$
$?, a \vee b$	2	$?$
b	3	$\neg\neg(a \vee b)$
$?, \neg(a \vee b)$	4	$?, a \vee b$
$?$	4	b
%	5	

Abbildung 2.6: Teildialog 3

in Dialogbäumen dargestellt werden. Der Dialogbaum zu den obigen Dialogen hat die in Abbildung 2.7 wiedergegebene Form.

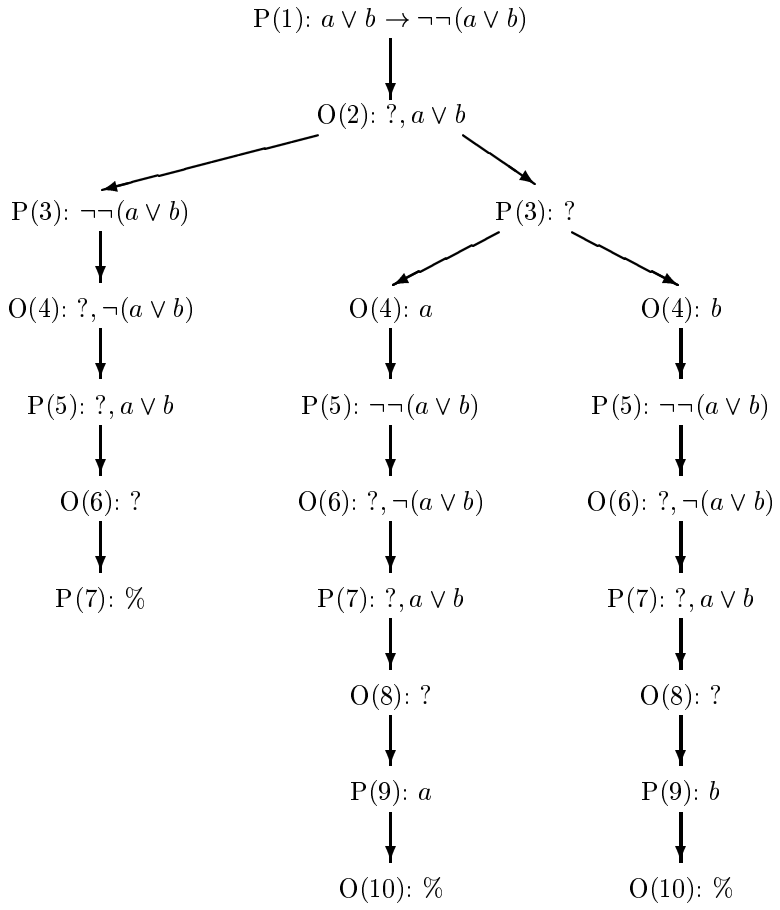

Abbildung 2.7: Dialogbaum

Dieser Dialogbaum verzweigt, wenn Wahlmöglichkeiten im Dialog bestehen. An den Blättern des Dialogbaums enden Dialoge. Nach Stekeler-Weithofer [17, S. 456] besitzt der Proponent eine *Spielstrategie* für eine These und bestimmte Hypothesen, wenn er durch geeignete Wahl seiner Züge, also durch die Wahl der richtigen Verzweigungen im Dialogbaum, stets verhindern kann, daß der Opponent einen Dialog gewinnt – unabhängig davon, welche seiner Wahlmöglichkeiten der Opponent realisiert. Spielstrategien können meist sehr leicht angegeben werden, da der Proponent durch die Rahmenregeln berechtigt ist, zusammengesetzte Aussagen des Opponenten mehrfach, also auch unendlich oft, anzugreifen und somit den Verlust eines Dialogs verhindern kann.

Der Proponent besitzt eine *Gewinnstrategie* für eine These und bestimmte Hypothesen, wenn er eine Spielstrategie kennt und darüber hinaus nachweisen kann, daß alle Dialoge, die gemäß der Spielstrategie geführt werden, nach endlich vielen Zügen enden. Da die Dialoge einer Spielstrategie nicht mit dem Gewinn des Opponenten enden dürfen, werden alle Dialoge nach einer Gewinnstrategie vom Proponenten gewonnen. Gewinnstrategien können häufig als spezielle Dialogbäume dargestellt werden, in denen alle Wahlmöglichkeiten des Opponenten enthalten sind

und die von den Wahlmöglichkeiten des Proponenten zu einem bestimmten Zeitpunkt jeweils nur eine nennen, welche zum Gewinn aller daraus entstehenden Dialoge führt. Nicht immer ist eine solche Charakterisierung einer Gewinnstrategie möglich. Der in Abbildung 2.8 wiedergegebene, stilisierte Dialog veranschaulicht eine Spielstrategie des Proponenten um die These $\forall n A(n)$ mit den Hypothesen $A(0)$ und $\forall n(A(n) \to A(n+1))$, wobei die Variabilitätsbereiche der Quantoren die natürlichen Zahlen sind.

Opponent		**Proponent**
$A(0)$	1	
$\forall n(A(n) \to A(n+1))$	2	$\forall n A(n)$
$t?2$	3	$0?2$
$A(0) \to A(1))$	4	$?, A(0)$
$A(1)$	5	$1?2$
$A(1) \to A(2))$	6	$?, A(1)$
$A(2)$	7	$2?2$
$A(2) \to A(3))$	8	$?, A(2)$
$\vdots$	$\vdots$	$\vdots$
$A(t-1)$	2t+1	$t-1?2$
$A(t-1) \to A(t))$	2t+2	$?, A(t-1)$
$A(t)$	2t+3	$A(t)$
%	2t+4	

Abbildung 2.8: Stilisierter Dialog

Unabhängig davon, welchen konkreten Wert der Opponent für t wählt, kann der Proponent den Gewinn des Dialogs nach endlich vielen Schritten erzwingen, da t eine endliche Zahl ist. Die Gewinnstrategie zu diesem Dialog läßt sich jedoch nicht als Baum darstellen.

Wenn der Proponent eine Gewinnstrategie für eine These A aufbauen kann, dann heißt A *allgemeingültig*. Wenn er eine Gewinnstrategie für den Dialog um die These A mit den Hypothesen $A_1, \ldots, A_n$ besitzt, dann heißt A *logische Folgerung* aus $A_1, \ldots, A_n$.

Die oben angegebene Formulierung der Rahmenregeln ist nur eine von zahlreichen möglichen Varianten. Verschiedene Formulierungen der Rahmenregeln können äquivalent zueinander in dem Sinne sein, daß sie zwar unterschiedliche Dialogverläufe zulassen, die Menge der Aussagen, deren Allgemeingültigkeit gezeigt werden kann, für die verschiedenen Formulierungen aber gleichbleibt. Andererseits können unterschiedliche Formulierungen der Rahmenregeln auch unterschiedliche Mengen allgemeingültiger Aussagen und somit unterschiedliche „Logiken“ ermöglichen.

Die Aussagen, deren Allgemeingültigkeit in Dialogen gemäß den oben angegebenen Rahmenregeln gezeigt werden kann, heißen effektiv allgemeingültig. Die Begründung der effektiven oder intuitionistischen Logik stand im Blickpunkt bei der Entwicklung der dialogischen Logik.

Eine Liberalisierung der Rahmenregeln für den Proponenten führt zu der Menge der klassisch gültigen Aussagen. Die angegebene Rahmenregel 5 wird dazu so abgeändert, daß der Proponent

einen beliebigen offenen Angriff beantworten darf, nicht nur den zuletzt vorgebrachten. Darüber hinaus wird dem Proponenten erlaubt, sich mehrfach gegen einen Angriff zu verteidigen. Die Menge der klassisch gültigen Formeln umfaßt die Menge der effektiv gültigen Formeln. Diese Inklusion ist echt, so daß durch diese Mengen unterschiedliche Begriffe der Allgemeingültigkeit definiert werden.

Besonders für das automatische Beweisen der Allgemeingültigkeit von Formeln ist es wünschenswert, nur endliche Dialogbäume zu betrachten. Auf diese Weise kann gewährleistet werden, daß das Verfahren stets nach endlicher Zeit stoppt und ersichtlich ist, ob eine Gewinnstrategie gefunden wurde oder nicht. Unendlich lange Dialoge können verhindert werden, indem die Anzahl der Angriffe, die der Proponent gegen eine Behauptung des Opponenten ausführen darf, beschränkt wird. Für die Regeln der klassischen Logik muß auch die Anzahl der Verteidigungen gegen einen Angriff beschränkt werden. Diese Beschränkungen verkleinern auch die Menge der im Dialog als allgemeingültig erweisbaren Aussagen. Wenn bei bestimmten Schranken keine Gewinnstrategie für eine These gefunden wird, so kann für größere Schranken eine solche Gewinnstrategie durchaus existieren.

Neben unendlich langen Dialogen können im Dialogbaum auch unendlich viele Verzweigungen auftreten. Dies ist der Fall bei Angriffen auf Allaussagen und Verteidigungen von Existenzaussagen, wenn die Variabilitätsbereiche der Quantoren unendliche Mengen sind. In beiden Situationen hat der Spieler unendlich viele Zugmöglichkeiten. Unendliche Verzweigungen in Dialogbäumen können verhindert werden, ohne die Menge der beweisbaren Formeln einzuschränken. Dabei wird unterschieden, welcher der beiden Spieler am Zug ist. Ist der Opponent am Zug, so wählt er aus den unendlich vielen Zugmöglichkeiten genau eine aus, nämlich den für den Proponenten schwierigsten Zug. Wenn der Proponent gegen diesen Zug gewinnen kann, so kann er gegen alle anderen Zugmöglichkeiten ebenfalls gewinnen. Für diesen schwierigsten Zug wählt der Opponent aus dem Variabilitätsbereich des Quantors ein Element, das bisher noch nicht im Dialog aufgetreten, also neu ist. Da der Dialog keine Aussagen über neue Elemente enthält, sind alle neuen Elemente gleichwertig.

Wenn der Proponent bei der Wahl eines Elements unendlich viele Möglichkeiten hat, so wird ein anderes Verfahren angewendet. Der Grundgedanke dabei ist, die Wahl des Elements so lange zu verzögern, bis nur noch endlich viele sinnvolle Wahlmöglichkeiten bestehen. Die Einzelheiten des Verfahrens sind im Kapitel 7 beschrieben.

Kapitel 3

Aufbau des Systems

Das System für dialogische Logik gliedert sich in vier Bereiche: *Benutzeroberfläche*, *Dialogspiel*, *Symbolverwaltung* und *Formelaufbau*, deren Verbindungen in Abbildung 3.1 veranschaulicht sind. Eine zentrale Rolle übernimmt die Dialogspielkomponente. Hier werden Dialogspiele um zuvor

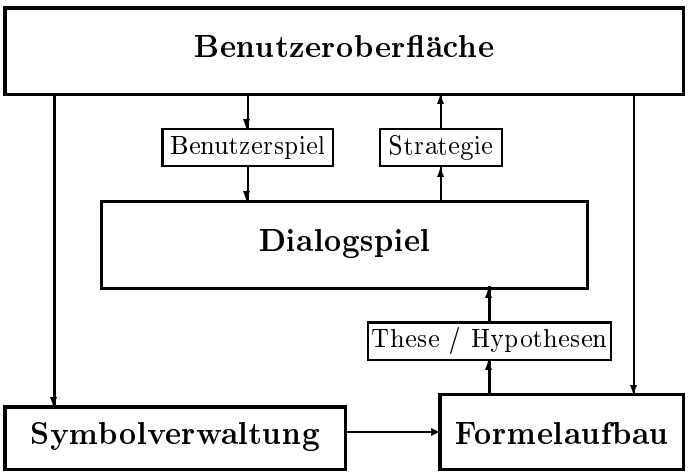

Abbildung 3.1: Aufbau des Systems

festgelegte Thesen und Hypothesen erzeugt. Das Ziel der Dialogspielkomponente ist es, eine Gewinnstrategie zu finden. Auf eine gefundene Strategie kann von der Benutzeroberfläche aus zugegriffen werden. Die Strategie kann in geeigneter Form ausgegeben werden, sie kann gespeichert und bei Bedarf geladen werden.

Über das Benutzerspiel kann der Benutzer mit der Dialogspielkomponente interagieren. Er übernimmt dabei die Rolle des Proponenten und versucht, im Dialog mit dem System, eine Gewinnstrategie zu finden.

Vor dem eigentlichen Dialogspiel müssen Formeln erzeugt werden, die in den Dialog als These und Hypothesen eingehen. Dies geschieht im Zusammenspiel der Benutzeroberfläche und der

Komponente zum Formelaufbau. Hier werden auch die Eigenschaften der Formeln festgelegt und Operationen, mit deren Hilfe Formeln bearbeitet werden können, zur Verfügung gestellt.

Formeln sind aufgebaut aus Symbolen, die durch den Benutzer definiert werden. Er kann dabei sowohl Variablen-, Konstanten-, Funktions- und Prädikatsymbole als auch logische Symbole, also Junktor- und Quantorsymbole, festlegen. Auf die definierten Symbole kann über die Symbolverwaltung zugegriffen werden. Da alle Symbole außer den logischen Symbolen Bezug auf Sorten nehmen, ist in diese Komponente auch die Sortenverwaltung integriert.

Die Benutzeroberfläche übernimmt die Kommunikation mit dem Benutzer. Sie ist in zwei Teilbereiche unterteilt: die *Textoberfläche* und die *graphische Oberfläche*. Der Aufbau der Benutzeroberfläche ist in Abbildung 3.2 dargestellt. Die Textoberfläche enthält den Parser für die Benutzerkom-

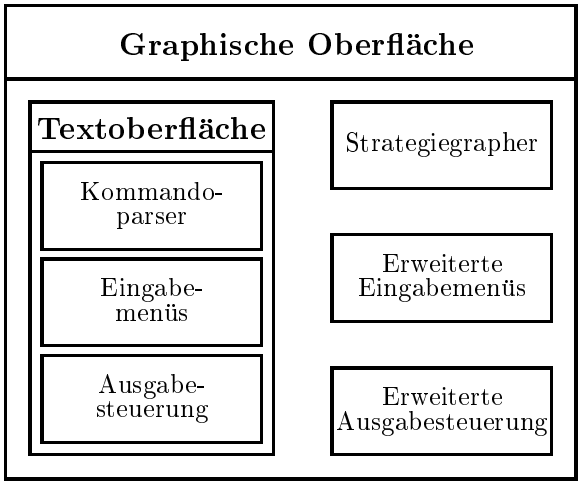

Abbildung 3.2: Aufbau der Benutzeroberfläche

mandos, die Ausgabesteuerung, welche Meldungen der einzelnen Komponenten aufbereitet und in geeigneter Weise ausgibt, sowie die Eingabemenüs für das Benutzerspiel. Die Textoberfläche ist in die graphische Oberfläche integriert, sie kann aber auch eigenständig benützt werden, wenn etwa die graphische Oberfläche nicht verwendet werden kann. Die graphische Oberfläche bietet über die Textoberfläche hinaus eine erweiterte Ausgabesteuerung, vor allem für die Ausgabe von Dialogspielen, sowie graphische Menüs für das Benutzerspiel und die Darstellung einer Gewinnstrategie als Baum.

Die Dialogspielkomponente umfaßt ebenfalls mehrere Teilbereiche, die in Abbildung 3.3 wiedergegeben sind. In der Komponente zum Baumaufbau wird der Spielbaum generiert. Aus dem Spielbaum werden Teilbäume, die nicht zum Gewinn aller Dialoge durch den Proponenten führen, eliminiert, um eine Gewinnstrategie zu finden. Die Sprechakte des Spielbaums werden von den Partikelregeln erzeugt. Die Rahmenregeln entscheiden, welche der erzeugten Sprechakte zu einem bestimmten Zeitpunkt im Dialog gesetzt werden dürfen. Da i. allg. mehrere Sprechakte einen Dialog fortsetzen können, werden diese nach lokalen Heuristiken geordnet, um eine Entscheidung im Dialog möglichst bald herbeizuführen. Die Daten des aktuellen Dialogs, also des bearbeite-

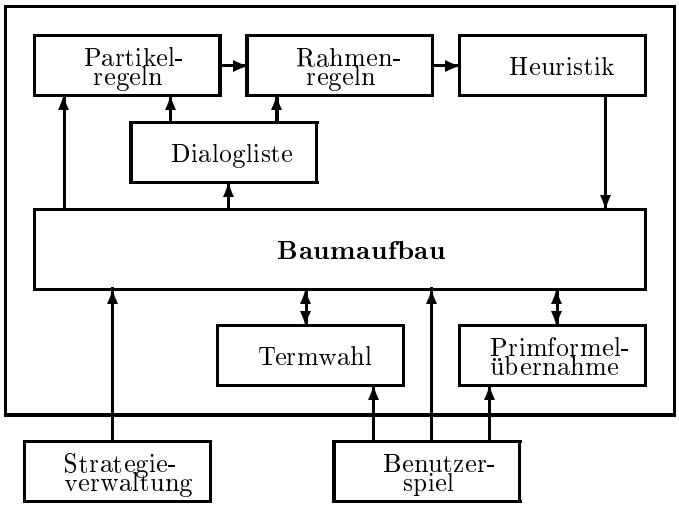

Abbildung 3.3: Aufbau der Dialogspielkomponente

ten Zweigs des Dialogbaums, werden in die Dialogliste eingetragen. Diese Daten werden von den Rahmenregeln benötigt, um über die Zulässigkeit von Sprechakten zu entscheiden.

Wenn im Dialog quantifizierte Aussagen auftreten, so müssen bei Angriffen oder Verteidigungen darauf Terme gewählt werden, die für die Quantorvariablen substituiert werden. Anstatt konkreter Terme werden dabei symbolische Terme, sogenannte *Termsymbole* eingesetzt. Erst wenn im Dialog Primformeln zu setzen sind, ist die konkrete Form der Termsymbole festzulegen. Dies geschieht in der Komponente zur Primformelübernahme.

Über das Benutzerspiel können sowohl der Baumaufbau selbst als auch die Termwahl und die Primformelübernahme gesteuert werden.

Alle Bestandteile des Systems außer der graphischen Oberfläche sind in der Programmiersprache Scheme implementiert. Um einen modularen Aufbau zu erreichen, wurde Yasos, eine objektorientierte Erweiterung aus der Scheme-Programmbibliothek SLIB [4] von T. Eigenschink, D. Love und A. Jaffer, verwendet. Yasos erlaubt es, die Komponenten des Systems als Objekte zu kapseln, auf welche nur über festgelegte Methoden zugegriffen werden kann. Mit den durch Yasos zur Verfügung gestellten Mitteln wurden fast alle Teile des Systems gestaltet. Ausnahmen bilden die Komponenten Baumaufbau und Primformelübernahme, die funktional implementiert wurden. In beiden Bereichen steht die Aufrufhierarchie von Funktionen im Vordergrund, die insbesondere beim Auftreten von Backtracking gezielt gesteuert werden muß.

Scheme bietet lediglich die Möglichkeit, Ausgaben in Form von Text darzustellen. Aus diesem Grund wurde für die graphische Benutzeroberfläche auf STk [7] zurückgegriffen. STk ist eine Implementation von Scheme durch Erick Gallesio, in die Tk, ein Toolkit von John K. Ousterhout [16] für graphische Benutzeroberflächen unter X11, integriert ist. Dieser Bereich des Systems ist unabhängig von den übrigen Teilen gestaltet. Um das System für dialogische Logik auch auf Rechnern ohne UNIX verwenden zu können, kann auf die Textoberfläche zurückgegriffen werden.

Kapitel 4

Sortenverwaltung

In der dialogischen Logik, wie auch in anderen Logikkalkülen, treten Sorten in Form von Variabilitätsbereichen der Quantoren auf. Der Variabilitätsbereich eines Quantors gibt an, welche Terme bei der Elimination des Quantors für die Quantorvariable eingesetzt werden dürfen. Im System für dialogische Logik erfüllen Sorten zwei Aufgaben:

- Sie erlauben, den korrekten Aufbau von Termen und Formeln zu überprüfen.
- Sie schränken bei der Wahl von Termen den Bereich, aus dem gewählt werden kann, ein.

Um die Korrektheit des Aufbaus von Formeln zu überprüfen, werden Konstanten- und Variablensymbolen Sorten zugeordnet. Funktionssymbolen werden eine geordnete Menge von Argumentsorten und eine Ergebnissorte zugeordnet. Auch zu Prädikatsymbolen müssen die Sorten der Argumente angegeben werden.

Bei der Termwahl, die bei Angriffen und Verteidigungen von quantifizierten Aussagen auftritt, werden Terme als Argumente in Prädikate eingesetzt. Dabei bestimmen nicht die Argumentsorten des Prädikatsymbols, aus welcher Menge von Termen gewählt werden kann. Vielmehr ist dies durch die Sorte der Quantorvariable, welche der Term ersetzen soll, festgelegt. Die Sorte der Quantorvariable spiegelt somit den Variabilitätsbereich des Quantors wider. Über die der Quantorvariable zugeordnete Sorte kann die Wahl von Termen stärker eingeschränkt werden, als nur durch die Argumentsorten der Prädikate. Voraussetzung dafür ist, daß auf den Sorten eine Hierarchie definiert ist. Da für die praktische Anwendung die Einschränkungen für die Termwahl und den Aufbau von Termen und Formeln nicht immer nötig sind, können sie durch die Verwendung einer speziellen Sorte, der Universalsorte, außer Kraft gesetzt werden.

Im System für dialogische Logik werden Sorten in einem Objekt verwaltet, das durch die Funktion

```
(make-sortenverwalter)
```

erzeugt wird. Beim Start des Systems wird ein solches Objekt mit dem Namen `sorte` generiert. Dieses Objekt verwaltet die durch den Benutzer deklarierten Sorten, eine Hierarchie auf diesen

Sorten sowie die Universalsorte. Die Universalsorte ist zu Beginn vorgegeben und trägt den Namen `universal`. Neue Sorten können über den Aufruf der Methode

`(deklariere sorte` *sortenname*`)`

deklariert werden. Der Parameter *sortenname* ist dabei eine Zeichenkette, die keine Sonderzeichen wie Leerstellen, Tabulatorzeichen, Klammern oder Kommata enthalten darf. Groß- und Kleinschreibung werden unterschieden, so daß „nat“ und „Nat“ als Namen verschiedener Sorten betrachtet werden. Ob eine Sorte deklariert ist, kann mit der Methode

`(ist-deklariert? sorte` *sortenname*`)`

überprüft werden. Das Ergebnis des Aufrufs ist ein Wahrheitswert.

Auf den deklarierten Sorten kann eine Ordnung definiert werden. Durch den Befehl

`(uebergeordnet sorte` $sortenname_1$ $sortenname_2$`)`

wird die Sorte mit dem Namen $sortenname_1$ der Sorte mit dem Namen $sortenname_2$ übergeordnet. Das bedeutet, daß überall dort, wo Terme der übergeordneten Sorte auftreten dürfen, auch Terme der untergeordneten Sorte verwendet werden können. Diese Ordnung der Sorten ist reflexiv, transitiv und antisymmetrisch, also eine Halbordnung. Die Relationen „übergeordnet“ und „untergeordnet“ sind nicht strikt, so daß jede Sorte sich selbst übergeordnet und untergeordnet ist. Durch die Aufrufe

`(uebergeordnet sorte` $sortenname_1$ $sortenname_2$`)`
`(uebergeordnet sorte` $sortenname_2$ $sortenname_1$`)`

werden zwei Sorten gleichgesetzt, d. h. Terme der einen Sorte können überall dort stehen, wo Terme der anderen Sorte auftreten dürfen. Sorten können nicht nur direkt gleichgesetzt werden. Wenn die Sorten `s1`, `s2`, `s3`, `s4`, `s5` und `s6` deklariert sind, so kann durch die Befehle

```
(uebergeordnet sorte "s1" "s2")
(uebergeordnet sorte "s2" "s1")
(uebergeordnet sorte "s2" "s3")
(uebergeordnet sorte "s3" "s4")
(uebergeordnet sorte "s2" "s5")
(uebergeordnet sorte "s5" "s6")
```

die in Abbildung 4.1 dargestellte Hierarchie aufgebaut werden, wobei die Pfeile jeweils von der übergeordneten zur untergeordneten Sorte zeigen. Durch den Aufruf

```
(uebergeordnet sorte "s4" "s1")
```

entsteht der in Abbildung 4.2 dargestellte Zyklus. Die Sorten `s1`, `s2`, `s3` und `s4` sind damit gleichgesetzt und jede dieser Sorten ist den Sorten `s5` und `s6` übergeordnet. Die Universalsorte nimmt in der Ordnung der Sorten eine besondere Stellung ein. Sie ist jeder Sorte übergeordnet und keiner anderen Sorte untergeordnet. Die Universalsorte kann durch den Aufruf der Methode

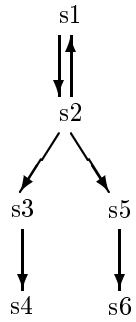

Abbildung 4.1: Sortenhierarchie 1

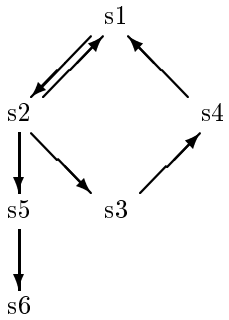

Abbildung 4.2: Sortenhierarchie 2

`(universal sorte` *sortenname*`)`

geändert werden. Voraussetzung dafür ist, daß die neue Universalsorte bereits deklariert ist und noch keiner anderen Sorte explizit über- oder untergeordnet wurde. Die ursprüngliche Universalsorte wird gelöscht, sie ist nach dem Aufruf nicht mehr deklariert. Die Methode

`(ist-universal? sorte` *sortenname*`)`

überprüft, ob die Sorte mit dem angegebenen Namen die Universalsorte ist und gibt einen entsprechenden Wahrheitswert zurück.

Die Beziehung zwischen zwei Sorten wird durch die Methode

`(ist-uebergeordnet? sorte` $sortenname_1$ $sortenname_2$`)`

überprüft. Dabei werden alle Sorten ermittelt, die der Sorte $sortenname_1$ untergeordnet sind. Das Ergebnis des Aufrufs gibt an, ob die Sorte $sortenname_2$ in den Untersorten von $sortenname_1$ enthalten ist. Die Menge der Sorten, die einer Sorte übergeordnet sind, wird von der Methode

`(uebersorten sorte` *sortenname*`)`

zurückgegeben, wobei *sortenname* die Sorte ist, deren Übersorten gebildet werden sollen. Deklarationen von Sorten können durch die Methode

`(undeklariere sorte` *sortenname*`)`

gelöscht werden. Dabei wird auch die Hierarchieliste angepaßt. Die Universalsorte kann auf diese Weise nicht gelöscht werden. Sie muß stattdessen mit Hilfe der Methode `universal` durch eine neue Universalsorte ersetzt werden. Der Aufruf

```
(loesche-deklarationen sorte)
```

löscht alle deklarierten Sorten und alle Ordnungsbeziehungen zwischen Sorten und setzt als Universalsorte die Sorte `universal`.

Über die Oberfläche sind nur die Methoden `deklariere`, `undeklariere`, `uebergeordnet` und `universal` für den Benutzer zugänglich. Die Methode `loesche-deklarationen` kann vom Benutzer nur beim Löschen aller Deklarationen, also der Sorten und der zum Formelaufbau benötigten Symbole verwendet werden.

Kapitel 5

Symbolverwaltung

Formeln, die im Dialogspiel auftreten, sind aufgebaut aus Variablen-, Konstanten-, Funktions- und Prädikatsymbolen sowie aus Junktor- und Quantorsymbolen. Bevor diese Symbole verwendet werden können, müssen sie deklariert werden. Dies ist aus zwei Gründen nötig:

1. Bei der Eingabe von Formeln durch den Benutzer muß die Struktur einer Formel erkannt werden. Dazu muß zu ermitteln sein, welche Rolle die Symbole in der Formel übernehmen, d. h., ob sie als Konstantensymbole, Funktionssymbole, ... auftreten. Darüber hinaus muß der Operatorenvorrang bekannt sein, der die Reihenfolge, in der Operationen anzuwenden sind, bestimmt.

2. Für die Manipulation von Formeln müssen Eigenschaften des Symbols, z. B. welche Sorten auftreten dürfen, festgelegt werden.

Nicht nur die Symbole zum Aufbau von Termen und Primformeln, auch die logischen Symbole, wie Junktoren und Quantoren, müssen deklariert werden. Es können beliebige Symbole deklariert werden, deren Verwendung im Dialogspiel durch geeignete Partikelregeln festgelegt wird. Auf diese Weise kann die im Dialogspiel verwendete Sprache durch Angabe von logischen Symbolen und Partikelregeln den Erfordernissen angepaßt werden.

Zu jedem Symboltyp existiert ein Objekt, in dem die Deklarationen verwaltet werden. Diese Objekte unterscheiden sich in den Daten, die verwaltet werden und in den Methoden, mit denen auf diese Daten zugegriffen werden kann. Alle Objekte besitzen jedoch die Methoden `deklariere`, `ist-deklariert?`, `undeklariere` und `loesche-deklarationen`, wie sie auch im Objekt zur Sortenverwaltung verfügbar sind. Damit der Typ eines Symbols eindeutig festgestellt werden kann, lehnen alle Objekte eine erneute Deklaration eines Symbols, das bereits in einem dieser Objekte deklariert wurde, ab. Die weiteren Eigenschaften der Objekte sind in den Abschnitten über die einzelnen Symboltypen beschrieben.

Durch die Benutzeroberfläche werden Kommandos zur Verfügung gestellt, um Variablen-, Konstanten-, Funktions- und Prädikatsymbole zu deklarieren und deklarierte Symbole zu löschen. Die

Deklarationen von logischen Symbolen können von der Benutzeroberfläche aus nicht manipuliert werden. Vielmehr sollen diese Deklarationen zusammen mit den Partikelregeln aus einer Datei geladen werden.

5.1 Variablensymbole

Variablensymbole werden von einem Objekt mit dem Namen `variablensymbol` verwaltet. Bei der Deklaration eines Variablensymbols muß eine Sorte angegeben werden:

`(deklariere variablensymbol` *symbolname sortenname*`).`

Der angegebene Sortenname muß bereits deklariert sein. Die Sorte eines Variablensymbols kann durch den Befehl

`(zugehoerige-sorte variablensymbol` *symbolname*`)`

bestimmt werden.

5.2 Konstantensymbole

Der Verwalter von Konstantensymbolen trägt den Namen `konstantensymbol`. Die Deklaration von Konstantensymbolen erfolgt analog der Deklaration von Variablensymbolen:

`(deklariere konstantensymbol` *symbolname sortenname*`).`

Im Unterschied zu Variablensymbolen können auch Mengen von Konstanten deklariert werden. Auf diese Weise ist es möglich, beispielsweise natürliche Zahlen in Formeln zu verwenden, ohne die benötigten Zahlen einzeln als Konstantensymbole deklarieren zu müssen. Konstantenmengen werden durch den Aufruf

`(deklariere-menge konstantensymbol` *konstantenmenge sortenname*`)`

deklariert. Der angegebenen Sorte darf noch keine andere Konstantenmenge zugeordnet sein. Für *konstantenmenge* können folgende Werte eingesetzt werden: „nat-symbole“, „nat0-symbole“, „integer-symbole“, „rat-symbole“, „real-symbole“ und „komplex-symbole“. Jede dieser Mengen umfaßt unendlich viele Symbole[1]:

[1] Die Symbolmengen wurden mit Hilfe der Scheme-Prädikate `number?`, `complex?`, `real?`, `rational?` und `integer?` implementiert, die über die genannten Darstellungen hinaus weitere Zahlenformate erlauben. Diese Formate sind im *Revised*^{4} *Report on the Algorithmic Language Scheme* [3] angegeben. STk, das für die Implementation der graphischen Oberfläche verwendet wurde, unterstützt in der Version 3.0 die Prädikate `complex?` und `rational?` nicht. Bei Verwendung der graphischen Oberfläche sind dadurch die entsprechenden Konstantenmengen mit der Konstantenmenge „real-symbole“ identisch.

nat-symbole : Diese Symbolmenge enthält Konstantensymbole, die die Form natürlicher Zahlen haben: „1“, „2“, „3“, ...

nat0-symbole : Die Symbolmenge „nat0-symbole“ umfaßt das Konstantensymbol „0“ sowie alle Konstantensymbole der Menge „nat-symbole“.

integer-symbole : Hierin enthalten sind Konstantensymbole, die die Form von ganzen Zahlen haben: ..., „-2“, „-1“, „0“, „1“, „2“, ...

rat-symbole : In der Menge „rat-symbole“ sind Symbole enthalten, die als rationale Zahlen interpretiert werden können. Die Elemente haben die Form eines Dezimalbruchs (z. B. „0.12“) oder eines Bruchs (z. B. „1/2“).

real-symbole : Die Elemente dieser Symbolmenge haben die Form von reellen Zahlen. Sie können beispielsweise in der Form „1.0e-2“ dargestellt werden.

komplex-symbole : In dieser Menge sind zusätzlich zu den Konstantensymbole der Menge „real-symbole“ Elemente der Form „1.2+2.1i“ enthalten.

Obwohl die Elemente der Symbolmengen aussehen wie natürliche Zahlen, rationale Zahlen, usw., sind sie lediglich Symbole, also Namen ohne jede weitere Bedeutung. Außer der Sorte der Symbole sind keine weiteren Eigenschaften bekannt.

Konstantensymbole, die durch eine Symbolmenge deklariert wurden, können durch den Aufruf

`(undeklariere-menge konstantensymbol` *symbolmenge*`)`

wieder gelöscht werden. Die Methode

`(ist-menge-deklariert? konstantensymbol` *symbolmenge*`)`

ermittelt, ob eine Symbolmenge bereits deklariert wurde.

Das Ergebnis der Methode `ist-deklariert?` gibt Auskunft darüber, ob ein Konstantensymbol deklariert ist. Dabei ist unerheblich, ob das Symbol einzeln oder durch eine Symbolmenge deklariert wurde.

Die Methode `zugehoerige-sorte` ermittelt wie bei Variablensymbolen die Sorte, die einem Konstantensymbol zugeordnet ist. Ein Konstantensymbol, das mittels einer Symbolmenge eingeführt wurde, kann i. allg. in mehreren Symbolmengen enthalten sein, weshalb einem solchen Symbol mehrere Sorten zukommen können. Es wird dann eine in der Sortenhierarchie *minimale* Sorte bestimmt. Das bedeutet, es gibt keine andere Sorte, die dem Konstantensymbol zukommt und die dieser minimalen Sorte untergeordnet ist. Die minimale Sorte ist nicht immer eindeutig bestimmt, wie in folgendem Beispiel verdeutlicht wird:

Seien die Sorten „nat“, „rat“ und „real“ deklariert, wobei die Sorte „rat“ der Sorte „nat“ übergeordnet wurde. Seien weiterhin die Konstantensymbole der Symbolmengen „nat-symbole“, „rat-symbole“ und „real-symbole“ mit den entsprechenden Sorten deklariert. Das Konstantensymbol „1“ ist in diesem Fall in allen drei Symbolmengen enthalten, weshalb ihm alle drei Sorten

zukommen. Von diesen Sorten sind „nat“ und „real“ minimal, da ihnen keine Sorten untergeordnet sind, die dem Konstantensymbol ebenfalls zukommen. Der Aufruf der Methode

```
(zugehoerige-sorte konstantensymbol "1")
```

liefert eine dieser beiden Sorten. Wenn in der Sortenhierarchie zusätzlich die Sorte „real“ der Sorte „rat“ übergeordnet wird, dann ist die Sorte „nat“ die einzige minimale Sorte, die dem Konstantensymbol zukommt.

5.3 Funktionssymbole

Funktionssymbole werden von dem Objekt `funktionssymbol` verwaltet. Ein Funktionssymbol wird durch den Befehl

`(deklariere funktionssymbol` *symbolname argumentsorten ergebnissorte position bindung repraesentation*`)`

deklariert. Für *argumentsorten* ist eine Liste von Sorten mit mindestens einem Element anzugeben. Durch die Argumentsorten wird die Anzahl und Art der Terme, auf die das Funktionssymbol angewendet werden kann, festgelegt. Der Parameter *ergebnissorte* ist der Name einer deklarierten Sorte, die einem mit dem Funktionssymbol aufgebauten Term zukommen soll.

Funktionssymbole können in Termen als Infix- oder Präfixsymbole dargestellt werden. Der Parameter *position* bestimmt, welche der beiden Notationen verwendet werden soll. Er wird in Form der Zeichenketten „infix“ oder „praefix“ angegeben.

Die Bindung bestimmt die Reihenfolge, in der Operatoren angewendet werden. Sie wird benötigt bei der Eingabe von Formeln, um deren Struktur festzustellen, und bei der Ausgabe, um unnötige Klammern zu vermeiden. Bindungen werden in Form einer Liste von bis zu zwei Zahlen angegeben. Diese Zahlen müssen positiv und kleiner als die maximale Bindung 1000 sein. Je größer sie sind, desto höher ist die Priorität bei der Anwendung des Symbols. Für Infixfunktionssymbole muß die Liste zwei Zahlenwerte enthalten: die linksseitige und die rechtsseitige Bindungsstärke. Durch diese beiden Werte kann die Assoziativität eines Symbols festgelegt werden. Für linksassoziative Funktionssymbole ist die linksseitige Bindung kleiner als die rechtsseitige. Wenn beispielsweise das Funktionssymbol „+“ als Infixsymbol und mit den Bindungen `'(2 3)` deklariert wurde, so wird der Term $a + b + c$ als $(a + b) + c$ interpretiert. Die meisten mathematischen Operationen sind linksassoziativ. Eine Ausnahme bildet die Exponentiation. Wenn das Symbol „^“ für die Exponentiation deklariert wurde, so soll der Term $a\hat{}b\hat{}c$ als $a\hat{}(b\hat{}c)$ interpretiert werden. Um dies zu erreichen, muß die linksseitige Bindung größer als die rechtsseitige sein. Präfixfunktionssymbole benötigen keine Bindungen, da die Argumente der Funktion geklammert werden, wodurch die Reihenfolge der Operatorenanwendung festgelegt ist. Deshalb ist hier eine leere Liste als Bindung anzugeben.

Der Parameter *repraesentation* ist eine Zeichenkette, die für die Ausgabe des Symbols beim Drucken von Formeln verwendet wird. Diese Zeichenkette kann beliebig gewählt werden und unterliegt nicht den Beschränkungen für Symbolnamen. So kann beispielsweise die Zeichenkette „(+)“ als Repräsentation eines Symbols verwendet werden, nicht aber als Name.

Die Methoden

```
(repraesentation funktionssymbol symbolname)
(argumentsorten funktionssymbol symbolname)
(ergebnissorte funktionssymbol symbolname)
(bindung funktionssymbol symbolname)
```

geben die Daten eines Symbols in der bei der Deklaration verwendeten Form zurück. Die Methoden

```
(ist-praefix? funktionssymbol symbolname)
(ist-infix? funktionssymbol symbolname)
```

liefern Wahrheitswerte entsprechend der angegebenen Position eines Funktionssymbols.

5.4 Prädikatsymbole

Das Objekt zur Verwaltung von Prädikatsymbolen trägt den Namen `praedikatsymbol`. Bei der Deklaration müssen ein Symbolname, Argumentsorten, die Position und eine Repräsentation angegeben werden:

```
(deklariere praedikatsymbol symbolname argumentsorten position repraesentation).
```

Die Position und die Repräsentation können wie bei Funktionssymbolen gewählt werden. Der Parameter *argumentsorten* kann im Unterschied zu Funktionssymbolen auch eine leere Liste enthalten. Nullstellige Prädikatsymbole können für Primformeln ohne Argumente verwendet werden.

Die Methoden

```
(repraesentation praedikatsymbol symbolname)
(argumentsorten praedikatsymbol symbolname)
(ist-praefix? praedikatsymbol symbolname)
(ist-infix? praedikatsymbol symbolname)
```

stehen wie bei Funktionssymbolen zur Verfügung.

5.5 Junktorsymbole

Junktorsymbole werden von dem Objekt `junktorsymbol` verwaltet. Die Deklaration erfolgt durch den Aufruf

`(deklariere junktorsymbol` *symbolname stelligkeit position bindung repraesentation*`).`

Für die Stelligkeit sind die Werte 1 oder 2 erlaubt. Einstellige Junktorsymbole müssen die Position „praefix“ haben, zweistellige Junktorsymbole sind als Infixsymbole zu deklarieren. Wie bei Funktionsymbolen ist die Bindung eines Infixjunktorsymbols eine Liste mit zwei Zahlenwerten. Bei Präfixjunktorsymbolen hingegen ist keine leere Liste, sondern eine Liste mit einem Element anzugeben, da die Argumente nicht geklammert werden. Wenn beispielsweise das Präfixjunktorsymbol „-“ und das Infixjunktorsymbol „&&“ deklariert sind, so entscheidet der zu „-“ angegebene Bindungswert, ob eine Formel $-a \,\&\&\, b$ als $-(a \,\&\&\, b)$ oder $(-a) \,\&\&\, b$ intepretiert wird. Ist der Wert kleiner als die linksseitige Bindung von „&&“, so wird die erste Form gewählt, ist der Wert größer, so trifft die zweite Interpretation zu.

Mittels der Methoden

`(repraesentation junktorsymbol` *symbolname*`)`
`(bindung junktorsymbol` *symbolname*`)`
`(ist-praefix? junktorsymbol` *symbolname*`)`
`(ist-infix? junktorsymbol` *symbolname*`)`
`(ist-einstellig? junktorsymbol` *symbolname*`)`
`(ist-zweistellig? junktorsymbol` *symbolname*`)`

können die Eigenschaften eines Junktorsymbols ermittelt werden.

5.6 Quantorsymbole

Das Objekt `quantorsymbol` verwaltet die Deklarationen von Quantorsymbolen. Bei der Deklaration eines Quantorsymbols müssen eine Bindung und eine Repräsentation angegeben werden:

`(deklariere quantorsymbol` *symbolname bindung repraesentation*`).`

Für die Bindung ist eine Liste mit einer Zahl, für die Repräsentation wie bei anderen Symbolen eine beliebige Zeichenkette anzugeben. Die Methoden `repraesentation` und `bindung` liefern die Angaben der Deklaration zurück.

Kapitel 6

Formelaufbau

6.1 Erzeugung von Formelobjekten

Im Dialogspiel werden zu Beginn logische Formeln als These und Hypothesen gesetzt. Diese Formeln werden im Verlauf des Dialogs in Teilformeln zerlegt, bis schließlich auf der Ebene der Primformeln eine Entscheidung herbeigeführt wird.

Formeln sind im System für dialogische Logik als Objekte implementiert, die selbst wieder Objekte enthalten können. Es wird unterschieden zwischen *Term-* und *Formelobjekten.* Termobjekte sind *Variablenterm-*, *Konstantenterm-* und *Funktionstermobjekte*, von denen die ersten beiden atomare Objekte sind, die keine anderen Objekte enthalten. Sie werden mit Hilfe der Funktionsaufrufe

`(make-variablenterm` *symbolname*`)`
`(make-konstantenterm` *symbolname*`)`

erzeugt. Als Symbolname muß ein bereits deklariertes Variablen- bzw. Konstantensymbol angegeben werden. Funktionstermobjekte verknüpfen über ein Funktionssymbol mehrere Termobjekte zu einem einzigen Objekt. Dies geschieht durch den Aufruf

`(make-funktionsterm` *symbolname* $termobjekt_1$ ... $termobjekt_n$`)`.

Für *symbolname* muß ein zuvor deklariertes Funktionssymbol angegeben werden. Die Parameter $termobjekt_1, \ldots,$ $termobjekt_n$ enthalten die Argumente, die durch das Funktionssymbol verknüpft werden. Dabei muß die Anzahl der Argumente mit der Stelligkeit des Funktionssymbols übereinstimmen. Wenn beispielsweise das zweistellige Funktionssymbol „f“ und die Variablensymbole „x“ und „y“ deklariert wurden, dann erzeugt

```
(make-funktionsterm "f"
   (make-variablenterm "x")
   (make-variablenterm "y"))
```

ein neues Termobjekt. Jedem Termobjekt ist eine Sorte zugeordnet. Bei Konstanten- und Variablentermobjekten ist dies die Sorte des zugehörigen Symbols, bei Funktionstermobjekten die Ergebnissorte des Funktionssymbols. Wenn ein Funktionstermobjekt erzeugt werden soll, so wird überprüft, ob die Sorten der Argumente zu den deklarierten Argumentsorten des Funktionssymbols passen, also ob sie Untersorten der jeweiligen Argumentsorten sind.

Außer den genannten Termobjekten existieren *Termsymbolobjekte*. Diese werden bei der Strategiebildung erzeugt. Ihre Verwendung wird im Abschnitt 7.7.2 besprochen.

Bei Formelobjekten werden *Primformelobjekte*, *Junktorformelobjekte* und *Quantorformelobjekte* unterschieden. Der Aufruf

`(make-primformel` *symbolname termobjekt*$_1$ ... *termobjekt*$_n$`)`

verknüpft die Objekte *termobjekt*$_1$, ..., *termobjekt*$_n$ über das Prädikatsymbol *symbolname* zu einem Primformelobjekt. Die Stelligkeit und die Argumentsorten des Prädikatsymbols müssen wie bei Funktionstermobjekten berücksichtigt werden. Wenn das Prädikatsymbol mit einer leeren Argumentsortenliste deklariert wurde, ist im Funktionsaufruf lediglich der Symbolname anzugeben.

Junktorformelobjekte werden je nach Stelligkeit des Junktorsymbols durch die Funktionen

`(make-junktorformel` *symbolname formelobjekt*`)`
`(make-junktorformel` *symbolname formelobjekt*$_1$ *formelobjekt*$_2$`)`

generiert. Dabei können als Argumente Primformelobjekte oder zusammengesetzte Formelobjekte angegeben werden.

Quantorformelobjekte werden aus einem Quantorsymbol, einem Variablensymbol und einem Formelobjekt mit Hilfe des Aufrufs

`(make-quantorformel` *quantorsymbol variablensymbol rumpfformelobjekt*`)`

erzeugt. Dabei ist es unerheblich, ob das Variablensymbol in der Rumpfformel enthalten ist und ob es frei oder gebunden vorkommt. Wenn beispielsweise die Variablensymbole „x“ und „y“, das zweistellige Infixprädikatsymbol „<“ und das Quantorsymbol „ALL“ deklariert wurden, so erzeugt der Aufruf

```
(make-quantorformel "ALL" "x"
   (make-primformel "<"
     (make-variablenterm "x")
     (make-variablenterm "y")))
```

ein neues Formelobjekt, das für die Formel $\forall x : x < y$ steht.

Neben den beschriebenen Formelobjekten existiert ein *Leerformelobjekt*. Dieses Objekt wird im Dialogspiel bei Sprechakten verwendet, in denen keine Behauptung gesetzt wird. Dazu wird nur ein einziges Leerformelobjekt benötigt, das den Namen `leerformel` trägt. Neue Objekte dieses

Typs können nicht erzeugt werden. Das Leerformelobjekt steht für eine Formel, die weder eine Primformel noch eine zusammengesetzte Formel ist.

Der Benutzer erzeugt Formelobjekte nur bei der Angabe der These und der Hypothesen eines Dialogs. Er verwendet dabei keinen der beschriebenen Funktionsaufrufe, sondern gibt eine aus deklarierten Symbolen zusammengesetzte Zeichenkette an, die die Formel beschreibt. Aus dieser Darstellung der Formel wird mit Hilfe der genannten Funktionen von der Benutzeroberfläche ein Formelobjekt generiert.

6.2 Methoden von Term- und Formelobjekten

Formelobjekte müssen im Dialog beispielsweise in Teilformeln zerlegt werden, sie müssen modifiziert oder ausgegeben werden. Dazu stellen Formel- und Termobjekt zahlreiche Methoden zur Verfügung, die im folgenden beschrieben werden.

6.2.1 Methoden zur Klassifikation von Objekten

Alle Term- und Formelobjekte besitzen Methoden, die es erlauben, ihren Typ festzustellen:

(`term?` *objekt*)
(`formel?` *objekt*)
(`konstantenterm?` *objekt*)
(`variablenterm?` *objekt*)
(`atomarer-term?` *objekt*)
(`formel?` *objekt*)
(`primformel?` *objekt*)
(`junktorformel?` *objekt*)
(`quantorformel?` *objekt*)
(`termsymbol?` *objekt*).

Jede dieser Methoden liefert einen der Wahrheitswerte `#t` oder `#f`. Die Methode `atomarer-term?` ergibt `#t`, wenn sie auf Konstanten- oder Variablentermobjekte angewendet wird, für alle anderen Objekte der Wert `#f`. Für alle der hier besprochenen Term- und Formelobjekte liefert die Methode `termsymbol?` den Wert `#f`. Auf Termsymbolobjekte angewendet ergibt sie den Wert `#t`.

6.2.2 Selektormethoden

Über Selektormethoden kann auf die Elemente, aus denen Term- und Formelobjekte zusammengesetzt sind, zugegriffen werden. Die Methode

(`hauptsymbol` *objekt*)

ermittelt für Term- und Primformelobjekte das Symbol, mit dessen Hilfe das Objekt aufgebaut wurde. Für Konstanten- und Variablentermobjekte ist dies das zugehörige Konstanten- bzw. Variablensymbol, für Funktionstermobjekte das Funktionssymbol und für Primformelobjekte das Prädikatsymbol. Junktor- und Quantorformelobjekte besitzen kein Hauptsymbol, weshalb die Methode

bei diesen Objekttypen den Wert `undef` liefert. Stattdessen besitzen Junktorformel- und Quantorformelobjekte eine Hauptverknüpfung, nämlich das zugehörige Junktor- bzw. Quantorsymbol. Dieses Symbol wird von der Methode

`(hauptverknuepfung` *objekt*`)`

ermittelt. Auf Term- oder Primformelobjekte angewendet ergibt sie den Wert `undef`.

Die Argumente von Funktionsterm- und Primformelobjekten werden durch den Aufruf

`(argumente` *objekt*`)`

in Form einer Liste von Termobjekten zurückgegeben. Für alle anderen Objekte sowie für Primformelobjekte ohne Argumente ist das Ergebnis der Methode eine leere Liste.

Bei allen Selektorfunktionen, die Objekte oder Listen von Objekten als Werte liefern, sind diese identisch (im Sinne des Scheme-Prädikats `eq?`) mit den bei der Erzeugung des Term- oder Formelobjekts angegebenen Argumentobjekten. Das bedeutet, daß Modifikationen an den von den Selektormethoden zurückgegebenen Objekten sich auch auf das zusammengesetzte Objekt auswirken können. Aus diesem Grund erzeugen die weiter unten erläuterten Modifikatormethoden Kopien von Objekten, welche anschließend modifiziert werden.

Die Komponenten von Junktorformeln können mit Hilfe der Methoden

`(unterformel` *objekt*`)`
`(linke-unterformel` *objekt*`)`
`(rechte-unterformel` *objekt*`)`

selektiert werden. Die Methode `unterformel` liefert für Formeln mit einstelligem Junktorsymbol das Formelobjekt, auf welches das Junktorsymbol angewendet wurde. Analog geben die Methoden `linke-unterformel` und `rechte-unterformel` für Objekte mit zweistelligem Junktorsymbol das erste bzw. zweite der bei der Erzeugung angegebenen Formelobjekte zurück. Wenn die Methoden auf andere als die genannten Objekte angewendet werden, so geben sie den Wert `undef` zurück.

Die Quantorvariable und der Rumpf eines Quantorformelobjekts können durch die Methoden

`(quantorvariable` *objekt*`)`
`(rumpf` *objekt*`)`

bestimmt werden. Die erste Methode gibt dabei ein Variablensymbol, die zweite ein Formelobjekt zurück. Für andere Objekte als Quantorformelobjekte liefern sie den Wert `undef`. Der Aufruf

`(liste-freier-variablen` *objekt*`)`

kann auf alle Term- und Formelobjekte angewendet werden. Er gibt eine Liste zurück, die alle Variablensymbole, die in dem Objekt frei vorkommen, genau einmal enthält. Für Konstantentermobjekte ist dies eine leere Liste, für Variablentermobjekte enthält die Liste das Variablensymbol als einziges Element. Bei Funktionsterm-, Primformel- und Junktorformelobjekten wird die Vereinigung der freien Variablen der Argumente bzw. Unterformeln gebildet. Für Quantorformelobjekte wird diese Liste ermittelt, indem aus der Liste der freien Variablen des Rumpfs das als Quantorvariable verwendete Variablensymbol entfernt wird, soweit es enthalten ist.

6.2.3 Modifikatormethoden

Die Methode

`(substituiere-variable` *objekt variablensymbol termobjekt*`)`

erlaubt es, alle freien Vorkommen eines Variablensymbols in einem Term- oder Formelobjekt durch das Objekt *termobjekt* zu ersetzen. Der Rückgabewert dieser Methode ist ein geändertes Objekt. Es wird verhindert, daß sich Modifikationen auch auf andere Objekte auswirken, indem bei Modifikationen stets Kopien der Objekte erzeugt werden. Analog ersetzt die Methode

`(substituiere-termsymbol` *objekt termsymbol termobjekt*`)`

Termsymbole durch Termobjekte. Im Unterschied zu Variablensymbolen wird nicht zwischen freien und gebundenen Vorkommen von Termsymbolen unterschieden. Vielmehr werden alle Vorkommen des Termsymbols durch das Termobjekt ersetzt. Termsymbole werden im Verlauf eines Dialogspiels in Formeln eingetragen. Wenn im Dialog eine Gewinnstrategie gefunden wurde, so können die Sprechakte des Strategiebaums Termsymbole enthalten. Die Methode `substituiere-termsymbol` ersetzt diese am Ende des Dialogspiels durch die Termobjekte, für die sie stehen.

6.2.4 Methoden für die Ausgabe von Objekten

Um Dialoge und Strategien ausgeben zu können, müssen für Term- und Formelobjekte Zeichenketten erzeugt werden, welche die Terme oder Formeln, für die die Objekte stehen, wiedergeben. Jedes Objekt stellt deshalb eine Methode

`(drucke` *objekt modus*`)`

zur Verfügung, die eine solche Zeichenkette erzeugt. Der Modus bestimmt, in welcher Weise ein Term oder eine Formel ausgegeben wird. Es sind die Grundmodi `normalmodus` und `einfachmodus` möglich, die mit dem Nebenmodus `klammermodus` kombiniert werden können. Ein Druckmodus wird mit Hilfe der Funktionsaufrufe

`(erstelle-modus` *grundmodus*`)`
`(erstelle-modus` *grundmodus nebenmodus*`)`

erzeugt. Die Funktion

`(druckvariante?` *druckmodus grund-/nebenmodus*`)`

stellt fest, ob ein Grund- oder Nebenmodus in dem Druckmodus enthalten ist. Wenn der Grundmodus `einfachmodus` gewählt ist, dann wird zur Ausgabe von Funktions-, Junktor-, Prädikat- und Quantorsymbolen in Formeln der jeweilige Symbolname verwendet. Im Grundmodus `normalmodus` wird statt des Symbolnamens die bei der Deklaration angegebene Repräsentation des Symbols benützt. Ist der Nebenmodus `klammermodus` nicht gewählt, so werden überflüssige Klammern in der Darstellung einer Formel vermieden. Ob Klammern notwendig sind, wird auf Grund von Bindungen entschieden. Jedes Objekt gibt beim Aufruf der Methode

`(bindung` *objekt*`)`

seine Bindung zurück. Die Bindung eines Objekts ist die zum Hauptsymbol bzw. zur Hauptverknüpfung des Objekts gehörende Bindung. Die Bindungen bleiben unberücksichtigt, wenn der Klammermodus gewählt ist. Formeln und Terme werden in diesem Fall vollständig geklammert wiedergegeben.

6.2.5 Methoden für die Unifikation

Im Dialogspiel dürfen nach den üblichen Rahmenregeln Primformeln vom Proponenten nur gesetzt werden, indem er sie vom Opponenten übernimmt. Dazu werden die vom Opponenten gesetzte und die vom Proponenten zu setzende Primformel *unifiziert*. Da die Primformeln Termsymbole enthalten können, die für noch unbestimmte Terme stehen, müssen sie für die Übernahme nicht vollkommen, sondern nur 'bis auf die Termsymbole' übereinstimmen. Das Verfahren der Unifikation ist im Abschnitt 7.7.4 beschrieben.

Termobjekte stellen für die Unifikation die Methode `unifiziere-term` zur Verfügung, Primformelobjekte besitzen die Methode `unifiziere-formel`. Wenn eine Unifikation erfolgreich ist, so wird von den Methoden eine Liste mit Termsymbolen zurückgegeben. Scheitert die Unifikation, so ist das Ergebnis der Wert `keine_unif`. Dieser Wert wird auch geliefert, wenn die Methoden auf Junktor- oder Quantorformelobjekte angewendet werden.

Termobjekte besitzen darüber hinaus die Methode `erweiterter-occur-check?`. Diese Methode gibt einen Wahrheitswert zurück, der im Unifikationsverfahren verwendet wird. Sie ist ebenfalls im Abschnitt 7.7.4 beschrieben.

6.2.6 Methoden zum Ersetzen von Teiltermen

Termobjekte und Primformelobjekte bieten die Möglichkeit, das Vorkommen eines Teilterms durch einen anderen Term zu ersetzen. Dies geschieht mittels der Methoden

`(rewrite-formel` *objekt hauptsymbol* $termobjekt_1$ $termobjekt_2$ *n*`)`
`(rewrite-term` *objekt hauptsymbol* $termobjekt_1$ $termobjekt_2$ *n*`).`

Diese Methoden suchen das n-te Vorkommen des angegebenen Symbols in dem Objekt. Das Symbol ist das Hauptsymbol des Objekts $termobjekt_1$ und kann ein Konstanten- oder Funktionssymbol sein. Bei der Suche nach dem n-ten Vorkommen wird zuerst das Hauptsymbol eines Objekts betrachtet und anschließend in den Argumenten eines zusammengesetzten Objekts weitergesucht, bis ein Termobjekt erreicht ist, dessen Hauptsymbol das n-te Vorkommen des Symbols ist. Es wird dann versucht, dieses Termobjekt mit $termobjekt_1$ zu unifizieren. Gelingt die Unifikation, so ist das Ergebnis des gesamten Ersetzungsprozesses ein neu erzeugtes Objekt, in dem das unifizierte Termobjekt durch $termobjekt_2$ ersetzt ist. Dieses Objekt wird zusammen mit einer bei der Unifikation erzeugten Liste von Termsymbolen zurückgegeben. Scheitert die Unifikation, dann ist

das Ergebnis der Wert `keine_unif`. Wenn kein n-tes Vorkommen des Symbols in dem Objekt gefunden wird, so geben die Rewrite-Methoden eine Zahl zurück.

Diese Methoden können benützt werden, um eine Logik mit Gleichheit im Dialogspiel zu realisieren. Sie werden in der aktuellen Version des Systems für dialogische Logik nicht verwendet.

Kapitel 7

Dialogspiel

Im Dialogspiel wird ein Dialogbaum aufgebaut. Der Dialogbaum besteht aus Knoten, welche die Sprechakte enthalten. Der erste Knoten des Dialogbaums enthält die Hypothesen des Opponenten. Alle weiteren Knoten sind abwechselnd dem Proponenten und dem Opponenten zugeordnet und enthalten jeweils einen Sprechakt des Spielers. Zu jedem Knoten werden mögliche Nachfolgerknoten erzeugt, indem Partikelregeln auf den enthaltenen Sprechakt angewendet werden. Die Rahmenregeln legen fest, welche der neu erzeugten und aus früheren Dialogschritten noch vorhandenen Sprechakte erlaubte Nachfolgerknoten sind. Sie greifen dazu auf die in der Dialogliste enthaltenen Daten zurück. Sind keine Nachfolgerknoten möglich, so hat der Spieler, der am Zug ist, den Dialog verloren.

In der Strategiebildung wird versucht, aus den möglichen Knoten des Proponenten jeweils einen auszuwählen, der für alle Nachfolgerknoten des Opponenten schließlich zum Gewinn durch den Proponenten führt. Alle anderen Proponentenknoten des Dialogbaums bleiben unberücksichtigt. Gelingt dies, so ist der entsprechend beschnittene Dialogbaum eine Gewinnstrategie des Proponenten. Der Dialogbaum wird dabei nicht vollständig aufgebaut, sondern nur soweit, wie es zur Bildung einer Gewinnstrategie notwendig ist. Eine Heuristik bestimmt, in welcher Reihenfolge die Knoten bei der Strategiebildung überprüft werden sollen. Der Benutzer kann über das Benutzerspiel diese Reihenfolge bei Zügen des Proponenten bestimmen.

7.1 Sprechakte

Die Schritte eines Dialogs heißen Sprechakte. Sie dienen dazu, Aussagen zu behaupten, frühere Behauptungen anzugreifen und sich gegen Angriffe zu verteidigen.

In Anlehnung an Felscher [5] und [6] werden Sprechakte im System für dialogische Logik aufgefaßt als Tupel $(\sigma\ \nu\ \delta\ \eta_1\ \eta_2\ \zeta)$.

Das Element σ wird als das Signum eines Sprechakts bezeichnet. Es gibt wieder, welcher der beiden Spieler den Sprechakt im Dialog gesetzt hat. In diesem Element können die Werte `o_signum` für

den Opponenten und `p_signum` für den Proponenten eingetragen werden.

Im Element ν wird jedem Sprechakt eines Dialogs eine eindeutige Identifikation zugeordnet. Durch die Identifikation kann in einem Angriffs- oder Verteidigungssprechakt angegeben werden, auf welche frühere Äußerung sich dieser bezieht. Die Identifikation einer Hypothese wird vom Benutzer als beliebige Zeichenkette festgelegt. Allen übrigen Sprechakten eines Dialogs werden Nummern in aufsteigender Reihenfolge zugeordet. Die Identifikationen sind nur innerhalb eines Dialogs eindeutig, nicht aber für den gesamten Dialogbaum, da nur innerhalb von Dialogen Bezüge zwischen Sprechakten hergestellt werden müssen.

Die Behauptung eines Sprechakts ist in dem Element δ als Formelobjekt enthalten. Für behauptungsfreie Sprechakte wird das Objekt `leerformel` eingetragen.

Ein Bezug zwischen Sprechakten wird mittels des Elements η_1 wiedergegeben. In diesem Feld kann die Identifikation eines früheren Sprechakts oder der Wert `undef`, falls kein Bezug hergestellt werden soll, enthalten sein.

Das Element η_2 nennt die Art eines Sprechakts. Mögliche Werte sind `these`, `hypothese`, `angriff`, `verteidigung` und `verlust`. Der Wert `verlust` wird verwendet, um einen Sprechakt als den letzten eines Dialogs zu kennzeichnen.

Zu einigen Arten von Sprechakten müssen weitere Angaben gemacht werden. Bei Angriffen auf Konjunktionen ist anzugeben, ob die linke oder die rechte Teilformel als Verteidigung gefordert wird. Ebenso ist bei Angriffen auf Allaussagen ein Term zu nennen, für den die Verteidigung einzulösen ist. Diese Parameter können in das Element ζ eingetragen werden. Mögliche Werte sind `links` und `rechts`, ein Termobjekt oder `leerparameter`, wenn keine zusätzlichen Angaben gemacht werden müssen.

Die Elemente eines Sprechakts können mittels der Funktionen `sigma-selektor`, `ny-selektor`, `delta-selektor`, `eta1-selektor`, `eta2-selektor` und `zeta-selektor` selektiert werden. Die Funktion `eta1-selektor` überprüft zusätzlich, ob der Bezug des Sprechakts ein Paar ist. In diesem Fall wird das erste Element des Paares als Wert zurückgegeben. Auf diese Weise können Sprechakte gekennzeichnet werden, wie es im Kapitel 10 beschrieben ist.

Die Funktionsaufrufe

(`sigma-setzen` *signum sprechakt*)
(`ny-setzen` *identifikation sprechakt*)
(`delta-setzen` *behauptung sprechakt*)
(`eta1-setzen` *bezug sprechakt*)
(`eta2-setzen` *art sprechakt*)
(`zeta-setzen` *parameter sprechakt*)

erlauben es, die entsprechenden Elemente eines Sprechakts zu modifizieren.

Da den Sprechakten eines Dialogs eindeutige Identifikationen zugeordnet sind, können die Selektoren für Sprechakte erweitert werden zu Funktionen, die eine Identifikation auf die entsprechende Komponente eines Sprechakts abbilden. Die Funktionen

`(delta-funktion` *identifikation*`)`
`(eta1-funktion` *identifikation*`)`
`(eta2-funktion` *identifikation*`)`
`(zeta-funktion` *identifikation*`)`

leisten dies für die Behauptung, den Bezug, die Art und den Parameter von Sprechakten. Sie werden mit Hilfe der Dialogliste berechnet.

Sprechakte werden von Partikelregeln erzeugt. Allerdings tragen die Partikelregeln keine Werte in die Elemente σ und ν des Sprechakts ein. Erst bei der Realisierung eines Sprechakts, wenn er also im Dialog gesetzt wird, werden konkrete Werte in diese Felder eingetragen. Solche von den Partikelregeln erzeugte Sprechakte heißen zur Unterscheidung *Sprechaktformen*. Sie werden durch den Aufruf

`(sprechakt-form` *delta eta1 eta2 zeta*`)`

generiert. Dieser Aufruf gibt eine Liste mit einer Sprechaktform zurück, die die angegebenen Elemente enthält. Die Elemente Signum und Identifikation haben den Wert `undef`. Eine Liste mit mehreren Sprechaktformen wird durch den Aufruf

`(sprechaktliste` $sprechaktform_1$ `...` $sprechaktform_n$`)`

erzeugt.

Wenn ein Dialog endet, wird die Sprechaktform `verlustsprechaktform` realisiert und dadurch ein Verlustsprechakt im Dialog gesetzt. Sie enthält als Behauptung eine Leerformel, als Bezug den Wert `undef`, den Wert `verlust` als Art und den Wert `leerparameter` als Parameter.

Sprechakte werden bei der Strategiebildung durch die Funktion

`(realisiere-dialogschritt` *sprechaktform signum*`)`

realisiert. Das Signum gibt wieder, für welchen Spieler der Sprechakt gesetzt werden soll. Der Sprechakt wird bei der Realisierung in die Dialogliste eingetragen. Der Rückgabewert der Funktion ist ein Sprechakt, in dem das Signum und die Identifikation gesetzt sind.

7.2 Dialogliste

Die Dialogliste enthält die Daten des aktuell geführten Dialogs. Diese Daten werden von den Rahmenregeln verwendet, um zu entscheiden, welche Sprechakte in einem Dialogzustand realisiert werden dürfen. Auch die Partikelregeln ermitteln mit Hilfe der Dialogliste Eigenschaften von Sprechakten, die angegriffen oder verteidigt werden sollen.

Die Dialogliste besteht aus Objekten, die mittels der Aufrufe

`(make-startzustand)`
`(make-dialogzustand` *identifikation vorgaengerzustand*`)`

erzeugt werden. Die Funktion `make-startzustand` generiert das erste Objekt der Dialogliste. Dieses Objekt nimmt die vom Opponenten gesetzten Hypothesen eines Dialogs auf. Alle weiteren Objekte werden durch die Funktion `make-dialogzustand` erzeugt. Der Parameter *vorgaengerzustand* enthält das vorhergehende Objekt der Dialogliste. Durch diesen Verweis entsteht eine verkettete Liste von Objekten. Dem jeweils letzten Objekt wird der Name `dialogliste` zugeordnet. Der Parameter *identifikation* teilt dem erzeugten Objekt mit, welche Identifikation dem Sprechakt, der in dieses Objekt eingetragen wird, bei der Realisierung zugewiesen werden soll. Diese Identifikation wird von dem jeweils letzten Element der Dialogliste durch die Methode

`(ny-aktuell dialogliste)`

zurückgegeben. Zur Realisierung kann ein Sprechakt durch die Methode

`(setze-sprechakt dialogliste` *sprechakt*`)`

in das letzte Objekt der Dialogliste eingetragen werden. Im Startzustand kann diese Methode mehrfach aufgerufen werden, um mehrere Hypothesen einzutragen. In den anderen Dialogzuständen darf sie nur einmal angewendet werden. Damit der nächste Dialogschritt eingetragen werden kann, muß die Dialogliste um einen neuen Zustand durch den Aufruf

`(extend dialogliste)`

erweitert werden. Diese Methode generiert mittels der Funktion `make-dialogzustand` ein neues Objekt, dem als Vorgängerzustand das in `dialogliste` enthaltene Objekt übergeben wird. Als Identifikation wird der Wert 1 übergeben, wenn die Methode im Startzustand aufgerufen wird, ansonsten die inkrementierte Identifikation des Vorgängerzustands.

Zusätzlich stellen die Zustandsobjekte die Methoden `return`, `aktueller-dialog-zustand` und `failure` zur Verfügung, um die Dialogliste zu manipulieren. Diese Methoden werden bei der Strategiebildung verwendet, um beim Auftreten von Backtracking gespeicherte Dialogzustände wiederherzustellen und zu Vorgängerzuständen zurückzukehren. Die Methode

`(return dialogliste)`

wird aufgerufen, wenn ein Dialogabschnitt erfolgreich beendet wurde. Sie kehrt zum unmittelbar vorhergehenden Dialogzustand zurück, indem der Vorgänger des aktuellen Zustandsobjekts zum aktuellen Zustandsobjekt gemacht wird. Wenn ein Dialog scheitert, dann muß zu einem früheren Dialogzustand gesprungen werden, um von dort neue Zugmöglichkeiten zu versuchen. Der aktuelle Dialogzustand wird von der Methode

`(aktueller-dialog-zustand dialogliste)`

zurückgegeben. Dieser Zustand kann gespeichert werden und zu einem späteren Zeitpunkt durch den Aufruf

`(failure dialogliste` *dialogzustand*`)`

wiederhergestellt werden.

Die Dialogzustandsobjekte besitzen zahlreiche Methoden, die Auskunft über den bisherigen Dialogverlauf geben. Diese Methoden werden von den Partikel- und Rahmenregeln nicht direkt, sondern über Schnittstellenfunktionen aufgerufen. Partikelregeln können über die Methoden

(gesamter-sprechakt dialogliste *identifikation*)
(behauptung dialogliste *identifikation*)
(bezug dialogliste *identifikation*)
(sprechaktart dialogliste *identifikation*)
(sprechaktparameter dialogliste *identifikation*)

auf gesetzte Sprechakte bzw. deren Elemente zugreifen. Die zugehörigen Schnittstellenfunktionen sind:

(sprechakt-funktion *identifikation*)
(delta-funktion *identifikation*)
(eta1-funktion *identifikation*)
(eta2-funktion *identifikation*)
(zeta-funktion *identifikation*).

Diese Funktionen geben mit Hilfe der zugehörigen Methoden einen Sprechakt sowie die Behauptung, den Bezug, die Art und den Parameter des Sprechakts mit der übergebenen Identifikation zurück. Die Methode

(these-bereits-gesetzt? dialogliste)

wird über die Schnittstellenfunktion `these-gesetzt?` ohne Argumente aufgerufen. Der zurückgegebene Wahrheitswert gibt an, ob im Dialog bereits eine These gesetzt wurde.

Um in den Rahmenregeln die Zahl von Angriffswiederholungen beschränken zu können, ermittelt die Methode

(anzahl-gleicher-angriffe dialogliste *identifikation*),

wie oft der Sprechakt mit der angegebenen Identifikation bereits angegriffen wurde. Sie wird über die Funktion

(angriffswiederholungen *identifikation*)

aufgerufen. Die Methode `anzahl-gleicher-verteidigungen` und die zugehörige Schnittstellenfunktion `verteidigungswiederholungen` leisten dasselbe für Wiederholungen von Verteidigungssprechakten. Die Methoden

(liste-offener-angriffe dialogliste *signum*)
(letzter-offener-angriff dialogliste *signum*)
(ist-offener-angriff? dialogliste *identifikation signum*)
(letzter-offener-angriff-verteidigbar? dialogliste *signum*)

geben Auskunft über die offenen Angriffe des Spielers mit dem angegebenen Signum. Nur die letzten drei dieser Methoden werden von den Rahmenregeln aufgerufen. Die zugehörigen Schnittstellenfunktionen sind:

```
(ny-letzter-offener-angriff signum)
(offener-angriff? identifikation signum)
(letzter-angriff-verteidigbar? signum).
```

Die erste Funktion gibt die Identifikation des letzten noch unbeantworteten Angriffssprechakts zurück oder liefert den Wert `undef`, wenn kein offener Angriff vorhanden ist. Die zweite Funktion liefert einen Wahrheitswert, der wiedergibt, ob der Sprechakt mit der angegebenen Identifikation ein offener Angriff des durch den Parameter *signum* bezeichneten Spielers ist. In den Partikelregeln wird für Angriffe auf Negationen üblicherweise keine Verteidigungsmöglichkeit angegeben. Logische Symbole ohne Verteidigung können bei der Definition von Partikelregeln in die Liste `junktoren_ohne_verteidigung` eingetragen werden. Diese Liste wird von der Methode `letzter-angriff-verteidigbar?` verwendet. Sie liefert den Wert `#f`, wenn die zuletzt durch den angegebenen Spieler angegriffene Behauptung als Hauptsymbol ein Junktorsymbol dieser Liste hat, ansonsten den Wert `#t`.

Auf die Identifikation des letzten im Dialog gesetzten Sprechakts kann über die Funktion

```
(ny-letzter-sprechakt)
```

zugegriffen werden. Diese Funktion liefert das Ergebnis des Methodenaufrufs:

```
(ny-aktuell dialogliste).
```

Für die Übernahme von Primformeln muß ermittelt werden, welche Primformeln ein Spieler bereits gesetzt hat. Dies leisten die Methoden

```
(gesetzte-primformeln dialogliste signum)
(gesetzte-primformeln-mit-ny dialogliste signum).
```

Die erste Methode liefert eine Liste der behaupteten Primformelobjekte des durch den Parameter *signum* bezeichneten Spielers. Die zweite Methode gibt eine Liste mit Paaren zurück. Jedes Paar enthält die Identifikation und die behauptete Primformel eines Sprechakts. Zu beiden Methoden existieren keine Schnittstellenfunktionen, da sie nicht von den Partikel- oder Rahmenregeln, sondern bei der Primformelübernahme aufgerufen werden.

Die Berechnung der offenen Angriffe, der gesetzten Primformeln und der Wiederholungen von Angriffen und Verteidigungen kann von jedem Zustandsobjekt aus dem in ihm enthaltenen Sprechakt und dem entsprechenden Ergebnis des Vorgängerzustands berechnet werden. Um die Berechnung zu beschleunigen, speichert jedes Zustandsobjekt die ermittelten Ergebnisse. Wenn zu einem weiter fortgeschrittenen Dialog diese Angaben ermittelt werden sollen, so muß die Berechnung nicht für alle Objekte der Dialogliste neu erfolgen, sondern lediglich für die seit der letzten Berechnung neu erzeugten Dialogzustandsobjekte.

7.3 Partikelregeln

Die Anwendung der Partikelregeln erzeugt Sprechakte, die im Dialog gesetzt werden können. Sie werden dazu in jedem Dialogschritt auf den zuletzt realisierten Sprechakt angewendet. Je nach Art des realisierten Sprechakts treffen unterschiedliche Partikelregeln zu. Jede der aktivierten Partikelregeln erzeugt Sprechaktformen, die zusammen mit den in früheren Dialogschritten erzeugten und noch nicht gelöschten Sprechaktformen von den Rahmenregeln überprüft werden.

Partikelregeln werden von dem Objekt `partikelregel` verwaltet. Dieses Objekt erlaubt es, Partikelregeln zu definieren, Definitionen zu löschen und alle definierten Partikelregeln auf einen Sprechakt anzuwenden. Die Definition einer Partikelregel erfolgt mittels der Methode

`(definiere partikelregel` *regelname regelrumpf*`).`

Als Regelname kann eine beliebige Zeichenkette angegeben werden. Der Regelrumpf ist eine Scheme-Funktion mit fünf Argumenten. Er wird durch den Aufruf

```
(p-regel ny delta eta1 eta2 zeta
   (regelbedingungen bedingung_1 ... bedingung_n)
   (sprechakt-form delta' eta1' eta2' zeta'))
```

erzeugt. Die Einträge `ny`, `delta`, `eta1`, `eta2` und `zeta` sind dabei die formalen Parameter der Funktion. In ihnen werden den Partikelregeln die für sie wichtigen Daten eines Sprechakts übergeben. Damit Partikelregeln aus einem Sprechakt auch mehrere Sprechaktformen erzeugen können, kann ein Regelrumpf auch durch einen Aufruf der Form

```
(p-regel ny delta eta1 eta2 zeta
   (regelbedingungen bedingung_1 ... bedingung_n)
   (sprechaktliste
     (sprechakt-form delta_1 eta1_1 eta2_1 zeta_1)
     ⋮
     (sprechakt-form delta_n eta1_n eta2_n zeta_n)))
```

generiert werden.

Bei der Anwendung einer Partikelregel werden die Regelbedingungen ausgewertet. Sie überprüfen Eigenschaften der in den Parametern übergebenen Daten eines Sprechakts und auch des Sprechakts, auf den sich dieser Sprechakt bezieht. Für die Argumente $bedingung_1, \ldots, bedingung_n$ können Scheme-Ausdrücke angegeben werden, die Wahrheitswerte liefern. In ihnen werden Scheme-Prädikate wie `eq?` oder `equal?`, logische Verknüpfungen, die Selektormethode `hauptverknuepfung` von Formelobjekten sowie die Funktionen `delta-funktion`, `eta1-funktion`, `eta2-funktion` und `zeta-funktion` verwendet. Die verwendbaren Funktionen und Methoden sind im Anhang B aufgelistet. Wenn alle Bedingungen erfüllt sind, wird als Ergebnis der Partikelregelanwendung eine Liste mit einer oder mehreren Sprechaktformen zurückgegeben. Scheitert mindestens eine der Bedingungen, dann ist der Rückgabewert eine leere Liste.

Zum Aufbau des Elements δ von neu zu erzeugenden Sprechaktformen können die Selektormethoden `unterformel`, `linke-unterformel`, `rechte-unterformel`, `rumpf` und `quantorvariable` sowie die Modifikatormethode `substituiere-variable` auf die Behauptung des übergebenen Sprechakts angewendet werden.

Wenn für die Konjunktion das Junktorsymbol „AND“ deklariert wurde, können zur Verwendung dieser Verknüpfung mit den beschriebenen Mitteln drei Partikelregeln definiert werden:

```
(definiere partikelregel
  "A_Kon"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen      (equal? (hauptverknuepfung delta) "AND"))
         (sprechaktliste
            (sprechakt-form     leerformel                          ;; delta
                                ny                                  ;; eta1
                                angriff                             ;; eta2
                                links )                             ;; zeta
            (sprechakt-form     leerformel                          ;; delta
                                ny                                  ;; eta1
                                angriff                             ;; eta2
                                rechts))))                          ;; zeta

(definiere partikelregel
  "V_Kon_L"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung (delta-funktion eta1))
                                     "AND")
                             (eqv? eta2 angriff)
                             (eqv? zeta links))
         (sprechakt-form     (linke-unterformel
                                 (delta-funktion eta1))             ;; delta
                             ny                                     ;; eta1
                             verteidigung                           ;; eta2
                             leerparameter)))                       ;; zeta

(definiere partikelregel
  "V_Kon_R"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung
                                         (delta-funktion eta1))
                                     "AND")
                             (eqv? eta2 angriff)
                             (eqv? zeta rechts))
         (sprechakt-form     (rechte-unterformel
                                 (delta-funktion eta1))             ;; delta
                             ny                                     ;; eta1
                             verteidigung                           ;; eta2
                             leerparameter)))                       ;; zeta
```

Die Regel „A_Kon“ erzeugt aus einem Sprechakt, in dem eine Konjunktion behauptet wird, zwei behauptungsfreie Angriffe, die sich durch die Parameter `links` und `rechts` unterscheiden. Die beiden anderen Regeln überprüfen, ob in dem übergebenen Sprechakt ein Angriff auf eine Kon-

junktion erfolgt. Sie ermitteln anhand des Parameters des Sprechakts, ob die linke oder die rechte Seite der Konjunktion eingefordert wurde und erzeugen mit Hilfe der Selektormethoden entsprechende Verteidigungssprechakte. In dieser Weise können auch Partikelregeln für die Disjunktion, Subjunktion und Negation definiert werden.

Die Partikelregel für Allaussagen besagt, daß beim Angriff auf eine solche Aussage ein Term gewählt werden muß, für den die Verteidigung einzulösen ist. Im System für dialogische Logik wird statt eines konkreten Terms ein Termsymbol gewählt. Diesem Termsymbol wird im Laufe des Dialogs ein konkreter Term zugeordnet. Da erzeugte Sprechakte im Dialog mehrfach realisiert werden können, darf bei der Erzeugung noch kein Termsymbol eingetragen werden. Ansonsten könnten durch die mehrmalige Realisierung mehrere Bindungen für ein Termsymbol entstehen. Um dies zu verhindern, wird bei der Erzeugung eines Sprechakts durch die Partikelregeln lediglich die zu ersetzende Quantorvariable im Parameter des Sprechakts vermerkt. Bei der Realisierung wird die Quantorvariable durch ein Termsymbol ersetzt.

Wenn das Quantorsymbol „ALL“ für den Allquantor deklariert wurde, dann lauten die Definitionen der entsprechenden Partikelregeln folgendermaßen:

```
(definiere partikelregel
  "A_All"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta)
                                     "ALL"))
         (sprechakt-form     leerformel                          ;; delta
                             ny                                  ;; eta1
                             angriff                             ;; eta2
                             (quantorvariable delta))))          ;; zeta

(definiere partikelregel
  "V_All"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung
                                          (delta-funktion eta1))
                                     "ALL")
                             (eqv? eta2 angriff))
         (sprechakt-form     (substituiere-variable              ;; delta
                                   (rumpf (delta-funktion eta1))
                                   (quantorvariable
                                       (delta-funktion eta1))
                                   zeta)
                             ny                                  ;; eta1
                             verteidigung                        ;; eta2
                             leerparameter)))                    ;; zeta
```

Die Partikelregel „A_All“ erzeugt einen Angriffssprechakt auf eine Allaussage und trägt dabei die Quantorvariable in das Element ζ des Sprechakts ein. Von der Partikelregel „V_All“ wird ein entsprechender Verteidigungssprechakt erzeugt. Sie ersetzt mittels der Modifikatormethode `substituiere-variable` die Quantorvariable im Rumpf der Quantorformel durch das im Parameter enthaltene Termsymbolobjekt, das dort bei der Realisierung des vorhergehenden Angriffs-

sprechakts eingesetzt wurde.

In ähnlicher Weise wird bei den Partikelregeln für Existenzaussagen verfahren. Sei „EXI“ als das Quantorsymbol für den Existenzquantor deklariert. Die Partikelregeln lauten dann:

```
(definiere partikelregel
  "A_Exi"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta)
                                     "EXI"))
         (sprechakt-form     leerformel                          ;; delta
                             ny                                  ;; eta1
                             angriff                             ;; eta2
                             leerparameter)))                    ;; zeta

(definiere partikelregel
  "V_Exi"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (eqv? eta2 angriff)
                             (equal? (hauptverknuepfung
                                          (delta-funktion eta1))
                                     "EXI"))
         (sprechakt-form     (rumpf (delta-funktion eta1))       ;; delta
                             ny                                  ;; eta1
                             verteidigung                        ;; eta2
                             (quantorvariable                    ;; zeta
                                 (delta-funktion eta1)))))
```

Die Regel „A_Exi“ erzeugt einen Angriffssprechakt für einen schlichten Zweifel an einer Existenzaussage. In der Regel „V_Exi“ wird ein Verteidigungssprechakt generiert, wobei noch kein Termsymbol für die Quantorvariable eingesetzt wird. Stattdessen wird der Rumpf der Existenzaussage unverändert als Behauptung übernommen und die zu ersetzende Quantorvariable als Parameter des Sprechakts vermerkt. Bei der Realisierung des Verteidigungssprechakts wird die angegebene Variable durch ein Termsymbolobjekt substituiert.

Die Definitionen aller üblicherweise verwendeten Partikelregeln sind zusammen mit den Deklarationen der logischen Symbole in der Datei `standard.par` im Verzeichnis `Regeln` enthalten. Hier wird auch die von den Rahmenregeln benötigte Liste `junktoren_ohne_verteidigung` belegt. Die Definitionen aller Partikel und der zugehörigen Partikelregeln aus dieser Datei sind im Anhang A aufgeführt.

Die Deklarationen von logischen Symbolen und die Definitionen der Partikelregeln können mit dem Befehl

`(lade-partikel` *dateiname*`)`

aus einer Datei geladen werden. An den Dateinamen werden das Verzeichnis `Regeln` und das Suffix „.par“ angefügt. Die vor dem Laden vorhandenen Junktor- und Quantorsymbole werden ebenso wie die Liste `junktoren_ohne_verteidigung` gelöscht.

Die Methoden

```
(ist-definiert? partikelregel regelname)
(undefiniere partikelregel regelname)
(alle-loeschen partikelregel)
```

überprüfen, ob eine Partikelregel definiert ist bzw. löschen eine oder alle Definitionen von Partikelregeln. Durch die Methode

```
(alle-anwenden partikelregel sprechakt)
```

werden die definierten Partikelregeln im Dialogspiel angewendet. Ihr wird ein realisierter Sprechakt übergeben und sie liefert eine Liste mit den daraus erzeugten Sprechaktformen zurück.

7.4 Rahmenregeln

Die Rahmenregeln haben im Dialogspiel die Aufgabe, aus den von den Partikelregeln erzeugten Sprechakten diejenigen auszuwählen, die im nächsten Dialogschritt realisierbar sind. Indem die Rahmenregeln nur bestimmte Dialogverläufe zulassen, legen sie die verwendete Logik fest. Sie bestimmen, ob eine These als effektiv oder klassisch-logisch allgemeingültig erwiesen werden soll. Um unterschiedliche Logiken zuzulassen, sind die Rahmenregeln nicht festgelegt, sondern können wie die Partikelregeln definiert werden. Die Definitionen werden von dem Objekt `rahmenregel` verwaltet. Die Methode

```
(definiere rahmenregel regelname regelrumpf)
```

definiert eine Rahmenregel. Als Regelname kann eine beliebige, auf den Rahmenregeln eindeutige Zeichenkette angegeben werden. Der Regelrumpf wird durch einen Aufruf der Form

```
(r-regel sigma delta eta1 eta2 zeta
   (regelbedingungen bedingung_1 ... bedingung_n)
   kennzeichen)
```

erzeugt. Dabei sind `sigma`, `delta`, `eta1`, `eta2` und `zeta` die formalen Parameter des Regelrumpfs. Für das Kennzeichen sind die Werte `sperre-sprechakt` und `loesche-sprechakt` zulässig. Die Bedingungen sind Scheme-Ausdrücke, die Wahrheitswerte liefern. In ihnen können Scheme-Prädikate und logische Operatoren sowie die Schnittstellenfunktionen der Dialogliste verwendet werden. Die für die Rahmenregeln zur Verfügung gestellten Funktionen und Methoden sind im Anhang B aufgelistet. Die Definitionen von Rahmenregeln können wie die der Partikelregeln durch die Methoden `undefiniere` und `alle-loeschen` gelöscht, durch die Methode `ist-definiert?` überprüft werden.

Der Funktionsaufruf

```
(lade-rahmenregeln dateiname)
```

löscht alle definierten Rahmenregeln und lädt neue Definitionen aus einer Datei im Verzeichnis `Regeln`, wobei an den Dateinamen das Suffix „.rah“ angefügt wird.

Die Methode

`(alle-anwenden rahmenregel` *signum neue-sprechakte alte-sprechakte*`)`

wendet alle definierten Rahmenregeln auf die bisher erzeugten Sprechakte an, um diese in realisierbare und gesperrte Sprechakte einzuteilen. Dieser Methode werden das Signum des Spielers, der den nächsten Sprechakt setzen soll, die aus dem letzten Dialogschritt von den Partikelregeln neu erzeugten Sprechakte sowie alle früher erzeugten und noch nicht gelöschten alten Sprechakte übergeben. Die alten Sprechakte wurden von den Rahmenregeln bereits im vorherigen Dialogschritt in realisierbare und gesperrte Sprechakte eingeteilt. Außerdem wird unterschieden, von welchem Spieler sie gesetzt werden können. Somit teilen sich die alten Sprechakte in vier Mengen auf: die im letzten Dialogschritt realisierbaren und gesperrten Sprechakte des Proponenten und die realisierbaren und gesperrten Sprechakte des Opponenten.

Bei der neuen Anwendung der Rahmenregeln sollen die neu erzeugten Sprechakte und die alten Sprechakte des aktuellen Spielers eingeteilt werden in realisierbare und gesperrte Sprechakte. Darüber hinaus werden solche Sprechakte ermittelt, die in keinem späteren Dialogzustand realisierbar sind und somit gelöscht werden können.

Bei der Definition werden die Rahmenregeln auf Grund des im Regelrumpf angegebenen Kennzeichens in zwei Klassen eingeteilt: sperrende und löschende Regeln. Auf jeden zu überprüfenden Sprechakt werden zuerst die löschenden Rahmenregeln angewendet. Sobald bei einer dieser Regeln alle Bedingungen des Regelrumpfs erfüllt sind, kann ein Sprechakt gelöscht werden. Auf die nicht gelöschten Sprechakte werden im nächsten Schritt die sperrenden Regeln angewendet. Trifft eine dieser Regeln zu, so wird ein Sprechakt für den nächsten Dialogschritt gesperrt. Alle anderen Sprechakte sind realisierbar. Der Rückgabewert der Methode ist eine Liste mit den im aktuellen Dialogzustand realisierbaren und gesperrten Sprechakten beider Spieler, wobei die Sprechakte des Spielers, der nicht am Zug ist, unverändert aus den alten Sprechakten übernommen werden.

Die verwendeten Rahmenregeln wurden in Anlehnung an Felscher [5] und [6] formalisiert. Für die effektive Logik werden folgende Regeln benützt:

Regel E : Ein Sprechakt des Opponenten muß sich stets auf den unmittelbar vorhergehenden Sprechakt des Proponenten beziehen.

```
(definiere rahmenregel
  "RR_E_O"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen (eqv? sigma o_signum)
                             (not (eq? eta1 (ny-letzter-sprechakt))))
           loesche-sprechakt))
```

Regel D-These : Der Proponent darf die These höchstens einmal setzen.

```
(definiere rahmenregel
  "RR_D_These_P"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen (eqv? eta2 these)
                             (these-gesetzt?))
           loesche-sprechakt))
```

Regel D00 : Der Proponent setzt als ersten Sprechakt die These.

```
(definiere rahmenregel
  "RR_D00_P"
  (r-regel sigma delta eta1 eta2 zeta
    (regelbedingungen (not (eqv? eta2 these))
                      (not (these-gesetzt?)))
     sperre-sprechakt))
```

Regel D11 : Der Proponent darf nur auf den letzten offenen Angriff des Opponenten antworten.

```
(definiere rahmenregel
"RR_D11_P"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen (eqv? sigma p_signum)
                             (eqv? eta2 verteidigung)
                             (not (eqv? eta1
                                        (ny-letzter-offener-angriff
                                             o_signum))))
           sperre-sprechakt))
```

Regel D12a : Der Proponent darf einen Angriff des Opponenten höchstens einmal beantworten. Verteidigungen zu nicht mehr offenen Angriffen werden gelöscht.

```
(definiere rahmenregel
  "RR_D12a_P"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen (eqv? sigma p_signum)
                             (eqv? eta2 verteidigung)
                             (not (offener-angriff? eta1 o_signum)))
           loesche-sprechakt))
```

Regel D12b : Wenn sich der Proponent gegen den letzten offenen Angriff nicht verteidigen kann, dann können alle früheren Verteidigungspflichten gelöscht werden.

```
(definiere rahmenregel
  "RR_D12b_P"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen (eqv? sigma p_signum)
```

```
                                (eqv? eta2 verteidigung)
                                (not (letzter-angriff-verteidigbar?
                                        o_signum)))
            loesche-sprechakt))
```

Regel D-Angriff : Die Zahl der Angriffswiederholungen des Proponenten auf eine Aussage des Opponenten ist durch die Angriffsschranke beschränkt.

```
(definiere rahmenregel
  "RR_D_Angriff_P"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen (eqv? sigma p_signum)
                             (eqv? eta2 angriff)
                             (>= (angriffswiederholungen eta1)
                                 angriff-schranke))
           loesche-sprechakt))
```

Die Regel D12b wurde eingeführt, um nach einem Angriff auf eine Negation alle früheren Verteidigungsmöglichkeiten zu löschen, so daß sie in späteren Dialogschritten nicht mehr betrachtet werden müssen. Durch die Regel D-Angriff wird verhindert, daß unendlich viele Angriffe auf einen Sprechakt ausgeführt werden können, wodurch unendlich lange Dialoge entstehen können. Die Angriffsschranke kann durch die Funktion

`(setze-angriff-schranke` *wert*`)`

gesetzt werden. Als Wert ist eine positive, ganze Zahl anzugeben.

FELSCHER verwendet weiterhin eine Rahmenregel D10, die festlegt, daß der Proponent eine Primformel erst behaupten darf, nachdem sie vom Opponenten behauptet wurde. Diese Reihenfolge wird durch das Verfahren der Primformelübernahme sichergestellt, so daß keine Rahmenregel dafür nötig ist.

Die genannten Rahmenregeln sind in der Datei `effektiv.rah` im Verzeichnis `Regeln` enthalten. Für klassisch-logische Dialoge werden die Regeln D11 und D12b entfernt, so daß der Proponent beliebige offene Angriffe beantworten darf. Die Regel D12a wird so modifiziert, daß der Proponent sich gegen einen Angriff auch mehrfach verteidigen darf. Die Anzahl der Verteidigungen muß wie bei den Angriffswiederholungen beschränkt werden, damit dadurch keine unendlichen Dialoge entstehen. Dazu wird eine Verteidigungsschranke eingeführt, die durch den Aufruf

`(setze-verteidigung-schranke` *wert*`)`

gesetzt werden kann. Das Ergebnis der Funktion `verteidigungswiederholungen` wird mit diesem Wert verglichen, um über die Zulässigkeit einer erneuten Verteidigung zu entscheiden. In den Regeln der klassisch-logischen Dialoge ist dem Proponenten darüber hinaus erlaubt, die These zu einem beliebigen Zeitpunkt im Dialog zu setzen.

Die Rahmenregeln für die klassische Logik sind in der Datei `klass.rah` im Verzeichnis `Regeln` enthalten.

In beiden Varianten von Dialogspielen wird die Übernahme von Primformeln nur durch die genannte Reihenfolge beim Setzen von Primformeln beschränkt. Für andere Logiken können weitere Bedingungen notwendig sein. Das Kapitel 10 beschreibt dies für den Fall der Modallogik. Um derartige Erweiterungen zu ermöglichen, müssen Regeln definierbar sein, welche die Übernahme von Primformeln einschränken. Eine solche Regel kann durch den Aufruf

```
(primformel-regel
   o-delta o-eta1 o-eta2 o-zeta
   p-delta p-eta1 p-eta2 p-zeta
   (regelbedingungen bedingung_1 ... bedingung_n)
   sperre-uebernahme)
```

erzeugt werden. In den formalen Parametern `o-delta`, ..., `o-zeta` und `p-delta`, ..., `p-zeta` werden beim Auswerten der Regel die Daten eines Sprechakts des Opponenten, in dem eine Primformel als Behauptung gesetzt wurde, und die Daten eines Sprechakts des Proponenten, mit dem er dieselbe Primformel setzen möchte, eingetragen. Die Behauptungen beider Sprechakte sind identisch. In den Regelbedingungen können Kriterien für die Einschränkung der Übernahme formuliert werden. Wenn die Bedingungen erfüllt sind, wird die Übernahme verhindert.

Die Regeln für die Primformelübernahme werden wie die übrigen Regeln mittels der Methode `definiere` in das Objekt zur Verwaltung der Rahmenregeln eingetragen. Sie werden jedoch nicht von der Methode `alle-anwenden` ausgewertet. Stattdessen überprüft die Methode

```
(uebernahme-moeglich? o-sprechakt p-sprechakt)
```

alle solchen Regeln und liefert den Wahrheitswert `#f`, wenn mindestens eine Regel erfüllt ist. Diese Methode wird bei der Primformelübernahme aufgerufen. Da für effektive und klassisch-logische Dialogspiele keine Primformelregeln definiert werden, gibt die Methode in diesem Fall stets den Wert `#t` zurück und ermöglicht damit die Übernahme.

7.5 Heuristik

Die Heuristik ordnet in jedem Schritt die realisierbaren Sprechakte und bestimmt damit die Reihenfolge, in der diese bei der Strategiebildung getestet werden sollen. Das Ziel ist, sowohl bei erfolgreichen Dialogspielen als auch bei gescheiterten Beweisversuchen die Zahl der überprüften Knoten zu minimieren.

Da die Partikel modifizierbar sind, die Heuristik jedoch festgelegt ist, muß die Entscheidung über die Reihenfolge der Sprechakte unabhängig von den in den Behauptungen verwendeten Partikeln sein. Aus diesem Grund wird nur eine sehr einfache, lokale Heuristik angewendet. Sie legt für den Opponenten fest, daß dieser möglichst wenige Primformeln behaupten soll. Deshalb werden

behauptungsfreie Sprechakte vor Sprechakten mit zusammengesetzten Aussagen getestet. Sprechakte mit Primformeln werden zuletzt überprüft. Auf diese Weise wird frühzeitig festgestellt, ob ein Dialogspiel scheitern wird.

Für den Proponenten sollen Sprechakte umso später getestet werden, je häufiger sie bereits wiederholt wurden. Dies gilt sowohl für Angriffs- als auch für Verteidigungssprechakte. Bei gleicher Wiederholungszahl wird unterschieden, welche Art von Sprechakt gesetzt werden soll und welche Behauptung der Sprechakt enthält. Es ist sinnvoll, Angriffe stets zu beantworten, soweit dies möglich ist. Da nach einer erfolgreichen Verteidigung mit einer Primformel ein Dialog gewonnen ist, sollen solche Sprechakte vor Verteidigungen mit zusammengesetzten Behauptungen getestet werden. Bei Angriffen sollen behauptungsfreie Sprechakte vor solchen mit Primformel überprüft werden. Zuletzt werden Angriffe mit zusammengesetzten Formeln getestet.

Die Heuristik wird durch die Funktionen `p-heuristik` und `o-heuristik` realisiert. Die Funktion `p-heuristik` wird in Opponentenknoten aufgerufen, um die Nachfolgersprechakte des Proponenten zu ordnen. Die Funktion `o-heuristik` ordnet die Nachfolgersprechakte des Opponenten in Proponentenknoten. Beiden Funktionen wird die Liste der realisierbaren Sprechakte übergeben. Sie bewerten die Sprechakte nach den genannten Kriterien und geben eine geordnete Liste zurück.

Da in erfolglosen Dialogspielen stets alle Nachfolgersprechakte des Proponenten getestet werden, hat die Heuristik für den Proponenten keinen Einfluß auf die Zahl der getesteten Knoten in solchen Dialogspielen. In gleicher Weise minimiert die Heuristik des Opponenten nicht die Zahl der getesteten Knoten in erfolgreichen Dialogspielen, sondern wirkt sich nur aus in Dialogspielen, die scheitern.

7.6 Thesen- und Hypothesenverwaltung

Die These und die Hypothesen, zu denen eine Gewinnstrategie gefunden werden soll, sind vor dem Start des Dialogspiels festzulegen. Sie werden von dem Objekt `thesenverwalter` verwaltet. Dieses Objekt besitzt Methoden, um Thesen und Hypothesen zu definieren, Definitionen zu überprüfen und zu löschen und um die entsprechenden Sprechakte in ein Dialogspiel zu übernehmen. Der Aufruf der Methode

`(definiere-hypothese thesenverwalter` *name formelobjekt*`)`

trägt eine Hypothese ein. Der Name muß eine für die Hypothesen eindeutige Zeichenkette sein. Er wird im Dialogspiel als Identifikation des zugehörigen Hypothesensprechakts verwendet. Das Formelobjekt ist die Behauptung dieses Sprechakts. Die Behauptung muß eine geschlossene Formel sein, darf also keine freien Variablen enthalten. Durch die Methoden

`(ist-hypothese-definiert? thesenverwalter` *name*`)`
`(undefiniere-hypothese thesenverwalter` *name*`)`

kann die Definition überprüft oder gelöscht werden.

Eine These wird durch die Methode

`(definiere-these thesenverwalter` *formelobjekt*`)`

definiert. Das Formelobjekt muß ebenfalls eine geschlossene Formel enthalten. Zu der These wird kein Name angegeben, da ihr erst bei der Realisierung im Dialogspiel eine Identifikation zugewiesen wird. Es darf nur eine einzige These für ein Dialogspiel definiert werden. Die Methoden

```
(ist-these-definiert? thesenverwalter )
(undefiniere-these thesenverwalter)
```

überprüfen bzw. löschen die Definition der These.

In das Dialogspiel können die These und die Hypothesen durch die Methoden

```
(gesetzte-these thesenverwalter )
(hypothesenliste thesenverwalter)
```

übernommen werden. Die erste Methode gibt eine Sprechaktform zurück, in der die definierte These als Behauptung enthalten ist. Die zweite Methode liefert eine Liste von Sprechakten. Diese enthalten jeweils eine definierte Hypothese als Behauptung, den zugehörigen Namen als Identifikation und als Signum den Wert `o_signum`. Sie können damit unmittelbar im Dialog gesetzt werden.

Nach einem Dialogspiel bleiben die Definitionen erhalten. Falls keine Gewinnstrategie gefunden wurde, kann mit evtl. erweiterten Hypothesen oder anderen Rahmenregeln ein neuer Beweis versucht werden. Um alle Definitionen zu löschen, wird die Methode

```
(loesche-alles thesenverwalter)
```

aufgerufen.

7.7 Strategiebildung

7.7.1 Baumaufbau

In der Strategiebildung wird ein Dialogbaum aufgebaut. Dabei wird versucht, Knoten des Dialogbaums auszuwählen, die eine Gewinnstrategie des Proponenten bilden. Solche Knoten werden als erfolgreiche Knoten bezeichnet. Ein Knoten des Proponenten ist erfolgreich, wenn alle seine Nachfolgerknoten erfolgreich sind oder wenn keine Nachfolgerknoten existieren. Ein Knoten des Opponenten ist erfolgreich, wenn er mindestens einen erfolgreichen Nachfolgerknoten hat. Bei der Überprüfung der Knoten wird im Dialogbaum nach dem Prinzip „Tiefe zuerst“ vorgegangen. Das bedeutet, es wird stets ein Dialog des Dialogbaums zu Ende geführt. Anschließend wird zu einer Verzweigung im Dialogbaum zurückgegangen und der nächste Dialog bis zu seinem Ende gespielt. Es müssen nicht alle möglichen Verzweigungen des Dialogbaums untersucht werden. Sobald zu einem Knoten des Proponenten ein Nachfolger gefunden wird, der nicht erfolgreich ist, ist auch

der Proponentenknoten gescheitert. Alle bis dahin noch nicht überprüften Nachfolger bleiben unberücksichtigt. Entsprechend muß bei den Nachfolgern eines Opponentenknotens nur so lange gesucht werden, bis ein einziger erfolgreicher Knoten gefunden wird.

Die Dialogbildung wird gestartet durch den Aufruf der Funktion

```
(starte-dialog).
```

Diese Funktion initialisiert die für die Termwahl und die Primformelübernahme benötigten Objekte und erzeugt einen Startzustand für die Dialogliste. Anschließend ruft sie Funktionen auf, die als die Knoten des Dialogbaums betrachtet werden. Diese Funktionen heißen `p-node`, `o-node` und `start-node`. Wenn ein Dialog beendet ist, weil keine Nachfolgerknoten mehr existieren, wird als Blatt des Dialogbaums ein Verlustknoten erzeugt. Die Funktionen `p-verlust-node` und `o-verlust-node` stellen diese Knoten dar.

Der erste Knoten des Dialogbaums wird durch die Funktion `start-node` gebildet. In diesem Knoten werden alle Hypothesen des Dialogs realisiert. Anschließend werden die Nachfolgersprechakte gebildet. Dazu werden neue Sprechakte erzeugt, indem die Partikelregeln auf alle Hypothesen angewendet werden und zu den resultierenden Sprechakten der Thesensprechakt hinzugenommen wird. Die Rahmenregeln ermitteln die realisierbaren und gesperrten Sprechakte, von denen die realisierbaren durch die Heuristik geordnet werden. Für jeden realisierbaren Sprechakt wird ein Proponentenknoten erzeugt, bis ein Knoten gefunden wird, der erfolgreich ist. Die erzeugten Sprechakte werden nicht explizit verwaltet, sondern sie werden in einer Liste von den Knoten an die Nachfolgerknoten weitergegeben.

Ein Proponentenknoten wird durch die Funktion `p-node` gebildet, der ein zu realisierender Sprechakt sowie die bisher erzeugten Sprechakte übergeben werden. Sie setzt einen Dialogschritt, indem sie die Dialogliste erweitert und den Sprechakt realisiert. Aus diesem und den bisher erzeugten Sprechakten werden mittels der Partikelregeln, der Rahmenregeln und der Heuristik die Nachfolgersprechakte des Opponenten bestimmt. Wenn keine Nachfolger existieren, dann beendet der Proponentenknoten erfolgreich, nachdem durch Aufruf der Funktion `o-verlust-node` ein Verlustsprechakt realisiert wurde. Ansonsten werden für alle Nachfolgersprechakte Opponentenknoten gebildet, die alle erfolgreich zurückkehren müssen, damit der übergeordnete Proponentenknoten erfolgreich beendet werden kann.

Die Funktion `o-node` bildet die Opponentenknoten. Sie realisiert wie die Funktion `p-node` den übergebenen Sprechakt und berechnet aus diesem und den bisher erzeugten Sprechakten, die ebenfalls übergeben werden, die realisierbaren Nachfolgersprechakte. Aus diesen werden wiederum Proponentenknoten gebildet, von denen einer erfolgreich zurückkehren muß, damit der Opponentenknoten erfolgreich ist. Wenn kein Nachfolgersprechakt möglich ist, dann wird die Funktion `p-verlust-node` aufgerufen und der Opponentenknoten scheitert.

Falls ein Opponentenknoten scheitert, muß nicht zum übergeordneten Proponentenknoten zurückgekehrt werden, da dieser ebenfalls scheitert. Vielmehr ist es sinnvoll, zum nächsthöheren Opponentenknoten zu springen. Zu diesem Zweck erzeugen Opponentenknoten *Failure-Continuations*.

Die Aktivierung einer Failure-Continuation bewirkt, daß zu dem Knoten, in dem sie erzeugt wurde, zurückgesprungen wird und dort die nächste Zugmöglichkeit getestet wird. Gleichzeitig wird der ursprüngliche Zustand der Dialogliste durch den Aufruf der Methode `failure` wiederhergestellt. Auch die für die Termwahl und die Primformelübernahme benötigten Daten werden in den Zustand, den sie bei der Erzeugung der Failure-Continuation hatten, versetzt. Jeder Opponentenknoten gibt die von ihm erzeugte Failure-Continuation über die Proponentenknoten an die nächsttieferen Opponentenknoten weiter, die die Failure-Continuation beim Scheitern aufrufen. Dieser Ablauf ist in Abbildung 7.1 wiedergegeben. Die Vorgehensweise zur Bildung von Failure-

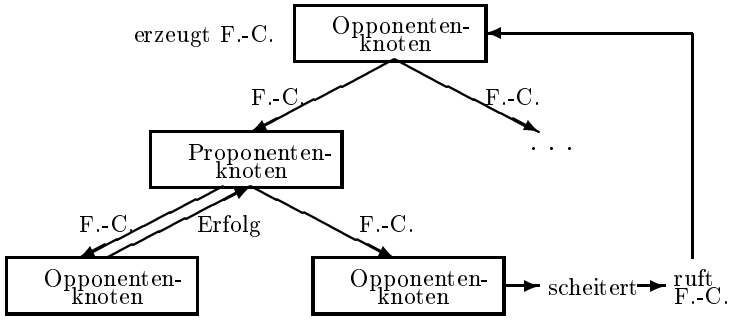

Abbildung 7.1: Verwendung von Failure-Continuations

Continuations und zur Korrektur des Zustands basiert auf dem von HAYNES [10] beschriebenen Verfahren[1].

Failure-Continuations werden auch bei der Primformelübernahme verwendet. Zur Unterscheidung von diesen heißen die beschriebenen Failure-Continuations, die lediglich Sprünge zwischen zwei Opponentenknoten bewirken, O-Failure-Continuations.

Wenn ein Knoten erfolgreich beendet wird, dann gibt er den bisher erzeugten Ausschnitt eines Strategiebaums zurück, nachdem die Dialogliste durch den Aufruf der Methode `return` korrigiert wurde. Proponentenknoten erzeugen dazu eine Liste mit dem realisierten Sprechakt und den Bäumen aller Nachfolgerknoten. Opponentenknoten geben eine Liste mit ihrem Sprechakt und einem Baum eines erfolgreichen Nachfolgerknotens zurück.

Wenn schließlich ein Proponentenknoten erfolgreich zum Startknoten zurückkehrt, so ist sein Rückgabewert eine Gewinnstrategie für das gesamte Dialogspiel. Sie wird in das Objekt `strategie` durch den Aufruf der Methode

`(setze strategie` *gewinnstrategie*`)`

eingetragen und ist damit der Benutzeroberfläche zur Ausgabe und Speicherung zugänglich.

[1] HAYNES beschreibt an gleicher Stelle einen *state-space*, mit dessen Hilfe Zustände korrigiert werden. Dieser state-space stellt ein wesentlich weiteres Konzept dar, als die im Abschnitt 7.2 beschriebene Dialogliste, die nur lineare Zustandsübergänge zuläßt.

7.7.2 Termwahl und Termsymbole

Wenn ein Quantor im Dialogspiel aus einer Behauptung eliminiert wird, muß ein Term angegeben werden, der für die Quantorvariable in die Behauptung eingesetzt wird. Der Proponent muß dabei den Term so wählen, daß er eine Gewinnstrategie finden kann. Er kann aus der Menge aller Terme, die durch die deklarierten Symbole gebildet werden können, dem *Herbrand-Universum*, wählen. Das Herbrand-Universum ist i. allg. eine unendliche Menge, so daß dabei unendlich viele Verzweigungen im Dialogbaum auftreten würden. Aus diesem Grund wählt der Proponent keinen konkreten Term, sondern ein Termsymbol, das für beliebige Terme stehen kann. Erst wenn der Proponent im weiteren Dialog eine Primformel übernehmen möchte und die vom Opponenten gesetzte oder die vom Proponenten zu setzende Primformel Termsymbole enthält, muß festgelegt werden, für welche Terme die Termsymbole stehen. Da es zu jedem Zeitpunkt nur endlich viele vom Opponenten gesetzte Primformeln gibt, existieren auch nur endlich viele Wahlmöglichkeiten. Durch die Übernahme einer Primformel werden Termsymbole an Terme gebunden. Diese Bindungen werden in eine Bindungsliste eingetragen. Ein gebundenes Termsymbol verhält sich so, als wäre es der Term, an den es gebunden wurde.

Wenn der Opponent einen Term wählen soll, dann genügt es, den für den Proponenten schwierigsten Fall zu wählen. Er nennt dazu als Term ein neues Symbol, das bislang nicht im Dialog aufgetreten ist. Kann der Proponent gegen diesen Fall gewinnen, so gewinnt er gegen jede andere Wahl ebenfalls. Als neues Symbol gibt der Opponent ein Termsymbol an. Dieses steht nicht, wie die vom Proponenten eingeführten Termsymbole, für einen konkreten Term, sondern wird als eigenständiges Symbol betrachtet und kann deswegen im weiteren Dialog nicht gebunden werden.

Termsymbole werden von dem Objekt mit dem Namen `termsymbol` verwaltet. Dieses Objekt stellt wie die Variablen- und Konstantensymbolverwalter die Methoden `deklariere`, `undeklariere`, `ist-deklariert?`, `loesche-deklarationen` und `zugehoerige-sorte` zur Verfügung. Bei der Deklaration muß ein Symbolname und ein Sortenname angegeben werden. Die Symbolnamen für neue Termsymbole generiert das Objekt `termsymbolerzeuger`. Es kann beim Start von Dialogspielen durch die Methode

```
(initialisiere termsymbolerzeuger)
```

initialisiert werden. Der Aufruf

```
(neues-symbol termsymbolerzeuger)
```

gibt ein bislang noch nicht verwendetes Termsymbol zurück. Die Termsymbole haben die Form „$n“, wobei n für eine Zahl steht. Aus einem Termsymbol wird ein Termsymbolobjekt durch die Funktion

`(make-termsymbol` *symbolname sortenname*`)`

erzeugt. Dabei wird der Name des Termsymbols und dessen Sorte angegeben. Ein Termsymbolobjekt besitzt die gleichen Methoden wie die übrigen Termobjekte. Wenn das zugehörige Termsymbol

noch keine Bindung hat, verhält sich das Objekt wie ein atomarer Term. Bei einem gebundenen Termsymbol wird ein Methodenaufruf an das gebundene Termobjekt weitergegeben, so daß das Termsymbolobjekt dessen Eigenschaften übernimmt.

Die Termwahl wird von den Proponenten- und Opponentenknoten veranlaßt. Bevor sie ihren Sprechakt realisieren, überprüfen sie, ob ein Term gewählt werden muß. Dies ist der Fall, wenn von den Partikelregeln in den Parameter des Sprechakts ein Variablensymbol eingetragen wurde und der Sprechakt eine Quantorformel angreift oder eine angegriffene Quantorformel verteidigt. Ein Knoten, in dem ein Term gewählt wird, heißt *Erzeugerknoten*. In ihm wird ein neues Termsymbol und ein zugehöriges Termsymbolobjekt erzeugt. Als Sorte des Termsymbols wird die Sorte der Quantorvariable verwendet. Für jedes freie Vorkommen der Quantorvariable wird das Termsymbolobjekt in die Behauptung sowie in den Parameter des Sprechakts eingesetzt. Schließlich wird das Termsymbol mit einer undefinierten Bindung und dem Signum des Spielers, für den es erzeugt wurde, in das Objekt `bindungsliste` eingetragen, das die Bindungen verwaltet. Dies geschieht durch die Methode

`(erweitere bindungliste` *termsymbolname signum*`).`

Anschließend kann der Erzeugerknoten den geänderten Sprechakt realisieren und Nachfolgersprechakte generieren. Das erzeugte Termsymbol bleibt im weiteren Dialogverlauf so lange unberücksichtigt, bis Primformeln übernommen werden sollen.

7.7.3 Primformelübernahme

Das Verfahren der Primformelübernahme wird angewendet, wenn ein Sprechakt des Proponenten realisiert werden soll, in dem eine Primformel behauptet wird. Ziel der Primformelübernahme ist es, unter den vom Opponenten bereits gesetzten Primformeln eine zu finden, die der vom Proponenten zu setzenden entspricht und übernommen werden darf. Wenn die Primformeln keine Termsymbole enthalten, müssen sie identisch sein, damit die Übernahme möglich ist. Andernfalls muß überprüft werden, ob die Termsymbole so durch Terme ersetzt werden können, daß beide Primformeln identisch werden. Das leistet die *Unifikation*. Bei erfolgreicher Unifikation werden die Termsymbole an die gefundenen Terme gebunden. Die Bindungen werden in die Bindungsliste eingetragen, in der die Daten für die Unifikation verwaltet werden. Wenn die Primformeln identisch oder unifizierbar sind, muß getestet werden, ob die Übernahme gemäß den Rahmenregeln erlaubt ist. Verhindert keine Regel die Übernahme, so darf der Proponent die Primformel setzen.

Bei der Strategiebildung wird in jedem Proponentenknoten überprüft, ob in dem zu realisierenden Sprechakt eine Primformel behauptet wird. Ein Knoten, in dem dies der Fall ist, heißt *Binderknoten*. Ein Binderknoten ruft vor der Realisierung des übergebenen Sprechakts die Funktion `primformel-uebernahme`, der er den Sprechakt des Proponenten mit der zu setzenden Primformel und die O-Failure-Continuation, die er vom übergeordneten Opponentenknoten erhalten hat, übergibt. Diese Funktion ermittelt durch die Methode `gesetzte-primformeln-mit-ny` der Dialogliste alle Primformelobjekte des Opponenten und versucht, jedes mit dem Primformelobjekt des

Proponenten zu unifizieren. Ist die Unifikation erfolgreich, so werden die Daten des Oppenentensprechakts, dessen Behauptung die zu übernehmende Primformel ist, ermittelt. Auf diesen Sprechakt und den Sprechakt des Proponenten werden mit Hilfe der Methode `uebernahme-moeglich?` des Objekts `rahmenregel` die Rahmenregeln für die Primformelübernahme angewendet. Falls alle Regeln die Übernahme erlauben, kehrt die Primformelübernahme zum Proponentenknoten zurück, der seinen Sprechakt realisiert und mit der Strategiebildung fortfährt. Scheitern alle Versuche, die Primformel zu übernehmen, so ist auch der Binderknoten gescheitert, weshalb die Primformelübernahme die O-Failure-Continuation aufruft und dadurch zum übergeordneten Opponentenknoten springt. Von dort wird mit einem anderen Nachfolgersprechakt des Proponenten die Strategiebildung fortgesetzt.

7.7.4 Unifikation und Bindungsliste

Ein Unifikationsversuch für zwei Primformelobjekte wird durch den Aufruf der Methode

(`unifiziere-formel` *primformelobjekt*$_1$ *primformelobjekt*$_2$)

gestartet. Diese Methode vergleicht die Hauptsymbole beider Primformelobjekte. Bei verschiedenen Symbolen können die Objekte nicht unifiziert werden. In diesem Fall gibt die Methode den Wert `keine_unif` zurück. Andernfalls wird versucht, jedes Argument des ersten Primformelobjekts mit dem entsprechenden Argument des zweiten Objekts zu unifizieren. Dazu stellen Termobjekte die Methode

(`unifiziere-term` *termobjekt*$_1$ *termobjekt*$_2$)

zur Verfügung. Wenn diese Methode für ein Argumentepaar scheitert, dann ist auch die Unifikation der Primformelobjekte gescheitert. Bei erfolgreicher Unifikation wird eine Liste mit neu gebundenen Termsymbolen zurückgegeben. Die Methode `unifiziere-formel` vereinigt die bei der Unifikation der Argumentepaare erzeugten Listen und gibt die Gesamtliste als Ergebnis der Unifikation zurück.

Die Unifikation von Funktionstermobjekten erfolgt analog der Unifikation von Primformelobjekten. Die Methode `unifiziere-term` vergleicht dabei die Hauptsymbole der Objekte und unifiziert die Argumentepaare. Das Ergebnis einer erfolgreichen Unifikation ist wiederum die Vereinigung der bei der Unifikation der Argumentepaaren erzeugten Listen mit neu gebundenen Termsymbolen. Auf ein Konstantentermobjekt angewendet, ist die Unifikation erfolgreich, wenn das zweite Termobjekt ebenfalls ein Konstantentermobjekt mit demselben Hauptsymbol ist. Der Rückgabewert ist in diesem Fall eine leere Liste, da keine Termsymbole gebunden wurden. Für Variablentermsymbole liefert die Methode `unifiziere-term` immer den Wert `keine_unif`, da in den zu unifizierenden Primformelobjekten keine Variablen mehr enthalten sein dürfen. Wenn ein Funktionstermobjekt oder ein Konstantentermobjekt mit einem Termsymbolobjekt unifiziert werden soll, geben diese den Methodenaufruf mit vertauschten Rollen an das Termsymbolobjekt weiter.

Ein Termsymbolobjekt kann stets mit einem anderen, welches dasselbe Termsymbol enthält, unifiziert werden. Dabei wird kein Termsymbol gebunden, so daß das Ergebnis dieser Unifikation eine leere Liste ist. In allen anderen Fällen, in denen ein Termsymbolobjekt mit einem Termobjekt unifiziert werden soll, müssen Einschränkungen beachtet werden.

Termsymbole, die vom Opponenten eingeführt wurden, dürfen nicht gebunden werden. Ein entsprechendes Termsymbolobjekt kann somit, bis auf den Fall identischer Termsymbole, nur mit einem Termsymbolobjekt des Proponenten unifiziert werden, soweit keine anderen Einschränkungen verletzt werden. Das Termsymbol des Proponenten wird dabei an das Termsymbolobjekt des Opponenten gebunden.

Wenn ein Proponententermsymbol mit einem Termobjekt unifiziert werden soll, muß die Sorte des Termsymbols der Sorte des Termobjekts übergeordnet sein, anderfalls scheitert die Unifikation. Ist das Termobjekt ebenfalls ein Termsymbolobjekt des Proponenten, so kann durch Vertauschen der Objekte diese Einschränkung stets erfüllt werden.

Über die Sortenbeschränkung hinaus muß überprüft werden, ob das Termsymbol in dem Termobjekt als echter Teilterm vorkommt. Das leistet die Methode

`(erweiterter-occur-check?` *termobjekt termsymbol*`)`.

In dieser Methode wird zusätzlich ermittelt, ob in dem Termobjekt ein vom Opponenten eingeführtes Termsymbol enthalten ist, das jünger als das zu überprüfende Termsymbol ist. Ist das Termsymbol selbst oder ein jüngeres Termsymbol des Opponenten in dem Termobjekt enthalten, dann liefert diese Methode den Wahrheitswert `#f` um anzuzeigen, daß eine Unifikation nicht möglich ist.

Das Alter eines Termsymbols wird erstmals bei seiner Erzeugung festgelegt. Alle später erzeugten Termsymbole sind jünger. In der Bindungsliste wird das Alter der Termsymbole gespeichert. Die Methoden

`(termsymbolalter bindungsliste` *termsymbol*`)`
`(ist-aelter? bindungsliste` $termsymbol_1$ $termsymbol_2$`)`

geben das Alter zurück bzw. vergleichen zwei Termsymbole bezüglich ihres Alters. Durch die Beachtung des Alters von Termsymbolen wird sichergestellt, daß kein Proponententermsymbol für ein vom Opponenten später eingeführtes Termsymbol stehen und der Proponent damit die Wahl des Opponenten vorausahnen kann. Es wird nicht verhindert, daß ein Proponententermsymbol an einen Term gebunden wird, der ein jüngeres Termsymbol des Proponenten enthält, wodurch der Proponent seine eigene Wahl vorwegnehmen kann. Allerdings muß dabei sichergestellt werden, daß die Beschränkung durch das Alter auch bei indirekten Bindungen über mehrere Termsymbole eingehalten wird. So könnte beispielsweise ein Proponententermsymbol P_1 an ein Termobjekt mit einem jüngeren Proponententermsymbol P_2 gebunden werden. P_2 könnte in einem zweiten Schritt an ein Termobjekt mit einem Termsymbol O_1 des Opponenten gebunden werden, wobei O_1 älter als P_2, aber jünger als P_1 ist. Das Termsymbol P_1 würde damit für einen Term stehen, der ein

jüngeres Opponententermsymbol enthält, was nicht erlaubt ist. Um dies zu verhindern, wird bei einer Bindung eines Termsymbols an ein Termobjekt das Alter der in dem Termobjekt enthaltenen Termsymbole angepaßt. Alle enthaltenen Termsymbole, die vom Proponenten eingeführt wurden und jünger sind, als das gebundene Termsymbol, erhalten dessen Alter, sie werden also gealtert.

Soweit keine der Beschränkungen des einführenden Spielers, der Sorte und des Alters verletzt sind, ist die Unifikation eines Termsymbolobjekts mit einem Termobjekt erfolgreich. Die erzeugte Bindung wird in die Bindungsliste durch den Aufruf

`(trage-bindung-ein bindungsliste` *termsymbol termobjekt*`)`

eingetragen. Dabei wird das Alter der im Termobjekt enthaltenen Termsymbole angepaßt. Das Termsymbolobjekt übernimmt das Verhalten des mit ihm unifizierten Termobjekts. Es kann dazu mit den Methoden

`(ist-gebunden? bindungsliste` *termsymbol*`)`
`(bindungswert bindungsliste` *termsymbol*`)`

der Bindungsliste feststellen, ob eine Bindung besteht und für welches Termobjekt es stehen soll. Alle Methodenaufrufe werden dann an dieses Termobjekt weitergegeben. Die Ausnahme bildet die Methode `substituiere-termsymbol`. Diese Methode wird von dem Termsymbolobjekt stets selbst bearbeitet.

Damit der Bindungwert eines Termsymbols in einem Schritt festgestellt werden kann, wird beim Eintragen einer neuen Bindung diese Bindung in alle in der Bindungsliste bereits vorhandenen Termobjekte eingesetzt. Das neue gebundene Termsymbol wird also in den früher eingetragenen Termobjekten durch seinen Wert ersetzt.

Am Ende eines Dialogspiels werden in den Sprechakten einer gefundenen Gewinnstrategie für alle enthaltenen Termsymbole ihre Bindungswerte eingetragen und die erzeugten Bindungen durch den Aufruf

`(loesche-bindungen bindungsliste)`

gelöscht.

7.7.5 Failure-Continuations

Bei der Primformelübernahme existieren i. allg. mehrere Primformeln des Opponenten, die übernommen werden können. Die Funktion `primformel-uebernahme` kehrt erfolgreich zu dem Binderknoten zurück, sobald eine Möglichkeit zur Übernahme gefunden wird. Im weiteren Verlauf des Dialogspiels können sich die durch die Unifikation gefundenen Bindungen von Termsymbolen als ungeeignet erweisen. In diesem Fall muß versucht werden, eine andere Primformel zu übernehmen, wodurch andere Bindungen entstehen. Dazu werden Failure-Continuations verwendet. Sie werden bei der Primformelübernahme erzeugt, falls Termsymbole in der Unifikation gebunden wurden. Sie

werden aufgerufen, wenn eine Bindung eines Termsymbols verhindert, daß eine Gewinnstrategie gefunden werden kann. Der in Abbildung 7.2 dargestellte Dialogausschnitt soll dieses Vorgehen verdeutlichen. In diesem Dialog hat der Opponent die Primformeln $P(f(a))$ und $P(a)$ gesetzt.

Opponent	**Proponent**
$P(f(a))$	
$P(a)$	$P(\$1) \wedge P(f(\$1))$
$L?$	

Abbildung 7.2: Dialogausschnitt

Der Proponent hat eine Konjunktion behauptet, von der der Opponent die linke Unterformel als Verteidigung fordert. In dieser Konjunktion ist '$1' ein Termsymbol, das noch keine Bindung hat. Zur Verteidigung hat der Proponent die Wahl, eine der beiden Primformeln $P(f(a))$ oder $P(a)$ zu übernehmen. Wählt er die erste Primformel, so wird das Termsymbol an den Term $f(a)$ gebunden. Dabei wird eine Failure-Continuation erzeugt. Nach der Übernahme ist der Dialog beendet, es muß aber ein zweiter Zweig bearbeitet werden, in dem der Opponent die Konjunktion angreift und die rechte Unterformel als Verteidigung fordert. In diesem Zweig erweist sich die für das Termsymbol gefundene Bindung als ungeeignet, da die Primformel $P(f(f(a)))$ nicht vom Proponenten gesetzt werden darf. Aus diesem Grund wird die erzeugte Failure-Continuation aufgerufen, die das Termsymbol an den Term a bindet, indem im linken Zweig die Primformel $P(a)$ übernommen wird. Mit dieser Bindung ist auch der rechte Dialogzweig erfolgreich.

Da bei jeder Primformelübernahme eine Failure-Continuation erzeugt wird, können i. allg. mehrere Failure-Continuations aufgerufen werden. Die Failure-Continuations werden deshalb in eine Liste eingetragen, so daß der neueste Eintrag am Anfang der Liste steht. Wenn die Bildung einer Gewinnstrategie scheitert, muß die zuletzt hergestellte Bindung korrigiert werden, indem die letzte erzeugte Failure-Continuation aufgerufen wird. Der Aufruf bewirkt, daß der Dialogbaum und die Bindungsliste in den Zustand, den sie bei der Erzeugung der Failure-Continuation hatten, versetzt werden. Durch die Übernahme einer anderen Primformel werden neue Bindungen erzeugt. Wenn alle an dieser Stelle im Dialogbaum möglichen Primformeln getestet wurden und keine der entstehenden Bindungen eine Gewinnstrategie zuläßt, muß die vorletzte Primformelübernahme korrigiert werden, indem die entsprechende Failure-Continuation aufgerufen wird.

Die Bindung eines Termsymbols hat nur für einen Teilbaum des Dialogbaums eine Bedeutung. Außerhalb dieses Ausschnitts ist das Termsymbol unbekannt. Sobald der Teilbaum erfolgreich verlassen wird, ist auch die Bindung erfolgreich und muß nicht mehr korrigiert werden. Die Failure-Continuation zu dieser Bindung darf nicht mehr aufgerufen werden und ist daher zu löschen. Aus diesem Grund werden den Failure-Continuations bei ihrer Erzeugung Kennzeichen angefügt, durch die festgestellt werden kann, wann sie zu löschen sind.

Ein Termsymbol wird von einem Erzeugerknoten als neues Symbol eingeführt und ist daher nur im Teilbaum unterhalb dieses Knotens sichtbar. Da ein Erzeugerknoten das von ihm eingeführte Termsymbol kennt, kann jede Failure-Continuation mit dem Termsymbol, dessen Bindung sie

korrigiert, gekennzeichnet werden. Wenn ein Erzeugerknoten erfolgreich ist, darf er die Failure-Continuation für sein Termsymbol löschen.

Der Sichtbarkeitsbereich für ein Termsymbol ist jedoch nicht immer durch seinen Erzeugerknoten beschränkt. Ist das noch ungebundene Termsymbol in einem Term enthalten, der an ein älteres Termsymbol gebunden wird, so erhält es das Alter dieses Termsymbols. Es wird dadurch behandelt, als wäre es bereits von dem Erzeugerknoten des älteren Termsymbols eingeführt worden. Der neue Sichtbarkeitsbereich ist damit der Teilbaum unterhalb des Erzeugerknotens des älteren Termsymbols. Sobald das Termsymbol gebunden wird, ändert sich sein Alter und der Bereich, in dem es bekannt ist, nicht mehr. Aus diesem Grund werden Failure-Continuations mit dem Termsymbol und dessen Alter gekennzeichnet. Zusammen mit ihrer Kennzeichnung werden sie in eine Liste eingetragen. Anhand der Kennzeichnung kann festgestellt werden, wann eine Failure-Continuation gelöscht werden darf. Dies soll durch ein Beispiel verdeutlicht werden. Sei die Liste

$$((t_1\ 3)\ (t_2\ 2)\ (t_3\ 2)\ (t_4\ 1)\ (t_5\ 2))$$

mit Kennzeichnungen gegeben. Die Failure-Continuations zu den Kennzeichnungen wurden dabei weggelassen. In dieser Liste haben die Termsymbole t_2, t_3 und t_5 dasselbe Alter. Die Bindung des Termsymbols t_5 wurde als erstes erzeugt. Da nur gebundene Termsymbole ihr Alter weitergeben, haben t_2 und t_3 ihr Alter von diesem Termsymbol erhalten. Damit war t_5 ursprünglich das älteste dieser Termsymbole und sein Erzeugerknoten ist im Dialogbaum den Erzeugerknoten der beiden anderen Termsymbole übergeordnet. Wenn der Erzeugerknoten für t_2 erfolgreich endet, darf der entsprechende Eintrag nicht gelöscht werden, da das Termsymbol bis zum Erzeugerknoten für t_5 sichtbar ist. Erst wenn dieser Knoten erfolgreich endet, dürfen alle Einträge, die als Alter den Wert 2 haben, gelöscht werden.

Die Liste mit den gekennzeichneten Failure-Continuations wird im Dialogbaum von den Proponentenknoten nach oben und nach unten weitergegeben. Opponentenknoten erhalten die Liste von den übergeordneten Knoten, geben sie jedoch nicht an die Nachfolgerknoten weiter, da ein Scheitern eines Nachfolgerknotens nicht das Scheitern des Opponentenknotens bewirkt und somit kein Backtracking ausgelöst werden darf. Sie fügen jedoch die vom einem erfolgreichen Knoten zurückgegebene Liste an die vorhandene Liste an und geben diese nach oben weiter.

Die Failure-Continuation-Liste wird erweitert, wenn ein Binderknoten die Übernahme einer Primformel veranlaßt. Die von der Primformelübernahme erzeugten Failure-Continuations werden an den Anfang der Liste gesetzt. Wenn ein Erzeugerknoten erfolgreich endet, entfernt er nach dem beschriebenen Verfahren unnötige Einträge aus der Liste.

Ein gescheiterter Opponentenknoten überprüft, ob in der Liste, die er erhalten hat, ein Eintrag steht. Ist dies der Fall, so ruft er die zuletzt erzeugte Failure-Continuation auf. Dadurch werden die Dialogliste und die Bindungsliste in den Zustand, den sie bei der Erzeugung der Failure-Continuation hatten, versetzt. Dies geschieht, indem bei der Erzeugung der Failure-Continuation beide Zustände durch die Methoden

```
(aktueller-dialog-zustand dialogliste)
(aktueller-bind-zustand bindungsliste)
```

ermittelt werden. Diese Zustände werden gespeichert und beim Aufruf der Failure-Continuation durch die Methoden

`(failure dialogliste` *dialogzustand*`)`
`(wiederherstellen bindungsliste` *bindungszustand*`)`

wiederhergestellt.

Enthält die Liste keine Einträge, so aktiviert der gescheiterte Opponentenknoten die O-Failure-Continuation, die er vom übergeordneten Knoten erhalten hat.

7.8 Benutzerspiel

Im Benutzerspiel kann der Anwender des Programms die Rolle des Proponenten übernehmen. Er entscheidet,

- welche Sprechakte des Proponenten realisiert werden,
- welche Terme bei der Elimination von Quantoren vom Proponenten gewählt werden und
- welche Primformeln übernommen werden.

Die Wahlmöglichkeiten des Opponenten werden wie im normalen Dialogspiel ohne den Einfluß des Benutzers erzeugt. Um eine Gewinnstrategie zu bilden, muß der Benutzer auf die so erzeugten Möglichkeiten geeignete Reaktionen finden.

Das Benutzerspiel wird aktiviert durch den Aufruf der Funktion `benutzerspiel-an` vor einem Dialogspiel. Nach dem Dialogspiel kann es durch die Funktion `benutzerspiel-aus` deaktiviert werden. Diese Funktionen belegen die Variable `benutzerspiel` mit einem Wahrheitswert. In den Opponentenknoten wird dieser Wert überprüft, wenn die untergeordneten Proponentenknoten gebildet werden sollen. Ist das Benutzerspiel aktiviert, dann werden die realisierbaren Proponentensprechakte nicht durch die Heuristik geordnet. Vielmehr wird durch den Aufruf

`(make-sprechaktselektor` *sprechaktliste*`)`

ein neues Objekt erzeugt, in das alle realisierbaren Sprechakte eingetragen werden. Jedes der so erzeugten Objekte besitzt die Methode `benutzerwahl`, die vom Opponentenknoten gerufen wird und den Benutzer mit Hilfe eines Menüs der Text- oder Graphikoberfläche einen Sprechakt auswählen läßt. Der gewählte Sprechakt wird zur Realisierung an den Knoten zurückgegeben. Wenn der mit diesem Sprechakt erzeugte Nachfolgerknoten scheitert, dann kann die Methode erneut aufgerufen werden, um einen weiteren Sprechakt zu wählen. Sobald der Benutzer keinen Sprechakt wählt, scheitert der Opponentenknoten, da kein erfolgreicher Nachfolgerknoten gebildet wurde.

Dasselbe Verfahren wird bei der Übernahme von Primformeln im Benutzerspiel angewendet. Die Funktion zur Primformelübernahme erzeugt durch den Aufruf

(`make-primformelselektor` *o-primformeln p-primformel*)

ein Objekt, in das alle bislang von Opponenten gesetzten Primformeln und die vom Proponenten zu setzende Primformel eingetragen werden. Das Objekt überprüft, welche Opponentenprimformeln mit der Primformel des Proponenten unifizierbar sind und gemäß den Rahmenregeln übernommen werden dürfen. Diese Primformeln werden beim nachfolgenden Aufruf der Methode `benutzerwahl` zur Wahl gestellt. Die gewählte Primformel wird an die Funktion `primformel-uebernahme` zurückgegeben, welche wie im automatischen Spiel die neuen Termsymbolbindungen berechnet und eine Liste mit den Failure-Continuations erzeugt. Wird eine der enthaltenen Failure-Continuations aktiviert, so wird der Zustand zum Zeitpunkt vor der Wahl wiederhergestellt und erneut die Methode `benutzerwahl` des Objekts aufgerufen. Wenn der Benutzer keine Primformel auswählt oder keine übernehmbare Primformel vorhanden ist, scheitert die Übernahme und es wird zu einer früheren Dialogstellung zurückgekehrt.

Für die Termwahl in Erzeugerknoten des Proponenten wird die Funktion

(`benutzer-termwahl` *sprechakt variable sorte*)

aufgerufen, der die zu ersetzende Variable sowie der Sprechakt, in dem sie auftritt, und ihre Sorte übergeben werden. Über die Text- oder Graphikoberfläche erhält der Benutzer die Möglichkeit, einen Term einzugeben. Dem Term muß eine Untersorte der Variablensorte zugeordnet sein. Der eingegebene Term darf keine Variablen enthalten, die Termsymbole des aktuellen Dialogs dürfen verwendet werden. Der gewählte Term wird in den Sprechakt eingesetzt, der anschließend realisiert werden kann. Wenn der Benutzer keinen Term eingibt, wird ein neues Termsymbol generiert und in den Sprechakt eingesetzt.

Am Ende eines Dialogs wird ein Verlustknoten erzeugt. Verlustknoten lassen im Benutzerspiel eine Mitteilung über den Gewinn oder den Verlust des Dialogs aus Sicht des Proponenten erzeugen. Der Benutzer kann dann wählen, ob er weiterspielen oder abbrechen möchte. Spielt er weiter, wird im Dialogbaum zu einer noch nicht bearbeiteten Zugmöglichkeit zurückgegangen und der Dialog fortgesetzt. Zum Abbrechen wird die Variable `benutzerabbruch` auf den Wert `#t` gesetzt. Diese Variable teilt allen Primformel- und Sprechaktselektoren mit, daß keine Elemente mehr ausgewählt werden sollen. Nachfolgende Aufrufe der Methode `benutzerwahl` kehren sofort ohne gewähltes Element zurück, so daß jede weitere Primformelübernahme oder Erzeugung von Proponentenknoten scheitert.

Wenn der Benutzer zu allen Wahlmöglichkeiten des Opponenten erfolgreiche Dialogfortsetzungen gefunden hat, endet das Dialogspiel, und die gefundene Gewinnstrategie wird in das Objekt zur Strategieverwaltung eingetragen. Bricht der Benutzer ab oder wählt er keine weiteren Sprechakte oder Primformeln aus, dann scheitert der Beweisversuch.

7.9 Strategieverwaltung

Die von der Strategiebildung erzeugte Gewinnstrategie wird in dem Objekt `strategie` verwaltet. Dieses Objekt erlaubt es, die Strategie einzutragen und zu löschen, sie in einer Datei zu speichern

und aus einer Datei zu laden. Die Methode

`(setze strategie` *strategiebaum*`)`

wird nach einem erfolgreichen Dialogspiel aufgerufen, um die Gewinnstrategie einzutragen. Der Parameter *strategiebaum* ist ein durch Listen aufgebauter Baum von Sprechakten. Die Methode ersetzt die Sprechakte durch Zeichenketten und speichert diesen Baum. Eine vorher gespeicherte Strategie wird dabei gelöscht. Durch den Aufruf

```
(stringausgabe strategie)
```

kann dieser Baum linear auf der Textoberfläche ausgegeben werden. Die Methode

```
(stringliste strategie)
```

gibt den Baum mit den Zeichenketten zurück. Diese Methode wird von der graphischen Oberfläche bei der Darstellung eines Strategiebaums verwendet.

Zum Speichern der Strategie in einer Datei bzw. zum Laden einer Strategie aus einer Datei können die Methoden

```
(speichere strategie dateiname)
(lade strategie dateiname)
```

verwendet werden. Der Aufruf

```
(loesche strategie)
```

entfernt eine gespeicherte Strategie. Von diesen Methoden sind von der Benutzeroberfläche aus die Aufrufe zum Laden und Speichern sowie zum Ausgeben der Strategie zugänglich.

Kapitel 8

Benutzeroberfläche

Die Benutzeroberfläche übernimmt die Kommunikation mit dem Anwender. Sie nimmt die Eingaben an, übersetzt sie mit Hilfe des Parsers in eine geeignete Darstellung und läßt die Anweisungen ausführen. Die erzeugten Mitteilungen werden aufbereitet und in geeigneter Weise ausgegeben. Im Benutzerspiel erzeugt die Oberfläche Menüs und leitet die Eingaben des Benutzers an die entsprechenden Komponenten weiter.

Die Benutzeroberfläche unterteilt sich in eine Textoberfläche und eine graphische Oberfläche. Die Textoberfläche benötigt für die Ein- und Ausgabe lediglich die von allen DOS- oder UNIX-Systemen zur Verfügung gestellten Möglichkeiten. Sie ist wie die übrigen Komponenten des Systems in Scheme implementiert. Die graphische Oberfläche ist mit den erweiterten Möglichkeiten von STk implementiert und erfordert ein X-Window-System. Sie bietet über die Textoberfläche hinaus eine erweiterte Ein- und Ausgabesteuerung sowie die graphische Darstellung von Strategiebäumen.

8.1 Textoberfläche

8.1.1 Kommandoeingabe

Der Benutzer kann Kommandos auf zwei Arten eingeben. Sie können eingetippt oder aus einer Datei mit mehreren Anweisungen geladen werden. Der erste Modus wird unmittelbar nach dem Starten des Systems aktiviert, indem die Funktion `read-loop` ohne Argumente aufgerufen wird. Diese Funktion liest alle Tastatureingaben, bis die Eingabezeile durch das Drücken der Return-Taste abgeschlossen wird. Die Eingabe wird der Funktion `kommando-ausfuehren` übergeben, welche die Zeichenkette übersetzen läßt und die gewünschte Aktion ausführt. Anschließend wird zur Funktion `read-loop` zurückgekehrt und die nächste Eingabezeile gelesen. Diese Funktion wird erst beendet, wenn durch eine entsprechende Anweisung das gesamte Programm beendet wird.

Im zweiten Eingabemodus wird die Funktion `lade-kommandodatei` aufgerufen, der ein Dateiname übergeben wird. Die Funktion öffnet die Datei und liest jeweils eine durch ein Newline-Zeichen

abgeschlossene Zeile. Die Zeile wird auf dem Bildschirm ausgegeben und dem Kommandoausführer übergeben. Anschließend wird die nächste Zeile gelesen. Nachdem die letzte Zeile der Datei gelesen wurde, wird wieder der erste Eingabemodus aktiviert.

8.1.2 Kommandoausführer

Der Kommandoausführer wird über die Funktion

`(kommando-ausfuehren` *kommandozeile*`)`

aufgerufen, der eine Zeichenkette mit der auszuführenden Anweisung übergeben wird. Die Funktion gibt die Zeichenkette weiter an den Kommandoparser, der ermittelt, welches Kommando ausgeführt werden soll und welche Argumente des Kommandos angegeben wurden. Wenn eine Zeichenkette fehlerhaft ist und nicht vom Kommandoparser akzeptiert wird, dann liefert dieser den Wert `#f`. Der Kommandoausführer gibt dann eine Fehlermeldung aus, welche die Stelle der Zeichenkette bezeichnet, an der ein Fehler gefunden wurde. Bei einer korrekt gebildeten Zeichenkette liefert der Kommandoparser eine Liste mit zwei Elementen: einem Scheme-Symbol und einer Liste mit Parametern. Das Symbol gibt an, welche Funktion bzw. welche Methode eines Objekts aufgerufen werden soll. Die Parameterliste enthält die Werte, die beim Aufruf übergeben werden müssen.

8.1.3 Parser

Der Parser besteht aus zwei Komponenten: dem *Kommandoparser* und dem *Formelparser*. Der Kommandoparser analysiert die vom Benutzer eingegebenen Anweisungen bis auf solche Abschnitte, die Formeln oder Terme darstellen sollen. Diese werden von dem Formelparser ausgewertet. Beide Parser werden von einem *Lexer* unterstützt, der die Zeichenkette in Abschnitte unterteilt, die unmittelbar verarbeitet werden können.

Lexer

Der Lexer zerlegt eine Zeichenkette in *Token*. Token sind einzelne Zeichen oder Zeichenketten, die von den Parsern als kleinste Einheiten verarbeitet werden. Die Zeichen '(', ')' und ',' werden stets als Token erkannt. Alle anderen Token müssen durch diese Zeichen oder Leerstellen voneinander getrennt werden. Dabei ist es unerheblich, wie viele Leerstellen zwischen zwei Token stehen. Insbesondere bei der Eingabe von Formeln muß die Trennung der Token beachtet werden. So wird die Zeichenkette „f(a,b)“ zerlegt in die Abschnitte „f“, „(“, „a“, „,“, „b“ und „)“, die entweder die angegebenen Sonderzeichen enthalten oder durch diese Zeichen voneinander getrennt sind. In der Zeichenkette „ABS(a+b)“ wird „a+b“ als ein einziges Token betrachtet, da die Elemente nicht voneinander getrennt sind. Nur wenn zwischen diese Zeichen Leerstellen eingefügt werden, werden sie als Token „a“, „+“ und „b“ erkannt. Wenn in der Zeichenkette ein Anführungszeichen auftritt,

dann werden die Zeichen ab dem ersten Anführungszeichen bis zu einem zweiten Anführungszeichen oder bis zum Zeilenende als ein einziger Abschnitt betrachtet, wobei die Anführungszeichen zu dem Token gehören.

Der Lexer ist ein Objekt, das den Namen `lexer` trägt. Dieses Objekt wird durch die Funktion `make-lexer` erzeugt. Die zu zerlegende Zeichenkette wird mittels der Methode

`(starte lexer` *zeichenkette*`)`

in den Lexer eingetragen. Diese Methode zerlegt die Zeichenkette in Token, die nacheinander durch Aufrufe der Methode

```
(next-token lexer)
```

angefordert werden können. Wenn das letzte Token der Zeichenkette zurückgegeben wurde, liefern alle weiteren Aufrufe dieser Methode den Wert `end_of_string`. Soll das zuletzt gelesene Token nochmals vom Lexer geliefert werden, muß es mit

`(unget lexer` *letztes-token*`)`

an den Lexer zurückgegeben werden. Die Methode

```
(ruecksetzen lexer)
```

gibt alle bisher gelesenen Token an den Lexer zurück, so daß der nachfolgende Aufruf der Methode `next-token` wieder das erste Token der Zeichenkette liefert.

Wenn eine Zeichenkette fehlerhaft aufgebaut ist und deshalb vom Kommandoparser nicht akzeptiert wird, sollte dem Benutzer mitgeteilt werden, an welcher Stelle der Zeichenkette der Fehler aufgetreten ist. Die Methode

```
(string-mit-position lexer)
```

setzt dazu die Token wieder zu einer Zeichenkette zusammen, wobei hinter dem letzten Token, das angefordert wurde, eine Markierung eingefügt wird. Dieses letzte Token ist dabei nicht das zuletzt angeforderte, sondern das letzte Token der Zeichenkette, das seit dem Eintragen der Zeichenkette in den Lexer mindestens einmal von der Methode `next-token` zurückgegeben wurde. Die Methode `ruecksetzen` verändert diesen Wert nicht. Der Kommandoausführer verwendet die erzeugte Zeichenkette bei der Fehlermeldung, wenn eine Anweisung des Benutzers nicht analysiert werden konnte.

Kommandoparser

Der Kommandoparser analysiert die vom Benutzer eingegebene Anweisung, indem er sie mit vorgegebenen Musterzeichenketten vergleicht. Eine Musterzeile enthält zum einen Token, die in der

Format-element	**Typ**	**akzeptierte Zeichenkette**	**Beispiel**
%s	String	beliebige Zeichenkette ohne Leerstellen und Sonderzeichen	ABS
%n	Nummer	Zeichenkette, die als Zahl interpretiert werden kann	-1.0
%ls	Stringliste	mehrere Stringelemente, durch runde Klammern zusammengefaßt	(nat real)
%ln	Nummernliste	mehrere Nummern, durch runde Klammern zusammengefaßt	(3 4 5)
%r	Repräsentation	wie Stringelemente oder beliebige Zeichenkette mit Leerstellen und Sonderzeichen, die durch Anführungszeichen begrenzt ist	"groesser als"
%f	Formel	Zeichenkette, die eine Formel darstellt	
%t	Term	Zeichenkette, die einen Term darstellt	

Tabelle 8.1: Formatelemente des Kommandoparsers

Anweisung in gleicher Form auftreten müssen. Zum anderen enthält sie Formatelemente, die festlegen, welchen Typ ein Ausdruck an der entsprechenden Stelle der Zeichenkette des Benutzers haben muß. In Tabelle 8.1 sind die möglichen Formatelemente angegeben. Durch die festgelegten Token wird erkannt, welcher Befehl ausgeführt werden soll. Mittels der Formatelemente können die Parameter zu diesem Befehl bestimmt werden. Die Parameter werden zu einer Liste zusammengefaßt, die zusammen mit einem Symbol, das den auszuführenden Befehl bezeichnet, an den Kommandoausführer übergeben wird. Den Formatelementen wird eine positive, ganze Zahl angefügt, um die Position des Parameters in dieser Liste anzugeben. Die Position '0' bezeichnet das erste Element der Parameterliste. Das letzte Element enthält den Parameter mit der höchsten Positionsnummer. Fehlen Positionen, so wird an den entsprechenden Stellen der Wert `undef` in die Parameterliste eingetragen.

Die Musterzeilen für den Kommandoparser sind nicht fest vorgegeben, sondern werden aus einer Datei geladen. Diese Datei muß eine Liste mit Befehlsbeschreibungen enthalten. Jede Befehlsbeschreibung ist selbst eine Liste mit einer Zeichenkette als Musterzeile und einem Scheme-Symbol zur Bezeichnung des Befehls. Der Kommandoparser vergleicht alle Musterzeilen mit der Anweisung des Benutzers, bis eine Übereinstimmung gefunden wird. Sei beispielsweise die Befehlsbeschreibung

```
(list "deklariere konstante %s0 %s1" 'konstante-dekl)
```

in der Datei enthalten. Die Zeichenkette „deklariere konstante epsilon real“ stimmt mit der Musterzeile sowohl in den festgelegten Token als auch in der Art und Anzahl der Parameter überein. Der Parser übergibt somit dem Kommandoausführer die Liste

```
(list 'konstante-dekl (list "epsilon" "real"))
```

mit der Befehlsbezeichnung und der Parameterliste.

Da auch die Musterzeile in Token zerlegt werden muß, existiert außer dem Lexer für die Anweisungen ein Lexer für die Musterzeilen. Dieser Lexer ist ein Objekt mit dem Namen `formatlexer`.

Ein Kommandoparser wird durch den Aufruf

`(make-kommandoparser` *suffix*`)`

erzeugt. Der Parameter *suffix* ist eine Zeichenkette, welche die Endung der Datei mit den Befehlsbeschreibungen enthält. Beim Start des Systems wird mittels dieser Funktion ein Objekt `kommando-parser` generiert, das das Suffix „.kmd“ verwendet. Durch Aufruf der Methode

`(init-trans kommando-parser` *dateiname*`)`

werden die Befehlsbeschreibungen aus einer Datei geladen, wobei an den Dateinamen der Pfad zu dem Verzeichnis `Sprache` und das Suffix des Parsers angefügt werden. Zuvor vorhandene Befehlsbeschreibungen werden dabei gelöscht. Beim Start des Systems wird mit dieser Methode die Datei `deutsch.kmd` geladen.

Wenn der Benutzer eine Befehlszeile eingegeben hat, ruft der Kommandoausführer die Methode

`(parse kommando-parser` *zeichenkette*`)`.

Diese Methode trägt die übergebene Zeichenkette in den Lexer und die erste Musterzeile in den Formatlexer ein. Ist das erste Token der Anweisung das Kommentarzeichen ‘;’ oder der Wert `end_of_string`, dann gibt die Methode das Symbol `'leerkommando` und eine leere Parameterliste an den Kommandoausführer zurück. Ansonsten werden die von beiden Lexern gelieferten Token verglichen. Passen sie nicht zueinander, wird die nächste Musterzeile in den Formatlexer eingetragen und der Befehlszeilenlexer durch die Methode `ruecksetzen` auf das erste Token der Befehlszeile zurückgesetzt.

Eine Übereinstimmung der Befehlszeile und der Musterzeile ist gefunden, wenn alle Token zueinander passen und beide Lexer im gleichen Schritt den Wert `end_of_string` liefern. Wenn keine passende Musterzeile gefunden wird, liefert der Kommandoparser den Wert `#f`.

Treten in der Musterzeile Formatelemente für Formeln oder Terme auf, so ruft der Kommandoparser die Methode `erzeuge-formel` bzw. `erzeuge-term` des Formelparsers auf, der den folgenden Abschnitt der Befehlszeichenkette untersucht und bei Erfolg ein Formel- bzw. Termobjekt liefert, das in die Parameterliste übernommen wird. Entdeckt der Formelparser einen Fehler, so liefert er kein Objekt. Der Kommandoparser testet in diesem Fall die nächste Musterzeile.

Formelparser

Der Formelparser analysiert Zeichenketten, die aus deklarierten Symbolen zum Formel- und Termaufbau sowie Klammern und Kommata zusammengesetzt sind und erzeugt ein entsprechendes Term- oder Formelobjekt. Er ist als prädiktiver Parser realisiert. Das Verfahren der prädiktiven Syntaxanalyse ist in AHO, SETHI, ULLMAN [1, S. 54ff] beschrieben.

Die vom Formelparser akzeptierten Ausdrücke können durch eine Grammatik beschrieben werden, in der Terminalsymbole und Nichtterminalsymbole enthalten sind. Die für Formeln und Terme verwendete Grammatik ist in Tabelle 8.2 dargestellt. Die Terminalsymbole sind in dieser Tabelle her-

Formel	::=	(Formel || **Präfixjunktor** Formel ||
		Quantorsymbol Variablensymbol Formel ||
		Präfixprädikatsymbol Argumenteliste Restformel ||
		Präfixfunktionssymbol Termliste Restterm Primformelrest ||
		Konstantensymbol Restterm Primformelrest ||
		Variablensymbol Restterm Primformelrest ||
		Termsymbol Restterm Primformelrest
Argumenteliste	::=	(Term Restliste || ϵ
Restliste	::=	, Term Restliste || ϵ
Restformel	::=	) Restformel || **Infixjunktor** Formel || ϵ
Primformelrest	::=	**Infixprädikatsymbol** Term Restformel
Term	::=	(Term || **Präfixfunktionssymbol** Termliste Restterm ||
		Konstantensymbol Restterm ||
		Variablensymbol Restterm ||
		Termsymbol Restterm
Termliste	::=	(Term Restliste
Restterm	::=	) Restterm || **Infixfunktionssymbol** Term || ϵ

Tabelle 8.2: Grammatik für den Formelparser

vorgehoben. Wenn die Grammatik vorgibt, daß ein Terminalsymbol gelesen werden soll, wird das nächste Token des Ausdrucks gelesen und mit dem Terminalsymbol verglichen. Die Terminalsymbole geben an, welchen Typ das Token haben muß. Zu jedem Nichtterminalsymbol der Grammatik existiert eine Funktion, die aufgerufen wird, wenn ein Ausdruck zu diesem Nichtterminalsymbol gelesen werden soll. Wahlmöglichkeiten in den Regeln der Grammatik entsprechen Verzweigungen in diesen Funktionen. Die Grammatik für Formeln und Terme wurde so gewählt, daß auf Grund des nächsten Tokens des Ausdrucks entschieden werden kann, welche Verzweigung zu wählen ist. Voraussetzung dafür ist, daß zu jedem Symbol eindeutig festgelegt ist, welchen Typ es hat. Dies wird bereits bei der Deklaration von Symbolen für den Formel- und Termaufbau verhindert, indem gleichnamige Symbole abgelehnt werden. Auf diese Weise wird vermieden, daß mehrere Verzweigungsmöglichkeiten getestet werden müssen. In der Grammatik sind ϵ-Regeln enthalten, die kein Token lesen. Diese Regeln werden nur verwendet, wenn keine andere Wahlmöglichkeit zutrifft. Das

Token, das gelesen wurde, um über die Wahlmöglichkeiten zu entscheiden, wird in diesem Fall an den Lexer zurückgegeben.

Im Verlauf des Parsens werden Term- und Formelobjekte erzeugt, die später evtl. durch Operatoren zu weiteren Objekten verknüpft werden. Dazu müssen die bislang erzeugten Objekte und die gelesenen Operatoren in Listen gespeichert werden. Eine Argumenteliste enthält die erzeugten Objekte, eine Operatorenliste die gelesenen Operatoren. Um feststellen zu können, welcher Operator anzuwenden ist, müssen zusätzlich die Bindungen der Operatoren in eine Bindungenliste[1] eingetragen werden. Die Listen werden als FIFO-Listen behandelt. Elemente werden an den Anfang einer Liste angefügt und vom Anfang der Liste weggenommen.

Wenn ein Terminalsymbol gelesen wird, dann wird eine zu diesem Symbol gehörende Funktion aufgerufen, die entscheidet, ob ein Objekt erstellt werden soll. Bei Konstanten- und Termsymbolen kann stets ein Objekt erzeugt werden, bei Variablensymbolen nur, wenn das Symbol in einem Term und nicht als Quantorvariable in einer Formel auftritt. Ein erzeugtes Objekt wird in die Argumenteliste eingetragen.

Bei Operatoren wie Funktions- Prädikat-, Junktor- und Quantorsymbolen haben die Bindungen die entscheidende Bedeutung für das Erzeugen von Objekten. Durch sie wird gewährleistet, daß die Operatoren in der richtigen Reihenfolge auf die Argumente angewendet werden. Das verwendete Verfahren basiert auf der in AHO, SETHI, ULLMAN [1, S. 247ff] beschriebenen Syntaxanalyse mit Operatorenvorrang.

Wenn bereits Argumente und Operatoren in den jeweiligen Listen enthalten sind und ein neuer Operator gelesen wird, dann müssen die Bindungen verglichen werden, um zu entscheiden, ob der neue Operator oder der erste der Operatorenliste das zuletzt erstellte Objekt bindet. Ist die linksseitige Bindung des neuen Operators kleiner oder gleich der rechtsseitigen Bindung des früheren Operators, so darf der frühere Operator gemäß seiner Stelligkeit auf die letzten erzeugten Argumente angewendet werden, um ein neues Objekt zu erzeugen. Der angewendete Operator, seine Bindung und die verwendeten Argumente werden aus den jeweiligen Listen entfernt und das neu erzeugte Objekt wird in die Argumenteliste eingetragen. Der Vergleich der Bindungen und das Erzeugen von Objekten werden wiederholt, wenn noch weitere Operatoren in der Operatorenliste enthalten sind. Dies geschieht so lange, bis ein Operator in der Operatorenliste auftritt, dessen Rechtsbindung kleiner ist als die Linksbindung des neu gelesenen Operators oder bis keine Operatoren mehr in der Operatorenliste sind. Erst dann wird der neue Operator in die Liste eingetragen und seine Rechtsbindung in der Bindungenliste vermerkt. Beim nächsten gelesenen Operator kann diese Bindung mit der Linksbindung des neuen Operators verglichen werden.

Über die Bindungen hinaus muß verhindert werden, daß Funktionssymbole Formelobjekte verknüpfen. Aus diesem Grund wird zwischen der Auswertung von Formeln und der Auswertung von Termen unterschieden. Die Auswertung von Termen hat stets Vorrang, auch wenn die Bindungen eine andere Reihenfolge erzeugen würden. Wenn ein vollständiger Term gelesen wurde

[1] Diese Bindungenliste in nicht zu verwechseln mit der Bindungsliste, die bei der Strategiebildung verwendet wird. Während diese die für die Unifikation benötigten Daten aufnimmt, enthält die hier beschriebene Bindungenliste die Bindungen von Operatoren.

und keine erlaubte Fortsetzung des Terms gefunden wird, dann kann mit Hilfe der speziellen Bindung `term_ende_bindung` die Auswertung des gesamten Terms veranlaßt werden. Diese Bindung ist kleiner als alle Funktionssymbolbindungen und bewirkt, daß jedes Funktionssymbol in der Operatorenliste stärker bindet und somit ausgewertet wird.

Nicht alle Operatoren besitzen zwei Bindungen. In diesem Fall werden für die verschiedenen Operatorentypen Standardbindungen verwendet, welche die richtige Reihenfolge bei der Auswertung sicherstellen. Klammern werden ebenfalls Standardbindungen zugeordnet, damit sie die Reihenfolge der Operatorenanwendung beeinflussen können. Wenn eine öffnende Klammer gelesen wird, soll kein Operator angewendet werden, da erst der Ausdruck in den Klammern ausgewertet werden muß. Eine öffnende Klammer wird deshalb sofort in die Operatorenliste eingetragen und als Rechtsbindung der Wert `klammer_auf_bindung` in der Bindungenliste vermerkt. Diese Bindung ist kleiner als alle anderen Bindungen, so daß die Klammer nicht als Operator angewendet wird. Wird eine schließende Klammer gelesen, so müssen alle Operatoren bis zu der öffnenden Klammer ausgewertet werden. Als Linksbindung der schließenden Klammer wird deshalb der Wert `klammer_zu_bindung` verwendet, der kleiner als alle Bindungen außer `klammer_auf_bindung` ist. Dieser Wert führt beim Vergleich der Bindungen dazu, daß alle nach der öffnenden Klammer gelesenen Operatoren stärker binden und damit ausgewertet werden. Als erster nicht angewendeter Operator verbleibt die öffnende Klammer, die zusammen mit ihrer Bindung aus den entsprechenden Listen entfernt wird, wodurch die Auswertung der Klammern beendet ist. Dieses Verfahren gewährleistet, daß Klammern stets paarweise auftreten müssen, auch wenn in der Grammatik beliebig viele schließende Klammern gelesen werden können. Das Verfahren funktioniert ebenfalls, wenn eine Liste von Argumenten für ein Präfixfunktionssymbol oder ein Präfixprädikatsymbol ausgewertet werden soll. Statt eines neuen Objekts werden dabei mehrere Objekte erzeugt, die als Argumente des Symbols verwendet werden. Die Kommata, welche die Argumentterme in der Zeichenkette trennen, werden beim Parsen ohne Aktion überlesen.

Wenn Präfixsymbole wie Präfixfunktions-, Präfixprädikat- Präfixjunktor- oder Quantorsymbole gelesen werden, dann werden diese ohne Vergleich mit bereits gelesenen Operatoren in die Operatorenliste eingetragen, da kein Argument links von ihnen steht, das von anderen Operatoren gebunden werden kann. Bei Präfixjunktor- und Quantorsymbolen wird als Rechtsbindung die bei der Deklaration als einzige Bindung angegebene Zahl in die Bindungenliste eingetragen. Präfixfunktions- und Präfixprädikatsymbolen folgt eine geklammerte Liste von Argumenten. Die Klammern stellen sicher, daß die Argumente ausgewertet werden, bevor das Präfixsymbol angewendet werden kann. Sobald ein Operator nach der schließenden Klammer gelesen wird, kann das Präfixsymbol auf die Argumente angewendet werden. Dazu muß für Präfixfunktions- und Präfixprädikatsymbole eine Rechtsbindung verwendet werden, die die Argumente stärker bindet als alle anderen Operatoren. In die Bindungenliste werden deshalb für solche Symbole die Werte `praefixfunktion_bindung` bzw. `praedikatsymbol_bindung` eingetragen. Beide Werte sind identisch und größer als alle anderen Bindungen.

Dieser Wert wird ebenfalls als Bindung für Infixprädikatsymbole verwendet. Wenn ein Infixprädikatsymbol gelesen wird, dann steht ein Term links von ihm. Dieser Term wurde ausgewertet, da

das Prädikatsymbol keine Fortsetzung eines Terms sein kann. Bei den Ausdrücken nach dem Prädikatsymbol wird durch den Vorrang der Termauswertung sichergestellt, daß die Bindung des Infixprädikatsymbols erst beim Lesen des nächsten logischen Symbols betrachtet wird. Da sie größer ist als alle anderen Bindungen, wird dann das Prädikatsymbol auf die Argumente angewendet.

Wenn das Ende einer Formel erreicht ist und keine gültige Fortsetzung gelesen wird, kann wie bei Termsymbolen mittels der Bindung `formel_ende_bindung` die Auswertung aller Operatoren veranlaßt werden.

In Tabelle 8.3 sind die verwendeten Bindungen zusammengefaßt. Wenn Symbole ohne Linksbin-

Symbolart	**linksseitige Bindung**	**rechtsseitige Bindung**
Infixfunktionssymbol	0<Linksbindung<1000	0<Rechtsbindung<1000
Präfixfunktionssymbol		1001 (`praefix_funktion_bindung`)
Infixprädikatsymbol		1001 (`praedikatsymbol_bindung`)
Präfixprädikatsymbol		1001 (`praedikatsymbol_bindung`)
Infixjunktorsymbol	0<Linksbindung<1000	0<Rechtsbindung<1000
Präfixjunktorsymbol		0<Bindung<1000
Quantorsymbol		0<Bindung<1000
öffnende Klammer		-2 (`klammer_auf_bindung`)
schließende Klammer	-1 (`klammer_zu_bindung`)	
Termende	0 (`term_ende_bindung`)	
Formelende	0 (`formel_ende_bindung`)	

Tabelle 8.3: Bindungen für den Formelparser

dung gelesen werden, müssen keine Auswertungen durchgeführt werden oder sind bereits durch den Vorrang der Termauswertung ausgeführt worden. Bei Einträgen ohne rechtsseitige Bindung werden Auswertungen ausgelöst, aber keine neuen Operatorsymbole in die Operatorenliste eingetragen.

Die Funktionen zum Parsen von Formeln und Termen sind zu einem Objekt mit dem Namen `formel-parser` zusammengefaßt. Dieses Objekt besitzt die Methoden

```
(erzeuge-formel formel-parser)
(erzeuge-term formel-parser),
```

welche nach dem beschriebenen Verfahren die vom Lexer gelieferten Token als Formelelemente und Termelemente interpretieren und auswerten. Wenn ein Token gelesen wird, das auf Grund der Grammatik keine Fortsetzung einer Formel oder eines Terms sein kann, dann wird das Verfahren beendet. Das Verfahren ist erfolgreich, wenn am Ende genau ein Objekt in der Argumenteliste enthalten und die Operatorenliste leer ist. Das erzeugte Objekt wird als Ergebnis zurückgegeben. Wenn während des Parsens ein Fehler auftritt, weil beispielsweise ein Funktionssymbol auf Argumente mit falschen Sorten angewendet werden soll, oder wenn am Ende eine falsche Anzahl von Elementen in der Argumenteliste oder der Operatorenliste enthalten ist, liefern die Methoden den Wert `undef` zurück.

8.1.4 Ausgabesteuerung

Die Ausgabesteuerung bearbeitet alle im Programm erzeugten Meldungen. Sie bereitet die Meldungen auf und gibt sie in geeigneter Weise aus. Die Aufbereitung und die Ausgabe sind modifizierbar und somit an die Bedürfnisse des Benutzers oder an verschiedene Oberflächen anpaßbar. Die graphische Oberfläche nutzt einige dieser Möglichkeiten, um die Art der Ausgabe zu ändern.

Meldungen werden erzeugt, indem ein Kennzeichen für die Meldung und Parameter der Meldung festgelegt werden. Ein Übersetzungsobjekt ermittelt den Text zu dem Kennzeichen und trägt die Parameter an den gewünschten Stellen ein. Die Texte zu den Meldungen sind nicht festgelegt, sondern werden wie die Befehlsbeschreibungen des Kommandoparsers aus einer Datei geladen. Übersetzungsobjekte werden durch den Aufruf

(`make-translator` *suffix*)

erzeugt. Sie besitzen die Methode

(`init-trans` *uebersetzungsobjekt dateiname*),

mit der eine Übersetzungstabelle geladen werden kann. An den Dateinamen werden dabei der Pfad zu dem Verzeichnis `Sprache` und das bei der Erzeugung des Objekts angegebene Suffix angefügt. In der Datei muß eine Liste mit Meldungsbeschreibungen enthalten sein. Meldungsbeschreibungen sind Listen mit einem Scheme-Symbol und einer Zeichenkette. Die Symbole dienen zur Identifikation der Meldungen und müssen deshalb eindeutig sein. Die Zeichenkette enthält den Text der Meldung, in den bei der Übersetzung die Parameter eingetragen werden. Die Stellen, an denen die Parameter eingefügt werden sollen, sind durch Formatelemente der Form „~n“ gekennzeichnet, wobei für n eine der Ziffern '0' bis '9' stehen kann.

Um eine Meldung zu übersetzen, wird die Methode

(`translate` *uebersetzungsobjekt kennzeichen parameterliste*)

aufgerufen. Der Wert *kennzeichen* enthält das Symbol zur Identifikation der Meldung, die Parameterliste beinhaltet die Parameter in Form von Zeichenketten. Die Methode ermittelt den Meldungstext zu dem Kennzeichen und trägt die Parameter für die Formatelemente ein. Die Ziffer jedes Formatelements gibt wieder, welcher Parameter an dieser Stelle eingetragen werden soll, wobei die Ziffer '0' für das erste Element der Liste steht. Der so aufbereitete Meldungstext wird als Ergebnis der Methode zurückgegeben.

Die Ausgabe der Meldung wird von Ausgabeobjekten übernommen, die durch die Funktion

(`make-ausgeber` *praefix uebersetzungsobjekt*)

erzeugt werden. Diese Objekte besitzen die Methode

(`ausgabe` *ausgabeobjekt kennzeichen parameterliste*).

Sie gibt das Kennzeichen und die Parameterliste an das bei der Erzeugung angegebene Übersetzungsobjekt weiter und erhält den Meldungstext zurück. Dieser wird über eine Ausgabefunktion angezeigt. Die Ausgabeobjekte besitzen eine Standardausgabefunktion, die zur Unterscheidung der Meldungen verschiedener Objekte das dem Ausgabeobjekt zugeordnete Präfix an die Meldung anfügt und diese Zeichenkette auf der Textoberfläche ausgibt. Die verwendete Ausgabefunktion jedes Ausgebers kann durch die Methode

`(neue-ausgabefunktion` *ausgabeobjekt funktion*`)`

geändert werden. Für den Parameter *funktion* muß dabei eine Scheme-Funktion mit einem Argument angegeben werden. Die Standardausgabefunktion wird durch den Aufruf

`(standardausgabefunktion` *ausgabeobjekt*`)`

wiederhergestellt. Über die Ausgabefunktion können die Ausgabeobjekte an verschiedene Oberflächen angepaßt werden, ohne die Objekte selbst zu ändern.

Durch die Methoden

`(schalte-ein` *ausgabeobjekt*`)`
`(schalte-aus` *ausgabeobjekt*`)`

werden Ausgabeobjekte aktiviert oder deaktiviert. Nur im aktivierten Zustand werden Meldungen tatsächlich ausgegeben, im deaktivierten Zustand werden sie unterdrückt. Neu erzeugte Ausgabeobjekte sind aktiviert.

Beim Start des Systems werden Objekte zur Ausgabe von Mitteilungen, Fehlermeldungen, Kontrollmeldungen, Sprechakten des Opponenten und Sprechakten des Proponenten erzeugt. Diese Objekte sind in Tabelle 8.4 wiedergegeben. Das Ausgabeobjekt `kontrollmeldung` dient zur Aus-

Übersetzungsobjekt	**Dateisuffix**	**Ausgabeobjekt**	**Meldungspräfix**
`mitteilungsuebersetzer`	„.mtt“	`mitteilung`	„==> “
`benutzerfehleruebersetzer`	„.ber“	`benutzerfehler`	„FEHLER: “
`kontrollmeldungsuebersetzer`	„.ktr“	`kontrollmeldung`	„Kontrollmeldung: “
`p-sprechaktuebersetzer`	„.spa“	`p-sprechakt`	„PRO“
`o-sprechaktuebersetzer`	„.spa“	`o-sprechakt`	„OPP“

Tabelle 8.4: Übersetzungs- und Ausgabeobjekte

gabe von Informationen über den Programmablauf. Es wird unmittelbar nach der Erzeugung deaktiviert und kann zum Testen des Programms aktiviert werden.

Neben diesen existieren die Objekte `direkt-uebersetzer` und `direkt-ausgeber`, die getrennt voneinander benützt werden können. Im Unterschied zu den übrigen Ausgabeobjekten besitzt der Direktausgeber keine Methode `ausgabe`, sondern die Methode

`(anzeige direkt-ausgeber` *zeichenkette*`)`,

welche die übergebene Zeichenkette ohne Übersetzung ausgibt. Diese Objekte werden verwendet, wenn die Aufbereitung und die Ausgabe von Meldungen zeitlich getrennt sind. Der Direktübersetzer verwendet als Dateisuffix die Zeichenkette „.dir“, der Direktausgeber hat kein Meldungspräfix.

Die Übersetzungstabellen der Übersetzer für Mitteilungen, Kontrollmeldungen, Fehlermeldungen und für den Direktübersetzer können zusammen durch die Funktion

`(lade-sprache` *dateiname*`)`

geändert werden. Die zu ladenden Dateien müssen sich in dem Verzeichnis `Sprache` befinden. Beim Start des Systems werden deutsche Meldungstexte geladen, indem diese Funktion mit dem Dateinamen „deutsch“ aufgerufen wird. Die Übersetzer für Sprechakte werden zu Beginn mit den Dateien `o_sprech.spa` bzw. `p_sprech.spa` initialisiert.

8.1.5 Eingabemenüs

Die Eingabemenüs der Textoberfläche werden beim Benutzerspiel verwendet, um Primformeln oder Sprechakte auswählen und Terme eingeben zu können. Die Funktion

`(benutzermenue` *ueberschrift eintraege gewaehlte-eintraege*`)`

zeigt ein Menü an, in dem ein Eintrag ausgewählt werden kann. Die Überschrift des Menüs wird als Zeichenkette übergeben. Das Argument *eintraege* ist eine Liste von Zeichenketten, aus denen eine ausgewählt werden soll. Diese Einträge werden im Menü mit Indizes beginnend ab ‘0’ numeriert und können vom Benutzer durch Eingabe der entsprechenden Nummer selektiert werden. Der Parameter *gewaehlte-eintraege* ist eine Liste mit Indizes. Wenn der Index eines Eintrags in der Liste enthalten ist, wird dieser Eintrag im Menü durch einen Stern gekennzeichnet. Der Benutzer kann im Menü einen gültigen Index oder eine leere Zeichenkette eingeben. Im ersten Fall ist das Ergebnis der Funktion der gewählte Index, im zweiten Fall wird eine leere Liste zurückgegeben, die wiedergibt, daß kein Eintrag ausgewählt wurde. Wenn bei der Primformelselektion keine übernehmbaren Primformeln vorhanden sind, wird statt des Eingabemenüs ein leeres Menü durch den Aufruf

`(leermenue` *menuetext*`)`

erzeugt. Diese Funktion gibt den Text aus und wartet auf die Bestätigung des Benutzers.

Die Funktion

`(termeingabe` *ueberschrift sortenname*`)`

erlaubt dem Benutzer, einen Term einzugeben. Sie akzeptiert eine Eingabe des Benutzers und generiert mittels der Methode `erzeuge-term` des Formelparsers ein Termobjekt. Es wird überprüft, ob dem Termobjekt eine Untersorte des übergebenen Sortennames zugeordnet ist, ob es einen geschlossenen Term darstellt und ob nur gültige Termsymbole in dem Term vorkommen. Ist eine

dieser Bedingungen nicht erfüllt, so wird eine neue Eingabe verlangt. Das Ergebnis der Funktion ist ein Termobjekt, das alle Bedingungen erfüllt, oder eine leere Liste, wenn kein Term eingegeben wurde.

Wenn ein Dialog beendet ist, dann werden von den Verlustknoten die Funktionen

```
(benutzer-gewinn-information)
(benutzer-verlust-information)
```

der Textoberfläche aufgerufen. Diese Funktionen geben eine entsprechende Meldung aus und lassen den Benutzer wählen, ob er weiterspielen oder abbrechen möchte. Wählt der Benutzer den Abbruch des Spiels, so wird das Kennzeichen `benutzerabbruch` gesetzt.

8.2 Graphische Oberfläche

Die graphische Oberfläche erweitert die Möglichkeiten der Textoberfläche. Dazu werden einige der Funktionen der Textoberfläche ersetzt, um eine graphische Darstellung zu ermöglichen. Auch die Funktionen zur Eingabe von Anweisungen müssen geändert werden, da die graphische Oberfläche im Unterschied zur Textoberfläche ereignisorientiert arbeitet. In der Textoberfläche ist stets eine Funktion aktiv, die den weiteren Ablauf bestimmt. Die Funktionen der graphischen Oberfläche enden hingegen nach der Ausführung einer Anweisung des Benutzers. Die Funktionen werden aufgerufen, wenn durch Aktionen des Benutzers bestimmte Ereignisse eintreten. Dadurch ist es beispielsweise möglich, Befehle nicht durch den Kommandoausführer aufzurufen, sondern über eine Menüleiste zu aktivieren. Den Menüeinträgen werden dazu festgelegte Aufrufe von Methoden oder Funktionen anderer Komponenten des Systems zugeordnet, die beim Auswählen der Menüeinträge ausgeführt werden. Die auf diese Weise aktivierbaren Befehle sind in Abschnitt 9.4 beschrieben.

Für die Ausgabe von Strategien stellt die graphische Oberfläche darüber hinaus neue Objekte und Methoden bereit, die in der Textoberfläche nicht verfügbar sind.

8.2.1 Textfeld

Das Textfeld integriert die Möglichkeiten der Textoberfläche in die graphische Umgebung. In diesem Feld werden die Anweisungen des Benutzers eingegeben und Meldungen des Programms ausgegeben. Im Unterschied zur Textoberfläche werden keine Sprechakte von Dialogspielen und keine Eingabemenüs dargestellt, da hierfür andere Felder verwendet werden.

Für die Ausgabe in das Textfeld müssen die Ausgabefunktionen der Ausgabeobjekte für Mitteilungen, Fehlermeldungen und Kontrollmeldungen sowie des Direktausgebers geändert werden. Dazu wird beim Start der graphischen Oberfläche die Methode `neue-ausgabefunktion` jedes dieser Objekte aufgerufen und eine geänderte Funktion eingetragen. Die neuen Ausgabefunktionen schreiben die Meldungen in das Textfeld und korrigieren den dargestellten Bereich des Felds, so daß die Meldung sichtbar ist.

Auch die von der Textoberfläche verwendete Funktion `read-loop` wird geändert, da diese Funktion auf die Eingabe des Benutzers wartet und dadurch die Verwendung der Menüleiste verhindern würde. Diese Funktion wird ersetzt durch eine Funktion, die lediglich eine Eingabemarke ausgibt, die Zeile, in der diese Marke steht, vermerkt und anschließend beendet wird. Zum Lesen der Eingabe wird eine Event-Funktion `kommandoeingabe` verwendet. Event-Funktionen werden in STk aufgerufen, sobald ein festgelegtes Ereignis eintritt. Die Funktion `kommandoeingabe` wird aktiviert, wenn im Textfeld die Return-Taste gedrückt wird. Sie holt die Zeichenkette ab der Eingabemarke und gibt diese an den Kommandoausführer weiter. Nachdem die Anweisung ausgeführt wurde, ruft sie die Funktion `read-loop` und wird schließlich beendet.

8.2.2 Dialogausgabe

Zur Ausgabe von Sprechakten stellt die graphische Oberfläche zwei Möglichkeiten zur Verfügung. Das *Tracefeld* zeigt wie die Textoberfläche die realisierten Sprechakte untereinander an. Diese Darstellung eignet sich vor allem für das automatische Spiel, um den Ablauf des Dialogspiels zu verfolgen. Die erfolgreichen und gescheiterten Dialoge können nach einem Dialogspiel anhand der Ausgaben in diesem Feld nachvollzogen werden.

Das *Dialogtableau* eignet sich für die Verwendung im Benutzerspiel. Es gibt den aktuellen Dialog wieder, wobei die Sprechakte der beiden Spieler in getrennten Feldern nebeneinander dargestellt werden. Wenn ein Dialog scheitert und zu einer Verzweigung im Dialogbaum zurückgegangen wird, dann werden die erfolglosen Sprechakte aus dem Tableau gelöscht, so daß lediglich der Anfang des Dialogs bis zu der Verzweigungsstelle erhalten bleibt. Die neuen Sprechakte setzen dieses Dialogstück fort. Am Ende eines Dialogspiels ist somit nur der zuletzt geführte Dialog im Tableau enthalten.

Die Dialogausgabefelder können über die Menüleiste ein- und ausgeblendet werden. Das Dialogtableau wird darüber hinaus angezeigt, wenn die Dialogausgabe oder das Benutzerspiel aktiviert wird. Zum Ein- und Ausblenden der Felder werden die Funktionen

```
(pack-tableau)
(pack-tracefeld)
(unpack-tableau)
(unpack-tracefeld)
```

aufgerufen. Die ersten beiden Funktionen zeigen die entsprechenden Felder an, die letzten beiden Funktionen entfernen sie.

Um die Sprechakte in den Ausgabefeldern anzeigen zu können, werden die Ausgabefunktionen der Objekte `p-sprechakt` und `o-sprechakt` durch den Aufruf der Methode `neue-ausgabefunktion` geändert. Die neuen Funktionen überprüfen bei der Ausgabe, welche Ausgabefelder aktiv sind und tragen die Sprechakte ein. In das Tracefeld wird ein neuer Sprechakt von beiden Ausgebern als letzter Sprechakt eingetragen und der sichtbare Bereich des Felds verschoben, so daß der neue Eintrag sichtbar ist. In das Dialogtableau werden Sprechakte von beiden Ausgabeobjekten

auf unterschiedliche Weise eingetragen. Die Proponentensprechakte werden in das rechte Feld des Dialogtableaus eingetragen, die Sprechakte des Opponenten in das linke Feld. Um zu gewährleisten, daß ein Sprechakt eines Spielers unterhalb des vorhergehenden Sprechakts des anderen Spielers steht und diese Ordnung auch bei Änderung der Schriftart nicht verschoben wird, trägt jedes Ausgabeobjekt den Sprechakt auch in das Feld des anderen Spielers ein. Allerdings korrigiert er die Schriftfarbe des Eintrags, so daß dieser nicht sichtbar ist und lediglich zum Einhalten des Abstands dient.

Darüber hinaus muß die Ordnung von Sprechakten gewährleistet werden, wenn im Dialog zu einer früheren Stellung zurückgegangen und eine andere Dialogfortsetzung gewählt wird. Bevor ein neuer Sprechakt eingetragen wird, müssen die letzten Sprechakte bis zu dem, der durch den neuen Sprechakt ersetzt werden soll, gelöscht werden. Die Ausgabefunktionen werden dabei von dem Objekt `tableauverwalter` unterstützt. Dieses Objekt besitzt die Methoden

```
(initialisiere-tableau tableauverwalter)
(korrigiere-tableau tableauverwalter).
```

Die erste Methode wird beim Start eines Dialogspiels zum Löschen des Tableaus verwendet. Die zweite Methode wird von den Ausgabefunktionen vor der Ausgabe eines Sprechakts aufgerufen. Die Methode stellt anhand der Dialogliste die Identifikation des realisierten Sprechakts fest und überprüft, ob ein Sprechakt mit dieser Identifikation bereits in das Tableau eingetragen wurde. Falls dies zutrifft, werden ab diesem Sprechakt alle Einträge gelöscht. Anschließend kann der neue Sprechakt von der Ausgabefunktion eingetragen werden.

8.2.3 Eingabemenüs

Die Eingabemenüs der graphischen Oberfläche sind in einem eigenen Fenster enthalten. Bei deaktiviertem Benutzerspiel wird dieses Fenster als Icon dargestellt. Erst wenn das Benutzerspiel über die Menüleiste oder durch eine Anweisung aktiviert wird, erscheint das Fenster in normaler Größe. Die Darstellungsform des Fensters wird durch die Funktion

```
(pack-menues)
(unpack-menues)
```

geändert. Die Funktion `unpack-menues` verkleinert das Fenster zu einem Icon, die Funktion `pack-menues` stellt die normale Größe des Fensters her.

Die Funktionen `benutzermenue`, `leermenue`, `termeingabe`, `benutzer-gewinn-information` und `benutzer-verlust-information` erzeugen die Menüs für dieses Fenster. Diese Funktionen der graphischen Oberfläche ersetzen die gleichnamigen Funktionen der Textoberfläche. Sie haben dieselben Eigenschaften wie diese Funktionen und realisieren lediglich eine andere Darstellung der Menüs.

8.2.4 Strategiegrapher

Der Strategiegrapher ermöglicht eine graphische Darstellung einer Gewinnstrategie. Die Gewinnstrategie wird dabei nicht wie in der Textoberfläche linear, sondern als Baum wiedergegeben. Die Graphik wird in einem separaten Fenster ausgegeben und kann ausgedruckt werden. Zum Drucken werden Dateien im PostScript-Format erzeugt, welche die Druckinformationen beinhalten. Dabei sind zwei Modi verfügbar. Zum einen kann die Graphik in eine einzige PostScript-Datei umgewandelt werden. Diese Datei kann i. allg. nicht ausgedruckt werden, da ein Strategiebaum die Größe einer DIN-A4-Seite meist überschreitet. Deshalb besteht die Möglichkeit, mehrere PostScript-Dateien zu erzeugen, von denen jede einen Ausschnitt des Baums wiedergibt, der ausgedruckt werden kann.

Die Funktionen zum Zeichnen eines Strategiebaums sind in dem Objekt `grapher` zusammengefaßt. Dieses Objekt erzeugt durch die Methode

`(zeichne-strategie grapher` *strategiebaum*`)`

die graphische Darstellung des Strategiebaums. Der Strategiebaum muß in der von der Methode `stringliste` des Objekts `strategie` gelieferten Form übergeben werden. Zur Darstellung der Graphik wird ein neues Fenster erzeugt, in das Knoten mit den Zeichenketten und Kanten zwischen den Knoten eingezeichnet werden. Ein Positionierungsalgorithmus bestimmt dabei die Position jedes Knotens im Fenster. Der Algorithmus gewährleistet, daß keine Knoten überlappen und eine platzsparende Darstellung des Baums gefunden wird. Die Positionierung jedes Knotens erfolgt in mehreren Schritten. Im ersten Schritt wird die Breite eines Knotens festgelegt. Es wird versucht, Knoten zu erzeugen, die eine vorgegebene Standardbreite nicht überschreiten. Bei längeren Zeichenketten führt dies zu einer mehrzeiligen Darstellung, wobei eine maximale Zeilenzahl nicht überschritten werden darf. Sind zuviele Zeilen notwendig, so wird die Breite über die Standardbreite erweitert, bis eine korrekte Zeilenzahl erreicht ist.

Die vertikale Position eines Knotens ergibt sich aus der Ebene des Baums, in der der Knoten enthalten ist. Sie ist für alle Knoten derselben Baumebene gleich. Für die horizontale Position erhält jeder Knoten eine Vorgabe. Dem obersten Knoten wird der linke Rand des Fensters als Position vorgegeben. Jeder Knoten plant nach dieser Vorgabe die Position seiner Nachfolgerknoten, so daß er zentriert über ihnen steht. Mit den entsprechenden Vorgaben läßt er die Nachfolgerknoten positionieren. Wenn eine Vorgabe nicht erfüllt werden kann, weil diese zu Überlappungen von Knoten führen würde oder weil der linke Rand überschritten werden würde, dann verschiebt der Knoten seine Position nach rechts, bis eine korrekte Position erreicht ist. Er gibt die neue Position an den übergeordneten Knoten zurück, der die weiteren Nachfolger entsprechend verschoben einplant. Wenn alle Nachfolger positioniert sind, korrigiert der Knoten seine Position, indem er sich zentriert über seine Nachfolger setzt. Die korrigierte Position wird an den übergeordneten Knoten weitergegeben. Ein Beispiel für einen mit diesem Verfahren erzeugten Strategiebaum ist in Abbildung 9.3 auf Seite 86 dargestellt.

Nach der Positionierung der Knoten werden diese mit den Kanten in ein *Canvas-Widget* eingezeichnet. Canvas-Widgets sind Fensterelemente, die von STk zur Ausgabe von Graphiken zur Verfügung

gestellt werden. Das Fenster enthält außer der Graphik eine Menüleiste, die Befehle zum Schließen des Fensters und zum Ausdrucken des Strategiebaums bereitstellt. Das Fenster bleibt sichtbar, bis es über die Menüleiste geschlossen wird. Auf diese Weise können mehrere Strategiebäume in verschiedenen Fenster gleichzeitig dargestellt werden.

Zum Drucken einer Graphik werden die Methoden

```
(drucke-gesamt druckverwalter identifikation)
(drucke-seiten druckverwalter identifikation)
```

des Objekts `druckverwalter` verwendet. Diesen Methoden wird von den Menübefehlen die Identifikation des Canvas-Widgets, das ausgedruckt werden soll, übergeben. Die Methoden verwenden das Kommando `postscript` von Canvas-Widgets, um eine PostScript-Datei zu erzeugen. Nach dem Aufruf kann der Name der zu erzeugenden Datei eingegeben werden. Die Methode `drucke-gesamt` erzeugt eine einzige Datei mit dem angegebenen Namen, welche die gesamte Graphik enthält. Die Methode `drucke-seiten` unterteilt die Graphik in mehrere Abschnitte, die jeweils die Größe einer DIN-A4-Seite im Querformat haben. Zu jedem Abschnitt wird eine eigene PostScript-Datei erzeugt. Die ausgedruckten Seiten können in Zeilen und Spalten angeordnet werden, so daß sich die gesamte Graphik ergibt. Um die Position jeder Seite zu kennzeichnen, werden an die Namen der erzeugten Dateien zwei Suffixe angefügt. Das erste Suffix gibt die Spalte der Seite an, das zweite Suffix die Zeile.

Kapitel 9

Bedienung und Beispielbeweise

9.1 Installation

Die Dateien des Systems für dialogische Logik sind in gepackter Form in der Datei `dialog.zip` enthalten. Zur Installation ist diese Datei in ein Verzeichnis zu kopieren, das als Hauptverzeichnis des Systems verwendet werden soll. Sie ist dort mittels der Funktionen `unzip` für UNIX-Systeme oder `pkunzip` für DOS-Systeme zu entpacken. Das Hauptverzeichnis enthält nach dem Entpacken die vom System stets benötigten Scheme-Dateien. Darüber hinaus werden die Unterverzeichnisse `Doku`, `GUI`, `Regeln` und `Sprache` erstellt. Das erste Unterverzeichnis enthält die Dokumentation des Systems in Form von LaTeX-Dateien und einer daraus erzeugten PostScript-Datei. Die für die graphische Oberfläche benötigten Dateien nimmt das Verzeichnis `GUI` auf. Das Verzeichnis `Regeln` beinhaltet die Dateien zur Definition der Partikel- und Rahmenregeln. Das letzte Verzeichnis enthält die Dateien mit den Meldungstexten und den Befehlsbeschreibungen.

9.2 Starten des Systems

Das System für dialogische Logik ist in Scheme implementiert. Damit es gestartet werden kann, muß die Scheme-Bibliothek SLIB [4] installiert sein. Aus dieser Bibliothek wird Yasos, eine objektorientierte Erweiterung von Scheme, verwendet. Yasos benötigt die Makrounterstützung, wie sie im Anhang des *Revised*^{4} *Report on the Algorithmic Language Scheme* [3] beschrieben ist. Diese Makrounterstützung kann aus der Scheme-Bibliothek geladen werden.

Zum Starten des Systems für dialogische Logik ist der Scheme-Interpreter im Hauptverzeichnis des Systems aufzurufen. Soweit der verwendete Interpreter nicht selbst die benötigte Makrounterstützung lädt, kann dies durch Laden der Datei `makro.scm` aus dem Hauptverzeichnis erfolgen. Das System wird zusammen mit der Textoberfläche durch Laden der Datei `diainit.scm` aus demselben Verzeichnis gestartet.

Zur Verwendung der graphischen Oberfläche werden das X-Window-System und die erweiterte Scheme-Implementation STk benötigt. STk ist im Hauptverzeichnis des Systems aufzurufen. Anschließend kann die graphische Oberfläche durch Laden der Datei `GUI/guiinit.stk` initialisiert werden. Es erscheint das Hauptfenster der graphischen Oberfläche, in dem die Menüleiste und das Eingabefeld deaktiviert sind. Durch Klicken mit der linken Maustaste auf das Bild des Aristoteles wird das System gestartet.

Wenn das System für dialogische Logik aus einem anderen als seinem Hauptverzeichnis gestartet werden soll, ist beim Laden der Dateien der entsprechende Pfad anzugeben. Darüber hinaus muß in der Umgebungsvariable `DIALOG_INIT_PATH` der Pfad zum Hauptverzeichnis des Systems enthalten sein. Das Trennzeichen für Verzeichnisse ('/' oder '\') ist als letztes Zeichen in die Pfadangabe aufzunehmen.

9.3 Anweisungen

Nach dem Starten des Systems können an der Eingabemarke der Textoberfläche bzw. im Textfeld der graphischen Oberfläche Anweisungen eingegeben werden. Das System wertet die Anweisungen aus und zeigt anschließend eine Meldung über den Erfolg oder Mißerfolg der Auswertung an. Alle nachfolgend beschriebenen Kommandos können entweder direkt eingegeben oder in eine Datei geschrieben werden. Die in einer Datei enthaltenen Kommandos werden durch die Anweisung

`lade datei` *dateiname*

ausgeführt. In der Kommandodatei dürfen Kommentarzeilen stehen, die mit einem einzelnen Semikolon als erstem Zeichen der Zeile gekennzeichnet werden.

Wenn eine eingegebene oder aus einer Datei gelesene Zeile nicht als Anweisung erkannt werden kann, meldet das System einen Syntaxfehler und bezeichnet die Stelle in der Anweisung, an der ein Fehler gefunden wurde.

Zum Beenden des Programms ist die Anweisung

`ende!`

einzugeben. In der graphischen Oberfläche werden durch diese Anweisung alle Fenster des Systems geschlossen.

9.3.1 Deklarationen

Zur Deklaration von Sorten und zum Festlegen einer Ordnung der Sorten werden die Anweisungen

`deklariere sorte` *sortenname*
`sorte` $sortenname_1$ `>=` $sortenname_2$
`sorte` $sortenname_1$ `=<` $sortenname_2$
`universalsorte` *sortenname*

verwendet. Die Sortennamen dürfen wie auch die Namen aller anderen Symbole keine Leerstellen, Klammern oder Kommata enthalten. Mit dem zweiten und dem dritten Kommando kann eine Hierarchie der Sorten aufgebaut werden. Beide Anweisungen sind bis auf die vertauschten Rollen der Sorten gleichwertig.

Die im Dialogspiel verwendeten Variablen- und Konstantensymbole werden durch die Anweisungen

`deklariere variable` *variablenname sortenname*
`deklariere konstante` *konstantenname sortenname*
`deklariere konstantenmenge` *mengenname sortenname*

deklariert. Bei der Deklaration von Konstantenmengen können die Mengennamen „nat-symbole“, „nat0-symbole“, „rat-symbole“, „real-symbole“ und „komplex-symbole“ angegeben werden, welche die Symbole der Mengen $\mathcal{N}$, $\mathcal{N}_0$, $\mathcal{Q}$, $\mathcal{R}$ und $\mathcal{C}$ deklarieren. Zu jeder Sorte darf nur eine Konstantenmenge deklariert werden.

Funktionssymbole werden durch die Anweisung

`deklariere funktion` *name* `(`$sorte_1$ `...` `) ->` *sorte* `praefix` *repraesentation*

für Präfixfunktionssymbole bzw. die Anweisung

`deklariere funktion` *name* `(`$sorte_1$ `...` `) ->` *sorte* `infix` *linksbindung*
rechtsbindung repraesentation

für Infixfunktionssymbole eingeführt. Die Bindungen können Werte zwischen 1 und 999 annehmen. Wenn die Repräsentation ohne Anführungszeichen angegeben wird, darf sie wie die Symbolnamen keine Leerstellen, Klammern oder Kommata enthalten. Durch Anführungszeichen begrenzt, kann eine beliebige Zeichenkette angegeben werden.

Prädikatsymbole werden durch die Anweisungen

`deklariere praedikat` *praedikatname* `(`$sorte_1$ `...` `)` `praefix` *repraesentation*
`deklariere praedikat` *praedikatname* `(`$sorte_1$ `...` `)` `infix` *repraesentation*
`deklariere aussagenvariable` *aussagenvariablenname repraesentation*

deklariert. Prädikatsymbole ohne Argumente können dabei entweder durch die erste Anweisung mit einer leeren Sortenliste oder mittels der dritten Anweisung deklariert werden.

Die Deklarationen von Sorten und Symbolen lassen sich durch die Kommandos

```
undeklariere sorte sortenname
undeklariere variable variablenname
undeklariere konstante konstantenname
undeklariere konstantenmenge mengenname
undeklariere funktion funktionsnname
undeklariere praedikat praedikatname
undeklariere aussagenvariable aussagenvariablenname
undeklariere alle deklarationen
```

löschen. Die letzte Anweisung löscht dabei alle zuvor eingeführten Sorten und Symbole.

9.3.2 Thesen und Hypothesen

Die These und die Hypothesen eines Dialogspiels werden durch die Anweisungen

```
these formel
hypothese name formel
```

festgelegt. Zu den Hypothesen werden Namen angegeben, mit denen sie im Dialog referenziert werden. Es dürfen beliebig viele Hypothesen aber nur ein These festgelegt werden. Die Formeln werden aus den deklarierten Symbolen und den logischen Partikeln aufgebaut. Soweit keine anderen Partikel geladen wurden, sind die Standardpartikel „ALL“, „EXI“, „AND“, „OR“, „NOT“ und „SUB“ verfügbar. In den Formeln dürfen Variablen nicht frei vorkommen.

Hypothesen und Thesen bleiben nach einem Dialogspiel erhalten. Zum Löschen werden die Anweisungen

```
undefiniere hypothese name
loesche thesen
```

verwendet. Die erste Anweisung löscht die Hypothese mit dem angegebenen Namen, die zweite löscht die These und alle Hypothesen.

9.3.3 Dialogspiel

Nachdem eine These und Hypothesen festgelegt wurden, wird durch die Anweisung

```
starte dialog
```

ein Dialogspiel begonnen. Am Ende des Dialogspiels gibt eine Meldung an, ob der Beweis erfolgreich war und eine Gewinnstrategie gefunden wurde oder ob er gescheitert ist. Die Ausgabe der erzeugten Dialogschritte wird durch die Anweisungen

```
dialogausgabe an
dialogausgabe aus
```

ein- und ausgeschaltet. Zu Beginn ist die Dialogausgabe deaktiviert. In der graphischen Oberfläche wird beim Aktivieren der Ausgabe das Dialogtableau eingeblendet. Mit dem Ausschalten der Dialogausgabe werden das Dialogtableau und auch das Tracefeld ausgeblendet.

Nach einem erfolgreichen Dialogspiel kann die gefundene Gewinnstrategie durch das Kommando

```
ausgabe strategie
```

linear auf der Textoberfläche oder im Textfeld ausgegeben werden. Die Anweisungen

`speichere strategie` *dateiname*
`lade strategie` *dateiname*

erlauben es, die Strategie in einer Datei zu speichern und sie aus einer Datei zu laden.

Der Ablauf eines Dialogspiels wird von den verwendeten Partikel- und Rahmenregeln sowie von der Angriffsschranke und der Verteidigungsschranke beeinflußt. Die Definitionen der Regeln können aus Dateien geladen werden, die sich im Verzeichnis `Regeln` befinden. Für die Partikelregeln steht derzeit lediglich die Datei `standard.par` zur Verfügung. Sie enthält die Deklarationen der Partikel „ALL“, „EXI“, „AND“, „OR“, „NOT“ und „SUB“ sowie die zugehörigen Partikelregeln. Diese Datei wird beim Start des Systems geladen. Wenn weitere Dateien für Partikelregeln verfügbar sind, dann ersetzt die Anweisung

`lade partikel` *dateiname*

die Standardregeln durch die Definitionen der angegebenen Datei. Der Dateiname muß dabei ohne das Verzeichnis und ohne die Erweiterung „.par“ für Partikelregeldateien angegeben werden.

Die Rahmenregeln werden durch die Anweisung

`lade rahmenregeln` *dateiname*

festgelegt. Sie lädt die Definitionen aus einer Datei, welche die Erweiterung „.rah“ für Rahmenregeln tragen muß. Durch Angabe der Dateinamen `effektiv` oder `klass` können die Regeln für effektive oder klassisch-logische Dialoge geladen werden. Zu Beginn werden die Regeln der effektiven Logik verwendet.

Die Angriffsschranke und die Verteidigungsschranke, die die Anzahl der Angriffe bzw. Verteidigungen desselben Sprechakts begrenzen, werden durch die Aufrufe

`setze angriffsschranke` *wert*
`setze verteidigungsschranke` *wert*

festgelegt. Es ist ein positiver, ganzzahliger Wert anzugeben. Diese Schranken sollten stets möglichst klein sein, da sie die Länge der vom System selbständig erzeugten Dialogspielen stark beeinflussen. Werden größere Werte für diese Schranken gewählt, so entstehen durch die Permutation von Sprechakten des Proponenten sehr viele zu untersuchende Dialoge, wodurch im automatischen Spiel nur nach sehr langer Zeit eine Gewinnstrategie gefunden werden kann. Beim Start des Systems werden sie mit dem Wert 1 vorbelegt.

Das Benutzerspiel wird durch die Anweisungen

```
benutzerspiel an
benutzerspiel aus
```

aktiviert und deaktiviert. Die graphische Oberfläche stellt nach dem Aktivieren des Benutzerspiels das Fenster mit den Eingabemenüs dar. In der Textoberfläche werden die Menüs erst dann ausgegeben, wenn eine Eingabe erforderlich ist. Die Bedienung dieser Menüs ist im Abschnitt 9.5 erläutert.

9.4 Menüsteuerung

In der graphischen Oberfläche können Aktionen außer durch Anweisungen auch über die Menüleiste ausgelöst werden. Die Abbildung 9.1 zeigt das Hauptfenster der graphischen Oberfläche mit eingeblendetem Dialogtableau hinter dem Fenster mit dem Eingabemenü.

Die Menüleiste enthält die Einträge „Kommando“, „Dialog“, „Regeln“, „Partikel“, „Schriftart“ und „Info“. Alle Einträge außer dem letzten können sowohl mit der Maus als auch durch Drücken der ALT-Taste und des jeweils unterstrichenen Buchstabens im Namen des Eintrags ausgewählt werden. Der Eintrag „Info“ wird über die Maus selektiert, wodurch ein Fenster mit einem Informationstext dargestellt wird. Alle anderen Menüeinträge zeigen eine Liste an, sobald sie ausgewählt werden. Die enthaltenen Einträge werden mit dem Mauszeiger oder durch Drücken des unterstrichenen Buchstabens ohne die ALT-Taste selektiert.

Der Menüpunkt „Kommando“ enthält lediglich den Eintrag „Ende!“, durch den das Programm beendet wird. Im Menüpunkt „Dialog“ sind die Einträge „Starte Dialog“, „Zeichne Strategie“, „Lösche Thesen“ sowie die Schalter „Benutzerspiel“, „Dialogtableau“ und „Dialog-Trace“ enthalten. Der Eintrag „Zeichne Strategie“ kann nach einem erfolgreichen Dialogspiel aufgerufen werden. Er erzeugt ein neues Fenster, in dem die Gewinnstrategie graphisch dargestellt ist. Die Schalter aktivieren oder deaktivieren das Benutzerspiel bzw. blenden die jeweiligen Dialogfelder ein und aus. Wenn das Benutzerspiel aktiviert wird, erscheint das Fenster mit den Eingabemenüs und das Dialogtableau wird eingeblendet.

Unter dem Menüpunkt „Regeln“ können durch Auswählen der Einträge „Effektiv“ oder „Klassisch“ die im Dialogspiel verwendeten Rahmenregeln festgelegt werden. Der Menüpunkt „Partikel“ bietet lediglich den Eintrag „Standard“ zum Laden der Standardpartikel an.

Im Menüpunkt „Schriftart“ stehen Schalter zum Wechseln der Schriftart und der Schriftgröße sowie Untermenüs zum Verändern der Schriftfarbe und der Hintergrundfarbe zur Verfügung. Die Untermenüs werden angezeigt, sobald sich der Mauszeiger über den jeweiligen Einträgen befindet. Ein selektierter Eintrag wirkt sich nur dann aus, wenn die gewählte Schriftart, Schriftgröße oder Farbe dem X-Window-System bekannt ist, ansonsten wird er ignoriert.

Die Fenster zur Darstellung von Strategiebäumen besitzen Menüleisten mit den Punkten „Kommando“ und „Drucken“. Ein solches Fenster ist in Abbildung 9.3 auf Seite 86 wiedergegeben. Unter dem ersten Menüpunkt ist der Eintrag „Schließen“ verfügbar, der das Fenster schließt. Der zweite Menüpunkt bietet die Einträge „Drucken Gesamt“ und „Drucken Seiten“ an. Nach dem Auswählen eines Eintrags erscheint ein Eingabefeld, in dem ein Dateiname angegeben werden kann. Mit Hilfe dieser Einträge können PostScript-Dateien zum Drucken des Strategiebaums erzeugt werden. Der Eintrag „Drucken Gesamt“ generiert eine einzige Datei, welche die gesamte Graphik enthält. Der Menüpunkt „Drucken Seiten“ erzeugt für jeden Ausschnitt der Größe einer DIN-A4-Seite eine eigene Datei, wobei zur Unterscheidung der Dateien die Spalte und die Zeile jedes Ausschnitts als Suffixe an den Dateinamen angefügt werden.

9.5 Beispielbeweise

Zur Beschreibung der Bedienung der Eingabemenüs im Benutzerspiel soll ein sehr einfaches Dialogspiel um die These $(\forall x P(x)) \rightarrow P(f(a))$ geführt werden. Der Variable ‘x’, der Konstante ‘a’ und den Argumenten der Funktion ‘f’ und des Prädikats ‘P’ soll die Sorte „nat“ zugeordnet sein. Die Deklarationen und die Definition der These lauten damit:

```
deklariere sorte nat
deklariere variable x nat
deklariere konstante a nat
deklariere funktion f (nat) -> nat praefix f
deklariere praedikat P (nat) praefix P
these (ALL x P(x)) SUB P(f(a)).
```

Nachdem diese Anweisungen eingegeben oder aus einer Datei geladen und das Benutzerspiel aktiviert wurden, kann das Dialogspiel gestartet werden. Im ersten Dialogschritt kann der Proponent nur die These setzen. Das Eingabemenü bietet deshalb nur diesen Sprechakt an, der ausgewählt werden muß. Die These wird im Dialog gesetzt und im folgenden Schritt vom Opponenten durch das Setzen des Antezedens angegriffen. Es ergibt sich die Stellung, die in Abbildung 9.1 für die graphische Oberfläche dargestellt ist. In der Textoberfläche erscheint ein Menü, in dem die Sprechakte durch Nummern indiziert sind. Der Benutzer hat nun die Wahl, entweder selbst die Behauptung des Opponenten anzugreifen oder sich gegen den Angriff durch Setzen des Sukzedens zu verteidigen. Eine Verteidigung kann jedoch nicht erfolgreich durchgeführt werden, da die Primformel $P(f(a))$ vom Proponenten nicht gesetzt werden darf. Wird im Menü dennoch dieser Sprechakt gewählt, so erscheint die Meldung, daß die Primformel nicht übernommen werden kann. Als erfolgreiche Fortsetzung bleibt nur der Angriff auf die Allaussage des Opponenten. Dazu ist auf

Abbildung 9.1: Eingabemenü der graphischen Oberfläche

der Textoberfläche der Index '1' einzugeben. Im Eingabemenü der graphischen Oberfläche ist der zweite Eintrag des Menüfelds zu selektieren und der 'OK'-Knopf zu aktivieren. Durch Aktivieren des Knopfes 'Zurück' oder eine leere Eingabe im Menü der Textoberfläche kehrt der Benutzer zum vorhergehenden Menü zurück.

Für den Angriff muß ein Term gewählt werden, daher erscheint nach dem Auswählen des Angriffssprechakts ein Menü zur Termwahl. Diese Situation ist in Abbildung 9.2 wiedergegeben. In der Textoberfläche wird dieselbe Meldung zusammen mit einer Eingabeaufforderung dargestellt. In diesem Menü kann ein Term angegeben oder ein Termsymbol gewählt werden. Um ein Term-

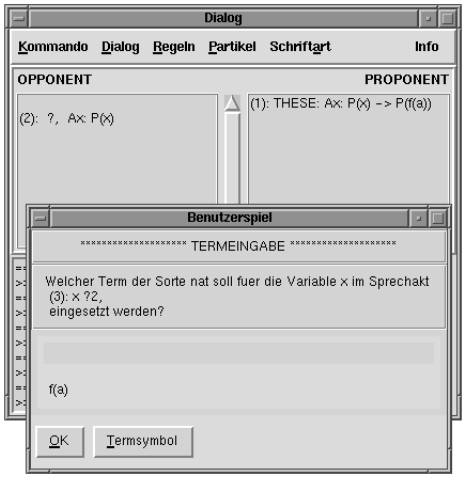

Abbildung 9.2: Termwahlmenü der graphischen Oberfläche

symbol zu wählen, ist auf der Textoberfläche lediglich die Return-Taste zu drücken. Im Menü der graphischen Oberfläche muß der Knopf 'Termsymbol' aktiviert werden. Da leicht ersichtlich ist, daß der Term $f(a)$ zum Ziel führt, kann dieser Term als Parameter des Angriffs gewählt werden.

Der Opponent ersetzt im Verteidigungssprechakt die Quantorvariable durch diesen Term. Die entstehende Primformel kann im nächsten Schritt vom Proponenten zum Einlösen seiner Verteidigungsmöglichkeit verwendet werden. Nach dem Auswählen dieses Sprechakts wird ein Menü zur Selektion der Primformel, die übernommen werden soll, angezeigt. Da der Opponent nur eine Primformel gesetzt hat, enthält das Menü nur einen Eintrag. Sobald die Primformel übernommen wird, ist der Dialog vom Proponenten gewonnen. Es erscheint ein Menü, in dem gewählt werden kann, ob weitergespielt oder abgebrochen werden soll. Wenn noch Zugmöglichkeiten des Opponenten bearbeitet werden müßten, würde beim Weiterspielen zu einer solchen Stellung im Dialogspiel zurückgegangen werden. Beim Abbrechen würden keine weiteren Zugmöglichkeiten mehr betrachtet werden und das Dialogspiel würde unmittelbar enden. Da in diesem Beispiel keine alternativen Zugmöglichkeiten des Opponenten vorhanden sind, wird sowohl beim Weiterspielen als auch beim Abbrechen der Dialog erfolgreich beendet.

In einem weiteren Beispiel soll die Verwendung von Termsymbolen im Dialog verdeutlicht werden. Seien die Symbole 'x', 'f' und 'P' wie im vorherigen Beispiel deklariert. Im Dialogspiel um die These $\forall x P(x) \rightarrow \forall x P(f(x))$ werden folgende Sprechakte erzeugt:

```
PRO(1): THESE: Ax: P(x) -> Ax: P(f(x))
OPP(2):  ?,  Ax: P(x)
PRO(3): Ax: P(f(x)) [[2]]
OPP(4): $1 ?,
PRO(5): $2 ?2,
OPP(6): P($2)
PRO(7): P(f($1)) [[4]]
OPP(8): %%%
```

Die Nummern in den doppelten eckigen Klammern hinter den Sprechakten des Proponenten geben an, auf welchen Sprechakt des Opponenten Bezug genommen wird.

Im vierten Dialogschritt greift der Opponent die Allaussage des Proponenten an und nennt das Termsymbol '$1', für das die Verteidigung erfolgen soll. Der Proponent kann die geforderte Primformel nicht setzen und greift seinerseits die Allaussage des Opponenten mit dem Termsymbol '$2' als Parameter an. Der Opponent antwortet mit der gewünschten Primformel, welche der Proponent zur Verteidigung übernimmt und den Dialog damit gewinnt. Im diesem Dialogschritt hat eine Unifikation der Primformeln $P(\$2)$ und $P(f(\$1))$ stattgefunden, wobei das vom Proponenten eingeführte Termsymbol '$2' an den Term 'f($1)' gebunden wurde. Dies ist aus der erzeugten Gewinnstrategie ersichtlich:

```
PRO(1): THESE: Ax: P(x) -> Ax: P(f(x))
OPP(2):  ?,  Ax: P(x)
PRO(3): Ax: P(f(x)) [[2]]
OPP(4): $1 ?,
PRO(5): f($1) ?2,
OPP(6): P(f($1))
PRO(7): P(f($1)) [[4]]
OPP(8): %%%
```

Durch die Unifikation werden die Primformeln identisch, so daß der Proponent die Primformel des Opponenten übernehmen kann.

Werden in der These das Antezedens und das Sukzedens vertauscht, scheitert das Dialogspiel:

```
PRO(1): THESE: Ax: P(f(x)) -> Ax: P(x)
OPP(2):  ?,  Ax: P(f(x))
PRO(3): Ax: P(x) [[2]]
OPP(4): $1 ?,
PRO(5): $2 ?2,
OPP(6): P(f($2))
PRO(3): $3 ?2,
OPP(4): P(f($3))
PRO(5): Ax: P(x) [[2]]
OPP(6): $4 ?,
```

Der erste Dialog wird vom Proponenten nach dem Setzen der Primformel durch den Opponenten im sechsten Schritt abgebrochen. Der Proponent müßte an dieser Stelle die Primformel $P(\$1)$ setzen. Da das Termsymbol '$1' vom Opponenten eingeführt wurde, darf es an keinen Term gebunden werden. Die Unifikation der Primformeln $P(\$1)$ und $P(f(\$2))$ scheitert damit. Der Proponent versucht anschließend eine neue Dialogfortsetzung ab dem dritten Dialogschritt und scheitert damit ebenfalls. Da er keine weiteren Alternativen wählen kann, bleibt der Beweisversuch erfolglos.

Abschließend sei in Abbildung 9.3 eine im Dialogspiel gefundene Gewinnstrategie dargestellt. Der Dialog wurde um die PEIRCEsche Formel $((A \to B) \to A) \to A$ als These geführt. Diese These ist nur klassisch logisch gültig. Indem als Hypothese das „Tertium non datur“ für die Aussagenvariable 'A' vorausgesetzt wird, ist die These auch im Dialog nach effektiven Regeln beweisbar. In der dargestellten Gewinnstrategie ist stets nur eine erfolgreiche Zugmöglichkeit des Proponenten aufgeführt. Wenn der Opponent mehrere Alternativen hat, so verzweigt die Gewinnstrategie bei diesen Zugmöglichkeiten. Alle Dialoge werden vom Proponenten gewonnen, so daß die Blätter des Strategiebaums Verlustsprechakte des Opponenten enthalten.

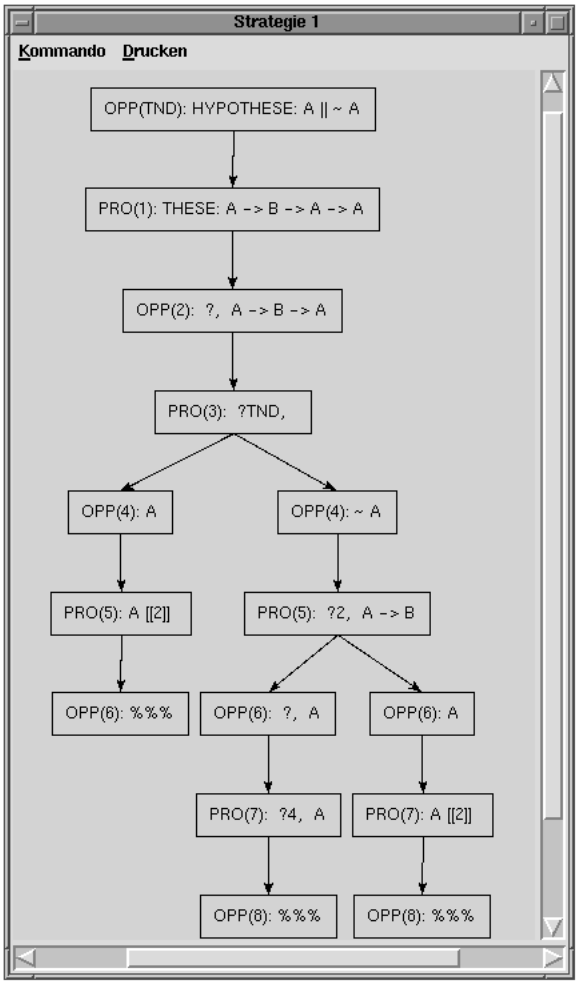

Abbildung 9.3: Strategiebaum

Kapitel 10

Resümee und Ausblick

In dieser Arbeit wurde der Aufbau des Systems für dialogische Logik beschrieben. Von zentraler Bedeutung sind dabei die im Abschnitt Strategiebildung dargestellten Verfahren zur Wahl von Termen, zur Primformelübernahme und zum Auslösen von Backtracking mittels Failure-Continuations. Durch die Verwendung von Termsymbolen bei der Termwahl werden unendliche Verzweigungen im Dialogbaum verhindert. Die Entscheidung, an welchen konkreten Term ein Termsymbol gebunden werden soll, wird so lange verzögert, bis bei der Übernahme von Primformeln nur noch endlich viele Möglichkeiten zu überprüfen sind. Da bei der Übernahme nicht entschieden werden kann, welche Bindungen die Bildung einer Gewinnstrategie zulassen, wird durch Failure-Continuations die Möglichkeit geschaffen, zu einem späteren Zeitpunkt Bindungen zu widerrufen und Alternativen auszuwerten.

Die Sorten- und Symbolverwaltung sowie der darauf basierende Aufbau von Formeln geben dem Anwender die Mittel an die Hand, eine Sprache zur Beschreibung des gewünschten Gegenstandsbereichs festzulegen. Die graphische Oberfläche ermöglicht eine aussagekräftige Darstellung von Dialogspielen und Gewinnstrategien. Die Integration einer Textoberfläche verhindert, daß der Kreis der Anwender durch die Systemanforderungen der graphischen Oberfläche eingeschränkt wird.

Für die Weiterentwicklung des Systems für dialogische Logik bieten sich mehrere Möglichkeiten an:

- Realisierung einer Logik mit Gleichheit:
  Die von Term- und Primformelobjekten zur Verfügung gestellten Rewrite-Methoden wurden im Hinblick auf diese Erweiterungsmöglichkeit implementiert. Um diese Methoden zu verwenden, muß eine Möglichkeit geschaffen werden, Gleichheitsregeln, wie beispielsweise $f(x) = g(h(x))$, zu definieren. Die Verwendung solcher Regeln sollte vom eigentlichen Dialogspiel getrennt werden und nur bei der Primformelübernahme möglich sein. Eine Primformel soll übernehmbar sein, wenn die Primformel des Opponenten und die Primformel des Proponenten durch Anwendung von Gleichheitsregeln so transformiert werden können, daß sie unifizierbar sind. Dazu ist eine Komponente nötig, die in dem Baum der möglichen Regelanwendungen nach geeigneten Folgen von Transformationen sucht.

- Verbesserte Visualisierung von Gewinnstrategien:
  Insbesondere für stark verzweigte Gewinnstrategien mit sehr vielen Knoten sind Hilfsmittel zu Darstellung der Struktur der Gewinnstrategie wünschenswert. Sie sollten es ermöglichen, Teilbäume oder Folgen von Sprechakten zu einem Knoten zusammenzufassen, um Darstellungen mit unterschiedlichem Grad der Detaillierung zu erreichen. Zu diesem Zweck können auch spezialisierte Grapher-Programme verwendet werden, denen eine erzeugte Gewinnstrategie übergeben wird.

- Statistische Informationen über die Strategiebildung:
  Nach einem Beweisversuch könnten Daten zur Strategiebildung abrufbar sein. Auf diese Weise könnten Informationen über die Anzahl der ausgewerteten Knoten, der erfolgreichen und gescheiterten Knoten, über die verwendeten Hypothesen, das Maximum der Angriffs- und Verteidigungswiederholungen oder die maximale Länge von Dialogen wiedergegeben werden.

- Verbesserte Rahmenregeln:
  Insbesondere bei höheren Angriffs- und Verteidigungsschranken entstehen durch die Permutation von Sprechakten sehr viele mögliche Dialoge, wodurch Beweisversuche sehr lange Zeit in Anspruch nehmen können. Verbesserte Rahmenregeln sollten versuchen, 'sinnlose' Wiederholungen von Sprechakten oder Permutationen von Sprechakten, die nicht zu neuen Zugmöglichkeiten führen, zu verhindern. Mittels der Dialogliste müßten derartigen Regeln weitere Informationen über den Dialogverlauf zur Verfügung gestellt werden.

Die Weiterentwicklung des Systems kann nicht nur durch Änderung der Implementation erfolgen. Ein Hauptziel der Gestaltung des Systems für dialogische Logik war die Erweiterbarkeit des Systems. Es sollte möglich sein, mit geringem Aufwand unterschiedliche Logiken zu realisieren. Diesem Ziel wurde Rechnung getragen, indem Partikel- und Rahmenregeln nicht feste Bestandteile des Systems, sondern modifizierbar sind.

Durch Erweiterung der Partikelregeln können neue logische Verknüpfungen im Dialogspiel verwendet werden. In der üblichen Formulierung der dialogischen Logik ist die Bisubjunktion '$\leftrightarrow$' nicht als Junktor enthalten. Im System für dialogische Logik kann diese als Infixjunktor deklariert werden. Die zugehörige Partikelregel könnte festlegen, daß eine Bisubjunktion durch einen schlichten Zweifel angegriffen werden kann und als Verteidigung die Konjunktion zweier Subjunktionen zu setzen ist. Ein nachfolgender Angriff auf die Konjunktion fordert eine der beiden Richtungen der Bisubjunktion als Verteidigung ein. Dieser Ablauf kann verkürzt werden, indem bereits beim Angriff auf die Bisubjunktion festgelegt wird, welche Richtung gezeigt werden soll. Die Partikelregel für den Angriff auf eine Behauptung „$A \leftrightarrow B$“ wird dazu analog dem Angriff auf eine Konjunktion formuliert. Als Parameter des Angriffs werden die Symbole '-> und '<- eingetragen. Die Regeln für die Verteidigung überprüfen den Parameter und erzeugen die Verteidigungssprechakte „$A \rightarrow B$“ oder „$B \rightarrow A$“.

Eine weitergehende Modifikation der Regeln soll am Beispiel der ontischen und deontischen Modallogik skizziert werden. In der ontischen Modallogik wird der Operator 'N' definiert, der angewendet

auf eine Aussage ausdrückt, daß diese Aussage notwendigerweise wahr ist. Mit einer Formel A ist auch NA eine Formel. Syntaktisch gleicht der neue Operator einem einstelligen Präfixjunktor und kann daher in dieser Weise deklariert werden. Die Behandlung des Modaloperators im Dialogspiel kann durch die *Schnittregel* festgelegt werden. Sie besagt, daß von einer Stellung

$$\begin{array}{l|l} NA_1 & \\ \vdots & \\ NA_n & NB \end{array}$$

zu der Stellung

$$\begin{array}{l|l} A_1 & \\ \vdots & \\ A_n & B \end{array}$$

übergegangen werden kann (vergleiche Kamlah/Lorenzen [11, S.225ff]). Wenn der Proponent also eine Modalaussage „NB“ durch Setzen von „B“ verteidigt, werden alle Behauptungen des Opponenten, die eine andere Hauptverknüpfung als den Modaloperator haben, gelöscht. Von den verbleibenden Behauptungen werden die Modaloperatoren entfernt. Diese Regel bewirkt einen Schnitt im Dialogspiel, weil der Proponent nach seiner Verteidigung der Modalaussage auf keinen Sprechakt vor dem Schnitt Bezug nehmen darf.

Die Schnittregel kann nicht direkt im System für dialogische Logik verwendet werden, da das Verfahren der Strategiebildung nicht zuläßt, daß ein Spieler in einem Schritt mehrere Sprechakte setzt. Die Behandlung modallogischer Aussagen kann jedoch in der von Haas [9, S. 224f] vorgeschlagenen Weise realisiert werden. Dabei muß im Dialogspiel unterschieden werden, zwischen Behauptungen, die aus der Diskussion um modallogisch zusammengesetzte Formeln hervorgegangen sind, und faktischen Behauptungen. Faktische Behauptungen sind dabei Formeln, die nicht modallogisch zusammengesetzt sind und nicht aus Modalaussagen hervorgegangen sind. Für Angriffe und Verteidigungen von Modalaussagen dürfen vom Proponenten keine faktischen Behauptungen verwendet werden, da diese nichts über die Notwendigkeit einer Formel aussagen. Dasselbe gilt für Angriffe und Verteidigungen von Formeln, die aus Modalaussagen hervorgegangen sind. Da in der ontischen Modallogik die Implikation $NA \Rightarrow A$ gilt, also notwendigerweise wahre Aussagen auch faktisch wahr sind, können aus Modalaussagen hervorgegangene Behauptungen vom Proponenten zum Angriff und zur Verteidigung faktischer Aussagen verwendet werden.

Als Partikelregel für den Modaloperator gibt Haas an, daß ein Angriff auf eine Aussage „NA“ durch den Sprechakt „N?“ ausgeführt wird. Zur Verteidigung wird die Aussage „A“ gesetzt und im Dialogspiel durch einen Stern als aus einer Modalaussage hervorgegangen gekennzeichnet. Alle Angriffe und Verteidigungen, die sich auf gekennzeichnete Sprechakte beziehen, werden ebenfalls gekennzeichnet. Im Dialogspiel darf der Proponent nur gekennzeichnete Primformeln übernehmen,

um gekennzeichnete Aussagen anzugreifen oder sich gegen gekennzeichnete Angriffe zu verteidigen. Zum Angriff und zur Verteidigung nicht gekennzeichneter Sprechakte dürfen alle Primformeln übernommen werden.

Sprechakte, die sich auf Modalaussagen beziehen, können im System für dialogische Logik gekennzeichnet werden, indem der Bezug dieser Sprechakte zu einem Paar mit einer Identifikation und dem Kennzeichen '* erweitert wird. Die Partikelregeln der Standardpartikel sind so zu modifizieren, daß zusätzlich die Regelbedingung `(not (pair? eta1))` überprüft wird. Da im Dialog um Modalaussagen für diese Verknüpfungen dieselben Regeln gelten, ist zu jeder Partikelregel eine neue Regel hinzuzunehmen, in der der Bezug ein Paar ist. Die Identifikation des Sprechakts, auf den Bezug genommen wird, ist durch `(car eta1)` zu bilden. Diese Regeln erzeugen Sprechakte, deren Bezug wieder ein Paar ist.

Zusätzlich sind folgende Partikelregeln für Modalaussagen einzuführen:

```
(definiere partikelregel
  "A_Mod"
  (p-regel ny delta eta1 eta2 zeta
           (regelbedingungen (not (pair? eta1))
                     (equal? (hauptverknuepfung delta) "N"))
            (sprechakt-form  leerformel                          ;; delta
                             ny                                  ;; eta1
                             angriff                             ;; eta2
                             'N)))                               ;; zeta

(definiere partikelregel
  "V_Mod"
  (p-regel ny delta eta1 eta2 zeta
           (regelbedingungen (not (pair? eta1))
                             (equal? (hauptverknuepfung (delta-funktion eta1))
                                     "N")
                             (eqv? eta2 angriff)
                             (eqv? zeta 'N))
           (sprechakt-form   (unterformel (delta-funktion eta1))   ;; delta
                             (cons ny '*)                          ;; eta1
                             verteidigung                          ;; eta2
                             leerparameter)))                      ;; zeta
```

Diese Regeln sind nicht auf gekennzeichnete Sprechakte anwendbar. Wenn eine Aussage aus einer Modalaussage hervorgegangen ist und selbst eine Modalaussage ist, erlauben die genannten Partikelregeln nicht, diese Aussage im Dialog weiter zu verwenden. Dadurch wird die Verwendung iterierter Modaloperatoren wie in der Formel NNA verhindert.

Die Partikelregeln kennzeichnen alle Sprechakte, die sich direkt oder indirekt auf Modalaussagen beziehen. Durch die Rahmenregeln muß sichergestellt werden, daß der Proponent zum Setzen einer Primformel in einem gekennzeichneten Sprechakt nur gekennzeichnete Primformeln des Opponenten übernimmt. Dies kann durch die Primformelregel

```
(definiere rahmenregel
  "PR_MOD_1"
```

```
  (primformel-regel o-delta o-eta1 o-eta2 o-zeta
                    p-delta p-eta1 p-eta2 p-zeta
       (regelbedingungen (pair? p-eta1)
                         (not (pair? o-eta1)))
       sperre-uebernahme))
```

errreicht werden.

Damit die vorgestellten Regeln zusammen mit den Rahmenregeln für effektive und klassische Logik verwendet werden können, sind über die von Haas vorgegebenen Regeln hinaus zwei weitere Rahmenregeln nötig. Diese Regeln sollen verhindern, daß der Proponent, nachdem er eine Modalaussage verteidigt hat, frühere Verteidigungsmöglichkeiten einlöst oder Behauptungen angreift, die nicht modallogisch zusammengesetzt sind oder aus Modalaussagen hervorgegangen sind. Diese Regeln lauten:

```
(definiere rahmenregel
  "RR_P_Schnitt_V"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen
             (eqv? sigma p_signum)
             (eqv? (eta2-funktion (- (ny-letzter-sprechakt) 1)) verteidigung)
             (eqv? (zeta-funktion (eta1-funktion
                                   (- (ny-letzter-sprechakt) 1)))
                   'N)
             (eqv? eta2 verteidigung)
             (not (and (pair? eta1)
                       (eq? (car eta1) (ny-letzter-sprechakt)))))
           loesche-sprechakt))

(definiere rahmenregel
  "RR_P_Schnitt_P"
  (r-regel sigma delta eta1 eta2 zeta
           (regelbedingungen
             (eqv? sigma p_signum)
             (eqv? (eta2-funktion (- (ny-letzter-sprechakt) 1)) verteidigung)
             (eqv? (zeta-funktion (eta1-funktion
                                   (- (ny-letzter-sprechakt) 1)))
                   'N)
             (eqv? eta2 angriff)
             (not (or (eqv? zeta 'N)
                      (pair? eta1))))
           loesche-sprechakt)).
```

Die ersten drei Regelbedingungen beider Regeln legen fest, daß diese Regeln nur angewendet werden sollen, wenn der Proponent am Zug ist und in seinem letzten Sprechakt eine Modalaussage verteidigt hat. Mittels dieser Regeln wird ein Schnitt im Dialogspiel in der Form bewirkt, daß der Proponent den modallogischen Dialog nicht mehr verlassen kann, nachdem er eine Modalaussage verteidigt hat. In den anderen Rahmenregeln der effektiven oder klassichen Logik muß berücksichtigt werden, daß der Bezug eines Sprechakts auch ein Paar sein kann.

In der deontischen Modallogik wird statt des Operators 'N' der Gebotsoperator 'O' eingeführt. Die Partikelregeln für die deontische Logik können analog der Regeln für die ontische Logik formuliert werden. Die genannte Primformelregel kann in derselben Form verwendet werden. Im Gegensatz zur ontischen Modallogik gilt jedoch nicht die Implikation $O\ A \Rightarrow A$. Etwas Gebotenes muß nicht faktisch wahr sein. Aus diesem Grund wird eine zweite Primformelregel definiert, welche die Übernahme gekennzeichneter Primformeln in nicht gekennzeichneten Sprechakten des Proponenten verbietet.

Durch die skizzierten Regeln können Dialoge um modallogisch zusammengesetzte Thesen und Hypothesen geführt werden. Die Partikelregeln für modallogische Dialoge sind in den Dateien `ontik.par` bzw. `deontik.par` enthalten. Die Rahmenregeln für effektive und klassische Modallogik befinden sich in den Dateien `ontik_ef.rah`, `ontik_kl.rah`, `deon_ef.rah` und `deon_kl.rah`. Alle diese Dateien stehen im Verzeichnis `Regeln`.

Anhang A

Partikel und Partikelregeln

Die Partikel und die zugehörigen Partikelregeln werden vom System für dialogische Logik aus einer Datei geladen, die sich im Verzeichnis `Regeln` befinden muß und die Endung „.par“ trägt. In dieser Datei sind Deklarationen logischer Partikel, Definitionen von Partikelregeln sowie die Definition der Liste `junktoren_ohne_verteidigung` enthalten. Beim Start des Systems werden diese Definitionen aus der Datei `standard.par` geladen. Sie enthält folgende Definitionen:

```
;;**************************
;;*** Definition der Partikel
;;**************************
(deklariere junktorsymbol "AND" 2 "infix" '(3 4) "&&")
(deklariere junktorsymbol "OR" 2 "infix" '(1 2) "||")
(deklariere junktorsymbol "NOT" 1 "praefix" '(6) "~")
(deklariere junktorsymbol "SUB" 2 "infix" '(1 2) "->")

(deklariere quantorsymbol "ALL" '(5) "A")
(deklariere quantorsymbol "EXI" '(5) "E")

;;******************************
;;*** JUNKTOREN_OHNE_VERTEIDIGUNG
;;******************************
(define junktoren_ohne_verteidigung (list "NOT"))


;;**********************************
;;*** Aussagenlogische Partikelregeln
;;**********************************

;;***  DISJUNKTION
;;***  -----------

(definiere partikelregel
  "A_Dis"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta) "OR"))
         (sprechakt-form     leerformel                                  ;; delta
                             ny                                          ;; eta1
                             angriff                                     ;; eta2
                             leerparameter                               ;; zeta
))       )

(definiere partikelregel
  "V_Dis_L"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung (delta-funktion eta1))
                                     "OR")
```

```
                              (eqv? eta2 angriff))
         (sprechakt-form      (linke-unterformel (delta-funktion eta1));; delta
                              ny                                       ;; eta1
                              verteidigung                             ;; eta2
                              leerparameter                            ;; zeta
))       )

(definiere partikelregel
  "V_Dis_R"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen    (equal? (hauptverknuepfung (delta-funktion eta1))
                                      "OR")
                              (eqv? eta2 angriff))
         (sprechakt-form      (rechte-unterformel
                                (delta-funktion eta1))                 ;; delta
                              ny                                       ;; eta1
                              verteidigung                             ;; eta2
                              leerparameter                            ;; zeta
))       )

;;*** KONJUNKTION
;;*** -----------

(definiere partikelregel
  "A_Kon"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen      (equal? (hauptverknuepfung delta) "AND"))
         (sprechaktliste
            (sprechakt-form     leerformel                           ;; delta
                                ny                                   ;; eta1
                                angriff                              ;; eta2
                                links )                              ;; zeta
            (sprechakt-form     leerformel                           ;; delta
                                ny                                   ;; eta1
                                angriff                              ;; eta2
                                rechts )                             ;; zeta

) )      )

(definiere partikelregel
  "V_Kon_L"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen    (equal? (hauptverknuepfung (delta-funktion eta1))
                                      "AND")
                              (eqv? eta2 angriff)
                              (eqv? zeta links))
         (sprechakt-form      (linke-unterformel
                                (delta-funktion eta1))                 ;; delta
                              ny                                       ;; eta1
                              verteidigung                             ;; eta2
                              leerparameter                            ;; zeta
))       )

(definiere partikelregel
  "V_Kon_R"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen    (equal? (hauptverknuepfung (delta-funktion eta1))
                                      "AND")
                              (eqv? eta2 angriff)
                              (eqv? zeta rechts))
         (sprechakt-form      (rechte-unterformel
                                (delta-funktion eta1))                 ;; delta
                              ny                                       ;; eta1
                              verteidigung                             ;; eta2
                              leerparameter                            ;; zeta
))       )

;;*** SUBJUNKTION
;;*** -----------

(definiere partikelregel
```

```
  "A_Sub"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta) "SUB"))
         (sprechakt-form     (linke-unterformel delta)                ;; delta
                             ny                                       ;; eta1
                             angriff                                  ;; eta2
                             leerparameter                            ;; zeta
))       )

(definiere partikelregel
  "V_Sub"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung (delta-funktion eta1))
                                     "SUB")
                             (eqv? eta2 angriff))
         (sprechakt-form     (rechte-unterformel
                               (delta-funktion eta1))                 ;; delta
                             ny                                       ;; eta1
                             verteidigung                             ;; eta2
                             leerparameter                            ;; zeta
))       )

;;*** NEGATION
;;*** -----------

(definiere partikelregel
  "A_Neg"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta) "NOT"))
         (sprechakt-form     (unterformel delta)                      ;; delta
                             ny                                       ;; eta1
                             angriff                                  ;; eta2
                             leerparameter                            ;; zeta
))       )


;;**********************************
;;*** Quantorenlogische Partikelregeln
;;**********************************

;;***  ALLQUANTOR
;;***  ----------

(definiere partikelregel
  "A_All"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta) "ALL"))
         (sprechakt-form     leerformel                               ;; delta
                             ny                                       ;; eta1
                             angriff                                  ;; eta2
                             (quantorvariable delta)                  ;; zeta
))       )

(definiere partikelregel
  "V_All"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung (delta-funktion eta1))
                                     "ALL")
                             (eqv? eta2 angriff))
         (sprechakt-form     (substituiere-variable                   ;; delta
                                       (rumpf (delta-funktion eta1))
                                       (quantorvariable
                                         (delta-funktion eta1))
                                       zeta)
                             ny                                       ;; eta1
                             verteidigung                             ;; eta2
                             leerparameter                            ;; zeta
))       )

;;***  EXISTENZQUANTOR
;;***  ---------------
```

```
(definiere partikelregel
  "A_Exi"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (equal? (hauptverknuepfung delta) "EXI"))
         (sprechakt-form     leerformel                                  ;; delta
                             ny                                          ;; eta1
                             angriff                                     ;; eta2
                             leerparameter                               ;; zeta
))       )

(definiere partikelregel
  "V_Exi"
  (p-regel ny delta eta1 eta2 zeta
         (regelbedingungen   (eqv? eta2 angriff)
                             (equal? (hauptverknuepfung (delta-funktion eta1))
                                     "EXI"))
         (sprechakt-form     (rumpf (delta-funktion eta1))               ;; delta
                             ny                                          ;; eta1
                             verteidigung                                ;; eta2
                             (quantorvariable (delta-funktion eta1))     ;; zeta
))       )
```

Anhang B

Funktionen und Methoden

Das System für dialogische Logik stellt eine Vielzahl von Funktionen und Methoden zur Verfügung, die in Partikel- und Rahmenregeln verwendet werden können. Sie erlauben es, die Bedingungen in Partikel- und Rahmenregeln zu formulieren. Bei der Bildung von Sprechakten können Partikelregeln darüber hinaus die Methoden zur Erzeugung und Modifikation von Formeln anwenden. Dieser Anhang gibt eine Übersicht der verwendbaren Befehle.

Klassifikation von Formeln (Abschnitt 6.2.1)

(`primformel?` *objekt)*
(`junktorformel?` *objekt)*
(`quantorformel?` *objekt)*

Selektion von Formelelementen (Abschnitt 6.2.2)

(`hauptsymbol` *objekt)*
(`unterformel` *objekt)*
(`linke-unterformel` *objekt)*
(`rechte-unterformel` *objekt)*
(`quantorvariable` *objekt)*
(`rumpf` *objekt)*

Modifikation von Formeln (Abschnitt 6.2.3)

(`substituiere-variable` *objekt variablensymbol termobjekt)*

Erzeugung von Formeln (Abschnitt 6.1)

(`make-junktorformel` *symbolname formelobjekt)*
(`make-junktorformel` *symbolname* $formelobjekt_1$ $formelobjekt_2$*)*
(`make-quantorformel` *quantorsymbol variablensymbol rumpfformelobjekt)*

Schnittstellenfunktionen der Dialogliste (Abschnitt 7.2)

(`sprechakt-funktion` *identifikation)*
(`delta-funktion` *identifikation)*

(eta1-funktion *identifikation)*

(eta2-funktion *identifikation)*

(zeta-funktion *identifikation)*

(these-gesetzt?)

(angriffswiederholungen *identifikation)*

(ny-letzter-offener-angriff *signum)*

(offener-angriff? *identifikation signum)*

(letzter-angriff-verteidigbar? *signum)*

(ny-letzter-sprechakt)

Anhang C

Beispiel einer Kommandodatei

In diesem Anhang wird die Kommandodatei für einen komplexen Beweis wiedergegeben. In diesem Beweis soll die Irrationalität der Zahl $\sqrt{2}$ gezeigt werden. Das Vorgehen bei diesem Beweis wurde aus BEHNKE u. a. [2, S. 49ff] übernommen.

Als These wird die Aussage formuliert, daß die Zahl $\sqrt{2}$ nicht als Quotient zweier ganzer Zahlen p und q dargestellt werden kann, wobei p und q teilerfremd sein sollen. Insbesondere soll die Zahl 2 kein Teiler dieser Zahlen sein. Die These lautet damit:

$$\neg\exists p\exists q(2 = \frac{p^2}{q^2} \wedge \neg(\exists s 2s = p \wedge \exists t 2t = q)).$$

Zum Beweis der These müssen zwei arithmetische Aussagen vorgegeben werden. Zum einen soll verwendet werden, daß eine ganze Zahl, deren Quadrat geradzahlig ist, selbst geradzahlig ist. Zum anderen wird eine arithmetische Umformung definiert. Die Aussagen

$$\forall u(\exists t 2t = u^2 \rightarrow \exists s 2s = u)$$
$$\forall x\forall y\forall z(2x = y^2 \wedge 2z = y \rightarrow 2z^2 = x)$$

werden im Dialogspiel als Hypothesen gesetzt. Die Kommandodatei für diesen Beweis lautet:

```
; Kommandodatei für die Irrationalität von Wurzel 2
deklariere sorte nat
deklariere konstante 2 nat
deklariere variable p nat
deklariere variable q nat
deklariere variable r nat
deklariere variable s nat
deklariere variable t nat
deklariere variable u nat
deklariere variable x nat
deklariere variable y nat
deklariere variable z nat
deklariere funktion * (nat nat) -> nat infix 3 4 *
```

```
deklariere funktion ^ (nat nat) -> nat infix 6 5 ^
deklariere praedikat = (nat nat) infix =
hypothese A1 ALL u ((EXI t 2 * t = u ^ 2) SUB (EXI s 2 * s = u))
hypothese A2 ALL x ALL y ALL z ((2 * x = y ^ 2 AND 2 * z = y) SUB 2 * z ^ 2 = x)
these NOT EXI p EXI q (2 * q ^ 2 = p ^ 2 AND NOT ((EXI s 2 * s = p) AND
  (EXI s 2 * s = q)))
setze angriffsschranke 2
```

Die Gewinnstrategie zu dieser These und diesen Hypothesen ist zu lang, um in annehmbarer Zeit selbständig gefunden zu werden. Eine im Benutzerspiel erzeugte Gewinnstrategie soll an dieser Stelle nicht detailliert dargestellt werden. Die Abbildung C.1 veranschaulicht jedoch die Struktur des Dialogbaums.

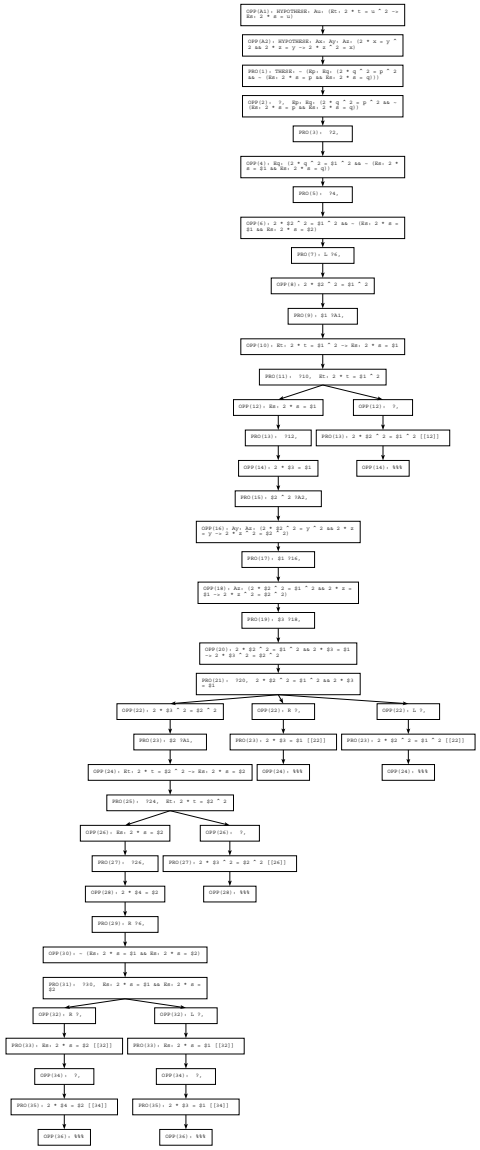

Abbildung C.1: Struktur der Gewinnstrategie

Literaturverzeichnis

[1] Aho, Alfred V., Ravi Sethi und Jeffrey D. Ullman. *Compilerbau Teil I.* Übers. Gerhard Barth u. Mitarbeiter. Bonn: Addison-Wesley, 1988.

[2] Behnke, Heinrich [u. a.], Hg. *Grundzüge der Mathematik Band I: Grundlagen der Mathematik. Arithmetik und Algebra.* 3., durchgesehene und erweiterte Auflage. Göttingen: Vandenhoek und Ruprecht, 1966.

[3] Clinger, William, und Jonathan Rees, Hg. „Revised^4 Report on the Algorithmic Language Scheme". *ACM Lisp Pointers*, Vol. 3, No. 4, 1991.

[4] Eigenschink, Todd R., Dave Love und Aubrey Jaffer. *SLIB : The Portable Scheme Library.* Version 2a3. Juni 1995

[5] Felscher, Walter. „Dialogues, Strategies and Intuitionistic Provability". *Annals of Pure and Applied Logic* 28 (1985): 217–254

[6] Felscher, Walter. „Dialogues as a Foundation for Intuitionistic Logic". *Handbook of Philosophical Logic.* Vol. III. Hg. Dov M. Gabbay und Franz Guenthner. Dordrecht: D. Reidel, 1986. 341–372.

[7] Gallesio, Erick. *STk Reference Manual.* Version 3.0. Université de Nice - Sophia Antipolis, Juli 1995.

[8] Haas, Gerrit. *Programme zur dialogischen Logik.* Arbeitsberichte des IMMD. Bd. 3, Nr. 4. Universität Erlangen-Nürnberg, Oktober 1970.

[9] Haas, Gerrit. *Konstruktive Einführung in die formale Logik.* Mannheim: Bibliographisches Institut, 1984.

[10] Haynes, Christopher T. „Logic Continuations". *Proceedings of the International Conference on Logic Programming.* 225 (1986). Berlin: Springer, 1986. 671–685.

[11] Kamlah, Wilhelm, und Paul Lorenzen. *Logische Propädeutik : Vorschule des vernünftigen Redens.* 2., verb. u. erw. Auflage, unveränd. Nachdruck. Mannheim: BI-Wissenschaftsverlag, 1990.

[12] Lorenz, Kuno. *Arithmetik und Logik als Spiele*. Diss. Kiel, 1961. (Teilnachdruck in P. Lorenzen/K. Lorenz: *Dialogische Logik*. 17–95).

[13] Lorenzen, Paul. „Logik und Agon". *Atti del XII Congresso Internazionale di Filosofia* (Venedig 1958). Bd. 4. Florenz: 1960. 187-194. (Nachdruck in P. Lorenzen/K. Lorenz: *Dialogische Logik*. 1–8).

[14] Lorenzen, Paul. „Ein dialogisches Konstruktivitätskriterium". *Infinitistic Methods. Proceedings of the Symposium on Foundations of Mathematics* (Warschau 1959). Oxford: Pergamon, 1961. 193–200. (Nachdruck in P. Lorenzen/K. Lorenz: *Dialogische Logik*. 9–16).

[15] Lorenzen, Paul, und Kuno Lorenz. *Dialogische Logik*. Darmstadt: Wissenschaftliche Buchgesellschaft, 1978.

[16] Ousterhout, John K. *Tcl und Tk : Entwicklung grafischer Benutzerschnittstellen für das X Window System*. Bonn: Addison-Wesley, 1995.

[17] Stekeler-Weithofer, Pirmin. *Grundprobleme der Logik : Elemente einer Kritik der formalen Vernunft*. Berlin: de Gruyter, 1986.

[18] Thiel, Christian. *Elementare Logik*. Kurseinheit 2. Fernuniversität Hagen, 1983.