User-Adaptive Proof Explanation

Armin Fiedler

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Saarbrücken, 2001

Dekan	Prof. Dr. Rainer Schulze-Pillot-Ziemen, Universität des Saarlandes, Saarbrücken
Prüfungvorsitzender	Prof. Dr. Dr. h.c. Wolfgang Wahlster, Universität des Saarlandes, Saarbrücken
1. Gutachter	Prof. Dr. Jörg Siekmann, Universität des Saarlandes, Saarbrücken
2. Gutachter	Prof. Dr. Werner Tack, Universität des Saarlandes, Saarbrücken
3. Gutachter	Prof. Dr. Frank Pfenning, Carnegie Mellon University, Pittsburgh, PA, U.S.A.
Kolloquium	18.10.2001, Universität des Saarlandes, Saarbrücken

Zusammenfassung

In jüngster Zeit werden automatische Beweissysteme für industrielle Zwecke immer wichtiger und im Mathematikunterricht zunehmend anwendbar. In vielen Anwendungen ist es wichtig, daß das Deduktionssystem seine Beweise dem menschlichen Benutzer in geeigneter Weise vermittelt. Daher werden spezielle Beweispräsentationssysteme entwickelt.

Allerdings sind auch die modernsten Beweispräsentationssysteme in mehrfacher Hinsicht unzulänglich. Erstens präsentieren sie lediglich die Beweise, bestenfalls in einem lehrbuchähnlichem Format, ohne zu motivieren, warum der Beweis so geführt wurde. Zweitens vernachlässigen alle heutigen Systeme die Benutzermodellierung und verzichten dadurch auf die Möglichkeit, die Präsentation an den jeweiligen Benutzer anzupassen, sowohl hinsichtlich der Abstraktionsebene, auf der der Beweis dargestellt wird, als auch hinsichtlich solcher Schritte, die trivial oder vom Benutzer leicht zu sehen sind und daher weggelassen werden sollten. Schließlich erlauben sie dem Benutzer keine Interaktion. Der Benutzer kann das System weder informieren, daß er einen Beweisteil nicht versteht, noch um eine andere Erklärung bitten. Ebensowenig kann er Folgefragen oder Fragen zum Hintergrund des Beweises stellen.

Als ersten Schritt diese Probleme zu beheben entwickeln wir in dieser Arbeit ein Berechnungsmodell für benutzeradaptive Beweiserklärung, die in dem generischen, benutzeradaptiven Beweiserklärungssystem ***P.rex*** (für ***p**roof **ex**plainer*) implementiert ist. Dafür benutzen wir Techniken aus drei verschiedenen Gebieten, nämlich erstens aus der mathematischen Logik, um Beweise aus verschiedenen Kalkülen mit mehreren Abstraktionsebenen zu repräsentieren, wobei die Korrektheit der Beweise sichergestellt wird, zweitens aus der Kognitionswissenschaft, um das mathematische Wissen und die mathematischen Fertigkeiten des Benutzers zu modellieren, und drittens aus der Sprachverarbeitung, um die Beweiserklärung zu planen und um Benutzereingaben zu erlauben und auf sie geeignet zu reagieren.

Abstract

Today, automated theorem provers are becoming more and more important in practical industrial applications and more and more useful in mathematical education. For many applications, it is important that a deduction system communicates its proofs reasonably well to the human user. To this end, proof presentation systems have been developed.

However, state-of-the-art proof presentation systems suffer from several deficiencies. First, they simply present the proofs, at best in a textbook-like format, without motivating why the proof is done as it is done. Second, they neglect the issue of user modeling and thus forgo the ability to adapt the presentation to the specific user, both with respect to the level of abstraction chosen for the presentation and with respect to steps that are trivial or easily inferable by the particular user and, therefore, should be omitted. Finally, they do not allow the user to interact with the system. He can neither inform the system that he has not understood some part of the proof, nor ask for a different explanation. Similarly, he cannot ask follow-up questions or questions about the background of the proof.

As a first step to overcome these deficiencies, we shall develop in this thesis a computational model of user-adaptive proof explanation, which is implemented in a generic, user-adaptive proof explanation system, called ***P.rex*** (for ***p**roof **ex**plainer*). To do so, we shall use techniques from three different fields, namely from computational logic to represent proofs from various calculi with several levels of abstractions ensuring the correctness of the proofs; from cognitive science to model the users mathematical knowledge and skills; and from natural language processing to plan the explanation of the proofs and to accept and appropriately react to the user's interactions.

Contents

Zusammenfassung

Motivation und Problemstellung

Seit Alters her gilt die Mathematik als eine „Königsdisziplin“, insofern als sie die herausragendsten kognitiven Fähigkeiten des Menschen fordert. Daher ist es nicht verwunderlich, daß bald nach der Entwicklung der ersten Computer und der Geburtsstunde der Künstlichen Intelligenz auf der Dartmouth-Konferenz im Jahre 1956 auch die ersten Systeme zum automatischen Beweisen mathematischer Sätze vorgestellt wurden (siehe z.B. [Davis, 1957; Newell *et al.*, 1957; Gelernter, 1959]).

In jüngster Zeit werden automatische Beweissysteme immer wichtiger für industrielle Anwendungen. So gehören sie heute zu den Standardwerkzeugen bei der Hard- und Softwareverifikation. Auch in mathematischen Anwendungen gibt es bereits erste ernstzunehmende Erfolge. Beispielsweise wurde 1996 für das sogenannte *Robbins Problem*, ein algebraisches Problem, das mehr als 60 Jahre ungelöst blieb, durch das Beweissystem EQP [McCune, 1997b] eine Lösung gefunden. Des weiteren sind Beweisplanungssysteme wie ΩMEGA [Benzmüller *et al.*, 1997] und Clam [Bundy *et al.*, 1990] in der Lage, nicht-triviale mathematische Sätze aus verschiedensten Gebieten zu lösen. Darüberhinaus werden Deduktionssysteme auch im Mathematikunterricht [Melis *et al.*, 2001] und (in beschränktem Maße) in der täglichen Arbeit des Mathematikers immer nützlicher.

In all diesen Anwendungen ist es wichtig, daß das Deduktionssystem seine Beweise dem menschlichen Benutzer in geeigneter Weise kommuniziert. Einige Systeme liefern als Ausgabe lediglich ein „Ja“ oder „Nein“, was dem Benutzer ein Verständnis des Beweises völlig unmöglich macht. Systeme mit aussagekräftigerer Ausgabe liefern dagegen eine formale Darstellung des Beweises. Allerdings basieren die meisten automatischen Beweiser auf maschinenorientierten Kalkülen wie dem Resolutionskalkül [Robinson, 1965], die bezüglich der Beweissuche optimiert sind und nicht notwendigerweise bezüglich ihrer Lesbarkeit. Da die Schlüsse in diesen Beweisen oft unnatürlich und obskur sind, sind die Beweise kaum lesbar und nur sehr schwer zu verstehen, sogar für Spezialisten. Daher wurden Algorithmen entwickelt [Andrews, 1980; Kursawe, 1982; Miller, 1984; Pfenning, 1987; Lingenfelder, 1990], die Beweise aus maschinenorientierten Kalkülen in mehr menschorientierte Kalküle wie den Kalkül des natürlichen Schließens (ND für *natural deduction*) [Gentzen, 1935] übertragen.

Allerdings stellten sich die resultierenden ND-Beweise als bei weitem nicht zufriedenstellend heraus. Das liegt daran, daß die ND-Beweise im Vergleich zum Originalbeweis sehr groß und umständlich sind. Darüberhinaus besteht im ND-Kalkül ein Inferenzschritt lediglich in der syntaktischen Manipulation von Quantoren und Junktoren, wogegen in von Menschen geführten Beweisen ein Ableitungsschritt oft durch die Anwendung einer Definition, eines Axioms, eines Lemmas oder eines Theorems (zusammengefaßt *Fakten* genannt) besteht. Basierend auf dieser Beobachtung wurden Abstraktionen von ND-Beweisen definiert, in denen Beweisschritte entweder durch ND-Inferenzregeln oder durch Faktenanwendungen begründet werden können

[Huang, 1994c; Horacek, 1999].

Da traditionelle automatische Beweiser im allgemeinen lediglich Beweise finden, die Mathematiker als einfach ansehen, wurden Ansätze entwickelt, die auf Beweisverfahren basieren, die der menschlichen Vorgehensweise eher entsprechen, wie z. B. Planungsansätze [Bundy *et al.*, 1990; Benzmüller *et al.*, 1997]. Die wesentliche Idee dabei besteht darin, Beweistechniken, wie sie von Mathematikern benutzt werden, in Planoperatoren einzubetten, die dann von einem Beweisplaner verwandt werden, um einen Beweisplan zu finden. Da ein solcher Beweisplan eine abstrakte Repräsentation eines Beweises darstellt und die entsprechende Abstraktionsebene viel besser zur Kommunikation geeignet ist als die Kalkülebene, entfällt die Notwendigkeit zur Beweistransformation auf der Kalkülebene.

Allerdings muß festgestellt werden, daß es für die praktische Anwendbarkeit von Beweissystemen nicht ausreicht, daß die Beweise in einem obwohl am Menschen orientierten, dennoch aber formalen Format präsentiert werden. Stattdessen sollten die Beweise dem Menschen so dargeboten werden, wie er es gewohnt ist, nämlich in natürlicher Sprache. Zu diesem Zweck wurden spezielle Beweispräsentationssysteme entwickelt.

Die bisher auf dem Gebiet der Deduktion entwickelten Beweispräsentationssysteme (z. B. [Chester, 1976; Edgar and Pelletier, 1993; Coscoy *et al.*, 1995; Dahn *et al.*, 1997]) sind jedoch in mehrfacher Hinsicht unzulänglich. Automatisch gefundene Beweise enthalten oft unabhängig von der Abstraktionsebene Beweisschritte, die logisch gesehen unabdingbar sind, von Menschen jedoch niemals explizit gemacht würden, da sie offensichtlich erscheinen. Da die meisten Beweispräsentationssysteme über keinerlei Techniken verfügen, die solche Schritte in der Beweisdarstellung auslassen könnten, präsentieren sie jeden einzelnen Beweisschritt. Dadurch werden die Beweise viel zu detailliert dargestellt, so daß selbst kleine Beweise nicht mehr leicht zu überschauen sind. Darüberhinaus benutzen diese Systeme zur Verbalisierung der Schritte meist Schablonen mit Versatzstücken vorgefertiger Sätze. Daher lassen die ausgegeben Texte Verbindungen zwischen den einzelnen Textsegmenten vermissen, wodurch der Text unzusammenhängend wirkt und ein Verständnis des Textes erheblich gestört wird. Die Situation wird noch verschlimmert, wenn Beweispräsentationssysteme nur die Inferenzregeln natürlichsprachlich ausdrücken, die Prämissen und Konklusionen der Beweisschritte aber als logische Formeln ausgeben, die oft unverständlich sind.

Um diese Probleme zu beheben wurden auf dem Gebiet der Sprachgenerierung weit ausgefeiltere Beweispräsentationssysteme entwickelt (z. B. [Huang and Fiedler, 1997; Holland-Minkley *et al.*, 1999]). Diese Systeme entscheiden, welche Information in der Präsentation enthalten sein soll, und planen die rhetorische Struktur des Textes im voraus, um einen kohärenten, das heißt inhaltlich zusammenhängenden Text zu erhalten. Darüberhinaus planen sie auch die innere Struktur der einzelnen Sätze und die Morphologie der einzelne Worte. Außerdem stellen sie genügend Variabilität in der Verbalisierung dadurch sicher, daß sie zwischen mehreren möglichen Alternativen auswählen. Sie vermeiden Redundanzen dadurch, daß sie Informationen miteinander kombinieren, und verbessern den Zusammenhalt der einzelnen Sätze durch Verwendung passender Referenzausdrücke. Zusammengenommen erlauben diese Techniken den heutigen Beweispräsentationssystemen die Beweise in fast lehrbuchähnlicher Form auszugeben.

Allerdings leiden auch die fortgeschrittensten Beweispräsentationssysteme an einigen prinzipiellen Schwächen. Erstens gehen diese Systeme von einem einzigen kommunikativen Ziel aus, nämlich den Beweis lehrbuchähnlich zu präsentieren. Während ein Mathematiker, der lediglich die Korrektheit eines Beweises überprüfen will, mit einem solchen *lehrbuchartigen* Beweis, in dem die Abfolge der Ableitungsschritte im Vordergrund steht, zufrieden ist, benötigt ein Schüler oder Student, der lernen will, wie man solche Beweise selbst führt, eine ausführlichere, unterrichtsähnliche

Erklärung des Beweises. In einer solchen *unterrichtsorientierten* Erklärung sollte der Grund, *warum* ein bestimmter Schritt gemacht wird, betont werden, anstatt lediglich festzustellen, *daß* der Schritt gemacht wurde [Leron, 1983].

Zweitens vernachlässigen alle heutigen Systeme die Benutzermodellierung und können deshalb die Präsentation nicht an den jeweiligen Benutzer anpassen. Sie präsentieren den Beweis auf einer fest vorgegebenen Abstraktionsebene, die für einen Anfänger zu hoch, für einen Experten aber zu niedrig sein mag, ohne zu erlauben zwischen den einzelnen Abstraktionsebenen hin und her zu wechseln. Ein ähnliches Problem ist, daß sie oft den Explizitheitsgrad nicht auf die kognitiven Fähigkeiten des Benutzers abstellen. Da sie die kognitiven Fertigkeiten des Benutzers nicht modellieren, können sie Schritte, die trivial oder leicht zu sehen sind, nicht einfach weglassen. [Horacek, 1999] führt einen Ansatz ein, wie man modellieren kann, welche Schritte leicht zu sehen sind.

Schließlich ist die Möglichkeit Fragen zu stellen und eine angepaßt Erklärung zu bekommen, falls der Benutzer Teile des Beweises nicht versteht, wünschenswert oder gar notwendig, zumindest für intelligente tutorielle Systeme für Mathematik und für mathematische Assistenzsysteme. Aber die heutigen Beweispräsentationssysteme erlauben dem Benutzer keine Interaktion. Der Benutzer kann das System weder informieren, daß er einen Beweisteil nicht versteht, noch um eine andere Erklärung bitten. Ebensowenig kann er Folgefragen oder Fragen zum Hintergrund des Beweises stellen.

Lösungsansatz

Als ersten Schritt diese Probleme zu beheben entwickeln wir in dieser Arbeit ein Berechnungsmodell für benutzeradaptive Beweiserklärung, die in dem generischen, benutzeradaptiven Beweiserklärungssystem ***P.rex*** (für ***proof explainer***) implementiert ist. Dabei stellen wir die folgenden Anforderungen an das System: Das System soll die Erklärung an den Benutzer anpassen, und zwar hinsichtlich der Abstraktionsebene, auf der der Beweis erläutert wird, dem Explizitheitsgrad, mit dem der Beweis erklärt wird, und den kognitiven Fähigkeiten des Benutzers, um leicht zu sehende Schritte wegzulassen. Weiterhin soll die Erklärung verschiedenen Darstellungsarten genügen, wie z. B. der lehrbuchartigen und der unterrichtsartigen Darstellung. Darüberhinaus soll das System dem Benutzer erlauben, mit dem System zu interagieren, wenn er mit einer Erklärung nicht zufrieden ist, und Folgefragen oder Fragen zum Hintergrund zu stellen. Die Metapher, die wir hier benutzen wollen, ist die des Mathematikers, der einen Beweis seinen Schülern oder seinen Kollegen erklärt.

Zur Problemlösung kombinieren wir Techniken aus drei verschiedenen Gebieten, nämlich aus der Kognitionswissenschaft, der mathematischen Logik und der Sprachverarbeitung, auf die wir im folgenden näher eingehen wollen.

Moderne Sprachgenerierungssysteme berücksichtigen das Wissen des Benutzers bei der Generierung von Erklärungen (vgl. z. B. [Cawsey, 1990; Paris, 1991a; Wahlster *et al.*, 1993]). Die meisten dieser Systeme passen sich an den Benutzer dadurch an, daß sie zwischen verschiedenen Diskursstrategien auswählen. Da Beweise stets reich an Inferenzen sind, muß die Beweiserklärung auch berücksichtigen, welche Inferenzen der Benutzer selbst machen kann [Zukerman and McConachy, 1993; Horacek, 1997b; 1999]. Wegen der Beschränkungen des menschlichen Gedächtnisses sind solche Inferenzen jedoch nicht ohne Probleme einfach verkettbar. Bei der Auswahl der zu vermittelnden Information hat sich die explizite Repräsentation der kognitiven Zustände des Benutzers als nützlich erwiesen [Walker and Rambow, 1994].

In der Kognitionswissenschaft wurden verschiedene Theorien der menschlichen

Kognition durch sogenannte *kognitive Architekturen* beschrieben, das heißt durch die festgelegte Struktur, die den kognitiven Apparat realisiert (z. B. [Newell, 1990; Meyer and Kieras, 1997a; Anderson and Lebiere, 1998]). Eine dieser Theorien ist ACT-R [Anderson and Lebiere, 1998], eine Theorie der Kognition, die deklaratives und prozedurales Wissen in ein deklaratives Gedächtnis und eine Produktionsregelbasis aufspaltet. Die Datenstruktur eines Kellers zur Verwaltung von Zielen erlaubt ACT-R ein Ziel dadurch zu erfüllen, daß es in Teilziele aufgeteilt wird, die unabhängig voneinander erfüllt werden können.

Prozedurales Wissen wird in ACT-R durch Produktionsregeln repräsentiert, deren Vorbedingungs- und Aktionsteile durch deklarative Strukturen definiert werden. Eine Produktionsregel kann nur dann angewandt werden, wenn ihre Vorbedingungen durch das derzeitig im deklarativen Gedächtnis vorhandene Wissen erfüllt werden. Jedes Element im deklarativen Gedächtnis ist mit einer numerischen Größe, Aktivität genannt, verknüpft, die den Zugriff auf das Element beeinflußt. Die Anwendung einer Produktionsregel verändert entweder das deklarative Gedächtnis oder mündet in einem beobachtbaren Ereignis. Wenn mehrere Produktionsregeln gleichzeitig anwendbar sind, entscheidet eine Konfliktlösungsheuristik, die aus einer rationalen Analyse der menschlichen Kognition abgeleitet wurde [Anderson, 1990], welche Produktionsregel schließlich angewandt wird.

ACT-R verbindet also die Möglichkeit zur *Benutzermodellierung* einerseits und zur *Planung* andererseits in einem einheitlichen Rahmen und ist daher als Basis zur benutzeradaptiven Dialogplanung besonders geeignet. Mit ACT-R modellieren wir einen Lehrer, der seinem Schüler einen mathematischen Beweis erklärt. Insbesondere modellieren wir die Annahmen des Lehrers über die kognitiven Zustände des Schülers während die Erklärung voranschreitet (was in dem Benutzermodell resultiert) und das Wissen des Lehrers über die mathematischen Theorien und wie man Beweise aus einer solchen Theorie erklärt (was zur Planung der Erklärung benötigt wird). Anhand der Annahmen über den kognitiven Zustand des Benutzers wählt der Planer eine Abstraktionsebene zur Erklärung aus und entscheidet, welche Schritte er als vom Benutzer leicht zu sehen annimmt.

Eine wichtige Voraussetzung, die *P.rex* die Auswahl zwischen verschiedenen Abstraktionsebenen erst erlaubt, ist die gleichzeitige Repräsentation eines Beweises auf mehreren Abstraktionsebenen. Diese Repräsentation, die auch die Schnittstelle zwischen *P.rex* und angeschlossenen automatischen Beweisern definiert, ist in TWEGA realisiert, einer Implementation einer Erweiterung des *logical frameworks* LF [Harper *et al.*, 1993], die dem *calculus of constructions* (λC) [Coquand and Huet, 1988] entspricht. LF und λC sind mächtige getypte λ-Kalküle, die erlauben andere logische Kalküle zu repräsentieren, insbesondere auch die Kalküle von anzuschließenden automatischen Beweisern. Sowohl der Eingabebeweis, der erklärt werden soll, als auch alle relevante Information aus den mathematischen Theorien, die der Beweis benötigt, werden in TWEGA kodiert und dann von *P.rex* benutzt. Insbesondere erlaubt TWEGA durch einen Expansionsmechanismus, der zu einem Ableitungsschritt einen Teilbeweis auf einer niedrigeren Abstraktionsebene definiert, einen Beweis auf mehreren Abstraktionsebenen gleichzeitig zu repräsentieren. Die typtheoretischen Grundlagen von TWEGA garantieren, daß ausschließlich solche Beweise repräsentiert werden können, die gemäß des Kalküls des angeschlossenen Beweisers korrekt sind.

Die erfolgreiche Kommunikation mittels eines Sprachgenerierungssystems setzt voraus, daß der zu übermittelnde Inhalt passend strukturiert ist, um eine kohärente semantische Organisation der Ausgabe zu gewährleisten. Für ein Erklärungssystem ist es weiterhin wichtig, Rückmeldungen und Folgefragen durch den Benutzer zu erlauben. Um mißverstandene Erklärungen richtigzustellen muß das System sowohl die verschiedenen Teile der Erklärung, als auch die Relationen zwischen diesen Teilen repräsentieren.

Eine Diskurstheorie, die solche Repräsentationen erlaubt, wurde von Mann und Thompson [1987] formuliert. Gemäß dieser RST (für *Rhetorical Structure Theory*) genannten Theorie können die Relationen zwischen Segmenten normalen englischen Texts durch eine endliche Menge von Relationen beschrieben werden. Basierend auf RST beschrieb Hovy [1993] Diskurse als Verschachtelung von Diskurssegmenten. Nach seiner Diskurstheorie enthält jedes Diskurssegment im wesentlichen das kommunikative Ziel, das der Sprecher mit diesem Segment erfüllen will, und entweder eine bis mehrere Untersegmente mit den dazugehörigen Relationen oder den semantischen Inhalt, der kommuniziert werden soll.

Grosz und Sidner [1986] entwickelten eine Diskurstheorie, die drei unterschiedliche, aber miteinander verwobene Komponenten unterscheidet, nämlich die linguistische Struktur, die intentionale Struktur und den attentionalen Zustand. Während die linguistische Struktur die Segmentierung des Diskurses beschreibt, beschreibt die intentionale Struktur, wie sich die Ziele der Diskurssegmente zueinander verhalten. Der attentionale Zustand ist eine Abstraktion des Aufmerksamkeitsfokus der Gesprächsteilnehmer während des Diskurses und modelliert die *Salienz* des Diskursinhalts, das heißt, inwiefern die Gesprächsteilnehmer auf den entsprechenden Inhalt in dem derzeitigen Kontext kognitiv zugreifen können.

Für *P.rex* definieren wir *Diskursstrukturbäume* zur Diskursrepräsentation. Diese Repräsentation paßt Hovys Diskurssegmente für unsere Zwecke an und verbindet sie mit den attentionalen Zuständen aus Grosz und Sidners Theorie. Dies erlaubt sowohl die Diskurssegmentierung, als auch den Aufmerksamkeitsfokus in einer gemeinsamen Umgebung zu modellieren. Da die attentionalen Zustände die Salienz modellieren, werden sie zur Auswahl des Explizitheitsgrades und der Referenzausdrücke benutzt. Die explizite Repräsentation des Diskurszwecks erlaubt die Präsentation in verschiedenen Darstellungsarten, wie die lehrbuchartige oder die unterrichtsartige Darstellung. Darüberhinaus berücksichtigen die Diskursstrukturbäume auch eingeschränkte Formen von Dialog, nämlich bestimmte Formen von Unterbrechungen und Klärungsdialogen. Dies ist eine notwendige Voraussetzung, um dem System die Repräsentation von Benutzerinteraktionen sowie die passende Reaktion darauf zu erlauben.

Architektur und Funktionsweise

Ein Überblick über die Architektur von *P.rex* ist in Abbildung 1 gegeben.

Wie bereits ausgeführt definiert TWEGA die Schnittstelle, über die automatische Beweiser an *P.rex* angebunden werden können. Dies geschieht dadurch, daß der Kalkül des Beweisers, die Beweise selbst sowie die zu ihnen in Beziehung stehenden mathematischen Konzepte in TWEGA kodiert werden. Die typtheoretischen Grundlagen von TWEGA garantieren die Korrektheit der repräsentierten Beweise.

Die zentrale Komponente des Systems ist der *Dialogplaner*. Er wählt und sortiert den Inhalt, der dargestellt werden soll, und organisiert ihn in einem Diskursstrukturbaum. Der Dialogplaner ist in ACT-R implementiert. Der Diskursstrukturbaum, der sowohl als Dialogplan, als auch als Diskursgedächtnis dient, wird im deklarativen Gedächtnis gespeichert. Die Planoperatoren, die den Diskursstrukturbaum aufbauen, sind als ACT-R-Produktionsregeln definiert. Der Dialogplaner modelliert den Benutzer dadurch, daß er das vermutete deklarative und prozedurale Wissen des Benutzers im deklarativen Gedächtnis bzw. in der Produktionsregelbasis ablegt.

Um nun einen konkreten Beweis zu erklären, nimmt der Dialogplaner zunächst den vermuteten kognitiven Zustand des Benutzers an, indem er sein deklaratives Gedächtnis und seine Produktionsregelbasis mit dem entsprechenden Benutzermodell abgleicht, das nach einer früheren Sitzung in der Datenbank der Benutzermodelle gespeichert wurde. Jedes Benutzermodell enthält Annahmen über das Wissen

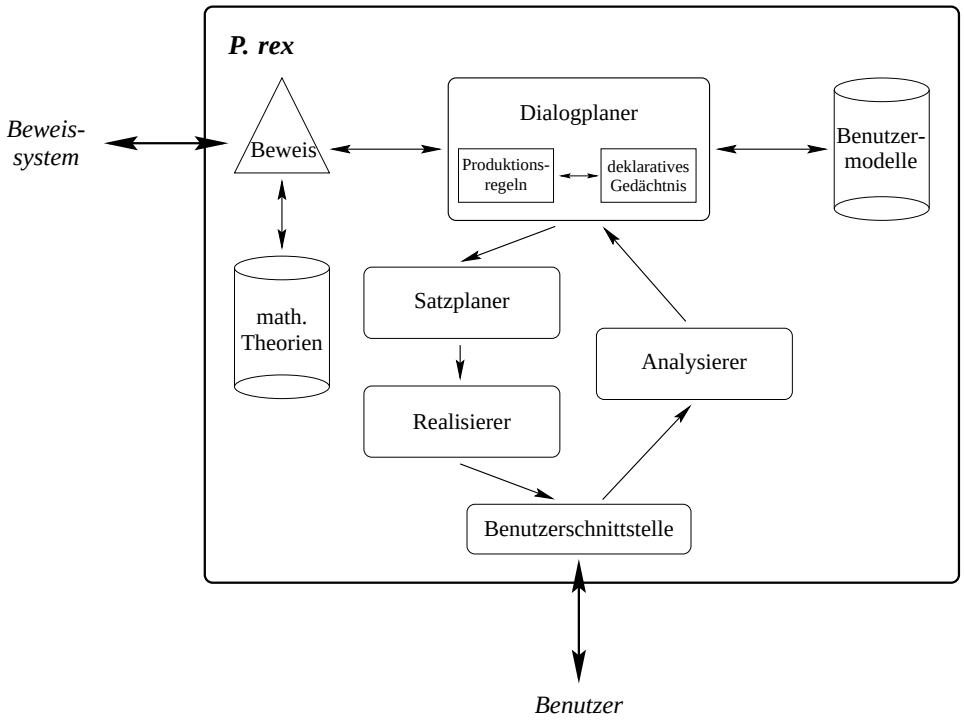


Abbildung 1. Die Architektur von *P.rex*.

des Benutzers, das für die Beweiserklärung relevant ist. Insbesondere enthält es Annahmen darüber, welche mathematischen Theorien, Definitionen, Beweise, Inferenzmethoden und mathematischen Fakten er kennt.

Danach setzt sich der Dialogplaner das übergeordnete Ziel, den Beweis zu zeigen. ACT-R versucht dieses Ziel dadurch zu erfüllen, daß es nacheinander Produktionsregeln anwendet, die Ziele entweder direkt erfüllen oder diese in Teilziele zerlegen. Dabei erzeugt der Dialogplaner nicht nur einen Dialogplan in Form eines Diskursstrukturbaumes, sondern verfolgt auch die kognitiven Zustände des Benutzers, während die Erklärung voranschreitet. Letzteres erlaubt dem System sowohl eine benutzerangepaßte Erklärung zu generieren, als auch flexibel auf die Interaktionen des Benutzers zu reagieren: Der Dialogplaner interpretiert zunächst die Interaktionen indem er neue Dialogziele erzeugt. Diese werden dann im Anschluß getrennt angegangen.

Der von dem Dialogplaner erzeugte Dialogplan wird dann von dem *Satzplaner* weiterverarbeitet. Um die interne Struktur der Sätze in der Ausgabe von *P.rex* zu planen wurde der Satzplaner von *PROVERB* [Fiedler, 1996; Huang and Fiedler, 1997] angepaßt und erweitert. Zu den Aufgaben des Satzplaners gehört es, Domänenkonzepte wo möglich zu aggregieren und sie in eine linguistische Struktur abzubilden. Diese spezifiziert die konkreten Worte und Referenzausdrücke, die die Domänenkonzepte ausdrücken, sowie den Skopus der Phrasen und Sätze.

Die linguistische Struktur wird dann durch den syntaktischen *Realisierer* TAG-GEN [Kilger and Finkler, 1995], der die korrekte Morphologie der Wörter sicherstellt, realisiert.

Die in *P.rex* verfolgte Anordnung von Dialogplaner, Satzplaner und Realisierer in einer Pipeline entspricht der Architektur der meisten Sprachgenerierungssysteme. Sie wurde daher von Reiter als Konsensarchitektur charakterisiert [Reiter, 1994].

Die von *P.rex* erzeugten Äußerungen werden dem Benutzer schließlich über eine

Benutzerschnittstelle dargeboten, die dem Benutzer darüberhinaus erlaubt, Anmerkungen, Fragen oder Befehle an das System zu richten. Ein *Analysierer* übersetzt die Interaktionen des Benutzers in neue Diskursziele, die anschließend von dem Dialogplaner angegangen werden. Da die Analyse natürlicher Sprache den Rahmen dieser Arbeit bei weitem sprengen würde, begnügen wir uns mit einem einfachen Analysierer, der nur eine kleine Menge vordefinierter Interaktionen zu verstehen vermag.

Diskussion

Der Dialogplaner von *P.rex* ist als *hybrider* Planer realisiert, das heißt, er kombiniert Planung mit Diskursrelationen, die für die Planung kleinerer Strukturen wie einzelner aufeinanderfolgender Äußerungen besonders geeignet ist, mit schemabasierter Planung, deren Vorteile in der Planung größerer Strukturen wie z. B. Absätze oder ganzer Textpassagen liegen. Zur Repräsentation des Dialogs baut er einen Diskursstrukturbaum auf, der Hovys RST-artigen Ansatz [1993] mit Grosz und Sidners Konzept attentionaler Zustände [1986] verknüpft. Die attentionalen Zustände werden zur Modellierung der Salienz benötigt. Der Dialogplaner kann so entscheiden, wie explizit er sich auf ein bereits eingeführtes Objekt bezieht.

Neben dieser Anpassung im Explizitheitsgrad mithilfe der attentionalen Zustände paßt der Dialogplaner seine Darstellung auch auf den Benutzer an, indem er den Beweis einerseits auf einer möglichst abstrakten, aber dem Benutzer noch bekannten Ebene erklärt, andererseits jegliche leicht vom Benutzer zu sehenende Schritte in der Erklärung wegläßt. Des weiteren kombiniert der Dialogplaner zwei Darstellungsarten in Abhängigkeit davon, wie gut sich der Benutzer mit dem jeweiligen Thema auskennt. Wo sich der Benutzer mit der zugrundeliegenden Theorie gut auskennt wird der Beweis lehrbuchartig dargestellt, während in unbekannteren Theorien eine unterrichtsartige Erklärung erfolgt. Dieser Wechsel zwischen den Darstellungsarten kann insbesondere innerhalb eines Beweise erfolgen.

Das System erlaubt es dem Benutzer jederzeit einzugreifen, wenn er mit der Erklärung nicht zufrieden ist. Es reagiert auf eine solche Unterbrechung mit der erneuten Planung der beanstandeten Beweisteile. Um diese Beweisteile zu identifizieren benutzt der Dialogplaner strukturelle Information, die in den Diskursstrukturbäumen explizit repräsentiert ist.

P.rex ist anfänglich mit 91 domänenunabhängigen Produktionsregeln ausgestattet. Durch die Definition von domänenabhängigen Schemata werden weitere auf die jeweilige mathematische Theorie spezialisierte Produktionsregeln hinzugefügt. Darüberhinaus können durch einen Lernprozeß in ACT-R zur Laufzeit weitere Produktionsregeln dynamisch generiert werden.

P.rex wurde bislang zur Erklärung von Dutzenden Beweisen aus der Gruppentheorie und der Theorie der Prädikatenlogik erster Stufe erfolgreich eingesetzt.

Überblick

Diese Arbeit ist wie folgt aufgebaut: Nach einer Einleitung in Kapitel 1 wird in Kapitel 2 in das Gebiet der Sprachgenerierung und der Beweispräsentation eingeführt. Weiterhin werden Anforderungen an die Beweiserklärung formuliert und ein Überblick über die Architektur von *P.rex* gegeben. Anschließend wird in Kapitel 3 TWEGA formal definiert und der ND-Kalkül, der von mehreren automatischen Beweisern benutzt wird, in TWEGA beispielhaft repräsentiert. Da einige Deduktionssysteme, wie z. B. ΩMEGA [Benzmüller *et al.*, 1997] auf komplexen Kalkülen basieren, die dynamisch repräsentiert werden müssen, wird in Kapitel 4 anhand des Kalküls von

ΩMEGA gezeigt, wie eine solche dynamische Repräsentation zur Laufzeit erfolgen kann.

In Kapitel 5 wird ACT-R eingeführt, die kognitive Theorie, die dem Dialogplaner von *P.rex* als Basis dient. Der Dialogplaner selbst wird anschließend in Kapitel 6 behandelt. In diesem zentralen Kapitel werden sowohl die Diskursstrukturbäume, als auch die Planoperatoren, die sie aufbauen, definiert. Insbesondere wird auf spezialisierte Planoperatoren, die die Beweiserklärung auf den jeweiligen Benutzer anpassen, sowie auf die Systemreaktion auf Benutzerinteraktionen eingegangen.

In Kapitel 7 werden dann die weiteren Komponenten des Systems besprochen, nämlich der Satzplaner, der Realisierer, die Benutzerschnittstelle sowie der Analysierer. Kapitel 8 schließt die vorliegende Arbeit ab.

Summary

Motivation and Problem

Mathematics has been considered one of the most sublime of the human cognitive skills. Therefore, soon after the advent of the first computers in the early fifties and with the advent of artificial intelligence in 1956 at the Dartmouth Conference, researchers began to devise systems for automated theorem proving (see e.g., [Davis, 1957; Newell *et al.*, 1957; Gelernter, 1959]).

Today, automated theorem provers are becoming more and more important in practical industrial applications. For example, they are standard routine in the verification of hardware and software, in deductive data bases and in logic programming. They also have the first considerable successes in mathematical applications. For example, in 1996, the so-called *Robbins Problem*, an algebraic problem that was open for more than 60 years, could be automatically proved by the automated theorem prover EQP [McCune, 1997b]. Proof planners such as ΩMEGA [Benzmüller *et al.*, 1997] and Clam [Bundy *et al.*, 1990] prove non-trivial mathematical theorems from assorted fields. Deduction systems are becoming useful in mathematical education [Melis *et al.*, 2001] and (to a limited extend) the mathematicians' everyday work.

For these applications, it is important that a deduction system communicates its proofs reasonably well to the human user. Some systems provide as output simply a "yes" or "no," thus precluding the user from scrutinizing the proof. More responsive systems return a formal account of the proof. However, most automated theorem provers are based on machine-oriented calculi such as resolution [Robinson, 1965], which are optimized for the process of proof search and not for human readability. Since the lines of reasoning of the output proofs are often unnatural and obscure, the proofs are very difficult to read and hardly comprehensible, even for specialists. Therefore, researchers developed algorithms [Andrews, 1980; Kursawe, 1982; Miller, 1984; Pfenning, 1987; Lingenfelder, 1990] to transform proofs from machine-oriented calculi into more human-oriented calculi such as the natural deduction (ND) calculus [Gentzen, 1935].

But the result of the transformation into the ND calculus turned out to be far from satisfactory. The reason is that the obtained ND proofs are very large and too involved in comparison to the original proof. Moreover, in the ND calculus, an inference step merely consists of the syntactic manipulation of a quantifier or a connective. In human-written proofs, in contrast, an inference step is often described in terms of the application of a definition, an axiom, a lemma or a theorem, collectively called an *assertion*. Based on this observation, abstractions of ND proofs have been defined, where a proof step may be justified either by an ND inference rule or by the application of an assertion [Huang, 1994c; Horacek, 1999].

Since traditional automated theorem provers find proofs for theorems that are usually considered easy by mathematicians, theorem provers based on a more human-

oriented approach such as the application of planning techniques have been developed [Bundy *et al.*, 1990; Benzmüller *et al.*, 1997]. The key idea here is that proof techniques as used by mathematicians are encoded into plan operators, which are used by the proof planner to find a proof plan. Because a proof plan is an abstract representation of a proof, it provides a level that is better suited for communication such that proof transformation at the calculus level becomes obsolete for proof planners.

To be of practical use for the human user, however, it does not suffice to communicate the proofs in an albeit human-oriented, but still formal format. Instead, the proofs should be output in the way people are used to, namely in natural language. To this end, proof presentation systems have been developed.

However, most proof presentation systems that have been developed so far in the field of automated theorem proving (e.g., [Chester, 1976; Edgar and Pelletier, 1993; Coscoy *et al.*, 1995; Dahn *et al.*, 1997]) suffer from several deficiencies. Regardless of the level of abstraction, automatically found proofs often include steps that are logically indispensable but people would not make explicit, because the steps are considered obvious. Since most proof presentation systems employ no techniques to exclude such steps from the presentation, they present every single step of the derivation. Thus, the output proofs are too detailed, such that even small proofs are not easy to follow. Moreover, these systems mostly use templates with canned sentence chunks for the verbalization of the steps. Therefore, the output texts lack interrelations between segments of the text. This leaves the text segments unconnected and, thus, severely obstructs the comprehension of the proof. The situation is even worse in those systems that verbalize only the inference rules in natural language, but output the premises and conclusions as logical formulae, which are still unreadable.

To overcome these problems proof presentation systems that employ more sophisticated techniques from the field of natural language generation have been developed (e.g., [Huang and Fiedler, 1997; Holland-Minkley *et al.*, 1999]). These systems decide which information to include in the presentation and they plan the rhetorical structure of the text in advance to achieve connected texts. They also plan the internal structure of the sentences and the morphology of the words. They ensure variation in the verbalization by choosing between several possible alternatives, avoid redundancies by aggregating information and improve the relatedness of the sentences by choosing appropriate referring expressions. These techniques together enable state-of-the-art proof presentation systems to output the proofs in an almost textbook-like format.

However, even the previously mentioned state-of-the-art proof presentation systems suffer from several deficiencies. Firstly, they usually presuppose a single communicative purpose, namely to present the proof in a *textbook-like* style. However, whereas a mathematician who wants to check the correctness of a proof is often satisfied with a textbook-like presentation, a student who wants to learn how to find a proof usually needs a more elaborate *classroom-style* explanation of the proof. In such a classroom-style explanation, the reason *why* a step is taken should be emphasized as opposed to just mention *that* the step is taken [Leron, 1983].

Secondly, the systems all neglect the issue of user modeling and thus forgo the ability to adapt the presentation to the specific user. They present the proof at a fixed level of abstraction that might be too high for a novice or too low for an expert without allowing for a transition between levels of abstraction. A similar problem is that they often do not adapt the degree of explicitness to the user's cognitive capabilities. Also, since they do not model the user's cognitive skills, they cannot omit steps that are trivial or easily inferable by the particular user. Horacek [1999] introduced an approach how to model which steps are easily inferable.

Finally, in case the user cannot understand some part of the proof, the pos-

sibility to ask questions about it and get an adapted explanation is desirable or even compulsory (for intelligent tutor systems for mathematics and mathematical assistant systems). However, current proof presentation systems do not allow the user to interact with the system. The user cannot tell the system when he did not understand a part of the proof and ask for another explanation of that part. Similarly, the user cannot ask follow-up questions or questions about the background of the proof.

Approach

As a first step to overcome these deficiencies, we shall develop in this thesis a computational model of user-adaptive proof explanation, which is implemented in a generic, user-adaptive proof explanation system, called ***P.rex*** (for ***proof explainer***). The demands we make on the system are the following: The system should adapt to the user with respect to the level of abstraction at which the proof is presented, the degree of explicitness used to explain the proof, and inferential capabilities of the user to omit inferable steps. The explanation should account for different presentation styles, such as the textbook-like style and the classroom-like style. Moreover, the system should allow the user to intervene if he is not satisfied with an explanation and to ask follow-up or background questions. The metaphor we have in mind is a human mathematician who teaches a proof to a student or else explains a proof to a colleague.

In our approach, we combine techniques from three different fields, namely cognitive science, computational logic and natural language processing, as we shall discuss in the following.

Modern natural language generation systems take into account the intended audience's knowledge in the generation of explanations (see e.g. [Cawsey, 1990; Paris, 1991a; Wahlster *et al.*, 1993]). Most of them adapt to the addressee by choosing between different discourse strategies. Since proofs are inherently rich in inferences, the explanation of proofs must also consider which inferences the audience can make [Zukerman and McConachy, 1993; Horacek, 1997b; 1999]. However, because of the constraints of the human memory, inferences are not chainable without costs. Explicit representation of the addressee's cognitive states proves to be useful in choosing the information to convey [Walker and Rambow, 1994].

In cognitive science various theories of human cognition have been described by means of a *cognitive architecture*, that is, the fixed structure that realizes the cognitive apparatus (e.g., [Newell, 1990; Meyer and Kieras, 1997a; Anderson and Lebiere, 1998]). One of these theories is ACT-R [Anderson and Lebiere, 1998], a cognitive architecture that separates declarative and procedural knowledge into a declarative memory and a production rule base, respectively. A goal stack allows ACT-R to fulfill a task by dividing it into subtasks, which can be fulfilled independently.

In ACT-R, procedural knowledge is represented in production rules, whose conditions and actions are defined in terms of declarative structures. A production rule can only apply if its conditions are satisfied by the knowledge currently available in the declarative memory. An item in the declarative memory is annotated with an activation that influences its retrieval. The application of a production rule modifies the declarative memory, or it results in an observable event. If several production rules are applicable a conflict resolution heuristic derived from a rational analysis of human cognition [Anderson, 1990] determines which production rule will eventually be applied.

Hence, ACT-R combines the abilities for *user modeling* and *planning* in a uniform framework and is therefore particularly well suited as a basis for a user-adaptive dialog planner. Using ACT-R, we model a teacher who explains mathematical

proofs to his student. In particular, we model the teacher's assumptions about the students cognitive states during the explanation (which establish the user model) and the teacher's knowledge of the mathematical theories and the way to explain the proof of a theorem in these theories (which is used to plan the explanation). The assumptions about the user's cognitive states are employed to choose the level of abstraction at which a proof is presented and to decide which steps the user can infer.

An important prerequisite that enables *P.rex* to choose among different levels of abstraction is the simultaneous representation of a proof at several levels of abstraction. This representation, which also serves as the interface between theorem provers and *P.rex*, is realized in TWEGA, an implementation of an extension of the *logical framework* LF [Harper *et al.*, 1993] that corresponds to the *calculus of constructions* (λC) [Coquand and Huet, 1988]. LF and λC are very powerful typed lambda calculi that allow us to represent other logical calculi, and thus give us the possibility to represent the calculus of any theorem prover that is to be connected to *P.rex*. The input proof to be explained as well as relevant information from the mathematical theories that relate to the proof are encoded in TWEGA and then used by *P.rex*. In particular, using an expansion mechanism that defines for any derivation step a subproof at the lower level of abstraction, TWEGA allows us to represent a proof at several levels of abstraction simultaneously.

Successful communication via a natural language generation system presupposes that the content to be conveyed is appropriately structured to ensure a coherent semantic organization. For an explanation system, it is also important to accept user feedback and follow-up questions. To be able to clarify misunderstood explanations, the system needs to represent the different parts of the explanation as well as the relations between them.

An appropriate discourse theory that allows for these representations was formulated by Mann and Thompson [1987]. This theory, called *Rhetorical Structure Theory (RST)*, states that the relations that hold between segments of normal English text can be represented by a finite set of relations. Based on RST, Hovy [1993] described discourse as the nesting of discourse segments. According to his discourse theory, each segment essentially contains the communicative goal the speaker wants to fulfill with this segment and either one to several discourse segments with inter-segment discourse relations or the semantic material to be communicated.

Grosz and Sidner [1986] developed a discourse theory that distinguishes three separate, but interrelated components, namely the linguistic structure, the intentional structure and the attentional state. Whereas the linguistic structure describes the segmentation of the discourse, the intentional structure captures how the purposes of the discourse segments relate to one another. The attentional state is an abstraction of the focus of attention of the participants as the discourse unfolds and models the salience of discourse contents.

For *P.rex*, we define *discourse structure trees* as a representation of discourses. This representation adapts Hovy's discourse segments and combines it with the concept of attentional spaces from Grosz and Sidner's theory. This allows us to model both the segmentation of the discourse and the focus of attention in a uniform framework. Since the attentional spaces model salience, they are employed to decide on the degree of explicitness and to choose appropriate referring expressions. Explicit representation of the discourse purpose allows for presentations using different styles, such as the textbook-like style or the classroom-like style. Moreover, discourse structure trees also account for restricted types of dialogs as well, namely certain types of interruptions and clarification dialogs. This is a necessary prerequisite for the system to represent user interactions and to appropriately react to them.

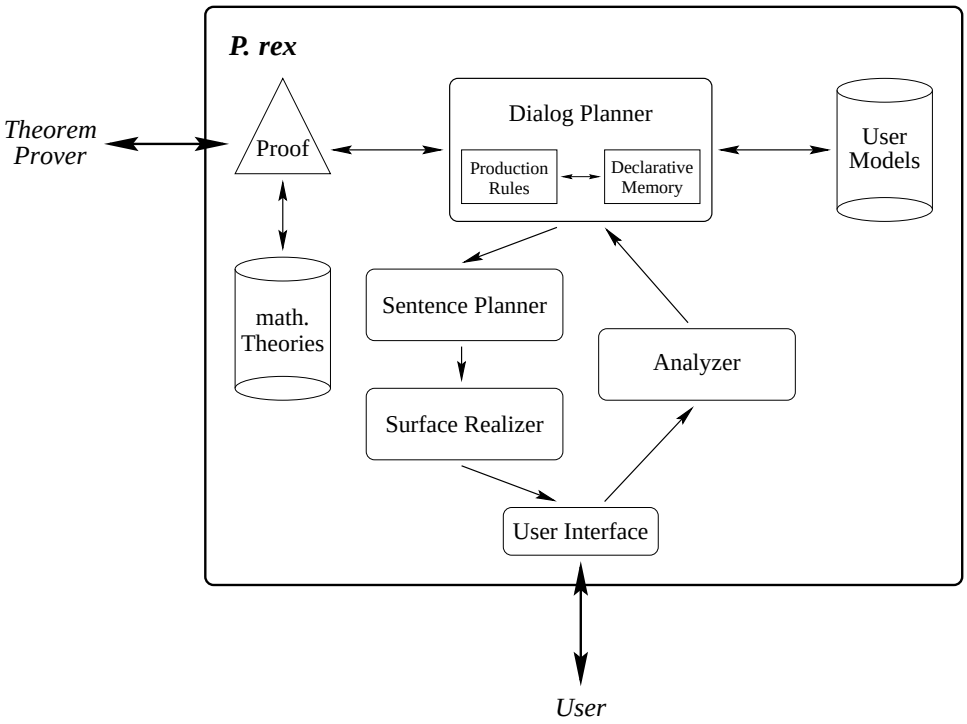


Figure 2. The Architecture of *P.rex*.

Architecture

An overview of the architecture of *P.rex* is displayed in Figure 2.3.

As mentioned previously TWEGA defines the interface between theorem provers and *P.rex*. The calculus of the prover, the input proof to be explained, as well as relevant information from the mathematical theories that relate to the proof are represented in TWEGA for further use by *P.rex*. can be connected to these theorem provers. The type-theoretic foundations of TWEGA guarantee that only those proofs can be represented that are correct with respect to the calculus of the corresponding prover.

The central component of *P.rex* is the *dialog planner*. It chooses the content and determines the order of the information to be conveyed, and organizes the pieces of information in a discourse structure tree. The dialog planner is implemented in ACT-R. The discourse structure tree, which serves both as dialog plan and as dialog history, is stored in the declarative memory. The plan operators of the dialog planner, which organize the discourse structure, are defined in terms of ACT-R productions. The dialog planner models the user by encoding his presumed declarative and procedural knowledge in the declarative memory and the production rule base, respectively.

In order to explain a particular proof, the dialog planner first assumes the user's cognitive state by updating its declarative and procedural memories. This is done by looking up the user's presumed knowledge in the user model, which was recorded during a previous session. An individual model for each user persists between the sessions. It is stored in the database of user models. Each user model contains assumptions about the knowledge of the user that is relevant to the proof explanation. In particular, it makes assumptions about which mathematical theories the user knows and which definitions, proofs, inference methods and mathematical facts

he knows.

After updating the declarative and procedural memories, the dialog planner sets the global goal to show the proof. ACT-R tries to fulfill this goal by successively applying productions that decompose or fulfill goals. Thereby, the dialog planner not only produces a dialog plan, but also traces the user's cognitive states in the course of the explanation. This allows the system both to always choose an explanation adapted to the user, and to react to the user's interactions in a flexible way: The dialog planner interprets the interaction in terms of applications of productions. Then it plans an appropriate response.

The dialog plan produced by the dialog planner is passed on to the *sentence planner*. We adapted and extended *PROVERB*'s micro-planner [Fiedler, 1996; Huang and Fiedler, 1997] to use it in *P.rex* to plan the internal structure of the sentences. The sentence planner aggregates domain concepts when possible and maps them into a linguistic structure. The linguistic structure specifies the lexical items and referring expressions that realize the domain concepts as well as the small-scale segmentation, that is, the scope of the phrases and sentences.

The linguistic structure is then realized by the *surface realizer* TAGGEN [Kilger and Finkler, 1995], which ensures the correct morphology of the surface words.

The organization of dialog planner, sentence planner and surface realizer in a pipeline corresponds to the architecture of most natural language generation systems and was therefore characterized as a consensus architecture [Reiter, 1994].

The utterances that are produced by *P.rex* are presented to the user via a *user interface* that allows the user to enter remarks, requests and questions. An *analyzer* receives the user's interactions, analyzes them and passes them on to the dialog planner. Since natural language understanding is beyond the scope of this work, we use a simplistic analyzer that understands a small set of predefined interactions.

Discussion

The dialog planner of *P.rex* is a *hybrid planner*, which combines planning with discourse relations as captured in productions with planning with schemata. Whereas discourse relations are particularly well suited for planning small-scale structures such as subsequent utterances, schemata are superior in planning large-scale structures such as paragraphs or sections. To represent the dialog the dialog planner constructs a discourse structure tree that combines the RST-like approach of Hovy [1993] with the concept of attentional spaces from Grosz and Sidner [1986]. The attentional spaces are employed to model the salience of the information. Based on the information in which focus space a fact was derived, the dialog planner decides how the fact can be referred to.

Beside adapting the degree of explicitness taking into account the attentional spaces, the dialog planner also adapts to the user by explaining the proof at the highest level of abstraction that it assumes to be known to the user and by omitting steps that it assumes the user can easily infer. Moreover, it combines two different presentation strategies depending on how familiar with the current subject the user is. Whereas a textbook-style presentation strategy is used for those parts of the proof where the user is more knowledgeable, a classroom-style explanation strategy is used for the unfamiliar parts. In particular, the system can switch between the strategies during an ongoing explanation.

The system allows the user to interrupt anytime if he is not satisfied with the current explanation and reacts to the interruption by accordingly replanning the parts of the proof the user complained. The dialog planner uses the structural information that is explicitly represented in the discourse structure trees to identify which parts of the explanation failed to convey successfully. Those parts are then

replanned.

Initially, *P.rex* is equipped with 91 domain independent productions. Further domain dependent productions are added by the definition of schemata tailored to the mathematical theory under consideration. Moreover the system can learn new productions via the production compilation process.

P.rex has been successfully used to explain dozens of proofs from group theory and the theory of first-order predicate logic.

Overview

This thesis is organized as follows: After an introduction in Chapter 1, we shall review relevant research in natural language generation and proof presentation, formulate requirements for proof explanation, and give an overview of the architecture of *P.rex* in Chapter 2. Next, in Chapter 3, we shall formally define TWEGA and give an example for the representation of a calculus that is used by several theorem provers. Some deduction systems such as ΩMEGA [Benzmüller *et al.*, 1997] are based on complex calculi that call for a dynamic representation of the proofs in TWEGA. Since such a dynamic representation cannot be precalculated, we shall show in Chapter 4 how a dynamic representation can be calculated at run time using as an example the calculus of ΩMEGA.

Chapter 5 is devoted to ACT-R, the theory of human cognition that serves as a basis for the dialog planner of *P.rex*. The dialog planner itself is the subject of Chapter 6. In this central chapter, we shall define the discourse structure trees and plan operators to construct them. In particular, we shall show how the system uses special plan operators to adapt its explanations to the user. Moreover, we shall discuss the system's reaction to a user's interactions.

Chapter 7 is then devoted to the front end components, namely the sentence planner and the linguistic realizer, the user interface, and the analyzer. Chapter 8 concludes the thesis.

Acknowledgments

My first thanks go to Jörg Siekmann, who introduced me to the field of artificial intelligence and accepted me as a Ph.D. student in the ΩMEGA group in Saarbrücken. He provided an excellent and highly motivating environment where I could concentrate on my work.

I am grateful to the members of my committee Jörg Siekmann, Werner Tack and Frank Pfenning who gave me helpful comments on my thesis.

The Graduiertenkolleg Kognitionswissenschaft (doctoral program in cognitive science) funded my work from April 1996 to March 1999.

Many fruitful discussions with my colleagues in the ΩMEGA group contributed to the success of this work. I am especially indebted to my office-mate Helmut Horacek who helped improve my knowledge of natural language generation and to our former project leader Michael Kohlhase who initially turned my attention to the idea of proof explanation and always showed great interest in my work. I also thank Christoph Benzmüller, Lassaad Cheikhrouhou, Andreas Franke, Andreas Meier, Erica Melis, Martin Pollet, Volker Sorge, Carsten Ullrich and Jürgen Zimmer for many discussions or technical help.

Frank Pfenning and Ken Koedinger invited me to stay at CMU for one year and provided an excellent working environment, where the core of my work came into being.

I am greatly indebted to Frank Pfenning who not only introduced me to logical frameworks, but also allowed me to use his system Twelf as a model for TWEGA. In many discussions, he helped me to achieve a deeper insight into computational logic. I also thank his then student Carsten Schürmann for many discussions regarding Twelf and its implementation.

In many discussions, Ken Koedinger gave me the psychologist's perspective on my work. Dieter Wallach first acquainted me with the details of ACT-R, and Christian Lebiere later answered all the questions regarding ACT-R I ever had.

Whenever I encountered problems with the English language when writing this thesis, Chris Scarpinatto was the native speaker and educated linguist I could always rely on to help me out.

My stay at CMU was very pleasant not only because of the professional environment, but also because of the personal friends I made. The first ones to mention are Chris Scarpinatto, Carsten Schürmann and Molly Bigelow, and Max Ritter who introduced me to the German crowd at CMU. An important part of my personal life in Pittsburgh was teaching and practicing aikido at Three Rivers Aikikai. Since I cannot list all of my aikido friends here, I representatively thank Garth Jones, Tara Meyer, Alexei Nikolaev, Liam Pedersen, Don Reed, Chris Scarpinatto, Klaus Sutner, and John Uddstrom.

In phases of hard work it is important to have good friends who offer their support and distract one from work for a little while. Therefore, I thank my friends in Saarbrücken, in particular Iris Himbert-Fiedler, Barbara Kessler, Holger Maier, Andreas Meier, Birgit Reeb, Jürgen Reeb, Martin Pollet, and Andrea Schmitt.

Ganz besonders herzlich möchte ich mich bei meiner lieben Frau Iris Himbert-Fiedler bedanken. Sie hat mich in all den Jahren mit aller Kraft unterstützt und geduldig meine Launen ertragen. Des weiteren möchte ich mich bei meinen Eltern dafür bedanken, daß sie mir meine Ausbildung ermöglichten und mich stets unterstützten.

Chapter 1

Introduction

Mathematics has been considered one of the most sublime of the human cognitive skills. Therefore, soon after the advent of the first computers in the early fifties and with the advent of artificial intelligence in 1956 at the Dartmouth Conference, researchers began to devise systems for automated theorem proving (see e.g., [Davis, 1957; Newell *et al.*, 1957; Gelernter, 1959]).

Today, automated theorem provers are becoming more and more important in practical industrial applications. For example, they are standard routine in the verification of hardware and software, in deductive data bases and in logic programming. They also have the first considerable successes in mathematical applications. For example, in 1996, the so-called *Robbins Problem*, an algebraic problem that was open for more than 60 years, could be automatically proved by the automated theorem prover EQP [McCune, 1997b]. Proof planners such as ΩMEGA [Benzmüller *et al.*, 1997] and Clam [Bundy *et al.*, 1990] prove non-trivial mathematical theorems from assorted fields. Deduction systems are becoming useful in mathematical education [Melis *et al.*, 2001] and (to a limited extend) the mathematicians' everyday work.

For these applications, it is important that a deduction system communicates its proofs reasonably well to the human user. Some systems provide as output simply a "yes" or "no," thus precluding the user from scrutinizing the proof. More responsive systems return a formal account of the proof. However, most automated theorem provers are based on machine-oriented calculi such as resolution [Robinson, 1965], which are optimized for the process of proof search and not for human readability. Since the lines of reasoning of the output proofs are often unnatural and obscure, the proofs are very difficult to read and hardly comprehensible, even for specialists. Therefore, researchers developed algorithms [Andrews, 1980; Kursawe, 1982; Miller, 1984; Pfenning, 1987; Lingenfelder, 1990] to transform proofs from machine-oriented calculi into more human-oriented calculi such as the natural deduction (ND) calculus [Gentzen, 1935].

But the result of the transformation into the ND calculus turned out to be far from satisfactory. The reason is that the obtained ND proofs are very large and too involved in comparison to the original proof. Moreover, in the ND calculus, an inference step merely consists of the syntactic manipulation of a quantifier or a connective. In human-written proofs, in contrast, an inference step is often described in terms of the application of a definition, an axiom, a lemma or a theorem, collectively called an assertion. Based on this observation, abstractions of ND proofs have been defined, where a proof step may be justified either by an ND inference rule or by the application of an assertion [Huang, 1994c; Horacek, 1999].

Since traditional automated theorem provers find proofs for theorems that are

usually considered easy by mathematicians, theorem provers based on a more human-oriented approach such as the application of planning techniques have been developed [Bundy *et al.*, 1990; Benzmüller *et al.*, 1997]. The key idea here is that proof techniques as used by mathematicians are encoded into plan operators, which are used by the proof planner to find a proof plan. Because a proof plan is an abstract representation of a proof, it provides a level that is better suited for communication such that proof transformation at the calculus level becomes obsolete for proof planners.

To be of practical use for the human user, however, it does not suffice to communicate the proofs in an albeit human-oriented, but still formal format. Instead, the proofs should be output in the way people are used to, namely in natural language. To this end, proof presentation systems have been developed.

However, most proof presentation systems that have been developed so far in the field of automated theorem proving (e.g., [Chester, 1976; Edgar and Pelletier, 1993; Coscoy *et al.*, 1995; Dahn *et al.*, 1997]) suffer from several deficiencies. Regardless of the level of abstraction, automatically found proofs often include steps that are logically indispensable but people would not make explicit, because the steps are considered obvious. Since most proof presentation systems employ no techniques to exclude such steps from the presentation, they present every single step of the derivation. Thus, the output proofs are too detailed, such that even small proofs are not easy to follow. Moreover, these systems mostly use templates with canned sentence chunks for the verbalization of the steps. Therefore, the output texts lack interrelations between segments of the text. This leaves the text segments unconnected and, thus, severely obstructs the comprehension of the proof. The situation is even worse in those systems that verbalize only the inference rules in natural language, but output the premises and conclusions as logical formulae, which are still unreadable.

To overcome these problems proof presentation systems that employ more sophisticated techniques from the field of natural language generation have been developed (e.g., [Huang and Fiedler, 1997; Holland-Minkley *et al.*, 1999]). These systems decide which information to include in the presentation and they plan the rhetorical structure of the text in advance to achieve connected texts. They also plan the internal structure of the sentences and the morphology of the words. They ensure variation in the verbalization by choosing between several possible alternatives, avoid redundancies by aggregating information and improve the relatedness of the sentences by choosing appropriate referring expressions. These techniques together enable state-of-the-art proof presentation systems to output the proofs in an almost textbook-like format.

However, even the previously mentioned state-of-the-art proof presentation systems suffer from several deficiencies. Firstly, they usually presuppose a single communicative purpose, namely to present the proof in a textbook-like style. However, whereas a mathematician who wants to check the correctness of a proof is often satisfied with a textbook-like presentation, a student who wants to learn how to find a proof usually needs a more elaborate classroom-style explanation of the proof. In such a classroom-style explanation, the reason *why* a step is taken should be emphasized as opposed to just mention *that* the step is taken [Leron, 1983].

Secondly, the systems all neglect the issue of user modeling and thus forgo the ability to adapt the presentation to the specific user. They present the proof at a fixed level of abstraction that might be too high for a novice or too low for an expert without allowing for a transition between levels of abstraction. A similar problem is that they often do not adapt the degree of explicitness to the user's cognitive capabilities. Also, since they do not model the user's cognitive skills, they cannot omit steps that are trivial or easily inferable by the particular user. Horacek [1999] introduced an approach how to model which steps are easily inferable.

Finally, in case the user cannot understand some part of the proof, the possibility to ask questions about it and get an adapted explanation is desirable or even compulsory (for intelligent tutor systems for mathematics and mathematical assistant systems). However, current proof presentation systems do not allow the user to interact with the system. The user cannot tell the system when he did not understand a part of the proof and ask for another explanation of that part. Similarly, the user cannot ask follow-up questions or questions about the background of the proof.

As a first step to overcome these deficiencies, we shall develop in this thesis a computational model of user-adaptive proof explanation, which is implemented in a generic, user-adaptive proof explanation system, called ***P.rex*** (for ***proof explainer***). The demands we make on the system are the following: The system should adapt to the user with respect to the level of abstraction at which the proof is presented, the degree of explicitness used to explain the proof, and inferential capabilities of the user to omit inferable steps. The explanation should account for different presentation styles, such as the textbook-like style and the classroom-like style. Moreover, the system should allow the user to intervene if he is not satisfied with an explanation and to ask follow-up or background questions. The metaphor we have in mind is a human mathematician who teaches a proof to a student or else explains a proof to a colleague.

In our approach, we combine techniques from three different fields, namely cognitive science, computational logic and natural language processing, as we shall discuss in the following.

Modern natural language generation systems take into account the intended audience's knowledge in the generation of explanations (see e.g. [Cawsey, 1990; Paris, 1991a; Wahlster *et al.*, 1993]). Most of them adapt to the addressee by choosing between different discourse strategies. Since proofs are inherently rich in inferences, the explanation of proofs must also consider which inferences the audience can make [Zukerman and McConachy, 1993; Horacek, 1997b; 1999]. However, because of the constraints of the human memory, inferences are not chainable without costs. Explicit representation of the addressee's cognitive states proves to be useful in choosing the information to convey [Walker and Rambow, 1994].

In cognitive science various theories of human cognition have been described by means of a cognitive architecture, that is, the fixed structure that realizes the cognitive apparatus (e.g., [Newell, 1990; Meyer and Kieras, 1997a; Anderson and Lebiere, 1998]). One of these theories is ACT-R [Anderson and Lebiere, 1998], a cognitive architecture that separates declarative and procedural knowledge into a declarative memory and a production rule base, respectively. A goal stack allows ACT-R to fulfill a task by dividing it into subtasks, which can be fulfilled independently.

In ACT-R, procedural knowledge is represented in production rules, whose conditions and actions are defined in terms of declarative structures. A production rule can only apply if its conditions are satisfied by the knowledge currently available in the declarative memory. An item in the declarative memory is annotated with an activation that influences its retrieval. The application of a production rule modifies the declarative memory, or it results in an observable event. If several production rules are applicable a conflict resolution heuristic derived from a rational analysis of human cognition [Anderson, 1990] determines which production rule will eventually be applied.

Hence, ACT-R combines the abilities for user modeling and planning in a uniform framework and is therefore particularly well suited as a basis for a user-adaptive dialog planner. Using ACT-R, we model a teacher who explains mathematical proofs to his student. In particular, we model the teacher's assumptions about the students cognitive states during the explanation (which establish the user model) and the teacher's knowledge of the mathematical theories and the way to explain

the proof of a theorem in these theories (which is used to plan the explanation). The assumptions about the user's cognitive states are employed to choose the level of abstraction at which a proof is presented and to decide which steps the user can infer.

An important prerequisite that enables *P.rex* to choose among different levels of abstraction is the simultaneous representation of a proof at several levels of abstraction. This representation, which also serves as the interface between theorem provers and *P.rex*, is realized in TWEGA, an implementation of an extension of the logical framework LF [Harper *et al.*, 1993] that corresponds to the calculus of constructions (λC) [Coquand and Huet, 1988]. LF and λC are very powerful typed lambda calculi that allow us to represent other logical calculi, and thus give us the possibility to represent the calculus of any theorem prover that is to be connected to *P.rex*. The input proof to be explained as well as relevant information from the mathematical theories that relate to the proof are encoded in TWEGA and then used by *P.rex*. In particular, using an expansion mechanism that defines for any derivation step a subproof at the lower level of abstraction, TWEGA allows us to represent a proof at several levels of abstraction simultaneously.

Successful communication via a natural language generation system presupposes that the content to be conveyed is appropriately structured to ensure a coherent semantic organization. For an explanation system, it is also important to accept user feedback and follow-up questions. To be able to clarify misunderstood explanations, the system needs to represent the different parts of the explanation as well as the relations between them.

An appropriate discourse theory that allows for these representations was formulated by Mann and Thompson [1987]. This theory, called *Rhetorical Structure Theory (RST)*, states that the relations that hold between segments of normal English text can be represented by a finite set of relations. Based on RST, Hovy [1993] described discourse as the nesting of discourse segments. According to his discourse theory, each segment essentially contains the communicative goal the speaker wants to fulfill with this segment and either one to several discourse segments with inter-segment discourse relations or the semantic material to be communicated.

Grosz and Sidner [1986] developed a discourse theory that distinguishes three separate, but interrelated components, namely the linguistic structure, the intentional structure and the attentional state. Whereas the linguistic structure describes the segmentation of the discourse, the intentional structure captures how the purposes of the discourse segments relate to one another. The attentional state is an abstraction of the focus of attention of the participants as the discourse unfolds and models the salience of discourse contents.

For *P.rex*, we define discourse structure trees as a representation of discourses. This representation adapts Hovy's discourse segments and combines it with the concept of attentional spaces from Grosz and Sidner's theory. This allows us to model both the segmentation of the discourse and the focus of attention in a uniform framework. Since the attentional spaces model salience, they are employed to decide on the degree of explicitness and to choose appropriate referring expressions. Explicit representation of the discourse purpose allows for presentations using different styles, such as the textbook-like style or the classroom-like style. Moreover, discourse structure trees also account for restricted types of dialogs as well, namely certain types of interruptions and clarification dialogs. This is a necessary prerequisite for the system to represent user interactions and to appropriately react to them.

P.rex has been successfully used for the explanation of dozens of proofs from various domains such as group theory and the theory of first-order predicate logic.

This thesis is organized as follows: First, in Chapter 2, we shall review relevant research in natural language generation and proof presentation, formulate require-

ments for proof explanation, and give an overview of the architecture of *P.rex*. Next, in Chapter 3, we shall formally define TWEGA and give an example for the representation of a calculus that is used by several theorem provers. Some deduction systems such as ΩMEGA [Benzmüller *et al.*, 1997] are based on complex calculi that call for a dynamic representation of the proofs in TWEGA. Since such a dynamic representation cannot be precalculated, we shall show in Chapter 4 how a dynamic representation can be calculated at run time using as an example the calculus of ΩMEGA.

Chapter 5 is devoted to ACT-R, the theory of human cognition that serves as a basis for the dialog planner of *P.rex*. The dialog planner itself is the subject of Chapter 6. In this central chapter, we shall define the discourse structure trees and plan operators to construct them. In particular, we shall show how the system uses special plan operators to adapt its explanations to the user. Moreover, we shall discuss the system's reaction to a user's interactions.

Chapter 7 is then devoted to the front end components, namely the sentence planner and the linguistic realizer, the user interface, and the analyzer. Whereas the sentence planner plans the internal structure of a sentence, the linguistic realizer plans the appearance of the words and produces the surface utterance. The user interface finally presents the surface sentence to the user and accepts his interaction. The interactions are interpreted by the analyzer and transformed into new discourse goals for the dialog planner to fulfill. Chapter 8 concludes the thesis.

Chapter 2

Natural Language Generation and Proof Presentation

Soon after the advent of the first computers in the early fifties, researchers began to build systems for automated theorem proving. To be of practical use for a mathematician, however, it is mandatory that these systems do not only communicate their machine-found proofs to the user, but also communicate them in the way a mathematician is used to, namely in natural language at the appropriate level of abstraction.

In Section 2.1 we shall review the generation of natural language with emphasis on the generation of explanations. Then, Section 2.2 will give an overview of proof presentation systems and formulate requirements for a proof explanation system. Section 2.3, finally, is devoted to the architecture of *P.rex*.

2.1 Natural Language Generation

The field of *natural language generation (NLG)* is concerned with the production of utterances in natural language as produced by human speakers or writers. Many NLG systems are deployed as the user interface for an underlying knowledge-based system. To fulfill its task the generator must decide on choices involving content and organization of the information to be conveyed *(what to say)* and choices involving the surface form of the produced utterances *(how to say it)*. Therefore, the processing used to be separated into two stages, called *content determination* (or *deep generation*) and *surface realization* (or *surface generation*).

Most early work focused on surface generation (e.g., [Chester, 1976; McDonald, 1984; Joshi, 1985; Matthiesen and Bateman, 1990]). Later, more and more researchers began to concentrate on deep generation (e.g., [McKeown, 1985; Hovy, 1988; Paris, 1988; Moore, 1989]). However, a major shortcoming of the two-staged process soon became apparent: the planning of the internal structure of the sentences was widely ignored. This observation led to the introduction of a mediating process, the *sentence planning* (cf. e.g., [Meteer, 1991; Hovy, 1992]).

In the following section we shall review a tentative reference architecture for NLG systems [Cahill *et al.*, 1999]. Then, in Section 2.1.2, we shall concentrate on the special requirements for the generation of explanations.

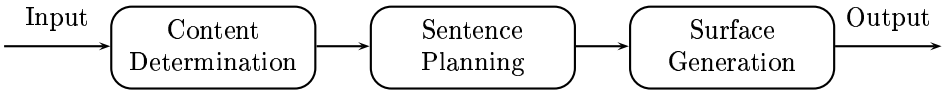


Figure 2.1. The consensus architecture for NLG systems according to Reiter [1994].

2.1.1 The Quest for a Reference Architecture

To generate natural language we have to determine which information is to be conveyed. The content must be appropriately structured to ensure a coherent semantic organization of the utterances. The utterances, in turn, must be internally structured to ensure a correct syntactic organization of the sentences.

Based on an analysis of five systems, Reiter [1994] claimed that there is a consensus about the overall architecture for NLG systems and argued that this consensus architecture is, at least partially, psycholinguistically plausible. As depicted in Figure 2.1 the consensus architecture consists of three components, which are arranged in a *pipeline*, that is, each component receives information only from its predecessor and sends information only to its successor. The functions of the components are the following:

Content Determination The *content determination* component takes the initial input to the generation system and produces from it a *semantic representation*, that is, a specification of the meaning content of the output text in terms of domain entities. It determines what information should be communicated and organizes this information in a rhetorically coherent manner. This component is often also called *macro-planner*, *text planner* or, in the case of dialog systems, *dialog planner*.

Sentence Planning The *sentence planner* converts the semantic representation into an abstract *linguistic specification* that specifies content words and grammatical relationships. It maps domain concepts and relations into content words and grammatical relations. Moreover, it generates referring expressions for individual domain entities and groups propositions into clauses and sentences. In the literature, the sentence planner is also called *micro-planner*.

Surface Generation The *surface generation* component finally realizes the abstract linguistic specification by expressing the content words and grammatical relations in the target language and, thus, produces the actual text. This component is also often referred to as *surface realization* or *linguistic realization* component.

Motivated by Reiter's observation of a consensus architecture, Cahill and colleagues [1999] examined eighteen further system and concluded that the consensus architecture is broadly supported. However, as they pointed out, the architecture is really defined by its data interfaces rather than by its processing components: The information is transformed from input through semantic representation and linguistic specification to text. To grasp the linguistic operations that are to be performed, Cahill and colleagues described a *reference architecture for generation systems (RAGS)* in terms of a functional model without imposing a processing strategy. Without aiming to be exhaustive the functional model consists of the following seven modules [Cahill *et al.*, 1999]:

Lexicalization *Lexicalization* is the mapping from a concept to a lexical item, where we do not differentiate between lexical alternatives. The decision of

which lexical alternative is chosen to appear in the produced text is called *lexical choice*.

Aggregation *Aggregation* is any process that merges two or more pieces of information that are separate at some other level.

Rhetorical Structuring *Rhetorical structuring* means the determination of rhetorical relations between pieces of information, that is, it determines how the pieces of information are related by the text structure.

Referring Expressions Choice of *referring expressions* means to decide how to refer to concepts or entities.

Ordering *Ordering* means the linearization of the pieces of information in the output text.

Segmentation *Segmentation* is the grouping of the pieces of information into clauses, sentences or large-scale structures such as paragraphs.

Coherence *Coherence* means the senseful progression from one piece of information to the next. It includes phenomena such as centering, salience and theme processing. Such processing is actually relatively rare in the examined systems.

Trying to assign the functional modules to the components of Reiter's consensus architecture, Cahill and colleagues found that the majority of the examined systems perform rhetorical structuring, segmentation and ordering in the content determination component. Whereas both coherence and lexicalization are approximately equally often located in the content determination component and the sentence planner, aggregation and choice of referring expressions are usually ascribed to the sentence planner. Lexical choice, finally, is mostly a task of the surface generator.

Note that the functional modules need not be arranged in a pipeline architecture. Therefore, Cahill and colleagues [1999] proposed to design an NLG system in two stages. First, a *data model* defines the functional modules entirely in terms of the data types they manipulate and the operations they can perform. Then, a more specific *process model*—for example, Reiter's consensus architecture—constrains the order and level of instantiation of different data types in the data model.

2.1.2 The Generation of Explanations

Beside the functional modules identified in the reference architecture, further functionalities are required for an explanation system.

When we want to generate explanations, it is important to tailor the explanations to the current audience. To do so, we need both a model of the user, that is, detailed knowledge about the user's knowledge and skills in the domain under consideration, and flexible discourse strategies that take into account the user's expertise [Paris, 1988; 1991b]. The ability of a system to adapt its behavior to the different users is called *user adaptivity*.

Since feedback from the user himself can be an important source for user modeling, Moore and Swartout suggested a reactive approach to explanation [Moore and Swartout, 1991]. In their approach, the system is devised as a dialog system that monitors the effects of its utterances on the user and accepts feedback from him. If the feedback indicates an unsatisfying explanation the system recovers by providing an alternative explanation. In particular, the system actively seeks feedback from the user to determine if he is following. Moreover, the system answers follow-up questions taking into account previous explanations.

To be able to respond to follow-up questions in context, the system must interpret questions in context. To be able to clarify misunderstood explanations the system itself must understand the explanation it produced, that is, it must represent the different parts of the explanation as well as the relations between parts. Moreover, the system must have several response strategies for each type of question. Otherwise, the system cannot offer an alternative response even if it understands why a previous explanation was not satisfactory. The capability of a system to plan or interpret utterances taking into account the discourse as it has developed so far is called *context sensitivity*.

Whereas in Moore and Swartout's approach, the user can give his feedback only after completion of an explanation, Mooney and colleagues emphasized that in long explanations the user must be able to interrupt the system at any time [Mooney *et al.*, 1991]. They advocate an incremental planner that determines some high-level structure of the explanations, but defers the detailed organization until the time of presentation.

To integrate these functionalities with the reference architecture, we add the following functional modules:

User Modeling *User modeling* means to represent the assumptions the system makes about the user's knowledge and skills with respect to the domain under consideration. The system may or may not validate and revise its assumptions based on the user's interactions.

Discourse History Recording The *discourse history* is recorded to allow the system to access previous discourse plans if needed.

In contrast to a pure generation system, an interactive system must also interpret input given by the user. To introduce this functionality, we add one further module:

Analysis *Analysis* means here that the system accepts interactions from the user and extracts new discourse goals from the interactions if needed.

User adaptivity and context sensitivity can then be achieved by enriching the rhetorical structuring component with different structuring strategies that take into account a user model and a discourse history, respectively.

When designing the process model, we can devise a hierarchical discourse planner, which decomposes the global discourse goal into increasingly detailed subgoals until atomic plan steps are reached. Then, in each planning cycle, we can allow the user to interrupt the system and to convey his utterances to the system. Note that it might be useful to restrict the set of possible interactions dependent on the discourse situation. The decomposition of the discourse goals can be done in a depth-first manner. As soon as an atomic step is reached, it is passed on for subsequent processing that eventually results in its realization. Thus, we achieve an incremental generation of the system's utterances as recommended in [Mooney *et al.*, 1991].

Before we turn our attention to the architecture of our proof explanation system *P.rex*, we shall give a brief overview of the field of proof presentation in the following section.

2.2 Proof Presentation

While the field of automated theorem proving matured in the last four decades, it became more and more apparent that the systems had to output proofs that can be more easily understood by mathematicians. To provide the proofs in the internal, machine-oriented formalisms of the theorem provers is by far not sufficient. Thus,

proof presentation systems had to be designed that presented proofs in natural language at an appropriate level of abstraction.

The problem of obtaining a natural language proof from a machine-found proof can be divided into two subproblems: First, the machine-found proof is transformed into a human-oriented calculus, which is much better suited for presentation. Second, the transformed proof is verbalized in natural language. In the following, we shall examine both stages in some more detail.

Already in the thirties, Gentzen devised a human-oriented calculus that aimed to reflect the way mathematicians reason, the *natural deduction (ND) calculus* [Gentzen, 1935]. However, most automated theorem provers are based on machine-oriented formalisms, such as resolution or matings. As a consequence, proofs encoded in such formalisms are not suited for a direct verbalization, because their lines of reasoning are often unnatural and obscure. To remedy this problem, researchers developed algorithms to transform proofs in machine-oriented calculi into ND proofs [Andrews, 1980; Kursawe, 1982; Miller, 1984; Pfenning, 1987; Lingenfelder, 1990].

Initially, the idea was that the transformation into an ND proof is sufficient for the subsequent verbalization. But the results of the transformations into the ND calculus turned out to be far from satisfactory. The reason is that the obtained ND proofs are very large and too involved in comparison to the original proof. Moreover, in the ND calculus, an inference step merely consists of the syntactic manipulation of a quantifier or a connective. Huang realized that in human-written proofs, in contrast, an inference step is often described in terms of the application of a definition, axiom, lemma or theorem, which he collectively calls *assertions*. Based on this observation, he defined an abstraction of ND proofs, called the *assertion level*, where a proof step may be justified either by an ND inference rule or by the application of an assertion, and gave an algorithm to abstract an ND proof to an assertion level proof [Huang, 1994c]. Based on Huang's ideas, Meier described an algorithm to transform refutation graphs (a data structure that represents resolution proofs) directly into assertion level proofs [Meier, 1997; 2000]. The assertion level proves to be much better suited for a subsequent verbalization of the proofs than a traditional calculus. However, the assertion level sometimes includes steps that include implicit applications of modus tollens, which prove to be difficult to comprehend for human beings. Therefore, [Horacek, 1999] introduced the *partial* assertion level, where these applications are made explicit.

Since traditional search-based automated theorem provers find only proofs for theorems that are usually considered easy by mathematicians, theorem provers based on human-oriented approaches such as planning have been developed [Bundy *et al.*, 1990; Benzmüller *et al.*, 1997]. The key idea here is that proof techniques as used by mathematicians are encoded into plan operators, which are used by the proof planner to find a proof plan. Because a proof plan is an abstract representation of a proof, it provides a level that is better suited for presentation, such that proof transformation becomes obsolete for proof planners.

Now, let us turn our attention to approaches to verbalize proofs. One of the earliest proof presentation systems was EXPOUND [Chester, 1976]. It translated the formal proofs directly into English. Even though sophisticated techniques were developed to plan the paragraphs and sentences that made up the written proof, the system verbalized every single step of the formal proof in a template driven way, such that even small proofs are still not easy to follow.

Proofs were also used as test input for early versions of MUMBLE [McDonald, 1984], an NLG system that adopted more advanced generation techniques. However, its main concern was not proof presentation, but to show the feasibility of its two-staged architecture for the generation of natural language.

Whereas the previously mentioned systems focused on problems in natural language generation and used formal proofs only as well-defined input for the gener-

ation process, research in the field of automated theorem proving addressed the readability of proofs as well. The following systems were developed in the field of automated theorem proving with the aim to obtain human-readable proofs.

The χ-proof system [Felty and Miller, 1988] was one of the first theorem provers that was designed with a natural language output component. The system showed every step of the derivation by using predefined templates with English words, but left the formulae in their logical form. The same is true for the pseudo-natural language presentation components of Coq [Coscoy *et al.*, 1995] and the proof system *Theorema* [Buchberger, 1997]. The latter allows the user in addition to hide or unhide proof parts that he considers too detailed or not detailed enough, respectively.

Natural Language Explainer [Edgar and Pelletier, 1993] was devised as a back end for the natural deduction theorem prover THINKER. Employing several isolated strategies, it was the first system to acknowledge the need for higher levels of abstraction when explaining proofs.

Another presentation system that uses templates with canned sentence chunks to verbalize proofs is ILF [Dahn *et al.*, 1997]. It has been connected to several automated theorem provers, whose output proofs are presented in natural language at the logic level of the machine-oriented formalism of the respective prover.

None of these systems employs modules for rhetorical structuring, sentence planning or linguistic realization. The following systems, in contrast, take into account approaches that have been developed in the field of natural language generation.

PROVERB [Huang, 1994a; Huang and Fiedler, 1997] can be seen as the first serious attempt to build a generic, comprehensive system that produces adequate argumentative texts from machine-found proofs. It takes as input ND proofs, which are abstracted to the assertion level before any subsequent processing starts. Adopting Reiter's consensus architecture (cf. Section 2.1.1), *PROVERB* consists of a macro-planner, which chooses the information to be conveyed, a micro-planner, which plans the internal structure of the sentences, and a linguistic realizer, which produces the output text. These three components are designed such that they cover the functionalities of the modules of the reference architecture (cf. Section 2.1.1).[1] The system uses presentation knowledge and linguistic knowledge to plan the proof texts, which are output in a textbook-like format.

Another recently developed NLG system that is used as a back end for a theorem prover is the presentation component of Nuprl [Holland-Minkley *et al.*, 1999]. The system consists of a pipeline of two components. It employs a content planner that selects the information to be included in the output text and decides how to refer to the information in the given context. The text plan is then passed on to the surface realizer FUF [Elhadad and Robin, 1992], which chooses the words and outputs the actual sentences.

To sum up, to remedy the problem that many theorem provers return proofs in their internal, machine-oriented formalisms, which are very hard to understand, more and more human-oriented interfaces for theorem provers have been developed. But these interfaces are mostly used in the theorem proving community, that is, by people who usually have an insight in the provers' formalisms. The systems present proofs mostly at a very low level of abstraction and none of the systems adapts its output to the user or can handle follow-up questions. Whereas this might be tolerable for the developers of the theorem provers, it is certainly not acceptable for mathematicians or mathematics students who want to work with theorem provers.

Before we shall formulate requirements for a proof explanation system, we shall take a closer look at *PROVERB*, the most sophisticated proof presentation system to date.

[1] Actually, *PROVERB* is one of the systems analyzed in [Cahill *et al.*, 1999].

2.2.1 *PROVERB*

The proof presentation system *PROVERB* [Huang, 1994a; Huang and Fiedler, 1997] was developed at Saarland University in Saarbrücken. Following Reiter's consensus architecture (cf. Section 2.1.1), it employs a pipeline of a macro-planner, a micro-planner and a surface realizer. We shall give a brief overview of each of these components.

Macro-Planner The macro-planner [Huang, 1994a] takes as input a natural deduction proof, where each proof step is justified by a natural deduction inference rule (cf. Section 3.1). First, such a proof is lifted to the so-called *assertion level* [Huang, 1994c], where proof steps are no longer just an application of an inference rule, but may be justified in terms of the application of a definition, axiom, lemma or theorem. The assertion level is hence a higher level of abstraction, which is much better suited for presentation than the original logic-level proof.

Next, the assertion-level proof is traversed using plan operators to choose the information that should be conveyed to the user and to organize this information into a linear order. While the traversal of the proof, the macro-planner performs rhetorical structuring, ordering, and large-scale segmentation into paragraphs.

The result of the macro-planner is a sequence of speech acts organized in attentional spaces. This sequence of speech acts constitutes the text plan, which is the input for the micro-planner. The macro-planner will be discussed in more detail in Section 6.1.5.

Micro-Planner The micro-planner [Fiedler, 1996; Huang and Fiedler, 1997] transforms the text plan into a linguistic specification of the output text. It performs the context sensitive choice of referring expressions, aggregation, lexicalization and lexical choice, as well as small-scale segmentation into sentences.

The output of the micro-planner is a tree structure, which is passed on to the linguistic realizer. We shall describe the micro-planner of *PROVERB* in more detail in Section 7.1.

Linguistic Realizer *PROVERB* uses the surface generator TAG-GEN [Kilger and Finkler, 1995] as its linguistic realizer. TAG-GEN, which was developed at the German Research Center for Artificial Intelligence (DFKI) in Saarbrücken, is responsible for the correct syntax and morphology of the produced sentences.

The output text produced by *PROVERB* is finally processed by the document preparation system LaTeX. An example of a proof verbalized by *PROVERB* is given in Figure 2.2.

PROVERB was designed with the aim of producing natural-language proofs similar to proofs found in mathematical textbooks. Hence, it presents machine-found proofs at the assertion level in a textbook-style format. Yet, the assertion level is often still at a too low level of abstraction to be acceptable by expert mathematicians. *PROVERB* uses presentation knowledge and linguistic knowledge to plan the overall text and the sentences.

Since *PROVERB* was not devised as an explanation system, it lacks any facilities for user modeling, user adaptation and user interaction. Also, the internal representation of the text plan is not powerful enough to allow for plan revision in case some proof steps do not communicate successfully.

However, the mentioned facilities are certainly necessary if theorem provers are to be accepted as a tool by mathematicians or mathematics students, who are mostly

Theorem:
Let F be a group, let U be a subgroup of F, and let 1 and 1_U be unit elements of F and U. Then 1_U equals 1.
Proof:
Let F be a group, let U be a subgroup of F, and let 1 and 1_U be unit elements of F and U.
Because 1_U is an unit element of U, $1_U \in U$. Therefore, there is x such that $x \in U$.
Let u_1 be such an x. Since $u_1 \in U$ and 1_U is an unit element of U, $u_1 * 1_U = u_1$. Since F is a group, F is a semigroup. Since U is a subgroup of F, $U \subset F$. Because $U \subset F$ and $1_U \in U$, $1_U \in F$. Similarly, because $u_1 \in U$ and $U \subset F$, $u_1 \in F$. Then, 1_U is a solution of $u_1 * x = u_1$.
Because $u_1 \in F$ and 1 is an unit element of F, $u_1 * 1 = u_1$. Since 1 is an unit element of F, $1 \in F$. Then, 1 is a solution of $u_1 * x = u_1$.
Therefore, 1_U equals 1. This conclusion is independent of the choice of u_1. ■

Figure 2.2. An example proof produced by *PROVERB* taken from [Huang and Fiedler, 1997].

not familiar with the formalisms theorem provers are based on. In the following section, we shall discuss the requirements for a system that explains proofs.

2.2.2 Requirements for the Explanation of Proofs

When designing a proof explanation system, we should study a mathematics teacher and examine how he explains proofs to his students.

In education, human teachers use *natural language* for the presentation and explanation. Mathematics teachers often experience that students are demotivated by an overload of formulae. Hence, it can be expected that a proof explanation system, in particular if used by novices, is much more accepted if it also communicates derivations and, to some extend, formulae in natural language.

Many concepts and ideas are much easier to understand when they are depicted graphically; the inclusion of graphs and diagrams in addition to natural language is standard routine in mathematics communication. The computer allows us to go beyond these traditional ways of presentation and to include parameterized animations as well, which, for example, display how a diagram changes when parameters are varied. That is, a *multi-modal* user interface would be desirable.

The major advantage of a teacher in comparison to a textbook is that the students can interact with the teacher during the lesson. For example, they can ask the teacher when they do not understand a derivation. These forms of *interaction* should be supported by an intelligent explanation system as well.

To communicate a proof, the teacher has to present individual proof steps choosing a degree of *explicitness*. Usually, he does not mention all proof steps explicitly by giving the premises, the conclusion and the inference method. Often, he only hints implicitly at some proof steps (e.g., by giving only the inference method) when the hint is assumed to suffice for the student to reconstruct the proof step. Other proof steps are completely omitted when they are obvious or easy to infer.

But a presentation that consists of a mere enumeration of proof steps is often unnatural and tedious to follow. Therefore, teachers add many *explanatory comments*, which motivate a step or explain the structure of a subproof, for example, by stressing that a case analysis follows.

Both decisions whether a proof step is only hinted at or omitted, and whether an explanatory comment is given are usually made *relative to the context*. For example, if a premise A of a proof step with conclusion B is used immediately after it was derived, the teacher only hints at it by saying: "Therefore, B holds." But if the premise A was derived a while ago he explicitly mentions it by saying: "Since A, we obtain B." A similar argument holds for explanatory comments. For example, if it is obvious that there is a case analysis it is not explicitly introduced, but the cases are presented right away.

The level of *abstraction* at which the proof is presented plays a major role. For example, the author of a textbook has an idea of his intended audience and adapts his presentation to that audience. Likewise, a teacher takes into account the abilities of his students when he decides at which level of abstraction he presents a derivation. To choose between different levels of abstraction, an intelligent proof explanation system needs a *user model*, which records the facts and inference methods the user knows.

Finally, a proof explanation system should also account for different *presentation strategies* with respect to the purpose of the session. For example, different strategies are needed when mere mathematical facts are to be conveyed in contrast to when mathematical skills are to be taught. In the former case, a proof can be presented in *textbook style* where the essential proof steps are shown. In the latter case, the presentation should be structured differently to convey also control knowledge, which explains *why* the various proof steps are taken as opposed to just show *that* they are taken [Leron, 1983]. We say, the proof is presented in *classroom style*. These two strategies reflect the difference in the difficulty of checking the correctness of a proof versus finding a proof.

P.rex is the first attempt to build an interactive, user-adaptive proof explanation system. Except for multi-modality, we shall address all the requirements formulated in this section. Although graphs and diagrams play an important role when included in a presentation, they occur only in certain mathematical theories and even in these theories, they do not occur very often. Therefore, we decided for the moment to neglect the generation of graphs and diagrams in our system. However, there is active research (e.g., at DFKI as well as Saarland University in Saarbrücken [Wahlster *et al.*, 1993; Horacek, 1997a]) on the combination of natural language with graphics. In particular, [Wahlster *et al.*, 1993] points out that a three-staged architecture analogous to Reiter's consensus architecture is also appropriate for multi-modal generation. We will incorporate these results later into *P.rex*.

We shall now describe the architecture of *P.rex* in the following section.

2.3 The Architecture of *P.rex*

P.rex is a generic proof explanation system that can be connected to different theorem provers. Except for multi-modality, it fulfills all requirements given in Section 2.2.2. It includes the functional modules of the reference architecture for NLG systems as reviewed in Section 2.1.1 as well as the additional functionalities for explanation systems described in Section 2.1.2. In this section, we shall give an overview of the architecture of *P.rex*, which is displayed in Figure 2.3.

As the interface between theorem provers and *P.rex*, we define the formal language Twega, the implementation of an extension of the logical framework LF [Harper *et al.*, 1993] that corresponds to the calculus of constructions (λC) [Coquand and Huet, 1988]. LF and λC are very powerful calculi from type theory that allow us to represent other logical calculi and, thus, to represent the proofs of most (if not all) currently implemented theorem provers. Hence, Twega ensures that *P.rex* can be connected to these theorem provers. The type-theoretic foundations of

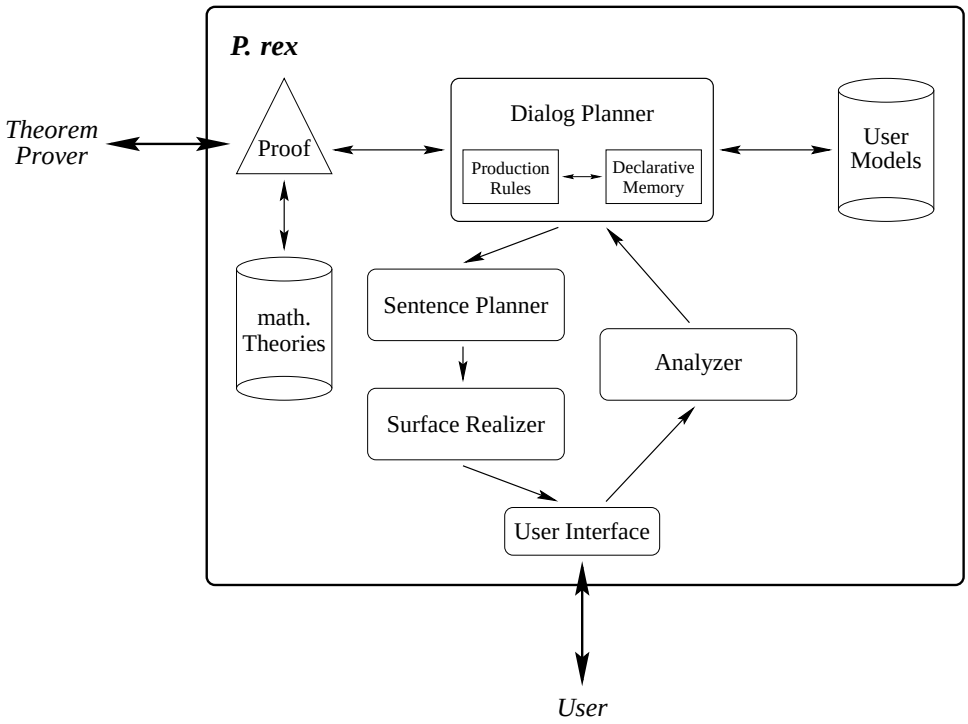


Figure 2.3. The Architecture of *P.rex*.

TWEGA guarantee that only those proofs can be represented that are correct with respect to the calculus of the corresponding prover. The calculus of the prover, the input proof to be explained, as well as relevant information from the mathematical theories that relate to the proof are represented in TWEGA for further use by *P.rex*. We shall define TWEGA formally in Chapter 3. The connection of the theorem prover ΩMEGA [Benzmüller *et al.*, 1997] to *P.rex* will be described in detail in Chapter 4.

The central component of *P.rex* is the *dialog planner*. It chooses the content and determines the order of the information to be conveyed, and organizes the pieces of information in a rhetorical structure, called *discourse structure*. The discourse structure also specifies the large-scale segmentation of the discourse into paragraphs.

The dialog planner is implemented in ACT-R, a production system that aims to model the human cognitive apparatus [Anderson and Lebiere, 1998]. ACT-R separates declarative and procedural knowledge into a declarative memory and a production rule base, respectively. A goal stack allows ACT-R to fulfill a task by dividing it into subtasks, which can be fulfilled independently. We shall review ACT-R briefly in Chapter 5.

The plan operators of the dialog planner, which organize the discourse structure, are defined in terms of ACT-R productions. A discourse history is represented in the declarative memory by storing conveyed information. Moreover, presumed declarative and procedural knowledge of the user is encoded in the declarative memory and the production rule base, respectively. This establishes that the dialog planner is modeling the user.

In order to explain a particular proof, the dialog planner first assumes the user's cognitive state by updating its declarative and procedural memories. This is done by looking up the user's presumed knowledge in the user model, which was recorded during a previous session. An individual model for each user persists between the

sessions. It is stored in the database of *user models*. Each user model contains assumptions about the knowledge of the user that is relevant to the proof explanation. In particular, it makes assumptions about which mathematical theories the user knows and which definitions, proofs, inference methods and mathematical facts he knows.

After updating the declarative and procedural memories, the dialog planner sets the global goal to show the proof. ACT-R tries to fulfill this goal by successively applying productions that decompose or fulfill goals. Thereby, the dialog planner not only produces a dialog plan, but also traces the user's cognitive states in the course of the explanation. This allows the system both to always choose an explanation adapted to the user, and to react to the user's interactions in a flexible way: The dialog planner interprets the interaction in terms of applications of productions. Then it plans an appropriate response. Chapter 6 is devoted to the comprehensive description of the dialog planner.

The dialog plan produced by the dialog planner is passed on to the *sentence planner*. We adapted and extended *PROVERB*'s micro-planner [Fiedler, 1996; Huang and Fiedler, 1997] to use it in *P.rex* to plan the internal structure of the sentences. The sentence planner aggregates domain concepts when possible and maps them into a linguistic structure. The linguistic structure specifies the lexical items and referring expressions that realize the domain concepts as well as the small-scale segmentation, that is, the scope of the phrases and sentences.

The linguistic structure is then realized by the syntactic generator TAG-GEN [Kilger and Finkler, 1995], which ensures the correct morphology of the surface words. Note that dialog planner, sentence planner and surface realizer are organized in a pipeline as in Reiter's consensus architecture (cf. Section 2.1.1).

The utterances that are produced by *P.rex* are presented to the user via a *user interface* that allows the user to enter remarks, requests and questions. An *analyzer* receives the user's interactions, analyzes them and passes them on to the dialog planner. Since natural language understanding is beyond the scope of this work, we use a simplistic analyzer that understands a small set of predefined interactions. We shall discuss the sentence planner, the surface realizer, the user interface and the analyzer in Chapter 7.

Chapter 3

The Representation of Proofs

In order to explain a proof, a system needs an internal representation of the theorem under consideration as well as its proof, the calculus in which the proof was derived, and the mathematical theory behind the theorem (i.e., definitions that are used in the theorem or in the proof, and more basic theorems and lemmata that are applied in the proof). Moreover, it should be guaranteed that a proof is correct in the sense that it constitutes a valid derivation of the theorem by a given deductive system. Hence, we have to define a language to formulate mathematical facts and provably correct derivations. To this end, let us examine how mathematics is usually formalized.

A *logic* consists of a *language* to formulate expressions and of a *semantics* assigning meaning to expressions, in particular to a specific subcategory of expressions, called *formulae*. *Propositions* are formulae of which we want to establish their *truth* given a set of formulae, called *assumptions*. The means to establish the truth of a proposition is a *deductive system*, which is defined by a set of *axioms*, which are true formulae, and a set of *inference rules*, which yield true formulae when given true formulae. A logic and an appropriate deductive system constitute a *calculus*.

Martin-Löf provides another perspective on calculi by introducing the notion of a *judgment* (such as "φ is true") as something we may know by virtue of a *proof* [Martin-Löf, 1985; 1996]. For him the notions of judgment and proof are more basic than the notions of proposition and truth, because the meaning of a proposition is explained via the inference rules used to establish its truth. A judgment is *derivable* if and only if it can be established by a deduction using the given axioms and inference rules. The set of derivable judgments can be seen as the least set that contains the axioms and that is closed under the inference rules. Thus, the axioms and inference rules provide a semantic definition for a language, the so-called *proof theoretic semantics*.

In the field of automated theorem proving various logics and calculi are used in the implementation of a deduction system. Examples for such logics are *first-order predicate logic*, *sorted logics* or *higher-order logics*. In practical systems, inference rules that are based on such logics are then used to build calculi, for example, variants of the *resolution*, *tableaux*, and *natural deduction* calculus. Since we aim at a generic system, it would be inappropriate to restrict *P.rex* to one of these calculi alone. Instead, we employ a *meta-language* in which we can formulate in a uniform way the various calculi of candidate theorem provers.

A well investigated theory suited as a meta-language for the representation of different calculi is the *typed lambda calculus* [Church, 1940]. There are several versions of typed lambda calculi, which differ in their expressiveness. Examples are the simply typed lambda calculus ($\lambda\!\rightarrow$), the polymorphic typed lambda calculus ($\lambda 2$), the LF logical framework (sometimes called λP) [Harper *et al.*, 1993], and the

calculus of constructions (λC, sometimes denoted by λPω or CC) [Coquand and Huet, 1988].

The syntax of a typed lambda calculus is given by the set of *terms* that can be expressed. The set of terms is restricted to the set of *valid* terms by a *type system*, which is a deductive system that defines the *types* that can be assigned to terms. The previously mentioned examples of typed lambda calculi differ in their respective type systems. For a calculus to be of practical value the problems of *type checking ("Is a given type the type of a given term?")* and *typability ("What is the type of a given term?")* must be decidable.

It has been observed by several logicians (e.g., [Howard, 1980; de Bruijn, 1970]) that propositions in intuitionistic logic can be interpreted as types in a typed lambda calculus where the proof of the truth of such a proposition corresponds to a term of the respective type. In the sense of this so-called *propositions-as-types* interpretation, intuitionistic propositional logic corresponds to $\lambda\!\!\rightarrow$, intuitionistic first-order predicate logic corresponds to λP, and intuitionistic higher-order logic corresponds to λC. In the propositional case this correspondence is an isomorphism, called *Curry-Howard isomorphism*[1] (for an in-depth presentation see, e.g., [Thompson, 1991; Barendregt, 1992; Geuvers, 1993]).

By adopting the proposition-as-types principle we would restrict our system to an intuitionistic logic, which does not allow for the representation of some proofs. For example, a proof of a theorem that is formulated in classical logic cannot be represented in intuitionistic logic. Hence, still assuming Martin-Löf's point of view, we employ an alternative, albeit related, representation technique, often called *judgments-as-types* [Harper *et al.*, 1993]. This technique is characterized by mapping judgments to types and their proofs to terms, thus reducing the problem of *proof checking ("Is a given derivation a proof for a given judgment?")* to the problem of type checking.

First, we shall present in Section 3.1 the natural deduction calculus as an example for a calculus various automated theorem provers are based on. Next, we shall review in Section 3.2 the notion of a *pure type system* (PTS) as a uniform way to describe many typed lambda calculi and state some of their properties. Moreover, we shall give eight typed lambda calculi as examples of PTSs, which together constitute a fine structure of the calculus of constructions λC. Then, in Section 3.3, we shall define TWEGA, an implementation of λC, which we use in *P.rex* as the formalism mathematics is represented in. In Section 3.4 finally, we shall elaborate on the judgment-as-types paradigm by showing how an example calculus, namely the natural deduction calculus, can be represented in TWEGA.

3.1 The Natural Deduction Calculus

In this section, we shall present the *natural deduction (ND) calculus*, which was introduced in [Gentzen, 1935] as a calculus whose inference rules mirror the actual practice of a mathematician as opposed to, for example, the Hilbert calculus or the resolution calculus. Therefore, it is particularly well suited for the explanation of proofs. The ND calculus is a calculus for first-order predicate logic. We essentially present its definition as given in [Pfenning, in prep.].

[1] The notions *propositions-as-types* and *Curry-Howard isomorphism* are sometimes used synonymously in the literature independently of the logic under consideration, although for example the correspondence between intuitionistic higher-order logic and the calculus of constructions λC is not an isomorphism: There are propositions that are not provable in the logic, but become provable when mapped into λC [Geuvers, 1993].

3.1.1 Syntax

The ND calculus is defined on the language of terms and formulae:

$$\begin{array}{lll} \textit{Terms} & \mathbf{T} & ::= \mathbf{V} \mid \mathbf{P}_f \mid \mathbf{C}_f(\mathbf{T},\ldots,\mathbf{T}) \\ \textit{Formulae} & \mathbf{W} & ::= \mathbf{C}_p(\mathbf{T},\ldots,\mathbf{T}) \mid \mathbf{W} \wedge \mathbf{W} \mid \mathbf{W} \vee \mathbf{W} \mid \mathbf{W} \supset \mathbf{W} \mid \neg \mathbf{W} \\ & & \quad \mid \forall \mathbf{V}.\mathbf{W} \mid \exists \mathbf{V}.\mathbf{W} \end{array}$$

where $\mathbf{V}$, $\mathbf{P}_f$, $\mathbf{C}_f$, and $\mathbf{C}_p$ are infinite collections of variables, function parameters, function symbols, and predicate symbols, respectively. Each function symbol and each predicate symbol has a unique arity. $\top, \bot \in \mathbf{C}_p$ are 0-ary predicate symbols. A (function or predicate) constant c is simply a (function or predicate) symbol with no argument, and we write c instead of $c()$.

$A \wedge B$ is the *conjunction* of A and B; $A \vee B$ is the *disjunction* of A and B. $A \supset B$ denoting *implication* means that A implies B. $\neg A$ is the *negation* of A. The *universal quantification* $\forall x.A$ means that A holds for all x, whereas the *existential quantification* $\exists x.A$ means that there is an x such that A holds. Formulae of the form $\forall x.A$ and $\exists x.A$ bind the variable x. $\top, \bot \in \mathbf{C}_p$ denote the predicate that is always true or false, respectively.

3.1.2 Deductive System

Let $\mathbf{P}_p$ denote an infinite collection of propositional parameters. The set of *valid derivations*, denoted by $\mathbf{D}$, is defined on formulae and propositional parameters by a deductive system. The main *judgment* of the ND calculus is the derivability of a formula C, written as $\vdash_{ND} C$, from *assumptions* (or *hypotheses*) $\vdash_{ND} A_1, \ldots, \vdash_{ND} A_n$. We just write $\vdash$ instead of $\vdash_{ND}$.

Notation: For a derivation $\mathcal{D} \in \mathbf{D}$ of a judgment $\vdash C$, we write

$$\mathcal{D} :: \vdash C \quad \text{or} \quad \begin{array}{c} \mathcal{D} \\ \vdash C \end{array}$$

In the ND calculus, each logical connective and quantifier is characterized by its *introduction* and *elimination* rules. The set of natural deduction rules $\mathbf{R}$ that constitutes the deductive system is defined in Table 3.1. An assumption $\vdash H$ is *discharged* by an inference rule if the conclusion no longer depends on $\vdash H$. A discharged assumption is denoted by $[\vdash H]$. We give discharged assumptions a label u as in $[\vdash H]^u$ and annotate the discharging rule R as R^u. $\mathbf{P}_d$ is the infinite collection of labels for discharged hypotheses. $\alpha[\beta/X]$ denotes the result of the substitution of β for all free occurrences of the variable X in α, renaming variables as necessary to avoid name clashes. Let us examine how the rules are justified:

Conjunction $\vdash A \wedge B$ is derivable if both $\vdash A$ and $\vdash B$ are derivable and vice versa. This accounts for the rules $\wedge I$, $\wedge E_l$, and $\wedge E_r$.

Disjunction $\vdash A \vee B$ is derivable if $\vdash A$ or $\vdash B$ is derivable, hence we have two introduction rules $\vee I_l$ and $\vee I_r$. Note that the inverse of this argument does not hold: If $\vdash A \vee B$ is derivable, we do not know whether $\vdash A$ or $\vdash B$ is derivable. But we can derive $\vdash C$ from $\vdash A \vee B$ if $\vdash C$ is derivable from both $\vdash A$ and $\vdash B$. This principle is called *case analysis* in mathematics. It gives rise to the rule $\vee E$.

Implication If we assume $\vdash A$ and if we can derive $\vdash B$ from $\vdash A$ we can derive $\vdash A \supset B$. This argument justifies the rule $\supset I$. If we have a derivation of $\vdash A \supset B$ and a derivation of $\vdash A$ we can obtain a derivation of $\vdash B$. The corresponding rule $\supset E$ has been extensively examined in philosophical logic under the name of *modus ponens*.

Table 3.1. The Inference Rules of the Natural Deduction Calculus

Conjunction	$\dfrac{\vdash A \quad \vdash B}{\vdash A \wedge B}\wedge I$ $\qquad$ $\dfrac{\vdash A \wedge B}{\vdash A}\wedge E_l$ $\quad$ $\dfrac{\vdash A \wedge B}{\vdash A}\wedge E_r$
Disjunction	$\dfrac{\vdash A}{\vdash A \vee B}\vee I_l$ $\quad$ $\dfrac{\vdash B}{\vdash A \vee B}\vee I_r$ $\qquad$ $\dfrac{\vdash A \vee B \quad \begin{array}{c}[\vdash A]^u \\ \vdots \\ \vdash C\end{array} \quad \begin{array}{c}[\vdash B]^v \\ \vdots \\ \vdash C\end{array}}{\vdash C}\vee E^{uv}$
Implication	$\dfrac{\begin{array}{c}[\vdash A]^u \\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B}\supset I^u$ $\qquad$ $\dfrac{\vdash A \supset B \quad \vdash A}{\vdash B}\supset E$
Negation	$\dfrac{\begin{array}{c}[\vdash A]^u \\ \vdots \\ \vdash p\end{array}}{\vdash \neg A}\neg I^u(p)$ $\qquad$ $\dfrac{\vdash \neg A \quad \vdash A}{\vdash C}\neg E$ where p is a new parameter
Truth	$\dfrac{}{\vdash \top}\top I$
Falsehood	$\dfrac{\vdash \bot}{\vdash C}\bot E$
Universal Quantification	$\dfrac{\vdash A[a/x]}{\vdash \forall x.A}\forall I(a)$ $\qquad$ $\dfrac{\vdash \forall x.A}{\vdash A[t/x]}\forall E$ where a is a new parameter
Existential Quantification	$\dfrac{\vdash A[t/x]}{\vdash \exists x.A}\exists I$ $\qquad$ $\dfrac{\vdash \exists x.A \quad \begin{array}{c}[\vdash A[a/x]]^u \\ \vdots \\ \vdash C\end{array}}{\vdash C}\exists E^u(a)$ where a is a new parameter

Negation If we can derive everything from an assumption, then there must be some inconsistency. So, if we can derive any judgment $\vdash p$ from an assumption $\vdash A$, we can conclude $\vdash \neg A$. Thus, we have the rule $\neg I$. Note that this rule is parametric in $p \in \mathbf{P}_p$. In particular, p must not occur in any undischarged hypothesis. Similarly, if we have a derivation of $\vdash A$ and a derivation of $\vdash \neg A$, then there is an inconsistency and we can derive any judgment $\vdash C$. Hence, the rule $\neg E$ is justified.

Truth We can always derive $\vdash \top$. Note that there is no elimination rule for $\top$.

Falsehood Since falsehood should not be derivable, there is no introduction rule for $\bot$. But if we can derive $\vdash \bot$ then there must be some inconsistency and we can derive everything. The corresponding rule $\bot E$ is therefore often called *ex falso quodlibet*.

Universal Quantification If we have $\vdash \forall x.A$ we can conclude $\vdash A[t/x]$ for every term t as formulated in rule $\forall E$. Similarly, if we can derive $\vdash A[a/x]$ for any new parameter $a \in \mathbf{P}_f$, which must not occur in any undischarged assumption, then we can also derive $\vdash \forall x.A$. This accounts for the rule $\forall I$.

Existential Quantification If we can derive $\vdash A[t/x]$ we know that $\vdash \exists x.A$, hence we have the rule $\exists I$. But if we have $\vdash \exists x.A$ we do not know for which term t $A[t/x]$ holds. We can only assume $\vdash A[a/x]$ for some new parameter a, which must not occur in any undischarged hypothesis. If we can derive $\vdash C$ from that assumption we can also derive $\vdash C$ from $\vdash \exists x.A$. This argument justifies the rule $\exists E$.

Example 3.1
The following derivation of the theorem $\vdash A \wedge B \supset B \wedge A$ is an example for a valid derivation in the ND calculus:

$$\dfrac{\dfrac{\dfrac{[\vdash A \wedge B]^u}{\vdash B} \wedge E_r \quad \dfrac{[\vdash A \wedge B]^u}{\vdash A} \wedge E_l}{\vdash B \wedge A} \wedge I}{\vdash A \wedge B \supset B \wedge A} \supset I^u$$

The two identical hypotheses are labeled by u each. $\supset I^u$ denotes that both hypotheses are discharged simultaneously by the rule $\supset I$. □

3.2 Pure Type Systems

Many variants of the typed lambda calculus can be described by the notion of a *pure type system* (PTS) [Barendregt, 1992]. The notation for PTSs is well suited to pinpoint subtle differences between the variants. Moreover, many important properties can be shown for PTSs in general or for restricted classes of PTSs. [Barendregt, 1992] gives an excellent introduction to PTSs that are defined on β-conversion. [Geuvers, 1993] extends the definition of PTSs to allow for η-conversion as well.

3.2.1 The Definition of Pure Type Systems

At first, we give the syntax of the language of typed lambda calculi, before we proceed to the definition of PTSs. Our definition follows mostly [Barendregt, 1992] and [Geuvers, 1993].

Definition 3.1 The set of all *terms* $\mathcal{T}$ is defined by the following abstract syntax

$$\mathcal{T} ::= \mathcal{V} \mid \mathcal{C} \mid \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}$$

where $\mathcal{V}$ and $\mathcal{C}$ are infinite collections of variables and constants, respectively. ■

We call a term of the form $\lambda x{:}A.B$ a *λ-abstraction* and a term of the form $\Pi x{:}A.B$ a *Π-abstraction* or *product*.

Notation: Usually, we write $A \to B$ instead of $\Pi x{:}A.B$, if x does not occur free in B. Moreover, $A[B/x]$ is the result of the substitution of B for all free occurrences of x in A, renaming variables as necessary to avoid name clashes.

To formulate judgments, we need the following definitions:

Definition 3.2 A *statement* is a pair $A : B$ with $A, B \in \mathcal{T}$. A is the *subject* and B is the *predicate* of $A : B$. A *declaration* is a pair $x{:}A$ with $x \in \mathcal{V}$ and $A \in \mathcal{T}$. ■

Note that a declaration is a statement for variables.

Definition 3.3 A *context* Γ is a finite ordered sequence of declarations, all with distinct variables:

$$Ctx ::= \cdot \mid Ctx, \mathcal{V}{:}\mathcal{T}$$ ■

Notation: We write $x{:}A \in \Gamma$ if $x{:}A$ occurs in Γ. We write $x \in \mathrm{dom}(\Gamma)$ if there is a declaration $x{:}A \in \Gamma$.

Having fixed the syntax of the language, we define some equality relations between terms via the notion of a *reduction*.

Definition 3.4 On $\mathcal{T}$, we define the following relations:

1. *β-reduction* is defined by the following contraction rule:
$$(\lambda x{:}A.B)C \to_\beta B[C/x]$$
2. *η-reduction* is defined by the following contraction rule:
$$\lambda x{:}A.Bx \to_\eta B \quad \text{if } x \text{ not free in } B$$
3. *$\beta\eta$-reduction* is defined as $\to_{\beta\eta} = \to_\beta \cup \to_\eta$
4. $\to_\beta^*$ ($\to_\eta^*$, $\to_{\beta\eta}^*$) is the reflexive, transitive closure of $\to_\beta$ ($\to_\eta$, $\to_{\beta\eta}$)
5. $=_\beta$ ($=_\eta$, $=_{\beta\eta}$), called *β- (η-, $\beta\eta$-)conversion*, is the symmetric, transitive closure of $\to_\beta^*$ ($\to_\eta^*$, $\to_{\beta\eta}^*$) ■

We read $A \to_\beta B$ as "A β-reduces to B in one step," $A \to_\beta^* B$ as "A β-reduces to B in many steps," and $A =_\beta B$ as "A is β-convertible to B." Similarly, we say "A η-reduces to B" or "A is $\beta\eta$-convertible to B" and so forth.

Definition 3.5 Let R be a reduction relation.

1. A term A is in R-normal form (short *R-nf*) if and only if there is no term A' such that $A \to_R A'$.
2. R is called *normalizing* if and only if every term A reduces to an R-normal form.
3. A term A is *strongly normalizing* for R if and only if all R-reduction sequences starting with A terminate.
4. R is called *strongly normalizing* if and only if every term A is strongly normalizing for R. ■

Now that we have the syntax for our language and the conversion between terms, we proceed to the definition of the type system for our language.

Definition 3.6 The *specification* of a pure type system is a triple $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

1. $\mathcal{S}$ is a subset of $\mathcal{C}$, called the *sorts*
2. $\mathcal{A}$ is a set of *axioms* of the form $s : s'$ with and $s, s' \in \mathcal{S}$
3. $\mathcal{R}$ is a set of *rules* of the form (s_1, s_2, s_3) with $s_1, s_2, s_3 \in \mathcal{S}$ ■

Notation: We write (s_1, s_2) for the rule (s_1, s_2, s_2).

Definition 3.7 The *pure type system* (PTS) λS determined by the specification $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ is defined by the notion of the *type derivation* $\vdash_{\lambda S}$ (we just write $\vdash$) that in turn is defined by the following axioms and rules:

(axioms)	$\dfrac{}{\vdash s : s'}$	if $(s : s' \in \mathcal{A})$
(start)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x{:}A}$	if $x \notin \mathrm{dom}(\Gamma)$
(weakening)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x{:}C \vdash A : B}$	if $x \notin \mathrm{dom}(\Gamma)$
(product)	$\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3}$	if $(s_1, s_2, s_3) \in \mathcal{R}$
(application)	$\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$	
(abstraction)	$\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$	
(conversion)	$\dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B \equiv B'}{\Gamma \vdash A : B'}$	

where $x \in \mathcal{V}$ and $s \in \mathcal{S}$. ■

Note that in the conversion rule, we use the notion of *definitional equality* ($\equiv$), which is not further specified. In [Barendregt, 1992] the conversion rule is only defined for β-conversion (i.e., $\equiv$ is defined as $=_\beta$). [Geuvers, 1993] calls these PTSs *pure type systems with β-conversion* (PTS_β) and also defines *pure type systems with $\beta\eta$-conversion* ($\mathrm{PTS}_{\beta\eta}$). We follow the second approach.

Definition 3.8 A *pure type systems with β-conversion* (PTS_β) is a PTS where $\equiv$ is defined as $=_\beta$. ■

Definition 3.9 A *pure type systems with $\beta\eta$-conversion* ($\mathrm{PTS}_{\beta\eta}$) is a PTS where $\equiv$ is defined as $=_{\beta\eta}$. ■

The type system restricts the set of terms to the set of *valid* terms.[2]

Definition 3.10 Let λS be a PTS. A term A is *valid in* λS if and only if there are a context Γ and a term B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. ■

Definition 3.11 Let Γ be a context and A a term.

1. A is called a *type* if and only if there is a sort $s \in \mathcal{S}$ such that $\Gamma \vdash A : s$.
2. A is called an *object* if and only if there is a type B such that $\Gamma \vdash A : B$. ■

[2]Some authors (e.g., [Barendregt, 1992; Geuvers, 1993; Ghani, 1997]) use the notion 'term' only for valid terms and call terms 'pseudo-terms' or 'pre-terms.'

Hence, PTSs stratify the terms in three levels: the level of objects, the level of types and the level of sorts. In the literature, the notion 'term' is often used although 'object' is meant, in particular when put into relation to 'type.' We adopt this usage if no confusion can arise.

Definition 3.12 Let A be a term.

1. A is *typable* if and only if there are a context Γ and a term B, such that $\Gamma \vdash A : B$.
2. A is *inhabited* if and only if there are a context Γ and a term M, such that $\Gamma \vdash M : A$. ■

Definition 3.13 Let Γ be a context and A be a term.

1. The problem whether there is a term B such that $\Gamma \vdash A : B$ is derivable is called *typability*.
2. Let B be a term. The problem whether $\Gamma \vdash A : B$ is derivable is called *type checking*. ■

A normal form important for PTS$_{\beta\eta}$s is based on β-reduction and η-expansion (i.e., the inverse of η-reduction).

Definition 3.14 Let $\Gamma \vdash_\Sigma U : V$, where U is in β-normal form. Then, the *long $\beta\eta$-normal form* (short *$\beta\eta$-lnf*) of U is defined as follows:

1. Let $U = \lambda x{:}A.M$. Then the $\beta\eta$-lnf of U in Γ is the term $\lambda x{:}A'.M'$, where A' is the $\beta\eta$-lnf of A in Γ and M' is the $\beta\eta$-lnf of M in $\Gamma, x{:}A$.
2. Let $U = \Pi x{:}A.B$. Then the $\beta\eta$-lnf of U in Γ is the term $\Pi x{:}A'.B'$, where A' is the $\beta\eta$-lnf of A in Γ and B' is the $\beta\eta$-lnf of B in $\Gamma, x{:}A$.
3. Let $U = cM_1 \ldots M_n$ and $\Pi x_1{:}P_1.\ldots.\Pi x_m{:}P_m.P$ (where P is not an abstraction) be the $\beta\eta$-lnf of V. Then the $\beta\eta$-lnf of U is the term $\lambda x_1{:}P_1'.\ldots.\lambda x_m{:}P_m'.cM_1' \ldots M_n' x_1' \ldots x_m'$, where P_i' is the $\beta\eta$-lnf of P_i in $\Gamma, x_1{:}P_1, \ldots, x_{i-1}{:}P_{i-1}$; M_i' is the $\beta\eta$-lnf of M_i in Γ; and x_i' is the $\beta\eta$-lnf of x_i in $\Gamma, x_1{:}P_1, \ldots, x_i{:}P_i$. ■

Notation: We write $A = \mathcal{NF}(A')$ if A is the $\beta\eta$-lnf of A'. Furthermore, $\mathcal{NF}(\mathcal{T}) = \{\mathcal{NF}(A) | A \in \mathcal{T}\}$.

Finally, let us define some restricted classes of PTSs.

Definition 3.15 A PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *functional* if and only if the following hold:

1. If $s_1 : s_2 \in \mathcal{A}$ and $s_1 : s_2' \in \mathcal{A}$ then $s_2 = s_2'$.
2. If $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1, s_2, s_3') \in \mathcal{R}$ then $s_3 = s_3'$. ■

Note that by this definition, $\mathcal{A}$ and $\mathcal{R}$ are partial functions in a functional PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$.

Definition 3.16 A PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *injective* if and only if it is functional and the following hold:

1. If $s_1 : s_2 \in \mathcal{A}$ and $s_1' : s_2 \in \mathcal{A}$ then $s_1 = s_1'$.
2. If $(s_1, s_2, s_3) \in \mathcal{R}$ and $(s_1', s_2', s_3) \in \mathcal{R}$ then $s_1 = s_1'$ and $s_2 = s_2'$. ■

Note that by this definition, $\mathcal{A}$ and $\mathcal{R}$ are injective functions in an injective PTS $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$.

Definition 3.17 A PTS $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ is *full* if and only if for all $s_1, s_2 \in \mathcal{S}$ there is an $s_3 \in \mathcal{S}$ such that $(s_1, s_2, s_3) \in \mathcal{R}$. ■

Definition 3.18 Let R be β- or $\beta\eta$-conversion. A PTS_R λS is *strongly normalizing* if and only if every term that is valid in λS is strongly normalizing for R. ■

3.2.2 Examples of Pure Type Systems

[Barendregt, 1992] introduces a collection of eight closely related lambda calculi, the so-called λ-*cube*. The λ-cube forms a natural fine structure of the *calculus of constructions* [Coquand and Huet, 1988]. It is organized according to the different possible dependencies between objects and types, as will be elaborated later in this section. Whereas [Barendregt, 1992] describes the λ-cube by PTS_βs, we extend the definition to $\mathrm{PTS}_{\beta\eta}$s.

Definition 3.19 The *cube of typed lambda calculi*, short λ-cube, consists of eight $\mathrm{PTS}_{\beta\eta}$s with sorts $\mathcal{S} = \{\star, \Box\}$, axioms $\mathcal{A} = \{\star : \Box\}$, and rules $\mathcal{R}$ as follows:

$\lambda\!\to$:	$(\star,\star)$			
$\lambda 2$:	$(\star,\star)$,	$(\Box,\star)$		
$\lambda\mathrm{P}$:	$(\star,\star)$,		$(\star,\Box)$	
$\lambda\underline{\omega}$:	$(\star,\star)$,			$(\Box,\Box)$
$\lambda\omega$:	$(\star,\star)$,	$(\Box,\star)$,		$(\Box,\Box)$
$\lambda\mathrm{P2}$:	$(\star,\star)$,	$(\Box,\star)$,	$(\star,\Box)$	
$\lambda\mathrm{P}\underline{\omega}$:	$(\star,\star)$,		$(\star,\Box)$,	$(\Box,\Box)$
$\lambda\mathrm{P}\omega$:	$(\star,\star)$,	$(\Box,\star)$,	$(\star,\Box)$,	$(\Box,\Box)$

■

Note that all eight systems in the λ-cube are functional and injective. Moreover, the last system is full. The table in the definition of the λ-cube makes explicit the inclusion relation $\subseteq$ between the respective systems: $\lambda(\mathcal{S},\mathcal{A},\mathcal{R}_1) \subseteq \lambda(\mathcal{S},\mathcal{A},\mathcal{R}_2)$ if and only if $\mathcal{R}_1 \subseteq \mathcal{R}_2$. The notion '$\lambda$-cube' suggests another way to present the relations between the systems, namely graphically as in Figure 3.1 where the arrows denote the inclusion relation $\subseteq$. The orientation of the arrows conveys the following information: vertical arrows indicate the addition of rule $(\Box,\star)$; arrows to the right mean the addition of rule $(\star,\Box)$; and arrows pointing into the depth denote the addition of rule $(\Box,\Box)$.

The system $\lambda\!\to$ is the simply typed lambda calculus (for an in-depth presentation see, e.g., [Barendregt, 1992]). $\lambda 2$ is the *polymorphic* or *second-order* typed

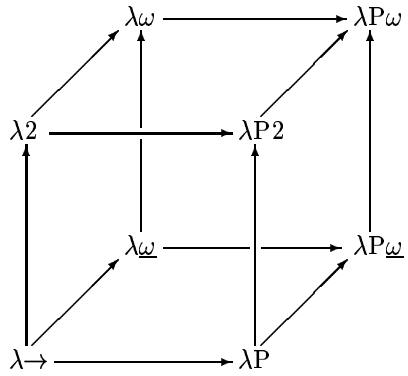


Figure 3.1. The λ-cube [Barendregt, 1992].

lambda calculus. It is essentially the system F of [Girard, 1972]. The system $\lambda\omega$ is the higher-order version of $\lambda 2$ and is essentially the system $F\omega$ of [Girard, 1972]. The system λP was introduced under the name logical framework (LF) by [Harper *et al.*, 1993]. λP2 is studied under the same name in [Longo and Moggi, 1988]. Finally, the largest system, λPω, which is often called λC or CC, is a version of the calculus of constructions [Coquand and Huet, 1988].

The λ-cube gives a fine structure of λC by exposing the various forms of dependencies in the subsystems. Let us call expressions of the category $\Box$ kinds (e.g., $K : \Box$ establishes that K is a kind), expressions of the category $\star$ types (e.g., $A : \star$ means that A is a type), and expressions of some type objects (e.g., a is an object in $a : A$, where $A : \star$). In the systems of the λ-cube, objects and types are mutually dependent and there are four different dependencies: (1) objects depending on objects, (2) objects depending on types, (3) types depending on objects, and (4) types depending on types. Each rule in $\mathcal{R}$ introduces one of these dependencies by allowing for the construction of objects or types featuring the respective dependency. Let us examine the different dependencies more closely.

(1) *Objects depending on objects:* the rule $(\star, \star)$ allows for the construction of types $\Pi x{:}A_1.A_2$, where A_1, A_2 are types. Hence the judgment $A{:}\star \vdash A \to A : \star$ can be derived. (Recall that $A_1 \to A_2$ is a shorthand for $\Pi x{:}A_1.A_2$).

In the judgment

$$A{:}\star, s{:}A \to A, z{:}A \vdash sz : A$$

the object sz depends on the object z. Consider for example the case where $A = \mathsf{nat}$ denotes the natural numbers, s denotes the successor function and z denotes 0. Then, the judgment states that sz is of type nat.

(2) *Objects depending on types:* the rule $(\Box, \star)$ is indispensable to form types $\Pi x{:}K.A$, where K is a kind and A is a type, like the type $\Pi\alpha{:}\star.\alpha \to \alpha$.

In the judgment

$$I{:}\Pi\alpha{:}\star.\alpha \to \alpha, A{:}\star \vdash IA : A \to A$$

the object IA depends on the type A. Consider for example the case where $A = \mathsf{nat}$ and $I = \lambda\alpha{:}\star.\lambda x{:}\alpha.x$, the latter denoting the polymorphic identity function. Then $IA = \lambda x{:}A.x$ denotes the identity function for natural numbers.

(3) *Types depending on objects:* the rule $(\star, \Box)$ is needed to construct kinds $\Pi x{:}A.K$, where A is a type and K is a kind. Thus the judgment $A{:}\star \vdash A \to \star : \Box$ is derivable.

In the judgment

$$A{:}\star, V{:}A \to \star, a{:}A \vdash Va : \star$$

the type Va depends on the object a. Let $A = \mathsf{nat}$ and V the collection of types of vectors indexed by their length. Then $V5$ denotes the type of vectors of length 5.

(4) *Types depending on types:* the rule $(\Box, \Box)$ enables us to build kinds $\Pi x{:}K_1.K_2$, where K_1, K_2 are kinds, as for example the kind $\star \to \star$.

A simple example of a type depending on another type is the type $A \to A$ that depends on the type A. In

$$A{:}\star, F{:}\star \to \star \vdash FA : \star$$

the type FA depends on the type A. Let $A = \mathsf{nat}$ and $F = \lambda\alpha{:}\star.\alpha \to \alpha$ the collection of self-mapping functions, then $FA = \mathsf{nat} \to \mathsf{nat}$ is the type of all functions from the natural numbers to the natural numbers. Note that $\vdash \lambda a{:}\star.a \to a : \star \to \star$.

Collections of types that depend on objects or types (e.g., V and F) are called *type families*.

Many important properties of the systems of the λ-cube can be shown by proving them for PTSs in general.

3.2.3 Properties of Pure Type Systems

Without giving detailed proofs, we state a selection of properties of PTSs, which we shall use later on. Unless otherwise mentioned, [Barendregt, 1992] gives the proofs for PTS$_\beta$s and [Geuvers, 1993] gives the proofs for arbitrary PTSs. We start with properties that hold for all PTSs.

Lemma 3.1 (Free Variable Lemma) *Let $\Gamma = x_1:A_1, \ldots, x_n:A_n$ be a context, and let M, B be terms such that $\Gamma \vdash M : B$. Furthermore, let $FV(U)$ denote the set of free variables in a term U. Then the following hold:*

(i) *The variables $x_1, \ldots, x_n$ all are distinct.*

(ii) *$FV(M), FV(B) \subseteq \{x_1, \ldots, x_n\}$.*

(iii) *$FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $1 \leq i \leq n$.*

Proof: By induction on the derivation of $\Gamma \vdash M : B$. ∎

Lemma 3.2 (Substitution Lemma) *Let $\Gamma, x:A, \Gamma' \vdash M : B$, and let $\Gamma \vdash N : A$. Then*

$$\Gamma, \Gamma'[N/x] \vdash M[N/x] : B[N/x]$$

Proof: By induction on the derivation of $\Gamma, x:A, \Gamma' \vdash M : B$. ∎

Lemma 3.3 (Stripping Lemma) *Let Γ be a context and M, N, R be terms. Then the following hold:*

(i) *If $\Gamma \vdash s : R$ where $s \in \mathcal{S}$, then $R \equiv s'$ with $s : s' \in \mathcal{A}$ for some $s' \in \mathcal{S}$.*

(ii) *If $\Gamma \vdash x:R$ where $x \in \mathcal{V}$, then $R \equiv A$ with $x:A \in \Gamma$ for some $A \in \mathcal{T}$.*

(iii) *If $\Gamma \vdash \Pi x:A.B : R$, then $\Gamma \vdash A : s_1$, $\Gamma, x:A \vdash B : s_2$, and $R \equiv s_3$ with $(s_1, s_2, s_3) \in \mathcal{R}$ for some $s_1, s_2, s_3 \in \mathcal{S}$.*

(iv) *If $\Gamma \vdash \lambda x:A.M : R$, then $\Gamma, x:A \vdash M : B$, $\Gamma \vdash \Pi x:A.B : s$, and $R \equiv \Pi x:A.B$ for some $B \in \mathcal{T}$ and $s \in \mathcal{S}$.*

(v) *If $\Gamma \vdash MN : R$, then $\Gamma \vdash M : \Pi x:A.B$, and $\Gamma \vdash N : A$ with $R \equiv B[N/x]$ for some $A, B \in \mathcal{T}$.*

Proof: By induction on the derivation of $\Gamma \vdash Q : R$, where Q is the subject of the statement in the premise of the respective implication. ∎

Corollary 3.1 *Let A be a valid term. Then the following hold:*

(i) *A is an object, a type, or a sort.*

(ii) *Every subterm of A is a valid term.*

The remaining theorems that we shall present here are restricted to certain PTSs. Note that the following theorem holds for functional PTSs and hence for all systems in the λ-cube.

Theorem 3.1 (Uniqueness of Types) *Let λS be a functional PTS, Γ a context and M, A, A' terms. If $\Gamma \vdash M : A$ and $\Gamma \vdash M : A'$, then $A \equiv A'$.*

Proof: By induction on the structure of M, using the Stripping Lemma 3.3. ■

Theorem 3.2 (Decidability of Type Checking and Typability)

(i) Type checking and typability are decidable for normalizing PTS_βs.

(ii) Type checking and typability are decidable for λP and λC.

Proof:

(i) Decidability of type checking and typability is proved for normalizing PTS_βs in [van Benthem Jutting, 1993]. Since it can be shown that all systems of the λ-cube defined on PTS_βs are strongly normalizing, type checking and typability are decidable for them.

(ii) Decidability of type checking and typability is proved for λP in [Coquand, 1991]. [Ghani, 1997] shows the same result for λC by proving the computability of the long $\beta\eta$-normal form and showing that $A =_{\beta\eta} A'$ if and only if $\mathcal{NF}(A) = \mathcal{NF}(A')$. This proof draws on the proof of the existence of the long $\beta\eta$-normal form in λC [Dowek *et al.*, 1993].[3]

3.3 The System TWEGA

In the previous section, we first introduced PTSs as a uniform way to describe typed lambda calculi. In terms of PTSs, we then presented the λ-cube as a fine structure of the calculus of constructions λC. Finally, we presented several important properties of λP and λC, for example the uniqueness of types (Theorem 3.1) and the decidability of type checking (Theorem 3.2). Hence, both λP and λC are powerful calculi appropriate for our purposes, namely to represent mathematical facts and provably correct derivations. Most importantly, both calculi allow us to represent diverse logics such as first-order and higher-order logics, modal logics, or temporal logics. The representation will be according to the judgment-as-types paradigm, thus proof checking is reduced to type checking, which is decidable, as we now know.

In this section, we describe TWEGA, an implementation of λC extended by two additional features: signatures and constant definitions.

3.3.1 Syntax

The abstract syntax of TWEGA is given as follows (for the concrete syntax cf. Appendix A.1):

$$\textit{Terms} \quad \mathcal{T} ::= \mathcal{V} \mid \mathcal{C} \mid \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}$$

where $\mathcal{V}$ and $\mathcal{C}$ are infinite collections of variables and constants, respectively.

Constants have not been officially introduced in PTSs, but they can be simulated via variables by postulating an initial context containing these variables. Clearly, the initial context cannot be changed afterwards. Therefore, following the approach taken in the LF logical framework (which is λP) [Harper *et al.*, 1993] we split off this initial context to form a *signature*, which contains *constant declarations* as opposed to variable declarations in contexts. This allows us to represent each theorem as a judgment with an empty context.

[3] Note, however, that some doubt has been cast on the proof in [Ghani, 1997] in its full generality [Harper and Pfenning, 1999], but in most applications λP suffices and the proof in [Coquand, 1991] is unquestioned.

Similarly to [Pfenning and Schürmann, 1999], we generalize signatures by also allowing for *constant definitions*, which are semantically transparent, that is, they may be expanded both for type checking and operational syntax. The advantages of constant definitions are twofold. First, they allow for the formulation of more abstract concepts in terms of less abstract concepts. Second, constant definitions allow us to represent different levels of abstractions in one proof: If the application of a constant corresponds to a step at an abstract level, then its expansion corresponds to the proof of the correctness of that step and is located at the next lower level of abstraction.

In order to describe the basic judgments, we consider signatures (which contain only constant declarations and definitions) and contexts (which contain only variable declarations).

$$\begin{array}{ll} \textit{Signatures} & \textit{Sig} ::= \cdot \mid \textit{Sig}, \mathcal{C} : \mathcal{T} \mid \textit{Sig}, \mathcal{C}{=}\mathcal{T} : \mathcal{T} \\ \textit{Contexts} & \textit{Ctx} ::= \cdot \mid \textit{Ctx}, \mathcal{V} : \mathcal{T} \end{array}$$

We stipulate that constants and variables can appear only once in signatures and contexts, respectively. This can always be achieved through renaming.

Notation:

1. We write $c : A \in \Sigma$ and $c{=}M : A \in \Sigma$ if $c : A$ and $c{=}M : A$ occur in signature Σ, respectively. Similarly, we write $x : A \in \Gamma$, if $x : A$ occurs in context Γ.
2. We write $c \in \mathrm{dom}(\Sigma)$ if there is a type A such that $c : A \in \Sigma$ or if there is an object M of type A such that $c{=}M : A \in \Sigma$. Similarly, we write $x \in \mathrm{dom}(\Gamma)$ if there is a type A such that $x : A \in \Gamma$.
3. We write $\Sigma \subseteq \Sigma'$ if $c : A \in \Sigma'$ for all $c : A \in \Sigma$ and $c{=}M : A \in \Sigma'$ for all $c{=}M : A \in \Sigma$. Similarly, we write $\Gamma \subseteq \Gamma'$ if $x : A \in \Gamma'$ for all $x : A \in \Gamma$.

Intuitively, $c{=}M : A$ means that the constant c is defined as the object M of type A. M is the *expansion* of c.

3.3.2 Deductive System

Recall from Section 3.2.2 that λC is the $\mathrm{PTS}_{\beta\eta}$ $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$, with $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ as follows (from now on, we shall write type for $\star$ and kind for $\Box$, since these notations are more mnemonic):

$$\begin{aligned} \mathcal{S} &= \{\mathsf{type}, \mathsf{kind}\} \\ \mathcal{A} &= \{\mathsf{type} : \mathsf{kind}\} \\ \mathcal{R} &= \{(\mathsf{type}, \mathsf{type}), (\mathsf{kind}, \mathsf{type}), (\mathsf{type}, \mathsf{kind}), (\mathsf{kind}, \mathsf{kind})\} \end{aligned}$$

The type system stratifies terms into three levels: objects, types, and kinds. Let Σ be a signature, Γ a context, and A, A', B terms. The judgments are

$\vdash \Sigma\ \textit{Sig}$	Σ is a valid signature
$\vdash_\Sigma \Gamma\ \textit{Ctx}$	Γ is a valid context
$\Gamma \vdash_\Sigma A : B$	A and B are valid terms
$\Gamma \vdash_\Sigma A \equiv A'$	A is definitionally equal to A'

The notion of definitional equality we consider here is $\beta\eta$-conversion. The judgments are defined via the following inference rules:

Valid Signatures

$$\frac{}{\vdash \cdot\ Sig}\mathsf{sigemp}$$

$$\frac{\vdash \Sigma\ Sig \quad \vdash_\Sigma A : s \quad c \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, c{:}A\ Sig}\mathsf{sigdecl}$$

$$\frac{\vdash \Sigma\ Sig \quad \vdash_\Sigma A : s \quad \vdash_\Sigma M : A \quad c \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, c{=}M{:}A\ Sig}\mathsf{sigdefn}$$

where $c \in \mathcal{C}\backslash\mathcal{S}$ and $s \in \mathcal{S}$.

Valid Contexts

$$\frac{\vdash \Sigma\ Sig}{\vdash_\Sigma \cdot\ Ctx}\mathsf{ctxemp}$$

$$\frac{\vdash_\Sigma \Gamma\ Ctx \quad \Gamma \vdash_\Sigma A : s \quad x \notin \mathrm{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}A\ Ctx}\mathsf{ctxdecl}$$

where $x \in \mathcal{V}$ and $s \in \mathcal{S}$.

Valid Terms

$$\frac{\vdash_\Sigma \Gamma\ Ctx}{\Gamma \vdash_\Sigma \mathsf{type} : \mathsf{kind}}\mathsf{axiom} \qquad \frac{\vdash_\Sigma \Gamma\ Ctx \quad x{:}A \in \Gamma}{\Gamma \vdash_\Sigma x : A}\mathsf{start}$$

$$\frac{\vdash_\Sigma \Gamma\ Ctx \quad c{:}A \in \Sigma}{\Gamma \vdash_\Sigma c : A}\mathsf{declstart} \qquad \frac{\vdash_\Sigma \Gamma\ Ctx \quad c{=}M{:}A \in \Sigma}{\Gamma \vdash_\Sigma c : A}\mathsf{defnstart}$$

$$\frac{\Gamma \vdash_\Sigma A : s_1 \quad \Gamma, x{:}A \vdash_\Sigma B : s_2}{\Gamma \vdash_\Sigma \Pi x{:}A.B : s_2}(s_1, s_2) \qquad \text{for } s_1, s_2 \in \mathcal{S}$$

$$\frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma MN : B[N/x]}\mathsf{appl}$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma M : B \quad \Gamma \vdash_\Sigma \Pi x{:}A.B : s}{\Gamma \vdash_\Sigma \lambda x{:}A.M : \Pi x{:}A.B}\mathsf{abstr}$$

$$\frac{\Gamma \vdash_\Sigma A : B \quad \Gamma \vdash_\Sigma B' : s \quad \Gamma \vdash_\Sigma B \equiv B'}{\Gamma \vdash_\Sigma A : B'}\mathsf{conv}$$

where $x \in \mathcal{V}$ and $s \in \mathcal{S}$.

Note that we added a non-empty context to the rule axiom and that we allowed for the occurrence of the declaration in an arbitrary place in the context in the rules declstart, defnstart and start. This allows us to omit the rule weakening, which always can be simulated by redoing the steps of the type derivation with adapted contexts.

Definitional Equality

$$\frac{\vdash_\Sigma \Gamma\ Ctx \quad c{=}M{:}A \in \Sigma}{\Gamma \vdash_\Sigma c \equiv M}\mathsf{cngdefn} \qquad \frac{\Gamma \vdash_\Sigma A : B}{\Gamma \vdash_\Sigma A \equiv A}\mathsf{refl}$$

$$\frac{\Gamma \vdash_\Sigma A \equiv A'}{\Gamma \vdash_\Sigma A' \equiv A}\mathsf{symm} \qquad \frac{\Gamma \vdash_\Sigma A \equiv A' \quad \Gamma \vdash_\Sigma A' \equiv A''}{\Gamma \vdash_\Sigma A \equiv A''}\mathsf{trans}$$

$$\frac{\Gamma \vdash_\Sigma (\lambda x{:}A.M)N : B}{\Gamma \vdash_\Sigma (\lambda x{:}A.M)N \equiv M[N/x]}\mathsf{beta} \qquad \frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.B}{\Gamma \vdash_\Sigma M \equiv \lambda x{:}A.Mx}\mathsf{eta}$$

$$\frac{\Gamma \vdash_\Sigma A \equiv A' \quad \Gamma, x{:}A \vdash_\Sigma B \equiv B'}{\Gamma \vdash_\Sigma \Pi x{:}A.B \equiv \Pi x{:}A'.B'}\mathsf{cngprod}$$

$$\frac{\Gamma \vdash_\Sigma A \equiv A' \quad \Gamma, x{:}A \vdash_\Sigma M \equiv M'}{\Gamma \vdash_\Sigma \lambda x{:}A.M \equiv \lambda x{:}A'.M'}\mathsf{cngabstr}$$

$$\frac{\Gamma \vdash_\Sigma A \equiv A' \quad \Gamma \vdash_\Sigma B \equiv B'}{\Gamma \vdash_\Sigma AB \equiv A'B'}\mathsf{cngappl}$$

3.4 Judgments as Types

In this section, we shall examine the judgments-as-types paradigm with an example, namely the representation of the natural deduction (ND) calculus (cf. Section 3.1). [Pfenning, in prep.] gives an excellent presentation of this paradigm and the representation of the ND calculus in the LF logical framework.

Our general approach is to represent deductions of an object calculus as objects in TWEGA and judgments of the object calculus as types in TWEGA. The type system of TWEGA will ensure that only the representation of a valid deduction is well-typed. Hence, checking the validity of a deduction *(proof checking)* in the object calculus is reduced to type checking in TWEGA.

Many calculi allow for reasoning from hypotheses. For example, in the ND calculus we can conclude the judgment $\vdash P \supset Q$ if we can infer $\vdash Q$ from the hypothesis $\vdash P$. A judgment that φ is derivable under a hypothesis ψ is called *hypothetical judgment*. We can substitute a derivation of ψ for every use of ψ in order to obtain a derivation that no longer depends on the hypothesis ψ (i.e., the hypothesis plays the role of a variable for a derivation). Hence, hypothetical judgments can be represented by a function that maps a derivation of the hypothesis to a derivation of the conclusion. An application of this function corresponds to the previously mentioned substitution of the derivation of ψ for every use of ψ.

Another phenomenon that often occurs is reasoning with parameters. For example, in the ND calculus, we can conclude that $\vdash \forall x.P$ if we can show that $\vdash P[a/x]$, where a is a new parameter, which does not occur in any undischarged hypothesis. A judgment that φ is derivable with parameter p is called a *parametric judgment*. We can substitute an expression ψ for every occurrence of p in the derivation of φ in order to obtain a derivation that no longer depends on the parameter p (i.e., the parameter plays the role of a variable for an expression). Thus, parametric judgments can be represented by a function that maps an expression to a derivation of the instantiated conclusion. Again, an application of this function corresponds to the substitution of the expression for the parameter throughout the derivation of the conclusion.

Let us now proceed to an illustrative example, the representation of the ND calculus in TWEGA.

3.4.1 The Representation of the Natural Deduction Calculus

The representation of the ND calculus consists of two main parts: the representation of terms and formulae and the representation of the valid derivations. The representation technique we use, called *higher-order abstract syntax*, goes back to Church [1940]. Its main principle is to represent constructs that bind variables by λ-abstraction. Hence, the variables bound by universal and existential quantifications are represented by variables in TWEGA. Since hypotheses and parameters play the role of variables in the derivation, they are represented by variables as well.

In this section, we simultaneously construct the signature Σ^{ND} (we just write Σ) that represents the connectives and inference rules of the ND calculus in TWEGA and define the encoding $\ulcorner\cdot\urcorner$ that encodes terms, formulae, propositional parameters and valid derivations of the ND calculus by TWEGA terms in long $\beta\eta$-normal form.

Terms and Formulae

We distinguish terms and formulae by attributing them with different types. In keeping with general practice, we call the type of terms i and the type of formulae and propositional parameters o. Hence, we start with the signature Σ that only contains:

$$\begin{array}{lcl} \mathsf{i} & : & \mathsf{type} \\ \mathsf{o} & : & \mathsf{type} \end{array}$$

For every n-ary function symbol f in $\mathbf{C}_f$ and for every m-ary predicate symbol P in $\mathbf{C}_p$ we add to Σ declarations

$$\begin{array}{lcl} \mathsf{f} & : & \underbrace{\mathsf{i}\to\cdots\to\mathsf{i}}_{n \text{ times}}\to\mathsf{i} \\ \mathsf{P} & : & \underbrace{\mathsf{i}\to\cdots\to\mathsf{i}}_{m \text{ times}}\to\mathsf{o} \end{array}$$

Hence, since $\top, \bot \in \mathbf{C}_p$, we add

$$\begin{array}{lcl} \mathsf{true} & : & \mathsf{o} \\ \mathsf{false} & : & \mathsf{o} \end{array}$$

Since the connectives $\wedge$, $\vee$, $\supset$ and $\neg$ compose formulae from smaller formulae we also need the following declarations in Σ:

$$\begin{array}{lcl} \mathsf{and} & : & \mathsf{o}\to\mathsf{o}\to\mathsf{o} \\ \mathsf{or} & : & \mathsf{o}\to\mathsf{o}\to\mathsf{o} \\ \mathsf{imp} & : & \mathsf{o}\to\mathsf{o}\to\mathsf{o} \\ \mathsf{not} & : & \mathsf{o}\to\mathsf{o} \end{array}$$

Universal and existential quantification take a variable and a formula to form a new formula. Recall that employing higher-order abstract syntax we represent variable binding constructs by λ-abstractions. Hence, we declare two constants that take λ-abstractions to form formulae:

$$\begin{array}{lcl} \mathsf{forall} & : & (\mathsf{i}\to\mathsf{o})\to\mathsf{o} \\ \mathsf{exists} & : & (\mathsf{i}\to\mathsf{o})\to\mathsf{o} \end{array}$$

Then, the encoding of terms, formulae and propositional parameters is given by

$$\begin{array}{rcl} \ulcorner x \urcorner & = & x \qquad \text{for } x \in \mathbf{V} \\ \ulcorner a \urcorner & = & a \qquad \text{for } a \in \mathbf{P}_f \cup \mathbf{P}_p \\ \ulcorner f(t_1,\ldots,t_n) \urcorner & = & \mathsf{f}\ \ulcorner t_1 \urcorner \ldots \ulcorner t_n \urcorner \\ \ulcorner P(t_1,\ldots,t_m) \urcorner & = & \mathsf{P}\ \ulcorner t_1 \urcorner \ldots \ulcorner t_m \urcorner \\ \ulcorner \top \urcorner & = & \mathsf{true} \\ \ulcorner \bot \urcorner & = & \mathsf{false} \\ \ulcorner A \wedge B \urcorner & = & \mathsf{and}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\ \ulcorner A \vee B \urcorner & = & \mathsf{or}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\ \ulcorner A \supset B \urcorner & = & \mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\ \ulcorner \neg A \urcorner & = & \mathsf{not}\ \ulcorner A \urcorner \\ \ulcorner \forall x.A \urcorner & = & \mathsf{forall}\ \lambda x{:}\mathsf{i}.\ulcorner A \urcorner \\ \ulcorner \exists x.A \urcorner & = & \mathsf{exists}\ \lambda x{:}\mathsf{i}.\ulcorner A \urcorner \end{array}$$

Note that forall and exists both take as argument a λ-abstraction of type $\mathsf{i}\to\mathsf{o}$.

Valid Derivations

In order to represent derivations, we need a representation for judgments. A judgment $\vdash C$ will be represented by a type family nd that is indexed by a formula C

(thus justifying the notion 'judgments-as-types'). Hence, we add to Σ the declaration

$$\mathsf{nd} \;:\; \mathsf{o} \to \mathsf{type}$$

and the encoding of a judgment is given by

$$\ulcorner \vdash C \urcorner \;=\; \mathsf{nd}\ \ulcorner C \urcorner$$

The methodology for encoding inference rules is the following. Recall that a hypothetical judgment can be represented by a function from the derivation of the hypothesis to the derivation of the conclusion. Therefore, when representing inference rules, hypothetical judgments are encoded by

$$\ulcorner \begin{array}{c} [\vdash A]^u \\ \vdots \\ \vdash C \end{array} \urcorner \;=\; \Pi u{:}\mathsf{nd}\ \ulcorner A \urcorner . \mathsf{nd}\ \ulcorner C \urcorner$$

Similarly, parametric judgments can be represented by a function from an expression to the derivation of the conclusion. Thus, if p is a parameter in $\vdash C$ in an inference rule to represent, we encode the judgment $\vdash C$ by

$$\ulcorner \vdash C \urcorner \;=\; \begin{cases} \Pi p{:}\mathsf{i}.\mathsf{nd}\ \ulcorner C \urcorner & \text{if } p \in \mathbf{P}_f \\ \Pi p{:}\mathsf{o}.\mathsf{nd}\ \ulcorner C \urcorner & \text{if } p \in \mathbf{P}_p \end{cases}$$

Note that representing hypothetical and parametric judgments as functions is in accord with higher-order abstract syntax, since hypotheses and parameters of a derivation can be considered as variables in the derivation.

An inference rule is then represented in the signature as a constant that can be considered as a function from the derivations of the premises of the rule to the derivation of the conclusion of the rule. Let $\vdash W_1, \ldots, \vdash W_n, \vdash C$ be judgments, where $\vdash W_1, \ldots, \vdash W_n$ may be hypothetical or parametric judgments. For each inference rule

$$\frac{\vdash W_1 \quad \ldots \quad \vdash W_n}{\vdash C} R$$

with $X_1, \ldots, X_m$ the meta-variables for formulae in $W_1, \ldots, W_n, C$ we declare in Σ a constant

$$\mathsf{R} \;:\; \Pi X_1{:}\mathsf{o}. \ldots . \Pi X_m{:}\mathsf{o}. \ulcorner \vdash W_1 \urcorner \to \cdots \to \ulcorner \vdash W_n \urcorner \to \ulcorner \vdash C \urcorner$$

We say that $X_1, \ldots, X_m$ are the *implicit arguments* to R. Now, let $A_1, \ldots, A_m$ be formulae; let $\mathcal{D}_1, \ldots, \mathcal{D}_n$ be derivations of the judgments $\vdash W_1', \ldots, \vdash W_n'$, where $W_i' = W_i[A_1/X_1, \ldots, A_m/X_m]$; and let $C' = C[A_1/X_1, \ldots, A_m/X_m]$. Then, the application of the inference rule R is encoded by

$$\ulcorner \frac{\begin{array}{ccc} \mathcal{D}_1 & & \mathcal{D}_n \\ \vdash W_1' & \cdots & \vdash W_n' \end{array}}{\vdash C'} R \urcorner \;=\; \mathsf{R}\ \ulcorner A_1 \urcorner \ldots \ulcorner A_m \urcorner\ \ulcorner \mathcal{D}_1 \urcorner \ldots \ulcorner \mathcal{D}_n \urcorner$$

Let us consider the inference rules $\wedge I$, $\supset I$ and $\forall I$ as examples:

Rule $\wedge I$: We declare

$$\mathsf{andi} \;:\; \Pi A{:}\mathsf{o}. \Pi B{:}\mathsf{o}. \mathsf{nd}\ A \to \mathsf{nd}\ B \to \mathsf{nd}\ (\mathsf{and}\ A\ B)$$

and the encoding of an application is given by

$$\ulcorner \frac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \vdash A & \vdash B \end{array}}{\vdash A \wedge B} \wedge I \urcorner \;=\; \mathsf{andi}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner$$

Table 3.6. The Signature Σ^{ND}

Types

i	:	type
o	:	type
nd	:	$\mathsf{o} \to \mathsf{type}$

Function Symbols

f	:	$\mathsf{i} \to \cdots \to \mathsf{i} \to \mathsf{i}$

Predicate Symbols and Connectives

P	:	$\mathsf{i} \to \cdots \to \mathsf{i} \to \mathsf{o}$
true	:	o
false	:	o
and	:	$\mathsf{o} \to \mathsf{o} \to \mathsf{o}$
or	:	$\mathsf{o} \to \mathsf{o} \to \mathsf{o}$
imp	:	$\mathsf{o} \to \mathsf{o} \to \mathsf{o}$
not	:	$\mathsf{o} \to \mathsf{o}$
forall	:	$(\mathsf{i} \to \mathsf{o}) \to \mathsf{o}$
exists	:	$(\mathsf{i} \to \mathsf{o}) \to \mathsf{o}$

Inference Rules

andi	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ A \to \mathsf{nd}\ B \to \mathsf{nd}\ (\mathsf{and}\ A\ B)$
andel	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ (\mathsf{and}\ A\ B) \to \mathsf{nd}\ A$
ander	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ (\mathsf{and}\ A\ B) \to \mathsf{nd}\ B$
oril	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ A \to \mathsf{nd}\ (\mathsf{or}\ A\ B)$
orir	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ B \to \mathsf{nd}\ (\mathsf{or}\ A\ B)$
ore	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\Pi C{:}\mathsf{o}.\mathsf{nd}\ (\mathsf{or}\ A\ B) \to (\mathsf{nd}\ A \to \mathsf{nd}\ C) \to (\mathsf{nd}\ B \to \mathsf{nd}\ C) \to \mathsf{nd}\ C$
impi	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.(\mathsf{nd}\ A \to \mathsf{nd}\ B) \to \mathsf{nd}\ (\mathsf{imp}\ A\ B)$
impe	:	$\Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ (\mathsf{imp}\ A\ B) \to \mathsf{nd}\ A \to \mathsf{nd}\ B$
noti	:	$\Pi A{:}\mathsf{o}.(\Pi p{:}\mathsf{o}.\mathsf{nd}\ A \to \mathsf{nd}\ p) \to \mathsf{nd}\ (\mathsf{not}\ A)$
note	:	$\Pi A{:}\mathsf{o}.\Pi C{:}\mathsf{o}.\mathsf{nd}\ A \to \mathsf{nd}\ (\mathsf{not}\ A) \to \mathsf{nd}\ C$
truei	:	nd true
falsee	:	$\Pi C{:}\mathsf{o}.\mathsf{nd}\ \mathsf{false} \to \mathsf{nd}\ C$
foralli	:	$\Pi A{:}\mathsf{i} \to \mathsf{o}.(\Pi a{:}\mathsf{i}.\mathsf{nd}\ (A\ a)) \to \mathsf{nd}\ (\mathsf{forall}\ A)$
foralle	:	$\Pi A{:}\mathsf{i} \to \mathsf{o}.\mathsf{nd}\ (\mathsf{forall}\ A) \to (\Pi t{:}\mathsf{i}.\mathsf{nd}\ (A\ t))$
existsi	:	$\Pi A{:}\mathsf{i} \to \mathsf{o}.\Pi t{:}\mathsf{i}.\mathsf{nd}\ (A\ t) \to \mathsf{nd}\ (\mathsf{exists}\ A)$
existse	:	$\Pi A{:}\mathsf{i} \to \mathsf{o}.\Pi C{:}\mathsf{o}.\mathsf{nd}\ (\mathsf{exists}\ A) \to (\Pi a{:}\mathsf{i}.\mathsf{nd}\ (A\ a) \to \mathsf{nd}\ C) \to \mathsf{nd}\ C$

Rule $\supset I$: This rule has as premise a hypothetical judgment. Hence, we declare

$$\mathsf{impi} \quad : \quad \Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.(\mathsf{nd}\ A \to \mathsf{nd}\ B) \to \mathsf{nd}\ (\mathsf{imp}\ A\ B)$$

and define the encoding of a step in a derivation using $\supset I$ as

$$\ulcorner \dfrac{\begin{array}{c}[\vdash A]^u \\ \mathcal{D} \\ \vdash B\end{array}}{\vdash A \supset B} \supset I^u \urcorner \quad = \quad \mathsf{impi}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ (\lambda u{:}\mathsf{nd}\ \ulcorner A \urcorner.\ulcorner \mathcal{D} \urcorner)$$

Note that the type of $\lambda u{:}\mathsf{nd}\ \ulcorner A \urcorner.\ulcorner \mathcal{D} \urcorner$ is $\mathsf{nd}\ \ulcorner A \urcorner \to \mathsf{nd}\ \ulcorner B \urcorner$.

Rule $\forall I$: This rule has as premise a parametric judgment. Therefore, we declare

$$\mathsf{foralli} \quad : \quad \Pi A{:}\mathsf{i} \to \mathsf{o}.(\Pi a{:}\mathsf{i}.\mathsf{nd}\ (A\ a)) \to \mathsf{nd}\ (\mathsf{forall}\ A)$$

The application of $\forall I$ is encoded by

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D} \\ \vdash A[a/x]\end{array}}{\vdash \forall x.A} \forall I(a) \urcorner \quad = \quad \mathsf{foralli}\ (\lambda x{:}\mathsf{i}.\ulcorner A \urcorner)(\lambda a{:}\mathsf{i}.\ulcorner \mathcal{D} \urcorner)$$

The encoding of the remaining rules is similar. The complete signature Σ^{ND} is summarized in Table 3.6. The full definition of the encoding function $\ulcorner \cdot \urcorner$ is given in Table 3.7.

Table 3.7: The Definition of $\ulcorner \cdot \urcorner$

Formulae

$$\begin{array}{rcll}
\ulcorner x \urcorner & = & x & \text{for } x \in \mathbf{V} \\
\ulcorner a \urcorner & = & a & \text{for } a \in \mathbf{P}_f \cup \mathbf{P}_p \\
\ulcorner f(t_1, \ldots, t_n) \urcorner & = & \mathsf{f}\ \ulcorner t_1 \urcorner \ldots \ulcorner t_n \urcorner & \\
\ulcorner P(t_1, \ldots, t_m) \urcorner & = & \mathsf{P}\ \ulcorner t_1 \urcorner \ldots \ulcorner t_m \urcorner & \\
\ulcorner \top \urcorner & = & \mathsf{true} & \\
\ulcorner \bot \urcorner & = & \mathsf{false} & \\
\ulcorner A \wedge B \urcorner & = & \mathsf{and}\ \ulcorner A \urcorner\ \ulcorner B \urcorner & \\
\ulcorner A \vee B \urcorner & = & \mathsf{or}\ \ulcorner A \urcorner\ \ulcorner B \urcorner & \\
\ulcorner A \supset B \urcorner & = & \mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner & \\
\ulcorner \neg A \urcorner & = & \mathsf{not}\ \ulcorner A \urcorner & \\
\ulcorner \forall x.A \urcorner & = & \mathsf{forall}\ \lambda x{:}\mathsf{i}.\ulcorner A \urcorner & \\
\ulcorner \exists x.A \urcorner & = & \mathsf{exists}\ \lambda x{:}\mathsf{i}.\ulcorner A \urcorner &
\end{array}$$

Derivations

$$\begin{array}{rcl}
\ulcorner \vdash C \urcorner & = & \mathsf{nd}\ \ulcorner C \urcorner \\
\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdash A & \vdash B\end{array}}{\vdash A \wedge B} \wedge I \urcorner & = & \mathsf{andi}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner \\
\ulcorner \dfrac{\begin{array}{c}\mathcal{D} \\ \vdash A \wedge B\end{array}}{\vdash A} \wedge E_l \urcorner & = & \mathsf{andel}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ \ulcorner \mathcal{D} \urcorner
\end{array}$$

continued on next page

continued from previous page

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash A \wedge B\end{array}}{\vdash B}\wedge E_r \urcorner = \mathsf{ander}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ \ulcorner\mathcal{D}\urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash A\end{array}}{\vdash A \vee B}\vee I_l \urcorner = \mathsf{oril}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ulcorner\mathcal{D}\urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash B\end{array}}{\vdash A \vee B}\vee I_r \urcorner = \mathsf{orir}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ulcorner\mathcal{D}\urcorner$$

$$\ulcorner \dfrac{\begin{array}{ccc}[\vdash A]^u & [\vdash B]^v & \\ \mathcal{D}_0 & \mathcal{D}_1 & \mathcal{D}_2\\ \vdash A \vee B & \vdash C & \vdash C\end{array}}{\vdash C}\vee E^{uv} \urcorner = \begin{array}{l}\mathsf{ore}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ \ulcorner C\urcorner\ \ulcorner\mathcal{D}_0\urcorner\\ \quad(\lambda u{:}\mathsf{nd}\ \ulcorner A\urcorner.\ulcorner\mathcal{D}_1\urcorner)\\ \quad(\lambda v{:}\mathsf{nd}\ \ulcorner B\urcorner.\ulcorner\mathcal{D}_2\urcorner)\end{array}$$

$$\ulcorner \dfrac{\begin{array}{c}[\vdash A]^u\\ \mathcal{D}\\ \vdash B\end{array}}{\vdash A \supset B}\supset I^u \urcorner = \mathsf{impi}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ (\lambda u{:}\mathsf{nd}\ \ulcorner A\urcorner.\ulcorner\mathcal{D}\urcorner)$$

$$\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash A \supset B & \vdash A\end{array}}{\vdash B}\supset E \urcorner = \mathsf{impe}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ \ulcorner\mathcal{D}_1\urcorner\ \ulcorner\mathcal{D}_2\urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}[\vdash A]^u\\ \mathcal{D}\\ \vdash p\end{array}}{\vdash \neg A}\neg I^u(p) \urcorner = \mathsf{noti}\ \ulcorner A\urcorner\ (\lambda p{:}\mathsf{o}.\lambda u{:}\mathsf{nd}\ \ulcorner A\urcorner.\ulcorner\mathcal{D}\urcorner)$$

$$\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash A & \vdash \neg A\end{array}}{\vdash C}\neg E \urcorner = \mathsf{note}\ \ulcorner A\urcorner\ \ulcorner C\urcorner\ \ulcorner\mathcal{D}_1\urcorner\ \ulcorner\mathcal{D}_2\urcorner$$

$$\ulcorner \dfrac{}{\vdash \top}\top I \urcorner = \mathsf{truei}$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash \bot\end{array}}{\vdash C}\bot E \urcorner = \mathsf{falsee}\ \ulcorner C\urcorner\ \ulcorner\mathcal{D}\urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash A[a/x]\end{array}}{\vdash \forall x.A}\forall I(a) \urcorner = \mathsf{foralli}\ (\lambda x{:}\mathsf{i}.\ulcorner A\urcorner)\ (\lambda a{:}\mathsf{i}.\ulcorner\mathcal{D}\urcorner)$$

continued on next page

continued from previous page

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash \forall x.A\end{array}}{\vdash A[t/x]}\forall E \urcorner \quad = \quad \mathsf{foralle}\ (\lambda x{:}\mathsf{i}.\ulcorner A\urcorner)\ \ulcorner\mathcal{D}\urcorner\ \ulcorner t\urcorner$$

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ \vdash A[t/x]\end{array}}{\vdash \exists x.A}\exists I \urcorner \quad = \quad \mathsf{existsi}\ (\lambda x{:}\mathsf{i}.\ulcorner A\urcorner)\ \ulcorner t\urcorner\ \ulcorner\mathcal{D}\urcorner$$

$$\ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & [\vdash A[a/x]]^u\\ \vdash \exists x.A & \mathcal{D}_2\\ & \vdash C\end{array}}{\vdash C}\exists E^u(a) \urcorner \quad = \quad \begin{array}{l}\mathsf{existse}\ (\lambda x{:}\mathsf{i}.\ulcorner A\urcorner)\ \ulcorner C\urcorner\ \ulcorner\mathcal{D}_1\urcorner\\ \qquad (\lambda a{:}\mathsf{i}.\lambda u{:}\mathsf{nd}\ \ulcorner A[a/x]\urcorner.\ulcorner\mathcal{D}_2\urcorner)\end{array}$$

Example 3.2
As an example for the representation of a derivation consider again the derivation of the theorem $\vdash A \wedge B \supset B \wedge A$ in the ND calculus from Example 3.1 on page 23:

$$\dfrac{\dfrac{\dfrac{[\vdash A \wedge B]^u}{\vdash B}\wedge E_r \quad \dfrac{[\vdash A \wedge B]^u}{\vdash A}\wedge E_l}{\vdash B \wedge A}\wedge I}{\vdash A \wedge B \supset B \wedge A}\supset I^u$$

Note that $\mathsf{A}{:}\mathsf{o}, \mathsf{B}{:}\mathsf{o} \in \Sigma$, since A and B are 0-ary predicate symbols in the derivation. The derivation is represented by the TWEGA term

$$\mathsf{impi}\ (\mathsf{and}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{and}\ \mathsf{B}\ \mathsf{A})\ (\lambda u{:}\mathsf{nd}\ (\mathsf{and}\ \mathsf{A}\ \mathsf{B}).\mathsf{andi}\ \mathsf{B}\ \mathsf{A}\ (\mathsf{ander}\ \mathsf{A}\ \mathsf{B}\ u)\ (\mathsf{andel}\ \mathsf{A}\ \mathsf{B}\ u))$$

which has the type

$$\mathsf{nd}\ (\mathsf{imp}\ (\mathsf{and}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{and}\ \mathsf{B}\ \mathsf{A}))$$

and can be added to Σ as the following constant declaration:

$$\begin{array}{rl}\mathsf{thm'} = & \mathsf{impi}\ (\mathsf{and}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{and}\ \mathsf{B}\ \mathsf{A})\\ & \quad (\lambda u{:}\mathsf{nd}\ (\mathsf{and}\ \mathsf{A}\ \mathsf{B}).\mathsf{andi}\ \mathsf{B}\ \mathsf{A}\ (\mathsf{ander}\ \mathsf{A}\ \mathsf{B}\ u)\\ & \qquad\qquad (\mathsf{andel}\ \mathsf{A}\ \mathsf{B}\ u))\\ : & \quad \mathsf{nd}\ (\mathsf{imp}\ (\mathsf{and}\ \mathsf{A}\ \mathsf{B})\ (\mathsf{and}\ \mathsf{B}\ \mathsf{A}))\end{array}$$

A better strategy is to abstract from A and B, before adding the constant definition. This allows us to use the theorem as a derived inference rule in later derivations. Hence, we add

$$\begin{array}{rl}\mathsf{thm} = & \lambda A{:}\mathsf{o}.\lambda B{:}\mathsf{o}.\mathsf{impi}\ (\mathsf{and}\ A\ B)\ (\mathsf{and}\ B\ A)\\ & \quad (\lambda u{:}\mathsf{nd}\ (\mathsf{and}\ A\ B).\mathsf{andi}\ B\ A\ (\mathsf{ander}\ A\ B\ u)\\ & \qquad\qquad (\mathsf{andel}\ A\ B\ u))\\ : & \quad \Pi A{:}\mathsf{o}.\Pi B{:}\mathsf{o}.\mathsf{nd}\ (\mathsf{imp}\ (\mathsf{and}\ A\ B)\ (\mathsf{and}\ B\ A))\end{array}$$

□

3.4.2 Adequacy of the Representation

So far, we showed how the ND calculus can be represented in TWEGA. The question now is: Is this representation adequate? In this section, we shall show how an *adequacy theorem* is formulated. Then we state and prove the adequacy theorem for the representation of the ND calculus in TWEGA. To do so, we need the following definitions:

Definition 3.20 An encoding function $\langle\cdot\rangle : \mathcal{A} \to \mathcal{B}$ is *compositional* if and only if for all expressions $A, B \in \mathcal{A}$ the following holds:

$$\langle A[B/x]\rangle = \langle A\rangle[\langle B\rangle/\langle x\rangle]$$ ■

Hence, in a compositional encoding function, substitution commutes with the encoding. In particular, by adopting higher-order abstract syntax, substitution in the logical system is encoded as substitution in TWEGA.

Definition 3.21 A signature is an *adequate* representation of a logical system if and only if there is an encoding that is a compositional bijection between syntactic entities of the logical system and long $\beta\eta$-normal forms in that signature. ■

Hence, the notion of adequacy ensures that an encoding does not introduce any additional entities and that it encodes all entities uniquely.

Now, we state and prove the adequacy theorem for the encoding of the natural deduction calculus by splitting it into three parts concerning $\mathbf{T}$, $\mathbf{W} \cup \mathbf{P}_p$ and $\mathbf{D}$.

Theorem 3.3 (Adequacy of the Encoding of the ND Calculus: Terms) *The encoding $\ulcorner\cdot\urcorner$ is a compositional bijection between*

> *terms $t \in \mathbf{T}$ with free variables in $\mathbf{V}' \subset \mathbf{V}$ and with parameters in $\mathbf{P}_f{}' \subset \mathbf{P}_f$*

and

> *terms $\mathsf{t} \in \mathcal{NF}(\mathcal{T})$ with $\Gamma \vdash_\Sigma \mathsf{t} : \mathsf{i}$, where $z{:}\mathsf{i} \in \Gamma$ for every $z \in \mathbf{V}' \cup \mathbf{P}_f{}'$.*

Proof: The encoding function $\ulcorner\cdot\urcorner$ on terms is evidently injective and maps every term $t \in \mathbf{T}$ into a term in $\beta\eta$-lnf of type i. We show surjectivity by defining the inverse function $\llcorner\cdot\lrcorner$ to $\ulcorner\cdot\urcorner$ as follows:

$$\begin{array}{rcl} \llcorner x\lrcorner & = & x \\ \llcorner a\lrcorner & = & a \\ \llcorner \mathsf{f}\ \mathsf{t}_1 \ldots \mathsf{t}_n\lrcorner & = & f(\llcorner\mathsf{t}_1\lrcorner, \ldots, \llcorner\mathsf{t}_n\lrcorner) \end{array}$$

Let $t \in \mathbf{T}$. We show that $\llcorner\ulcorner t\urcorner\lrcorner = t$ by induction over the structure of t:

Case $t = x \in \mathbf{V}'$**:** Then $\llcorner\ulcorner x\urcorner\lrcorner = \llcorner x\lrcorner = x$.

Case $t = a \in \mathbf{P}_f{}'$**:** Then $\llcorner\ulcorner a\urcorner\lrcorner = \llcorner a\lrcorner = a$.

Case $t = f(t_1, \ldots, t_n)$**:** By induction hypothesis $\llcorner\ulcorner t_i\urcorner\lrcorner = t_i$. Hence,

$$\begin{array}{rcl} \llcorner\ulcorner f(t_1, \ldots, t_n)\urcorner\lrcorner & = & \llcorner \mathsf{f}\ \ulcorner t_1\urcorner\ \ldots\ \ulcorner t_n\urcorner\lrcorner \\ & = & f(\llcorner\ulcorner t_1\urcorner\lrcorner, \ldots, \llcorner\ulcorner t_n\urcorner\lrcorner) \\ & = & f(t_1, \ldots, t_n) \end{array}$$

Finally, we show the compositionality property. Let $t, u \in \mathbf{T}$ and $x \in \mathbf{V}' \cup \mathbf{P}_f{}'$. We show $\ulcorner t[u/x]\urcorner = \ulcorner t\urcorner[\ulcorner u\urcorner/\ulcorner x\urcorner]$ by induction over the structure of t:

Case $t = x$**:** Then $\ulcorner x[t'/x]\urcorner = \ulcorner t'\urcorner = x[\ulcorner t'\urcorner/x] = \ulcorner x\urcorner[\ulcorner t'\urcorner/\ulcorner x\urcorner]$

Case $t = a \in \mathbf{V}' \cup \mathbf{P}_f{}'$, $a \neq x$**:** Then $\ulcorner a[t'/x]\urcorner = \ulcorner a\urcorner = \ulcorner a\urcorner[\ulcorner t'\urcorner/\ulcorner x\urcorner]$

Case $t = f(t_1, \ldots, t_n)$**:** By induction hypothesis $\ulcorner t_i[t'/x]\urcorner = \ulcorner t_i\urcorner[\ulcorner t'\urcorner/\ulcorner x\urcorner]$. Hence,

$$\begin{array}{rcl} \ulcorner f(t_1, \ldots, t_n)[t'/x]\urcorner & = & \ulcorner f(t_1[t'/x], \ldots, t_n[t'/x])\urcorner \\ & = & \mathsf{f}\ \ulcorner t_1[t'/x]\urcorner\ \ldots\ \ulcorner t_n[t'/x]\urcorner \\ & = & \mathsf{f}\ \ulcorner t_1\urcorner[\ulcorner t'\urcorner/\ulcorner x\urcorner]\ \ldots\ \ulcorner t_n\urcorner[\ulcorner t'\urcorner/\ulcorner x\urcorner] \\ & = & (\mathsf{f}\ \ulcorner t_1\urcorner\ \ldots\ \ulcorner t_n\urcorner)[\ulcorner t'\urcorner/\ulcorner x\urcorner] \\ & = & \ulcorner f(t_1, \ldots, t_n)\urcorner[\ulcorner t'\urcorner/\ulcorner x\urcorner] \end{array}$$ ■

Note that Theorem 3.3 requires only that the substitution of terms into terms commute with the encoding. We now show that the substitution of formulae and derivations into terms commutes with the encoding as well.

Lemma 3.4 *Let $t \in \mathbf{T}$, $p \in \mathbf{P}_p$, $u \in \mathbf{P}_d$, $\varphi \in \mathbf{W}$, and $\mathcal{D} \in \mathbf{D}$. Then the following hold:*

(i) $\ulcorner t[\varphi/p]\urcorner = \ulcorner t\urcorner[\ulcorner\varphi\urcorner/\ulcorner p\urcorner] = \ulcorner t\urcorner$

(ii) $\ulcorner t[\mathcal{D}/u]\urcorner = \ulcorner t\urcorner[\ulcorner\mathcal{D}\urcorner/\ulcorner u\urcorner] = \ulcorner t\urcorner$

Proof: Trivially true, since p and u do not occur in t and hence, $\ulcorner p\urcorner$ and $\ulcorner u\urcorner$ do not occur in $\ulcorner t\urcorner$. ∎

Theorem 3.4 (Adequacy of the Encoding of the ND Calculus: Formulae) *The encoding $\ulcorner\cdot\urcorner$ is a compositional bijection between*

> *formulae and propositional parameters $\varphi \in \mathbf{W} \cup \mathbf{P}_p{}'$ (where $\mathbf{P}_p{}' \subset \mathbf{P}_p$) with free variables in $\mathbf{V}' \subset \mathbf{V}$ and with function parameters in $\mathbf{P}_f{}' \subset \mathbf{P}_f$*

and

> *terms $\mathsf{W} \in \mathcal{NF}(\mathcal{T})$ with $\Gamma \vdash_\Sigma \mathsf{W} : \mathsf{o}$, where $z{:}\mathsf{i} \in \Gamma$ for every $z \in \mathbf{V}' \cup \mathbf{P}_f{}'$ and $p{:}\mathsf{o} \in \Gamma$ for every $p \in \mathbf{P}_p{}'$.*

Proof: As in the case of terms, the encoding function $\ulcorner\cdot\urcorner$ is evidently injective and maps every formula or propositional parameter $\varphi \in \mathbf{W} \cup \mathbf{P}_p{}'$ into a term in $\beta\eta$-lnf of type o. We show surjectivity by extending to formulae and propositional parameters the inverse function $\llcorner\cdot\lrcorner$ from the proof of Theorem 3.3:

$$\begin{array}{rcl}
\llcorner p\lrcorner & = & p \\
\llcorner \mathsf{P}\ \mathsf{t}_1 \ldots \mathsf{t}_m\lrcorner & = & P(\llcorner \mathsf{t}_1\lrcorner, \ldots, \llcorner \mathsf{t}_m\lrcorner) \\
\llcorner \mathsf{true}\lrcorner & = & \top \\
\llcorner \mathsf{false}\lrcorner & = & \bot \\
\llcorner \mathsf{and\ A\ B}\lrcorner & = & \llcorner \mathsf{A}\lrcorner \wedge \llcorner \mathsf{B}\lrcorner \\
\llcorner \mathsf{or\ A\ B}\lrcorner & = & \llcorner \mathsf{A}\lrcorner \vee \llcorner \mathsf{B}\lrcorner \\
\llcorner \mathsf{imp\ A\ B}\lrcorner & = & \llcorner \mathsf{A}\lrcorner \supset \llcorner \mathsf{B}\lrcorner \\
\llcorner \mathsf{not\ A}\lrcorner & = & \neg \llcorner \mathsf{A}\lrcorner \\
\llcorner \mathsf{forall}\ \lambda x{:}\mathsf{i}.\mathsf{A}\lrcorner & = & \forall x.\llcorner \mathsf{A}\lrcorner \\
\llcorner \mathsf{exists}\ \lambda x{:}\mathsf{i}.\mathsf{A}\lrcorner & = & \exists x.\llcorner \mathsf{A}\lrcorner
\end{array}$$

Now, let $\varphi \in \mathbf{W} \cup \mathbf{P}_p{}'$. We show that $\llcorner\ulcorner\varphi\urcorner\lrcorner = \varphi$ by induction over the structure of φ:

Case $\varphi = p \in \mathbf{P}_p{}'$**:** Then $\llcorner\ulcorner p\urcorner\lrcorner = \llcorner p\lrcorner = p$.

Case $\varphi = P(t_1, \ldots, t_m)$**:** By Theorem 3.3 $\llcorner\ulcorner t_i\urcorner\lrcorner = t_i$. Hence,

$$\begin{array}{rcl}
\llcorner\ulcorner P(t_1, \ldots, t_m)\urcorner\lrcorner & = & \llcorner \mathsf{P}\ \ulcorner t_1\urcorner\ \ldots\ \ulcorner t_m\urcorner\lrcorner \\
& = & P(\llcorner\ulcorner t_1\urcorner\lrcorner, \ldots, \llcorner\ulcorner t_m\urcorner\lrcorner) \\
& = & P(t_1, \ldots, t_m)
\end{array}$$

Case $\varphi = \psi_1 \wedge \psi_2$**:** By induction hypothesis $\llcorner\ulcorner\psi_i\urcorner\lrcorner = \psi_i$. Hence,

$$\begin{array}{rcl}
\llcorner\ulcorner\psi_1 \wedge \psi_2\urcorner\lrcorner & = & \llcorner \mathsf{and}\ \ulcorner\psi_1\urcorner\ \ulcorner\psi_2\urcorner\lrcorner \\
& = & \llcorner\ulcorner\psi_1\urcorner\lrcorner \wedge \llcorner\ulcorner\psi_2\urcorner\lrcorner \\
& = & \psi_1 \wedge \psi_2
\end{array}$$

Cases $\varphi = \psi_1 \vee \psi_2$, $\varphi = \psi_1 \supset \psi_2$ **and** $\varphi = \neg\psi$**:** These cases are similar to the previous case.

Case $\varphi = \forall x.\psi$**:** By induction hypothesis $\llcorner\ulcorner\psi\urcorner\lrcorner = \psi$

$$\begin{array}{rcl} \llcorner\ulcorner\forall x.\psi\urcorner\lrcorner & = & \llcorner \mathsf{forall}\ \lambda x{:}\mathsf{i}.\ulcorner\psi\urcorner\lrcorner \\ & = & \forall x.\llcorner\ulcorner\psi\urcorner\lrcorner \\ & = & \forall x.\psi \end{array}$$

Case $\varphi = \exists x.\psi$**:** This case is similar to the previous one.

Next, we show the compositional property for propositional parameters. Let $p, p' \in \mathbf{P}_p{}', p \neq p'$ and $\varphi \in \mathbf{W}$. Then the following hold:

$$\begin{array}{c} \ulcorner p[\varphi/p]\urcorner = \ulcorner\varphi\urcorner = \ulcorner p\urcorner[\ulcorner\varphi\urcorner/\ulcorner p\urcorner] \\ \ulcorner p'[\varphi/p]\urcorner = \ulcorner p'\urcorner = \ulcorner p'\urcorner[\ulcorner\varphi\urcorner/\ulcorner p\urcorner] \end{array}$$

Finally, we show the compositionality property for formulae. Let $\varphi \in \mathbf{W}$, $t \in \mathbf{T}$ and $x \in \mathbf{V}' \cup \mathbf{P}_f{}'$. We show $\ulcorner\varphi[t/x]\urcorner = \ulcorner\varphi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]$ by induction over the structure of φ:

Case $\varphi = P(t_1, \ldots, t_n)$**:** By Theorem 3.3 and Lemma 3.4 $\ulcorner t_i[t/x]\urcorner = \ulcorner t_i\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]$. Hence,

$$\begin{array}{rcl} \ulcorner P(t_1, \ldots, t_n)[t/x]\urcorner & = & \ulcorner P(t_1[t/x], \ldots, t_n[t/x])\urcorner \\ & = & \mathsf{P}\ \ulcorner t_1[t/x]\urcorner\ \ldots\ \ulcorner t_n[t/x]\urcorner \\ & = & \mathsf{P}\ \ulcorner t_1\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]\ \ldots\ \ulcorner t_n\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \\ & = & (\mathsf{P}\ \ulcorner t_1\urcorner\ \ldots\ \ulcorner t_n\urcorner)[\ulcorner t\urcorner/\ulcorner x\urcorner] \\ & = & \ulcorner P(t_1, \ldots, t_n)\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \end{array}$$

Case $\varphi = \psi_1 \wedge \psi_2$**:** By induction hypothesis $\ulcorner\psi_i[t/x]\urcorner = \ulcorner\psi_i\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]$. Hence,

$$\begin{array}{rcl} \ulcorner(\psi_1 \wedge \psi_2)[t/x]\urcorner & = & \ulcorner\psi_1[t/x] \wedge \psi_2[t/x]\urcorner \\ & = & \mathsf{and}\ \ulcorner\psi_1[t/x]\urcorner\ \ulcorner\psi_2[t/x]\urcorner \\ & = & \mathsf{and}\ \ulcorner\psi_1\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]\ \ulcorner\psi_2\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \\ & = & (\mathsf{and}\ \ulcorner\psi_1\urcorner\ \ulcorner\psi_2\urcorner)[\ulcorner t\urcorner/\ulcorner x\urcorner] \\ & = & \ulcorner\psi_1 \wedge \psi_2\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \end{array}$$

Cases $\varphi = \psi_1 \vee \psi_2$, $\varphi = \psi_1 \supset \psi_2$ **and** $\varphi = \neg\psi$**:** These cases are similar to the previous case.

Case $\varphi = \forall y.\psi$**:** By induction hypothesis $\ulcorner\psi[t/x]\urcorner = \ulcorner\psi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]$. Hence, if $x \neq y$

$$\begin{array}{rcl} \ulcorner(\forall y.\psi)[t/x]\urcorner & = & \ulcorner\forall y.\psi[t/x]\urcorner \\ & = & \mathsf{forall}\ \lambda y{:}i.\ulcorner\psi[t/x]\urcorner \\ & = & \mathsf{forall}\ \lambda y{:}i.\ulcorner\psi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \\ & = & (\mathsf{forall}\ \lambda y{:}i.\ulcorner\psi\urcorner)[\ulcorner t\urcorner/\ulcorner x\urcorner] \\ & = & \ulcorner\forall y.\psi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \end{array}$$

Otherwise, if $x = y$

$$\begin{array}{rcl} \ulcorner(\forall x.\psi)[t/x]\urcorner & = & \ulcorner\forall x.\psi\urcorner \\ & = & \mathsf{forall}\ \lambda x{:}i.\ulcorner\psi\urcorner \\ & = & \mathsf{forall}\ \lambda x{:}i.\ulcorner\psi\urcorner[\ulcorner t\urcorner/x] \\ & = & (\mathsf{forall}\ \lambda x{:}i.\ulcorner\psi\urcorner)[\ulcorner t\urcorner/x] \\ & = & \ulcorner\forall x.\psi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \end{array}$$

Case $\varphi = \exists y.\psi$**:** This case is similar to the previous one. ∎

Note that Theorem 3.4 does not require that the substitution of formulae for propositional parameters into formulae and the substitution of derivations into formulae commute with the encoding. Therefore, we show the following lemma.

Lemma 3.5 *Let* $\psi \in \mathbf{W}$, $p \in \mathbf{P}_p$, $u \in \mathbf{P}_d$, $\varphi \in \mathbf{W}$, *and* $\mathcal{D} \in \mathbf{D}$. *Then the following hold:*

(i) $\ulcorner\psi[\varphi/p]\urcorner = \ulcorner\psi\urcorner[\ulcorner\varphi\urcorner/\ulcorner p\urcorner] = \ulcorner\psi\urcorner$

(ii) $\ulcorner\psi[\mathcal{D}/u]\urcorner = \ulcorner\psi\urcorner[\ulcorner\mathcal{D}\urcorner/\ulcorner u\urcorner] = \ulcorner\psi\urcorner$

Proof: Trivially true, since, by definition of $\mathbf{W}$, p and u do not occur in ψ and hence $\ulcorner p\urcorner$ and $\ulcorner u\urcorner$ do not occur in $\ulcorner\psi\urcorner$. ∎

Theorem 3.5 (Adequacy of the Encoding of the ND Calculus: Valid Derivations) *For every* $\varphi \in \mathbf{W} \cup \mathbf{P}_p$, *the encoding* $\ulcorner\cdot\urcorner$ *is a compositional bijection between*

> *valid derivations* $\mathcal{D} :: \vdash\varphi \in \mathbf{D}$ *with free variables in* $\mathbf{V}' \subset \mathbf{V}$ *and with parameters in* $\mathbf{P}' = \mathbf{P}_f{}' \cup \mathbf{P}_p{}'$ *(where* $\mathbf{P}_f{}' \subset \mathbf{P}_f$ *and* $\mathbf{P}_p{}' \subset \mathbf{P}_p$*)*

and

> *terms* $\mathsf{D} \in \mathcal{NF}(\mathcal{T})$ *with* $\Gamma \vdash_\Sigma \mathsf{D} : \mathsf{nd}\ \ulcorner\varphi\urcorner$, *where* $z{:}\mathsf{i} \in \Gamma$ *for every* $z \in \mathbf{V}' \cup \mathbf{P}_f{}'$ *and* $p{:}\mathsf{o} \in \Gamma$ *for every* $p \in \mathbf{P}_p{}'$.

Proof: First, we have to prove that $\ulcorner\cdot\urcorner$ is injective and maps every valid derivation $\mathcal{D} :: \vdash\varphi$ into a term $\mathsf{D} \in \mathcal{NF}(\mathcal{T})$ with $\Gamma \vdash_\Sigma \mathsf{D} : \mathsf{nd}\ \ulcorner\varphi\urcorner$. Evidently, $\ulcorner\cdot\urcorner$ is injective and D is in $\beta\eta$-lnf by the definition of $\ulcorner\cdot\urcorner$. This leaves us to show that D is of type $\mathsf{nd}\ \ulcorner\varphi\urcorner$. This can be done by straightforward induction over the last step of the derivation $\mathcal{D} :: \vdash\varphi$. Close inspection of the inference rules shows that if the last step is of the form

$$\frac{\begin{array}{ccc}\mathcal{D}_1 & & \mathcal{D}_k\\ \vdash\psi_1 & \cdots & \vdash\psi_k\end{array}}{\vdash\varphi}R$$

then $\ulcorner\mathcal{D} :: \vdash\varphi\urcorner = \mathsf{R}\ \ulcorner A_1\urcorner \ldots \ulcorner A_m\urcorner\ \ulcorner\mathcal{D}_1\urcorner \ldots \ulcorner\mathcal{D}_k\urcorner$ for some $A_1, \ldots, A_m$ and $\mathsf{R}\ \ulcorner A_1\urcorner \ldots \ulcorner A_m\urcorner \ulcorner\mathcal{D}_1\urcorner \ldots \ulcorner\mathcal{D}_k\urcorner$ is indeed of type $\mathsf{nd}\ \ulcorner\varphi\urcorner$. The same result holds for inference rules with premises that represent hypothetical or parametric judgments.

As before, we show surjectivity by extending to valid derivations the inverse function $\llcorner\cdot\lrcorner$ from the proof of Theorem 3.4 as follows (we show only some exemplary cases):

$$\llcorner\mathsf{nd}\ \mathsf{C}\lrcorner = \vdash\llcorner\mathsf{C}\lrcorner$$

$$\llcorner\mathsf{andi}\ \mathsf{A}\ \mathsf{B}\ \mathsf{D1}\ \mathsf{D2}\lrcorner = \frac{\begin{array}{cc}\llcorner\mathsf{D1}\lrcorner & \llcorner\mathsf{D2}\lrcorner\\ \vdash\llcorner\mathsf{A}\lrcorner & \vdash\llcorner\mathsf{B}\lrcorner\end{array}}{\vdash\llcorner\mathsf{A}\lrcorner \wedge \llcorner\mathsf{B}\lrcorner}\wedge I$$

$$\llcorner\mathsf{impi}\ \mathsf{A}\ \mathsf{B}\ (\lambda u{:}\mathsf{nd}\ \mathsf{A}.\mathsf{D})\lrcorner = \frac{\begin{array}{c}[\vdash\llcorner\mathsf{A}\lrcorner]^u\\ \llcorner\mathsf{D}\lrcorner\\ \vdash\llcorner\mathsf{B}\lrcorner\end{array}}{\vdash\llcorner\mathsf{A}\lrcorner \supset \llcorner\mathsf{B}\lrcorner}\supset I^u$$

$$\llcorner\mathsf{foralli}\ (\lambda x{:}\mathsf{i}.\mathsf{A})\ (\lambda a{:}\mathsf{i}.\mathsf{D})\lrcorner = \frac{\begin{array}{c}\llcorner\mathsf{D}\lrcorner\\ \vdash\llcorner\mathsf{A}[a/x]\lrcorner\end{array}}{\vdash\forall x.\llcorner\mathsf{A}\lrcorner}\forall I(a)$$

The remaining cases are similar.

Now, let $\mathcal{D} \in \mathbf{D}$. We show that $\llcorner\ulcorner\mathcal{D}\urcorner\lrcorner = \mathcal{D}$ by induction over the structure of $\mathcal{D}$:

Case $\mathcal{D} = \vdash\psi$**:** By Theorem 3.4 $\llcorner\ulcorner\psi\urcorner\lrcorner = \psi$. Hence, $\llcorner\ulcorner\vdash\psi\urcorner\lrcorner = \llcorner\mathsf{nd}\ \ulcorner\psi\urcorner\lrcorner = \vdash\llcorner\ulcorner\psi\urcorner\lrcorner = \vdash\psi$.

Case $\wedge I$**:** Let
$$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash\psi_1 & \vdash\psi_2\end{array}}{\vdash\psi_1\wedge\psi_2}\wedge I$$
By induction hypothesis $\llcorner\ulcorner\mathcal{D}_i\urcorner\lrcorner = \mathcal{D}_i$. Furthermore, by Theorem 3.4 $\llcorner\ulcorner\psi_i\urcorner\lrcorner = \psi_i$. Hence,
$$\begin{aligned}\ulcorner\ \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash\psi_1 & \vdash\psi_2\end{array}}{\vdash\psi_1\wedge\psi_2}\wedge I\ \urcorner &= \llcorner\mathsf{andi}\ \ulcorner\psi_1\urcorner\ \ulcorner\psi_2\urcorner\ulcorner\mathcal{D}_1\urcorner\ \ulcorner\mathcal{D}_2\urcorner\lrcorner\\ \llcorner\ \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash\psi_1 & \vdash\psi_2\end{array}}{\vdash\psi_1\wedge\psi_2}\wedge I\ \lrcorner &= \dfrac{\begin{array}{cc}\llcorner\ulcorner\mathcal{D}_1\urcorner\lrcorner & \llcorner\ulcorner\mathcal{D}_2\urcorner\lrcorner\\ \vdash\llcorner\ulcorner\psi_1\urcorner\lrcorner & \vdash\llcorner\ulcorner\psi_2\urcorner\lrcorner\end{array}}{\llcorner\ulcorner\psi_1\urcorner\lrcorner\wedge\llcorner\ulcorner\psi_2\urcorner\lrcorner}\wedge I\\ &= \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash\psi_1 & \vdash\psi_2\end{array}}{\vdash\psi_1\wedge\psi_2}\wedge I\end{aligned}$$

Case $\supset I$**:** Let
$$\mathcal{D} = \dfrac{\begin{array}{c}[\vdash\psi_1]^u\\ \mathcal{D}'\\ \vdash\psi_2\end{array}}{\vdash\psi_1\supset\psi_2}\supset I^u$$
By induction hypothesis $\llcorner\ulcorner\mathcal{D}'\urcorner\lrcorner = \mathcal{D}'$. Furthermore, by Theorem 3.4 $\llcorner\ulcorner\psi_i\urcorner\lrcorner = \psi_i$. Hence,
$$\begin{aligned}\ulcorner\ \dfrac{\begin{array}{c}[\vdash\psi_1]^u\\ \mathcal{D}'\\ \vdash\psi_2\end{array}}{\vdash\psi_1\supset\psi_2}\supset I^u\ \urcorner &= \llcorner\mathsf{impi}\ \ulcorner\psi_1\urcorner\ \ulcorner\psi_2\urcorner\ (\lambda u{:}\ulcorner\psi_1\urcorner.\ulcorner\mathcal{D}'\urcorner)\lrcorner\\ \llcorner\ \dfrac{\begin{array}{c}[\vdash\psi_1]^u\\ \mathcal{D}'\\ \vdash\psi_2\end{array}}{\vdash\psi_1\supset\psi_2}\supset I^u\ \lrcorner &= \dfrac{\begin{array}{c}[\vdash\llcorner\ulcorner\psi_1\urcorner\lrcorner]^u\\ \llcorner\ulcorner\mathcal{D}'\urcorner\lrcorner\\ \vdash\llcorner\ulcorner\psi_2\urcorner\lrcorner\end{array}}{\vdash\llcorner\ulcorner\psi_1\urcorner\lrcorner\supset\llcorner\ulcorner\psi_2\urcorner\lrcorner}\supset I^u\\ &= \dfrac{\begin{array}{c}[\vdash\psi_1]^u\\ \mathcal{D}'\\ \vdash\psi_2\end{array}}{\vdash\psi_1\supset\psi_2}\supset I^u\end{aligned}$$

Case $\forall I$**:** Let
$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}'\\ \vdash\psi[a/x]\end{array}}{\vdash\forall x.\psi}\forall I(a)$$
By induction hypothesis $\llcorner\ulcorner\mathcal{D}'\urcorner\lrcorner = \mathcal{D}'$. Furthermore, $\llcorner\ulcorner\psi\urcorner\lrcorner = \psi$ by Theo-

rem 3.4 and $\llcorner\ulcorner a\urcorner\lrcorner = a$ by Theorem 3.3. Hence,

$$\begin{aligned}
\ulcorner \frac{\begin{array}{c}\mathcal{D}'\\ \vdash\psi[a/x]\end{array}}{\vdash\forall x.\psi}\forall I(a) \urcorner &= \llcorner \mathsf{foralli}\ (\lambda x{:}\mathsf{i}.\ulcorner\psi\urcorner)\ (\lambda a{:}\mathsf{i}.\ulcorner\mathcal{D}'\urcorner)\lrcorner \\
&= \frac{\begin{array}{c}\llcorner\ulcorner\mathcal{D}'\urcorner\lrcorner\\ \vdash\llcorner\ulcorner\psi\urcorner[\ulcorner a\urcorner/\ulcorner x\urcorner]\lrcorner\end{array}}{\vdash\forall x.\llcorner\ulcorner\psi\urcorner\lrcorner}\forall I(\llcorner\ulcorner a\urcorner\lrcorner) \\
&= \frac{\begin{array}{c}\llcorner\ulcorner\mathcal{D}'\urcorner\lrcorner\\ \vdash\llcorner\ulcorner\psi[a/x]\urcorner\lrcorner\end{array}}{\vdash\forall x.\llcorner\ulcorner\psi\urcorner\lrcorner}\forall I(\llcorner\ulcorner a\urcorner\lrcorner) \\
&= \frac{\begin{array}{c}\mathcal{D}'\\ \vdash\psi[a/x]\end{array}}{\vdash\forall x.\psi}\forall I(a)
\end{aligned}$$

The remaining cases are similar to the previous three cases.

Finally, we show the compositionality property. Let $x \in \mathbf{V}' \cup {\mathbf{P}_f}'$, $p \in {\mathbf{P}_p}'$ and $u \in \mathbf{P}_d$. Moreover, let $t \in \mathbf{T}$, $\varphi \in \mathbf{W}$ and $\mathcal{D}' \in \mathbf{D}$. We show

$$\begin{array}{lrcl}
\text{(i)} & \ulcorner\mathcal{D}[t/x]\urcorner & = & \ulcorner\mathcal{D}\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner] \\
\text{(ii)} & \ulcorner\mathcal{D}[\varphi/p]\urcorner & = & \ulcorner\mathcal{D}\urcorner[\ulcorner\varphi\urcorner/\ulcorner p\urcorner] \\
\text{(iii)} & \ulcorner\mathcal{D}[\mathcal{D}'/u]\urcorner & = & \ulcorner\mathcal{D}\urcorner[\ulcorner\mathcal{D}'\urcorner/\ulcorner u\urcorner]
\end{array}$$

simultaneously by induction over the structure of $\mathcal{D}$:

Case $\mathcal{D} = \vdash\psi$:

$$\begin{array}{lrcl}
\text{(i)} & \ulcorner(\vdash\psi)[t/x]\urcorner & = & \ulcorner\vdash\psi[t/x]\urcorner \\
& & = & \mathsf{nd}\ \ulcorner\psi[t/x]\urcorner \\
& & = & \mathsf{nd}\ (\ulcorner\psi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]) \quad \text{by Theorem 3.4} \\
& & = & (\mathsf{nd}\ \ulcorner\psi\urcorner)[\ulcorner t\urcorner/\ulcorner x\urcorner] \\
& & = & \ulcorner\vdash\psi\urcorner[\ulcorner t\urcorner/\ulcorner x\urcorner]
\end{array}$$

(i) and (ii) are similar with Lemma 3.5 instead of Theorem 3.4.

Case $\wedge I$: Let $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \vdash\psi_1 & \vdash\psi_2\end{array}}{\vdash\psi_1 \wedge \psi_2}\wedge I$

(i) $$\ulcorner \left(\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdash \psi_1 & \vdash \psi_2\end{array}}{\vdash \psi_1 \wedge \psi_2} \wedge I \right)[t/x] \urcorner$$

$$= \ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1[t/x] & \mathcal{D}_2[t/x] \\ \vdash \psi_1[t/x] & \vdash \psi_2[t/x]\end{array}}{\vdash (\psi_1 \wedge \psi_2)[t/x]} \wedge I \urcorner$$

$$= \mathsf{andi}\ \ulcorner \vdash \psi_1[t/x] \urcorner\ \ulcorner \vdash \psi_2[t/x] \urcorner\ \ulcorner \mathcal{D}_1[t/x] \urcorner\ \ulcorner \mathcal{D}_1[t/x] \urcorner$$

$$= \mathsf{andi}\ \ulcorner \vdash \psi_1 \urcorner[\ulcorner t \urcorner / \ulcorner x \urcorner]\ \ulcorner \vdash \psi_2 \urcorner[\ulcorner t \urcorner / \ulcorner x \urcorner]\ \ulcorner \mathcal{D}_1 \urcorner[\ulcorner t \urcorner / \ulcorner x \urcorner]\ \ulcorner \mathcal{D}_1 \urcorner[\ulcorner t \urcorner / \ulcorner x \urcorner]$$
by Theorem 3.4 and induction hypothesis

$$= (\mathsf{andi}\ \ulcorner \vdash \psi_1 \urcorner\ \ulcorner \vdash \psi_2 \urcorner\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner)[\ulcorner t \urcorner / \ulcorner x \urcorner]$$

$$= \ulcorner \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \vdash \psi_1 & \vdash \psi_2\end{array}}{\vdash \psi_1 \wedge \psi_2} \wedge I \urcorner[\ulcorner t \urcorner / \ulcorner x \urcorner]$$

(ii) and (iii) are similar with Lemma 3.5 instead of Theorem 3.4.

The remaining cases are similar to the last case. ∎

In this chapter, we defined TWEGA as a suitable formalism to encode various calculi and to represent mathematical facts and derivations. An important feature of TWEGA that makes it well suited for our purposes is the concept of constant definitions, since these allow us to represent several levels of abstraction of a proof simultaneously.

Then, we showed how a concrete calculus, namely the natural deduction calculus of first-order predicate logic, can be represented in TWEGA using the judgments-as-types paradigm. Using this paradigm has two major advantages: First, the judgments-as-types paradigm reduces the problem of proof checking to the problem of type checking, and since the problem of type checking is decidable, this gives us a decision procedure to check the correctness of the proofs. Hence, TWEGA ensures that only such proofs can be encoded that are correct with respect to the represented calculus.[4] Second, the explicit representation of hypothetical and parametric judgments makes the judgment-as-types paradigm particularly well suited for the explanation of proofs.

Moreover, we showed that the representation in TWEGA is adequate. Close inspection shows that the type system rules $(\mathsf{kind}, \mathsf{type})$ and $(\mathsf{kind}, \mathsf{kind})$ have not been used in the definition of $\ulcorner \cdot \urcorner$. Hence, the encoding $\ulcorner \cdot \urcorner$ can already be used to represent the ND calculus in the logical framework LF, which can be seen as a fragment of λC. In the following chapter, we shall examine a more complex example which requires the full strength of λC for the encoding.

[4] However, note that it is also possible to represent inconsistent calculi in TWEGA that include incorrect inference rules. Nevertheless, only correct applications of such incorrect inference rules can be represented.

Chapter 4

The Encoding of Dynamic Deduction Systems

In the preceding chapter, we defined TWEGA as the formalism in *P.rex* to represent mathematical facts and derivations. Considering the case of the natural deduction calculus, we saw how a concrete calculus can be encoded into TWEGA. A similar approach can be taken for other deduction systems that implement a fixed calculus, that is, a calculus with a predefined set of inference rules. Hence, most provers can be connected to *P.rex* by providing a function that encodes the prover's calculus in TWEGA once and for all. We call such a function a *static encoding*.

However, many deduction systems, such as the ΩMEGA system [Benzmüller *et al.*, 1997], operate on a calculus that is not fixed, that is, a calculus whose set of inference rules can be extended later on. For example, in ΩMEGA the user can define new inference rules that may or may not be mapped into the application of a fixed series of inference rules that existed before. The user is even allowed to discard the built-in calculus and to provide a calculus of his own, that is, ΩMEGA can be seen as a logical framework. Hence, it is not possible to statically encode the calculus of the ΩMEGA system in TWEGA and to work with that for good.

Then, we should encode the calculus with which the user of ΩMEGA is currently working. But this is also impossible, since ΩMEGA allows for high-level inference rules, called *methods*, that have a flexible number of premises, that is, the exact number of premises cannot be calculated before the method is applied.

Our approach to cope with this problem is to define a function that encodes *applications* of inference rules instead of trying to encode inference rules *per se* in TWEGA. We call such a function a *dynamic encoding*.

First, we shall present in Section 4.1 the formalism in which proofs in ΩMEGA are represented. Next, in Section 4.2 we shall define the dynamic encoding of such proofs in TWEGA.

4.1 The ΩMEGA System

The system ΩMEGA is an interactive system supporting theorem proving in mathematics [Benzmüller *et al.*, 1997]. Its logic engine contains a proof planner, which may employ external reasoners, such as other deduction systems, constraint solvers, equation solvers or computer algebra systems. ΩMEGA makes use of the mathematical database MBase [Franke and Kohlhase, 2000], which organizes mathematical knowledge in a hierarchy of *theories*. Beside definitions, a theory in MBase contains theorems and methods for proving theorems, which both can be seen as derived inference rules that can be formulated in terms of inference rules of underlying

theories. Since the user of ΩMEGA can switch back and forth between theories or develop his own theories, the calculus of ΩMEGA is not fixed. Moreover, methods for proving theorems have in general a flexible number of premises. The exact number of premises is calculated when the method is applied in a proof. This number may vary from application to application. As an obvious example, consider the methods that are due to external reasoners. For instance, it is possible to define a method *SPASS*, which means that a subproof is assumed to be provable by the first order theorem prover SPASS [Weidenbach, 1997]. Clearly, the number of premises of a proof step proved by SPASS cannot be anticipated when the method is defined. However, when the method is applied the number of premises is clear. Certainly, such methods cannot be encoded statically. Instead, we encode the applications of the methods as they are embedded in the proof. That is, proofs in ΩMEGA are encoded dynamically.

The representation of formulae in ΩMEGA is based on a restricted polymorphic typed lambda calculus that we shall present first by giving its syntax and type system. Next, we shall present the representation of inference rules and proofs in ΩMEGA.

4.1.1 Syntax

The syntax of ΩMEGA is divided in *types* and *terms*. Type polymorphism is given by *schematic types*. We use the notions *polymorphic types* and *schematic types* interchangeably.

$$\begin{array}{ll} \textit{Simple Types} & \mathcal{T} ::= \mathcal{V} \mid \mathcal{C}\mathcal{T}\ldots\mathcal{T} \mid \mathcal{T}\rightarrow\mathcal{T} \\ \textit{Schematic Types} & \mathcal{S} ::= \mathcal{T} \mid \forall\mathcal{V}.\mathcal{S} \end{array}$$

where $\mathcal{V}$ and $\mathcal{C}$ are infinite collections of type variables and type function symbols, respectively, where each function symbol has a unique arity. Note that the schematic types include the simple types. In the following, we use τ, τ', τ_i to denote simple types and $\sigma, \sigma', \sigma_i$ to denote schematic types. A type $\gamma\tau_1\ldots\tau_n$ is called an *application type*; a type $\tau\rightarrow\tau'$ is called a *function type*.

Terms are defined as

$$\textit{Terms} \quad \mathbf{T} ::= \mathbf{V} \mid \mathbf{C} \mid \mathbf{TT} \mid \lambda\mathbf{V}.\mathbf{T}$$

where $\mathbf{V}$ and $\mathbf{C}$ are infinite collections of variables and function symbols, respectively. Each variable and each function symbol has a unique arity. In the following, we use e, e', e_i to denote ΩMEGA terms. A term ee' is called an *application*; a term $\lambda x.e$ is called a *λ-abstraction* or just *abstraction*.

In order to describe basic judgments, we consider *signatures*, which contain only (type and term) constant declarations, and *contexts*, which contain only (type and term) variable declarations.

$$\begin{array}{ll} \textit{Signatures} & \textsc{Sig} ::= \cdot \mid \textsc{Sig}, \mathcal{C}:\mathbf{N} \mid \textsc{Sig}, \mathbf{C}:\mathcal{S} \\ \textit{Contexts} & \textsc{Ctx} ::= \cdot \mid \textsc{Ctx}, \mathcal{V}:\textsc{type} \mid \textsc{Ctx}, \mathbf{V}:\mathcal{S} \end{array}$$

where $\mathbf{N}$ is the set of natural numbers. $\gamma:n$ means

$$\gamma: \underbrace{\textsc{type}\rightarrow\cdots\rightarrow\textsc{type}}_{n \text{ times}}\rightarrow\textsc{type}$$

By using this notation, the formal introduction of kinds in ΩMEGA is avoided. (See the usage of kinds in Chapter 3 for comparison.) We stipulate that constants and variables can appear only once in signatures and contexts, respectively. This can always be achieved through renaming.

Notation: We write $c \in \mathrm{dom}(\Sigma)$ and $c{:}\sigma \in \Sigma$ if $c{:}\sigma$ occurs in signature Σ; we write $\gamma \in \mathrm{dom}(\Sigma)$ and $\gamma{:}n \in \Sigma$ if $\gamma{:}n$ occurs in Σ. Similarly, we write $x \in \mathrm{dom}(\Gamma)$ and $x{:}\tau \in \Gamma$, if $x{:}\tau$ occurs in the context Γ; we write $\alpha \in \mathrm{dom}(\Gamma)$ and $\alpha{:}\,\textsc{type} \in \Gamma$, if $\alpha{:}\,\textsc{type}$ occurs in Γ.

4.1.2 Type System

Let Σ be a signature, Γ a context, σ a simple or schematic type, and e a term. The judgments are

$\vdash \Sigma\ \mathit{Sig}$	Σ is a valid signature
$\vdash_\Sigma \Gamma\ \mathit{Ctx}$	Γ is a valid context
$\Gamma \vdash_\Sigma \sigma : \textsc{type}$	σ is a valid simple type
$\Gamma \vdash_\Sigma \sigma : \textsc{ptype}$	σ is a valid schematic type
$\Gamma \vdash_\Sigma e : \sigma$	e is a valid term

The judgments are defined via the following inference rules:

Valid Signatures

$$\frac{}{\vdash \cdot\ \mathit{Sig}}\textsc{sigemp}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \quad \gamma \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, \gamma{:}n\ \mathit{Sig}}\textsc{sigtype}$$

$$\frac{\vdash \Sigma\ \mathit{Sig} \quad \vdash_\Sigma \sigma : \textsc{ptype} \quad c \notin \mathrm{dom}(\Sigma)}{\vdash \Sigma, c{:}\sigma\ \mathit{Sig}}\textsc{sigterm}$$

where $\gamma \in \mathcal{C}$, $c \in \mathbf{C}$ and n is a natural number.

Valid Contexts

$$\frac{}{\vdash_\Sigma \cdot\ \mathit{Ctx}}\textsc{ctxemp}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash_\Sigma \Gamma, \alpha{:}\,\textsc{type}\ \mathit{Ctx}}\textsc{ctxtype}$$

$$\frac{\vdash_\Sigma \Gamma\ \mathit{Ctx} \quad \Gamma \vdash_\Sigma \tau : \textsc{type} \quad x \notin \mathrm{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\tau\ \mathit{Ctx}}\textsc{ctxterm}$$

where $\alpha \in \mathcal{V}$ and $x \in \mathbf{V}$.

Valid Types

$$\frac{\alpha{:}\,\textsc{type} \in \Gamma}{\Gamma \vdash_\Sigma \alpha{:}\,\textsc{type}}\textsc{typestart}$$

$$\frac{\gamma{:}n \in \Sigma \quad \Gamma \vdash_\Sigma \tau_1 : \textsc{type} \quad \cdots \quad \Gamma \vdash_\Sigma \tau_n : \textsc{type}}{\Gamma \vdash_\Sigma \gamma\tau_1 \ldots \tau_n : \textsc{type}}\textsc{typeappl}$$

$$\frac{\Gamma \vdash_\Sigma \tau : \textsc{type} \quad \Gamma \vdash_\Sigma \tau' : \textsc{type}}{\Gamma \vdash_\Sigma \tau \to \tau' : \textsc{type}}\textsc{typefunc}$$

where $\alpha \in \mathcal{V}$.

Valid Schematic Types

$$\frac{\Gamma \vdash_\Sigma \sigma : \textsc{type}}{\Gamma \vdash_\Sigma \sigma : \textsc{ptype}}\textsc{ptypestart}$$

$$\frac{\Gamma, \alpha{:}\,\textsc{type} \vdash_\Sigma \sigma : \textsc{ptype}}{\Gamma \vdash_\Sigma \forall\alpha.\sigma : \textsc{ptype}}\textsc{ptypeabstr}$$

Valid Terms

$$\frac{\vdash_\Sigma \Gamma\ Ctx \quad t:\sigma \in \Sigma}{\Gamma \vdash_\Sigma t:\sigma}\text{SIGSTART} \qquad \frac{x:\tau \in \Gamma}{\Gamma \vdash_\Sigma x:\tau}\text{CTXSTART}$$

$$\frac{\Gamma \vdash_\Sigma t:\forall\alpha.\sigma \quad \Gamma \vdash_\Sigma \tau:\text{TYPE}}{\Gamma \vdash_\Sigma t:\sigma[\tau/\alpha]}\text{PTERMINST}$$

$$\frac{\Gamma \vdash_\Sigma z:\tau' \to \tau \quad \Gamma \vdash_\Sigma t:\tau'}{\Gamma \vdash_\Sigma zt:\tau}\text{TERMAPPL}$$

$$\frac{\Gamma, x:\tau \vdash_\Sigma t:\tau' \quad \Gamma \vdash_\Sigma \tau \to \tau':\text{TYPE}}{\Gamma \vdash_\Sigma \lambda x.t:\tau \to \tau'}\text{TERMABSTR}$$

$$\frac{\Gamma, \alpha:\text{TYPE} \vdash_\Sigma t:\sigma \quad \Gamma \vdash_\Sigma \forall\alpha.\sigma:\text{PTYPE}}{\Gamma \vdash_\Sigma t:\forall\alpha.\sigma}\text{PTERMABSTR}$$

where $z \in \mathbf{V} \cup \mathbf{C}$.

Definition 4.1 Let Σ be a signature with $o:0 \in \Sigma$ and φ a term, such that $\vdash_\Sigma \varphi : \forall\alpha_1.\ldots.\forall\alpha_n.o$, $n \geq 0$. Then, φ is called a *formula*. ∎

Note that a judgment $\vdash_\Sigma \varphi : \forall\alpha_1.\ldots.\forall\alpha_n.o$ cannot be expressed in LF, but in λC, as we shall show later in Section 4.2.2.[1]

In the following, we use $\varphi, \varphi', \varphi_i$ to denote ΩMEGA formulae.

4.1.3 Inference Rules and Proofs

Having defined the syntax and the type system of ΩMEGA, a calculus can now be given by specifying a concrete signature and a set of inference rules. Consider, for example, the following signature Σ^{ND}:

$\iota:0, \quad o:0,$
$\top:o, \quad \bot:o,$
$\wedge:o\to o\to o, \quad \vee:o\to o\to o, \quad \supset:o\to o\to o, \quad \neg:o\to o,$
$\text{FORALL}:(\iota\to o)\to o, \quad \text{EXISTS}:(\iota\to o)\to o$

Recall that $\iota:0$ and $o:0$ mean $\iota:\text{TYPE}$ and $o:\text{TYPE}$, respectively.

If we use infix notation for $\wedge$, $\vee$ and $\supset$, and declare the following notations:

$\forall x.A$ means $\text{FORALL}(\lambda x.A)$
$\exists x.A$ means $\text{EXISTS}(\lambda x.A)$

then specifying the inference rules of the ND calculus as in Table 3.1 on page 22 gives us a higher-order variant of the ND calculus. Indeed, a similarly defined higher-order variant of the ND calculus is used in MBase as the base theory from which all theories inherit their basic properties.

Let us consider the derivation of $\vdash A \wedge B \supset B \wedge A$:

$$\dfrac{\dfrac{\dfrac{[\vdash A \wedge B]^u}{\vdash B}\wedge E_r \quad \dfrac{[\vdash A \wedge B]^u}{\vdash A}\wedge E_l}{\vdash B \wedge A}\wedge I}{\vdash A \wedge B \supset B \wedge A}\supset I^u$$

[1] Note that Definition 4.1 is not the standard way of defining formulae, since a formula φ may have type $\forall\alpha_1.\ldots.\forall\alpha_n.o$ for a $n > 0$. The standard way of defining a polymorphic formula is the following:

Definition 4.1' Let Σ be a signature with $o:0 \in \Sigma$, Γ a context with $\alpha_1:\text{TYPE}, \ldots, \alpha_n:\text{TYPE} \in \Gamma$, $n \geq 0$, and φ a term, such that $\vdash_\Sigma \varphi : o$. Then, φ is called a *formula*. ∎

Note that a judgment $\Gamma \vdash_\Sigma \varphi : o$ can be expressed in LF.

The non-standard definition of polymorphic formulae in Definition 4.1 is due to the unsatisfactory management of contexts in the proof plan data structure of ΩMEGA, which does not clearly separate the signature from the contexts for historic reasons. A reimplementation of the data structure with the proper management of contexts is planned.

Table 4.6. The derivation of $\vdash A \wedge B \supset B \wedge A$.

Label	*Antecedent*	*Succedent*	*Justification*
L_1	L_1	$\vdash A \wedge B$	HYP
L_2	L_1	$\vdash B$	$\wedge E_r(L_1)$
L_3	L_1	$\vdash A$	$\wedge E_l(L_1)$
L_4	L_1	$\vdash B \wedge A$	$\wedge I(L_2, L_3)$
L_5		$\vdash A \wedge B \supset B \wedge A$	$\supset I(L_4)$

If we take the two identical hypotheses as one node, the derivation tree becomes a directed acyclic graph. Making explicit the hypotheses each formula depends on, we can also use a table to display the derivation as in Table 4.6. Each line in the table corresponds to a node in the proof graph. *Labels* are used to identify the nodes. The *antecedent* of each node denotes the undischarged hypotheses on which the formula in the node, called *succedent*, depends. The *justification* finally describes the application of an inference rule that derives the node. Note that node L_5 does not depend on the hypothesis L_1, which is discharged by the application of inference rule $\supset I$ in line L_5. A hypothesis is implicitly discharged by deleting it from the antecedent.

The tabular display of proofs suggests the representation of proofs in ΩMEGA by a so-called *proof plan data structure* (PDS). A PDS is essentially a directed acyclic graph, where the nodes contain a label, an antecedent, a succedent, and an ordered list of justifications. The links between nodes are given by the premises in justifications.

The justifications of a node are ordered by their levels of abstraction. An *abstract* justification bridges a whole subgraph of the PDS as a single step. The bridged subgraph is called the *expansion* of the abstract justification. Consider the following derived inference rule

$$\frac{\vdash P \wedge Q}{\vdash Q \wedge P} \wedge \mathit{Comm}$$

whose expansion is:

$$\frac{\dfrac{\vdash P \wedge Q}{\vdash Q} \wedge E_r \quad \dfrac{\vdash P \wedge Q}{\vdash P} \wedge E_l}{\vdash Q \wedge P} \wedge I$$

An example for an abstract proof using the derived inference rule $\wedge \mathit{Comm}$ is given in Table 4.7.

Table 4.7. The derivation of $\vdash A \wedge B \supset B \wedge A$ at an abstract level.

Label	*Antecedent*	*Succedent*	*Justification*
L_1	L_1	$\vdash A \wedge B$	HYP
L_4	L_1	$\vdash B \wedge A$	$\wedge \mathit{Comm}(L_1)$
L_5		$\vdash A \wedge B \supset B \wedge A$	$\supset I(L_4)$

We can combine the proofs given in Table 4.6 and Table 4.7 in a single PDS that contains the abstract justification as well as its expansion. This combined PDS is shown in Table 4.8.

Table 4.8. The derivation of $\vdash A \wedge B \supset B \wedge A$ at two levels of abstraction.

Label	*Antecedent*	*Succedent*	*Justification*
L_1	L_1	$\vdash A \wedge B$	HYP
L_2	L_1	$\vdash B$	$\wedge E_r(L_1)$
L_3	L_1	$\vdash A$	$\wedge E_l(L_1)$
L_4	L_1	$\vdash B \wedge A$	$\wedge Comm(L_1)$ $\wedge I(L_2, L_3)$
L_5		$\vdash A \wedge B \supset B \wedge A$	$\supset I(L_4)$

After this informal introduction of the PDS, we now give a formal definition.[2] We start with the definition of a justification, which can be considered as an application of an inference rule.

Definition 4.2 A *justification* is a 3-tuple $J = (R, \overline{W}, \overline{P})$ where R is an inference rule, $\overline{W}$ is a sequence of formulae, called *parameters*, and $\overline{P}$ is a sequence of PDS nodes, called *premises*.[3] ∎

Although we need the notion of a *justification sequence* in the next definition, we postpone its formal definition until later. Informally speaking, in a justification sequence the justifications are ordered from most abstract to least abstract.

Definition 4.3 A *PDS node* is a 4-tuple $N = (l, \mathbb{H}, \varphi, \overline{J})$ where l is a label, $\mathbb{H}$ is a set of labels of hypothesis nodes, called *antecedent*, φ is a formula, called *succedent*, and $\overline{J}$ is a justification sequence.
A *hypothesis node* is a PDS node that contains its own label in its antecedent. ∎

The next definition specifies when a set of PDS nodes constitutes a proof.

Definition 4.4 Let $\mathbb{N}$ be a set of PDS nodes, let $\mathbb{P} \subseteq \mathbb{N}$ and $C \in \mathbb{N}$. $\mathbb{N}$ is a *proof graph of C from $\mathbb{P}$* if and only if one of the following holds:

1. $C \in \mathbb{P}$.
2. Let $C = (l, \mathbb{H}, \varphi, \overline{J})$. For each justification $J = (R, \overline{W}, \overline{P})$ in $\overline{J}$ and for each premise P in $\overline{P}$ there is a set $\mathbb{N}' \subsetneq \mathbb{N}$ that is a proof graph of P from $\mathbb{P}$. ∎

Since $\mathbb{N}' \subsetneq \mathbb{N}$ this clearly defines an acyclic graph.

We now define levels of abstraction.

Definition 4.5 Let $N = (l, \mathbb{H}, \varphi, \overline{J})$ be a PDS node, $J = (R, \overline{W}, \overline{P})$ be a justification in $\overline{J}$, and $\mathbb{P} = \{P \mid P \text{ in } \overline{P}\}$.

1. The *expansion* of J is a set $\mathbb{N}$ of PDS nodes that constitutes a proof graph of N' from $\mathbb{P}$, where N' denotes N with J deleted from its justification sequence. We say the expansion of J *proves* J.
2. A justification $J = (R, \overline{W}, \overline{P})$ is *more abstract* than a justification $J' = (R', \overline{W'}, \overline{P'})$ (we write $J \succ J'$ or $J' \prec J$) if and only if there is a justification J'' such that J'' justifies N in the expansion of J and $J'' = J'$ or $J'' \succ J'$.

[2] Since ΩMEGA contains a proof planner, the PDS is also defined with features other than the logical ones. But since these features are necessary only for the planning process, we can safely omit them here (cf. [Cheikhrouhou, in prep.] for the complete definition of the PDS).

[3] In ΩMEGA, term positions are allowed as parameters as well. But since we encode only applications of inference rules instead of inference rules themselves, we can safely neglect term positions here.

3. A *justification sequence* $\overline{J}$ is an ordered sequence of justifications $J_1, \ldots, J_n$ where $J_1 \succ \cdots \succ J_n$. ■

Finally, we define the PDS formally.

Definition 4.6 Let Σ be a signature. A *proof plan data structure* (PDS) is a 5-tuple $P = (\mathbb{T}, \mathbb{C}, \mathbb{A}, C, \mathbb{N})$ where $\mathbb{T} \subseteq \mathcal{C}$ is a set of 0-ary type function symbols and $\mathbb{C} \subseteq \mathbf{C}$ is a set of term function symbols of non-schematic type, such that for every $\gamma \in \mathbb{T}$ and every $c \in \mathbb{C}$ also $\gamma, c \in \Sigma$ (we say the type and term function symbols are *locally* declared in the PDS). Furthermore, $\mathbb{N}$ is a set of PDS nodes, $\mathbb{A} \subset \mathbb{N}$ is a set of hypothesis nodes, called *assumptions*, and $C \in \mathbb{N}$ is the conclusion node, where the set of the labels of the nodes in $\mathbb{A}$ is the antecedent of C. ■

Definition 4.7 A PDS $P = (\mathbb{T}, \mathbb{C}, \mathbb{A}, C, \mathbb{N})$ is *complete* if and only if $\mathbb{N}$ constitutes a proof graph of C from $\mathbb{A}$. ■

We finish this section by collecting the meaningful objects in ΩMEGA.

Definition 4.8 Let $\mathcal{N}$ be the set of PDS nodes, $\mathcal{J}$ the set of justification sequences, and $\mathcal{P}$ the set of complete PDSs. The set Ω of all ΩMEGA objects is given by

$$\Omega = \mathcal{S} \cup \mathbf{T} \cup \mathcal{N} \cup \mathcal{J} \cup \mathcal{P}$$ ■

4.2 Encoding ΩMEGA Objects

Our aim in this section is the definition of an encoding $\ulcorner \cdot \urcorner$ that encodes ΩMEGA objects by TWEGA terms in long $\beta\eta$-normal form. Recall from Section 3.4.1 that, as we defined the encoding for the ND calculus, we simultaneously constructed a signature Σ^{ND} to ensure the declaration of constants into which ND objects and inference rules were to be mapped. Once the signature was complete, a proof could easily be encoded by giving its derivation in TWEGA.

In Section 4.1 we pointed out that ΩMEGA's high-level methods have in general a flexible number of premises that become fixed when the methods are applied. Therefore, we cannot encode ΩMEGA's signature Σ_Ω statically, but have to construct the TWEGA signature Σ on the fly from the proof. Every single application of an inference rule is added to Σ during the encoding of a PDS. Even though this approach is more cumbersome, it has the advantage that only those parts of Σ_Ω that are used in the proof under consideration are actually encoded in TWEGA. Hence, the TWEGA signature Σ is not loaded with constants that are never used.

To formalize this dynamic approach, we implement the definition of $\ulcorner \cdot \urcorner$ by the function

$$\mathcal{E} : \mathit{Sig} \times \mathit{Ctx} \times \Omega \to \mathit{Sig} \times \mathcal{NF}(\mathcal{T}) \times \mathcal{NF}(\mathcal{T})$$

that simultaneously constructs a signature and the encoding of ΩMEGA objects. The function $\mathcal{E}$ satisfies the judgment-as-types principle. It maps a TWEGA signature Σ, a TWEGA context Γ and an ΩMEGA object ω into a TWEGA signature Σ' and two TWEGA terms U and V (both in $\beta\eta$-lnf), where $\Sigma \subseteq \Sigma'$ and $\Gamma \vdash_{\Sigma'} U : V$. Then, we consider the notation $\ulcorner \cdot \urcorner$ as an abbreviation for $\mathcal{E}(\Sigma, \Gamma, \omega)$ and define it as $\ulcorner \omega \urcorner = U$.

We define $\mathcal{E}$ inductively over ΩMEGA objects ω in terms of a rule system. We start with types and terms before we shall proceed to derivations.

But first, we introduce the following notation:

Notation: We write $\lambda\overline{x_n : A_n}.M$ for $\lambda x_1 : A_1. \ldots . \lambda x_n : A_n.M$ and $\Pi\overline{x_n : A_n}.B$ for $\Pi x_1 : A_1. \ldots . \Pi x_n : A_n.B$.

4.2.1 The Encoding of the Syntax

Since the syntax of ΩMEGA allows for the representation of higher-order logic, we represent it similarly to the approach for higher-order logic in the LF logical framework as presented in [Harper *et al.*, 1993]. We start with the signature Σ^{HOL}, which contains the following declarations:

$$\begin{array}{lll} \mathsf{hol} & : & \mathsf{type} \\ \Rightarrow & : & \mathsf{hol} \rightarrow \mathsf{hol} \rightarrow \mathsf{hol} \\ \mathsf{obj} & : & \mathsf{hol} \rightarrow \mathsf{type} \\ \Lambda & : & \Pi\alpha{:}\mathsf{hol}.\Pi\beta{:}\mathsf{hol}.(\mathsf{obj}\,\alpha \rightarrow \mathsf{obj}\,\beta) \rightarrow \mathsf{obj}\,(\alpha \Rightarrow \beta) \\ \mathsf{ap} & : & \Pi\alpha{:}\mathsf{hol}.\Pi\beta{:}\mathsf{hol}.\mathsf{obj}\,(\alpha \Rightarrow \beta) \rightarrow \mathsf{obj}\,\alpha \rightarrow \mathsf{obj}\,\beta \end{array}$$

Here, hol stands for the type of higher-order logic types; $\Rightarrow$ is the constructor of higher-order function types (we use it in infix notation); obj is used as the type of objects of higher-order types; Λ and ap are the abstraction and application operator of higher-order logic, respectively.

The Encoding of Types

Since the representation of ΩMEGA is based on a typed λ-calculus, every term t in ΩMEGA has a type σ. Hence, to encode t, we have to encode σ as well.

In ΩMEGA, types consist of type variables α, application types $\gamma\tau_1 \ldots \tau_n$, function types $\tau \rightarrow \tau'$, or they are schematic types $\forall\alpha_1.\ldots.\forall\alpha_n.\tau$.

Type Variables Let Σ be a signature, Γ a context and α a type variable. Then,

$$\ulcorner\alpha\urcorner \quad = \quad \alpha$$

is implemented by

$$\frac{\alpha{:}\mathsf{hol} \in \Gamma}{\mathcal{E}(\Sigma,\Gamma,\alpha) \quad = \quad (\Sigma,\alpha,\mathsf{hol})}\,\text{TVAR}$$

Note that we require that $\alpha{:}\mathsf{hol} \in \Gamma$. The rule TPOLY, which will be defined later on in this section, ensures that this requirement is always met.

Application Types Let Σ be a signature, Γ a context, γ an n-ary type constant and $\tau_1, \ldots, \tau_n$ types, $n \geq 0$. Then,

$$\ulcorner\gamma\tau_1 \ldots \tau_n\urcorner \quad = \quad \gamma\ulcorner\tau_1\urcorner \ldots \ulcorner\tau_n\urcorner$$

is implemented by

$$\frac{\begin{array}{rcl} \mathcal{E}(\Sigma_0,\Gamma,\tau_1) & = & (\Sigma_1,T_1,\mathsf{hol}) \\ & \vdots & \\ \mathcal{E}(\Sigma_{n-1},\Gamma,\tau_n) & = & (\Sigma_n,T_n,\mathsf{hol}) \end{array}}{\mathcal{E}(\Sigma,\Gamma,\gamma\tau_1 \ldots \tau_n) \quad = \quad (\Sigma',\gamma T_1 \ldots T_n,\mathsf{hol})}\,\text{TAPPL}$$

where $\Sigma_0 = \Sigma$

$$K = \underbrace{\mathsf{hol} \rightarrow \cdots \rightarrow \mathsf{hol}}_{n \text{ times}} \rightarrow \mathsf{hol}$$

$$\Sigma' = \begin{cases} \Sigma_n & \text{if } \gamma{:}K \in \Gamma \text{ or } \gamma{:}K \in \Sigma \\ \Sigma_n, \gamma{:}K & \text{otherwise} \end{cases}$$

Note that type function symbols are added to the signature and that K cannot contain any type variables. At first glance it might seem unnecessary to check whether $\gamma{:}K$ occurs in Γ, since only variables are allowed to occur in a context. But, since some ΩMEGA (type or term) function symbols are encoded by TWEGA variables, we do have to check here. We shall elaborate on this issue in Section 4.2.2 when we describe the encoding of the PDS.

Function Types Let Σ be a signature, Γ a context and τ and τ' types. Then,

$$\ulcorner \tau \to \tau' \urcorner \quad = \quad \ulcorner \tau \urcorner \Rightarrow \ulcorner \tau' \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl}\mathcal{E}(\Sigma, \Gamma, \tau) & = & (\Sigma', T, \mathsf{hol}) \\ \mathcal{E}(\Sigma', \Gamma, \tau') & = & (\Sigma'', T', \mathsf{hol})\end{array}}{\begin{array}{rcl}\mathcal{E}(\Sigma, \Gamma, \tau \to \tau) & = & (\Sigma'', T \Rightarrow T', \mathsf{hol})\end{array}}\ \text{TFUNC}$$

Schematic Types Let Σ be a signature, Γ a context, τ a simple type and $\alpha_1, \ldots, \alpha_n$, $n \geq 0$, all type variables occurring in τ. Then,

$$\ulcorner \mathbf{\forall} \alpha_1 . \ldots . \mathbf{\forall} \alpha_n . \tau \urcorner \quad = \quad \Pi \alpha_1 : \mathsf{hol} . \ldots . \Pi \alpha_n : \mathsf{hol.obj} \ulcorner \tau \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl}\mathcal{E}(\Sigma, \Gamma', \tau) & = & (\Sigma', T, \mathsf{hol})\end{array}}{\begin{array}{rcl}\mathcal{E}(\Sigma, \Gamma, \mathbf{\forall} \alpha_1 . \ldots . \mathbf{\forall} \alpha_n . \tau) & = & (\Sigma', \Pi \overline{\alpha_n : \mathsf{hol}} . \mathsf{obj}\, T, \mathsf{type})\end{array}}\ \text{TPOLY}$$

where $\Gamma' = \Gamma, \alpha_1 : \mathsf{hol}, \ldots, \alpha_n : \mathsf{hol}$.

Before we proceed to the encoding of terms, let us examine an example:

Example 4.1
The encoding of the type $\mathbf{\forall} \alpha . (\alpha \to o) \to o$ is done as follows:

$$\begin{array}{rcl}\ulcorner \mathbf{\forall} \alpha . (\alpha \to o) \to o \urcorner & = & \Pi \alpha : \mathsf{hol.obj} \ulcorner (\alpha \to o) \to o \urcorner \\ & = & \Pi \alpha : \mathsf{hol.obj} (\ulcorner \alpha \to o \urcorner \Rightarrow \ulcorner o \urcorner) \\ & = & \Pi \alpha : \mathsf{hol.obj} ((\ulcorner \alpha \urcorner \Rightarrow \ulcorner o \urcorner) \Rightarrow \ulcorner o \urcorner) \\ & = & \Pi \alpha : \mathsf{hol.obj} ((\alpha \Rightarrow o) \Rightarrow o)\end{array}$$

Note that the declaration $o : \mathsf{hol}$ is added to the signature by rule TAPPL when $\ulcorner o \urcorner$ is calculated. □

The Encoding of Terms

In ΩMEGA, terms consist of term function symbols c, term variables x, application terms ee', or abstraction terms $\lambda x . e$, or they are terms of schematic type.

Schematic types are instantiated in ΩMEGA by the rule PTERMINST without protocoling the types replacing type variables. Therefore, we cannot tell whether the type of a function symbol is instantiated or not without comparing it with the type with which is was declared. In TWEGA, term function symbols of instantiated schematic type are encoded with arguments that account for the types that substitute type variables. The arguments are calculated by comparing the schematic type with its instance by a function $\mathcal{I}$:

$$\mathcal{I} : \mathit{Sig} \times \mathit{Ctx} \times \mathcal{V} \times \mathcal{T} \times \mathcal{T} \to \mathcal{T}$$
$$\mathcal{I}(\Sigma, \Gamma, \alpha, S, T) = \begin{cases} \mathcal{S}(\Sigma, \Gamma, \mathcal{P}(\Sigma, \Gamma, \alpha, S), T)) & \text{if } \mathcal{P}(\Sigma, \Gamma, \alpha, S) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where the functions $\mathcal{P}$ and $\mathcal{S}$ are defined as follows:

$$\mathcal{P} : \mathit{Sig} \times \mathit{Ctx} \times \mathcal{T} \times \mathcal{T} \to \mathcal{P}$$
$$\mathcal{P}(\Sigma, \Gamma, t, t') = \text{the position } p \text{ of the first occurrence of } t \text{ in } t'$$

$$\mathcal{S} : \mathit{Sig} \times \mathit{Ctx} \times \mathcal{P} \times \mathcal{T} \to \mathcal{T}$$
$$\mathcal{S}(\Sigma, \Gamma, p, t) = \text{the subterm } t' \text{ at position } p \text{ in } t$$

where $\mathcal{P}$ is the set of all positions. Hence, if $\alpha:\mathsf{hol} \in \Gamma$, S a schematic type and T its instance, $\mathcal{I}(\Sigma, \Gamma, \alpha, S, T)$ returns the subterm t of T at position p, if the first occurrence of α in S is at position p.

Term Function Symbols Let Σ be a signature, Γ a context, Σ_Ω an ΩMEGA signature, and c a term function symbol of simple type τ with $c:\sigma \in \Sigma_\Omega$, where $\sigma = \forall\alpha_1.\ldots.\forall\alpha_n.\tau'$ and $\tau = \tau'[\tau_1/\alpha_1]\ldots[\tau_n/\alpha_n]$ for some types $\tau_1, \ldots, \tau_n$ and $n \geq 0$. Note that $\sigma = \tau$ if and only if $n = 0$. Then,

$$\ulcorner c \urcorner \quad = \quad c \ulcorner \tau_1 \urcorner \ldots \ulcorner \tau_n \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl}\mathcal{E}(\Sigma, \Gamma, \tau) & = & (\Sigma', T, \mathsf{hol}) \\ \mathcal{E}(\Sigma', \cdot, \sigma) & = & (\Sigma'', S, \mathsf{type})\end{array}}{\mathcal{E}(\Sigma, \Gamma, c) \quad = \quad (\Sigma''', cT_1 \ldots T_n, \mathsf{obj}\, T)}\text{SYM}$$

where $S = \Pi\overline{\alpha_n:\mathsf{hol}}.\mathsf{obj}\, T'$
$\Gamma' = \Gamma, \alpha_1:\mathsf{hol}, \ldots, \alpha_n:\mathsf{hol}$
$T_i = \mathcal{I}(\Sigma'', \Gamma', \alpha_i, T', T)$ for $1 \leq i \leq n$
$\Sigma''' = \begin{cases} \Sigma'' & \text{if } c \in \mathrm{dom}(\Gamma) \text{ or } c \in \mathrm{dom}(\Sigma'') \\ \Sigma'', c:S & \text{otherwise} \end{cases}$

Note that function symbols are added to the signature. S cannot contain any free (type or term) variables, since no free variables are allowed in σ. As in rule TCONST we have to check for the occurrence of c in Γ if the ΩMEGA object c corresponds to a TWEGA variable (cf. Section 4.2.2).

Term Variables Let Σ be a signature, Γ a context and x a term variable. Then,

$$\ulcorner x \urcorner \quad = \quad x$$

is implemented by

$$\frac{x:\mathsf{obj}\, T \in \Gamma}{\mathcal{E}(\Sigma, \Gamma, x) \quad = \quad (\Sigma, x, \mathsf{obj}\, T)}\text{VAR}$$

Note that we require that $x:\mathsf{obj}\, T \in \Gamma$. The rule ABSTR, which will be defined later in this section, ensures that this requirement is always met.

Application Terms Let Σ be a signature, Γ a context, e a term of type $\tau' \to \tau$ and e' a term of type τ'. Then,

$$\ulcorner e e' \urcorner \quad = \quad \mathsf{ap} \ulcorner \tau' \urcorner \ulcorner \tau \urcorner \ulcorner e \urcorner \ulcorner e' \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl}\mathcal{E}(\Sigma, \Gamma, e) & = & (\Sigma', u, \mathsf{obj}\,(T' \Rightarrow T)) \\ \mathcal{E}(\Sigma', \Gamma, e') & = & (\Sigma'', u', \mathsf{obj}\, T')\end{array}}{\mathcal{E}(\Sigma, \Gamma, ee') \quad = \quad (\Sigma'', \mathsf{ap}\, T'\, T\, u\, u', \mathsf{obj}\, T)}\text{APPL}$$

Abstraction Terms Let Σ be a signature, Γ a context, e a term of type τ and x a variable of type τ'. Then,

$$\ulcorner \lambda x.e \urcorner \quad = \quad \Lambda \ulcorner \tau' \urcorner \ulcorner \tau \urcorner \lambda x:\mathsf{obj} \ulcorner \tau' \urcorner . \ulcorner e \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl}\mathcal{E}(\Sigma,\Gamma,\tau') & = & (\Sigma',T',\mathsf{hol}) \\ \mathcal{E}(\Sigma',\Gamma',e) & = & (\Sigma'',u,\mathsf{obj}\,T)\end{array}}{\mathcal{E}(\Sigma,\Gamma,\lambda x.e) \quad = \quad (\Sigma'',\Lambda\,T'\,T\,\lambda x\!:\!\mathsf{obj}\,T'.u,\mathsf{obj}\,(T'\Rightarrow T))}\text{ABSTR}$$

where $\Gamma' = \Gamma, x\!:\!\mathsf{obj}\,T'$.
This rule ensures that $x\!:\!\mathsf{obj}\,T'$ is in the context in which e will be encoded.

Terms of Schematic Type Let Σ be a signature, Γ a context and e_σ a term of type $\sigma = \forall\alpha_1.\ldots.\forall\alpha_n.\tau$, where no free variables occur in σ. Furthermore, let e_τ of type τ be the result of the n-fold application of PTERMINST to e_σ and $\alpha_1,\ldots,\alpha_n$ (i.e., the terms e_σ and e_τ look alike, but the type of e_τ is an instance of the type of e_σ).

$$\ulcorner e_\sigma \urcorner \quad = \quad \lambda\alpha_1\!:\!\mathsf{hol}.\ldots.\lambda\alpha_n\!:\!\mathsf{hol}.\ulcorner e_\tau\urcorner$$

is implemented by

$$\frac{\mathcal{E}(\Sigma,\Gamma',e_\tau) \quad = \quad (\Sigma',u,T)}{\mathcal{E}(\Sigma,\Gamma,e_\sigma) \quad = \quad (\Sigma',\lambda\overline{\alpha_n\!:\!\mathsf{hol}}.u,\Pi\overline{\alpha_n\!:\!\mathsf{hol}}.T)}\text{POLY}$$

where $\Gamma' = \Gamma,\alpha_1\!:\!\mathsf{hol},\ldots,\alpha_n\!:\!\mathsf{hol}$.

Let us now examine an example for the encoding of terms.

Example 4.2
Let us calculate the encoding of the term $\forall x.x \supset x$. Recall that this is short for FORALL$(\lambda x.x \supset x)$. We write $\mathsf{ap}_{t_1,t_2}(e_1,e_2)$ for $\mathsf{ap}\,t_1\,t_2\,e_1\,e_2$ and $\Lambda_{t_1,t_2}(e)$ for $\Lambda\,t_1\,t_2\,e$.

$$\begin{array}{rcl}
\ulcorner\forall x.x\supset x\urcorner & = & \mathsf{ap}_{\ulcorner o\to o\urcorner,\ulcorner o\urcorner}(\ulcorner\text{FORALL}_o\urcorner,\ulcorner\lambda x.x\supset x\urcorner) \\
& = & \mathsf{ap}_{o\Rightarrow o,o}(\mathsf{forall}\ \ulcorner o\urcorner,\Lambda_{\ulcorner o\urcorner,\ulcorner o\urcorner}(\lambda x\!:\!\mathsf{obj}\ulcorner o\urcorner.\ulcorner x\supset x\urcorner)) \\
& = & \mathsf{ap}_{o\Rightarrow o,o}(\mathsf{forall}\ o,\Lambda_{o,o}(\lambda x\!:\!\mathsf{obj}\,o.\mathsf{ap}_{\ulcorner o\urcorner,\ulcorner o\urcorner}(\ulcorner x\supset\urcorner,\ulcorner x\urcorner))) \\
& = & \mathsf{ap}_{o\Rightarrow o,o}(\mathsf{forall}\ o,\Lambda_{o,o}(\lambda x\!:\!\mathsf{obj}\,o.\mathsf{ap}_{o,o}(\mathsf{ap}_{\ulcorner o\urcorner,\ulcorner o\to o\urcorner}(\ulcorner\supset\urcorner,\ulcorner x\urcorner),x))) \\
& = & \mathsf{ap}_{o\Rightarrow o,o}(\mathsf{forall}\ o,\Lambda_{o,o}(\lambda x\!:\!\mathsf{obj}\,o.\mathsf{ap}_{o,o}(\mathsf{ap}_{o,o\Rightarrow o}(\mathsf{imp},x),x)))
\end{array}$$

Note that the following declarations are added to the signature:

$$\begin{array}{rcl} o & : & \mathsf{hol} \\ \mathsf{forall} & : & \Pi\alpha\!:\!\mathsf{hol}.\mathsf{obj}\,((\alpha\Rightarrow o)\Rightarrow o) \\ \mathsf{imp} & : & \mathsf{obj}\,(o\Rightarrow o\Rightarrow o)\end{array}$$

The first declaration is added by rule TAPPL when $\ulcorner o\urcorner$ is calculated. The second and third declaration are added by rule SYM when calculating $\ulcorner\text{FORALL}_o\urcorner$ and $\ulcorner\supset\urcorner$, respectively. □

4.2.2 The Encoding of Derivations

To adopt the judgments-as-types principle, we represent the judgment $\vdash\varphi$ (meaning that φ can be derived in ΩMEGA) by a type family pf that is indexed by the term $\ulcorner\varphi\urcorner$ and its type. If formulae in ΩMEGA always had type o, it would suffice to add $\mathsf{pf}\ :\mathsf{obj}\,o\to\mathsf{type}$ to Σ^{HOL}, as suggested in [Harper *et al.*, 1993]. But since formulae

in ΩMEGA are in general of type $\forall\alpha_1.\ldots.\forall\alpha_n.o$ (cf. Definition 4.1 on page 50), we instead add the following declaration to Σ^{HOL}:

$$\mathsf{pf} \ : \Pi\omega{:}\mathsf{type}.\omega \to \mathsf{type}$$

As the following type derivation shows, the type of pf is valid:

$$\dfrac{\dfrac{}{\vdash \mathsf{type} : \mathsf{kind}}\mathsf{axiom} \quad \dfrac{\dfrac{\dfrac{\omega{:}\mathsf{type} \in \omega{:}\mathsf{type}}{\omega{:}\mathsf{type} \vdash \omega : \mathsf{type}}\mathsf{ctxstart} \quad \dfrac{}{\omega{:}\mathsf{type}, x{:}\omega \vdash \mathsf{type} : \mathsf{kind}}\mathsf{axiom}}{\omega{:}\mathsf{type} \vdash \omega \to \mathsf{type} : \mathsf{kind}}(\mathsf{type},\mathsf{kind})}{\vdash \Pi\omega{:}\mathsf{type}.\omega \to \mathsf{type} : \mathsf{kind}}(\mathsf{kind},\mathsf{kind})$$

Note that we use rule (kind,kind) in the derivation. This rule is not defined in LF, but it is defined in λC.[4]

Example 4.3

1. Let φ be a formula of type o. Then, $\ulcorner\varphi\urcorner$ has type $\mathsf{obj}\, o$. Hence,

$$\ulcorner\vdash\varphi\urcorner = \mathsf{pf}\,(\mathsf{obj}\, o)\,\ulcorner\varphi\urcorner$$

2. Let ψ be a formula of type $\forall\alpha.o$. Then, $\ulcorner\psi\urcorner$ has type $\Pi\alpha{:}\mathsf{hol.obj}\, o$. Hence,

$$\ulcorner\vdash\psi\urcorner = \mathsf{pf}\,(\Pi\alpha{:}\mathsf{hol.obj}\, o)\,\ulcorner\psi\urcorner \qquad \square$$

In the remainder of this chapter, we use τ, τ', τ_i to denote simple types as well as schematic types. Moreover, we use the following notation:

Notation: For an ΩMEGA type τ, we write

$$\ulcorner\tau\urcorner \quad = \quad \begin{cases} \mathsf{obj}\ulcorner\tau\urcorner & \text{if } \tau \text{ a simple type} \\ \Pi\overline{\alpha_n{:}\mathsf{hol}}.\mathsf{obj}\ulcorner\tau'\urcorner & \text{if } \tau = \forall\alpha_1.\ldots.\forall\alpha_n.\tau' \text{ a schematic type and } \tau' \text{ is a simple type} \end{cases}$$

The Encoding of PDS Nodes

Let Σ be a signature, Γ a context and $N = (l, \mathbb{H}, \varphi, \overline{J})$ a PDS node where $\{h_1, \ldots, h_n\} \subseteq \mathbb{H}$ are the labels of the hypotheses $\{H_1, \ldots, H_n\}$ that are newly introduced by N (i.e., $\{H_1, \ldots, H_n\}$ are discharged by a justification for which N is a premise). Furthermore, let $H_i = (h_i, \mathbb{H}_i, \varphi_i, \overline{J}_i)$ with $h_i \in \mathbb{H}_i$ and τ_i the type of φ_i for $1 \leq i \leq n$, and let $\mathcal{E}(\Sigma, \Gamma, \varphi) = (\Sigma, u, T)$. We distinguish the two cases where $l{:}\mathsf{pf}\,T\,u$ does or does not occur in Γ. If $l{:}\mathsf{pf}\,T\,u \in \Gamma$ then N was introduced as a

[4]Note that this approach to encode ΩMEGA judgments is only a temporary solution, which is necessary because of the current definition of formulae in ΩMEGA.

The declaration of pf with type $\Pi\omega{:}\mathsf{type}.\omega \to \mathsf{type}$ (which is not a valid type in LF, but is a valid type in λC) is too general. It allows us to also represent unmeaningful judgments such as $\mathsf{pf}\,(\mathsf{obj}\,(o \Rightarrow o \Rightarrow o))\,\mathsf{imp}$ (which would mean that ⊃ can be derived in ΩMEGA, but that is not possible, since ⊃ is not a formula). However, the definition of the encoding $\ulcorner\cdot\urcorner$ ensures that such pathological cases do not occur.

As soon as the proof plan data structure of ΩMEGA is reimplemented and polymorphic formulae can be defined in a standard way (cf. Footnote 1 on page 50), pf will be declared as $\mathsf{pf} \ : \mathsf{obj}\, o \to \mathsf{type}$ and the encoding of derivations described in this section will be appropriately changed.

Since LF allows for the declaration $\mathsf{pf} \ : \mathsf{obj}\, o \to \mathsf{type}$, and since only the need for a declaration $\mathsf{pf} \ : \Pi\omega{:}\mathsf{type}.\omega \to \mathsf{type}$ was the reason to choose λC as the theoretic basis for TWEGA, we will also eliminate the rules (kind,type) and (kind,kind) from the type system of TWEGA when the proof plan data structure of ΩMEGA has been reimplemented. This will reduce TWEGA to an implementation of LF, thus eliminating the theoretic problems with typability and decidability of type checking (cf. Footnote 3 on page 30).

hypothetical judgment, that is, as a hypothesis of some subproof in the PDS (cf. the last premise of the rules NODE and JUST-SEQ, respectively; both rules will be defined in a moment). Since labels of nodes are unique, we use them as the names for the bound variables of abstractions that encode hypothetical judgments. Note that we ensure in the implementation of $\mathcal{E}$ that every node is encoded only once by marking already encoded nodes accordingly.

Case $l : \mathsf{pf}\, T\, u \in \Gamma$**:**

$$\ulcorner N \urcorner = \lambda h_1 : \mathsf{pf} \ulcorner \tau_1 \urcorner \ulcorner \varphi_1 \urcorner . \ldots . \lambda h_n : \mathsf{pf} \ulcorner \tau_n \urcorner \ulcorner \varphi_n \urcorner . l$$

is implemented by

$$\frac{\begin{array}{rcl} \mathcal{E}(\Sigma_0, \Gamma, \varphi_1) & = & (\Sigma_1, u_1, T_1) \\ & \vdots & \\ \mathcal{E}(\Sigma_{n-1}, \Gamma, \varphi_n) & = & (\Sigma_n, u_n, T_n) \end{array}}{\mathcal{E}(\Sigma, \Gamma, N) \quad = \quad (\Sigma_n, D, C)}\ \text{HJNODE}$$

where $\Sigma_0 = \Sigma$
$C = \Pi \overline{h_n : \mathsf{pf}\, T_n\, u_n} . \mathsf{pf}\, T\, u$
$D = \lambda \overline{h_n : \mathsf{pf}\, T_n\, u_n} . l$

Let us consider as an example the PDS from Table 4.8 on page 52. The node L_1 does not introduce any new hypotheses. Hence, if $L_1 : \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \urcorner \in \Gamma$, $\ulcorner L_1 \urcorner = L_1$.

Case $l : \mathsf{pf}\, T\, u \notin \Gamma$**:**

$$\ulcorner N \urcorner = \lambda h_1 : \mathsf{pf} \ulcorner \tau_1 \urcorner \ulcorner \varphi_1 \urcorner . \ldots . \lambda h_n : \mathsf{pf} \ulcorner \tau_n \urcorner \ulcorner \varphi_n \urcorner . \ulcorner \overline{J} \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl} \mathcal{E}(\Sigma_0, \Gamma, \varphi_1) & = & (\Sigma_1, u_1, T_1) \\ & \vdots & \\ \mathcal{E}(\Sigma_{n-1}, \Gamma, \varphi_n) & = & (\Sigma_n, u_n, T_n) \\ \mathcal{E}(\Sigma_n, \Gamma_n, \overline{J}) & = & (\Sigma_{n+1}, D', \mathsf{pf}\, T\, u) \end{array}}{\mathcal{E}(\Sigma, \Gamma, N) \quad = \quad (\Sigma_{n+1}, D, C)}\ \text{NODE}$$

where $\Sigma_0 = \Sigma$
$\Gamma_n = \Gamma, h_1 : \mathsf{pf}\, T_1\, u_1, \ldots, h_n : \mathsf{pf}\, T_n\, u_n$
$C = \Pi \overline{h_n : \mathsf{pf}\, T_n\, u_n} . \mathsf{pf}\, T\, u$
$D = \lambda \overline{h_n : \mathsf{pf}\, T_n\, u_n} . D'$

In our example PDS, L_5 does not introduce any new hypotheses, but L_4 introduces L_1 as a new hypothesis. Hence, if $L_5 \notin \mathrm{dom}(\Gamma)$, $\ulcorner L_5 \urcorner = \ulcorner \supset I(L_4) \urcorner$. If $L_4 \notin \mathrm{dom}(\Gamma)$, $\ulcorner L_4 \urcorner = \lambda L_1 : \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \urcorner . \ulcorner \wedge \mathit{Comm}(L_1), \wedge I(L_2, L_3) \urcorner$.

The Encoding of Justification Sequences

Consider a PDS node N that is justified by a justification sequence $\overline{J} = J_0, \ldots, J_k$. Since $\overline{J}$ is a justification sequence, $J_0 \succ \ldots \succ J_k$. From the definition of the expansion we know that, for $0 \leq i < k$, J_i is proved by its expansion, whose last step is justified by J_{i+1}. Therefore, we encode J_i by a constant definition that also encodes the expansion of J_i. Since J_k is on the lowest level of abstraction, there is no expansion for it. Hence, we encode it by a constant declaration. We add both constant declarations and constant definitions to the signature.

Note that constant declarations and definitions that are to be added to the signature must not contain any free variables. This will be ensured by two functions $\mathcal{A}_\lambda$ and $\mathcal{A}_\Pi$, which are defined as follows:

Let Σ be a valid signature, Γ a valid context, and let M and A be terms such that $\Gamma \vdash_\Sigma M : A$ and $\Gamma \vdash_\Sigma A : s$ for $s \in \mathcal{S}$. Then we define

$$\begin{array}{l} \mathcal{A}_\lambda, \mathcal{A}_\Pi : Sig \times Ctx \times \mathcal{T} \to \mathcal{T} \\ \mathcal{A}_\lambda(\Sigma, \Gamma, M) \;=\; \lambda\overline{x_m : v_m}.M \text{ such that } \vdash_\Sigma \lambda\overline{x_m : v_m}.M : \Pi\overline{x_m : v_m}.A \\ \mathcal{A}_\Pi(\Sigma, \Gamma, A) \;=\; \Pi\overline{y_n : w_n}.A \text{ such that } \vdash_\Sigma \Pi\overline{y_n : w_n}.A : s \end{array}$$

Now, we can give the formal definition of the encoding of justification sequences. Let Σ be a signature and Γ a context. Furthermore, let $N = (L, \overline{H}, \varphi, \overline{J}_0)$ be a PDS node, and let $\overline{J}_i = J_i, \ldots, J_k$ for $0 \le i \le k$.

Case $i = k$**:** Hence $\overline{J}_k = J_k$. With an appropriately declared new constant c

$$\ulcorner J_k \urcorner = c z_1 \ldots z_l \ulcorner P_1 \urcorner \ldots \ulcorner P_n \urcorner \ulcorner \psi_1 \urcorner \ldots \ulcorner \psi_m \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl} \mathcal{E}(\Sigma_0, \Gamma, P_1) &=& (\Sigma_1, D_1, C_1) \\ &\vdots& \\ \mathcal{E}(\Sigma_{n-1}, \Gamma, P_n) &=& (\Sigma_n, D_n, C_n) \\ \mathcal{E}(\Sigma_n, \Gamma, \psi_1) &=& (\Sigma_{n+1}, u_1, T_1) \\ &\vdots& \\ \mathcal{E}(\Sigma_{n+m-1}, \Gamma, \psi_m) &=& (\Sigma_{n+m}, u_m, T_m) \\ \mathcal{E}(\Sigma_{n+m}, \Gamma', \varphi) &=& (\Sigma_{n+m+1}, u, T) \end{array}}{\mathcal{E}(\Sigma, \Gamma', \overline{J}_k) \;=\; (\Sigma', D, C)}\ \text{JUST}$$

where
$$\begin{array}{rcl} \overline{J}_k &=& J_k = (R, (\psi_1, \ldots, \psi_m), (P_1, \ldots, P_n)) \\ \Sigma_0 &=& \Sigma \\ \Gamma' &=& \Gamma, y_1 : C_1, \ldots, y_n : C_n, x_1 : T_1, \ldots, x_m : T_m \\ C' &=& \mathcal{A}_\Pi(\Sigma'', \Gamma, \Pi\overline{y_n : C_n}.\Pi\overline{x_m : T_m}.\mathsf{pf}\, T\, u) \\ &=& \Pi\overline{z_l : S_l}.\Pi\overline{y_n : C_n}.\Pi\overline{x_m : T_m}.\mathsf{pf}\, T\, u \\ \Sigma' &=& \Sigma_{n+m+1}, c : C' \\ D &=& c z_1 \ldots z_l D_1 \ldots D_n u_1 \ldots u_m \\ C &=& (\mathsf{pf}\, T\, u)[D_1/y_1] \ldots [D_n/y_n][u_1/x_1] \ldots [u_m/x_m] \end{array}$$

Considering again our example PDS, we have $\ulcorner \supset I(L_4) \urcorner = \mathsf{impi} \ulcorner A \urcorner \ulcorner B \urcorner \ulcorner L_4 \urcorner$, where the following declaration is added to the signature:

$$\begin{array}{rcl} \mathsf{impi} &:& \Pi A : \mathsf{obj}\, o.\Pi B : \mathsf{obj}\, o.(\mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \urcorner \to \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner B \wedge A \urcorner) \\ && \quad \to \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \supset B \wedge A \urcorner : \mathsf{type} \end{array}$$

Note that $(\mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \urcorner \to \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner B \wedge A \urcorner)$ is the type of $\ulcorner L_4 \urcorner$.

Case $i < k$**:** This case is very similar to the previous one. The difference is that the new constant is defined in the signature in terms of the expansion $\overline{J}_{i+1}$ of the justification J_i. Hence, we have to encode the expansion as well. Therefore, we need the last premise of the following rule.

With an appropriately defined new constant c

$$\ulcorner \overline{J}_i \urcorner = c z_1 \ldots z_l \ulcorner P_1 \urcorner \ldots \ulcorner P_n \urcorner \ulcorner \psi_1 \urcorner \ldots \ulcorner \psi_m \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl} \mathcal{E}(\Sigma_0, \Gamma, P_1) &=& (\Sigma_1, D_1, C_1) \\ &\vdots& \\ \mathcal{E}(\Sigma_{n-1}, \Gamma, P_n) &=& (\Sigma_n, D_n, C_n) \\ \mathcal{E}(\Sigma_n, \Gamma, \psi_1) &=& (\Sigma_{n+1}, u_1, T_1) \\ &\vdots& \\ \mathcal{E}(\Sigma_{n+m-1}, \Gamma, \psi_m) &=& (\Sigma_{n+m}, u_m, T_m) \\ \mathcal{E}(\Sigma_{n+m}, \Gamma', \overline{J}_{i+1}) &=& (\Sigma_{n+m+1}, D', C') \end{array}}{\mathcal{E}(\Sigma, \Gamma, \overline{J}_i) \;=\; (\Sigma', D, C)}\ \text{JUST-SEQ}$$

$$\begin{array}{lll}
\text{where} & J_i & = (R, (\psi_1, \ldots, \psi_m), (P_1, \ldots, P_n)) \\
& \Sigma_0 & = \Sigma \\
& \Gamma' & = \Gamma, y_1 : C_1, \ldots, y_n : C_n, x_1 : T_1, \ldots, x_m : T_m \\
& D'' & = \mathcal{A}_\lambda(\Sigma_{n+m+1}, \Gamma, \lambda\overline{y_n : C_n}.\lambda\overline{x_m : T_m}.D') \\
& & = \lambda\overline{z_l : S_l}.\lambda\overline{y_n : C_n}.\lambda\overline{x_m : T_m}.D' \\
& C'' & = \Pi\overline{z_l : S_l}.\Pi\overline{y_n : C_n}.\Pi\overline{x_m : T_m}.C' \\
& \Sigma' & = \Sigma_{n+m+1}, c{=}D'' : C'' \quad (c \text{ is a new symbol}) \\
& D & = cz_1 \ldots z_l D_1 \ldots D_n u_1 \ldots u_m \\
& C & = C'[D_1/y_1] \ldots [D_n/y_n][u_1/x_1] \ldots [u_m/x_m]
\end{array}$$

Note that the calculation of D'' by $\mathcal{A}_\lambda$ ensures that C'' does not contain any free (type or term) variables either.

For example, $\ulcorner \wedge Comm(L_1), \wedge I(L_2, L_3) \urcorner = \mathsf{andcomm} \ulcorner A \urcorner \ulcorner B \urcorner \ulcorner L_1 \urcorner$, where the following is added to the signature:

$$\begin{array}{lll}
\mathsf{andcomm} & = & \lambda A : \mathsf{obj}\, o.\lambda B : \mathsf{obj}\, o.\lambda L_1 : \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \urcorner . \ulcorner \wedge I(L_2, L_3) \urcorner \\
& & : \Pi A : \mathsf{obj}\, o.\Pi B : \mathsf{obj}\, o.\mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner A \wedge B \urcorner \to \mathsf{pf}\,(\mathsf{obj}\, o) \ulcorner B \wedge A \urcorner \\
& & : \mathsf{type}
\end{array}$$

Note that during the calculation of $\ulcorner \wedge I(L_2, L_3) \urcorner$ the declaration of the constant andi is also added to the signature by the rule JUST.

To simplify the implementation multiple occurrences of justifications in different justification sequences are distinguished and individually encoded by $\mathcal{E}$ using a new constant for every occurrence.

The Encoding of the PDS

Let Σ be a signature, Γ a context and let $P = (\{\gamma_1, \ldots, \gamma_m\}, \{c_1, \ldots, c_n\}, \mathbb{A}, N, \mathbb{N})$ be a complete PDS.

The locally declared (type and term) function symbols $\gamma_1, \ldots, \gamma_m, c_1, \ldots, c_n$ can be seen as parameters of the PDS, which play the role of variables whose scopes encompass the whole PDS. Employing higher-order syntax, we represent such function symbols by variables. Therefore, in the rules TAPPL and SYM we had to check for the occurrence of variables that encode function symbols in the respective contexts.

Let $\tau_1, \ldots, \tau_n$ be the types of $c_1, \ldots, c_n$, respectively. Recall from Definition 4.6 that γ_i is 0-ary for $1 \leq i \leq m$ and τ_i is not schematic for $1 \leq i \leq n$.

$$\ulcorner P \urcorner = \ulcorner N \urcorner$$

is implemented by

$$\frac{\begin{array}{rcl}
\mathcal{E}(\Sigma_0, \Gamma, \gamma_1) & = & (\Sigma_1, \gamma_1, \mathsf{hol}) \\
& \vdots & \\
\mathcal{E}(\Sigma_{m-1}, \Gamma, \gamma_m) & = & (\Sigma_m, \gamma_m, \mathsf{hol}) \\
\mathcal{E}(\Sigma_m, \Gamma, \tau_1) & = & (\Sigma_{m+1}, T_1, \mathsf{hol}) \\
& \vdots & \\
\mathcal{E}(\Sigma_{m+n-1}, \Gamma, \tau_n) & = & (\Sigma_{m+n}, T_n, \mathsf{hol}) \\
\mathcal{E}(\Sigma_{m+n}, \Gamma', N) & = & (\Sigma_{m+n+1}, D, C)
\end{array}}{\mathcal{E}(\Sigma, \Gamma, P) = (\Sigma', D, C)}\,\text{PDS}$$

$$\begin{array}{lll}
\text{where} & \Sigma_0 & = \Sigma \\
& \Gamma' & = \Gamma, \gamma_1 : \mathsf{hol}, \ldots, \gamma_m : \mathsf{hol}, c_1 : \mathsf{obj}\, T_1, \ldots, c_n : \mathsf{obj}\, T_n \\
& D' & = \mathcal{A}_\lambda(\Sigma_{n+1}, \Gamma', D) = \lambda\overline{x_l : S_l}.D \\
& C' & = \Pi\overline{x_l : S_l}.C \\
& \Sigma' & = \Sigma_{m+n+1}, c{=}D' : C' \quad (c \text{ is a new symbol})
\end{array}$$

Let P be the PDS of Table 4.8. Then, $\ulcorner P \urcorner = \ulcorner L_5 \urcorner$ and the following definition is

added to the signature:

$$\begin{aligned}\mathsf{THM} \;=\; & \lambda A{:}\mathsf{obj}\,o.\lambda B{:}\mathsf{obj}\,o.\ulcorner L_5\urcorner \\ & : \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\supset B\wedge A\urcorner \\ & : \mathsf{type}\end{aligned}$$

Example 4.4
We now give the encoding of the PDS of Table 4.8 (see page 52). For the sake of readability, we do not descend into the encoding of formulae. (Recall that, for example, $\ulcorner A\wedge B\urcorner = \mathsf{ap}\,o\,o\,(\mathsf{ap}\,o\,(o\Rightarrow o)\,\mathsf{and}\,A)\,B$.) Nevertheless, we give the declarations of the connectives.

$$\begin{aligned}
o \;&:\; \mathsf{hol} : \mathsf{type}\\
\mathsf{and} \;&:\; \mathsf{obj}\,(o\Rightarrow o\Rightarrow o) : \mathsf{type}\\
\mathsf{imp} \;&:\; \mathsf{obj}\,(o\Rightarrow o\Rightarrow o) : \mathsf{type}\\
\mathsf{pf} \;&:\; \Pi\omega{:}\mathsf{type}.\omega\to\mathsf{type} : \mathsf{kind}\\
\mathsf{ander} \;&:\; \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\urcorner\to\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner B\urcorner : \mathsf{type}\\
\mathsf{andel} \;&:\; \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\urcorner\to\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\urcorner : \mathsf{type}\\
\mathsf{andi} \;&:\; \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.\mathsf{pf}\,(\mathsf{obj}\,o)\,B\to\mathsf{pf}\,(\mathsf{obj}\,o)\,A\to\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner B\wedge A\urcorner\\
&\quad : \mathsf{type}\\
\mathsf{andcomm} \;&=\; \lambda A{:}\mathsf{obj}\,o.\lambda B{:}\mathsf{obj}\,o.\lambda L_1{:}\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\urcorner.\\
&\qquad \mathsf{andi}\;A\;B\;(\mathsf{ander}\;A\;B\;L_1)\;(\mathsf{andel}\;A\;B\;L_1)\\
&\quad : \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\urcorner\to\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner B\wedge A\urcorner\\
&\quad : \mathsf{type}\\
\mathsf{impi} \;&:\; \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.(\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\urcorner\to\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner B\wedge A\urcorner)\\
&\qquad\qquad\to\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\supset B\wedge A\urcorner\\
&\quad : \mathsf{type}\\
\mathsf{THM} \;&=\; \lambda A{:}\mathsf{obj}\,o.\lambda B{:}\mathsf{obj}\,o.\\
&\qquad \mathsf{impi}\;A\;B\;\lambda L_1{:}\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\urcorner.\mathsf{andcomm}\;A\;B\;L_1\\
&\quad : \Pi A{:}\mathsf{obj}\,o.\Pi B{:}\mathsf{obj}\,o.\mathsf{pf}\,(\mathsf{obj}\,o)\ulcorner A\wedge B\supset B\wedge A\urcorner\\
&\quad : \mathsf{type}
\end{aligned}$$

□

4.3 Correctness and Adequacy of the Encoding

After having defined the encoding of ΩMEGA objects, we want to ensure that the encoding behaves as intended. Therefore, we prove the correctness of $\mathcal{E}$ in Section 4.3.1 and show its adequacy in Section 4.3.2.

4.3.1 Correctness of the Encoding

In this section, we aim at proving that $\mathcal{E}$ meets its specifications as spelled out in Section 4.2 on page 53, namely that $\Sigma\subseteq\Sigma'$ and $\Gamma\vdash_{\Sigma'} U:V$. We tackle this problem by splitting the correctness theorem into several lemmata with respect to the correctness of the encoding of the constituents of PDSs.

Before we start, we formulate a lemma that we often apply tacitly in the remainder of this section.

Lemma 4.1 *Let Σ and Σ' be valid signatures, where $\Sigma\subseteq\Sigma'$. Furthermore, let Γ be a valid context and $U,V\in\mathcal{T}$.*

$$\textit{If } \Gamma\vdash_{\Sigma} U:V \textit{ then } \Gamma\vdash_{\Sigma'} U:V.$$

Proof: Easy by structural induction. ■

First, we prove that $\mathcal{E}$ meets its specification when we encode types.

Lemma 4.2 (Correctness for Types) *Let $\tau \in \mathcal{S}$, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, \tau) = (\widehat{\Sigma}, U, V)$. Then the following hold:*

(i) $\Sigma \subseteq \widehat{\Sigma}$ *and* $\widehat{\Sigma}$ *is valid.*

(ii) $\Gamma \vdash_{\widehat{\Sigma}} U : V$ *and* $\Gamma \vdash_{\widehat{\Sigma}} V : s$ *for some* $s \in \mathcal{S}$. *In particular,* $s = \mathsf{type}$ *if* τ *is a simple type and* $s = \mathsf{kind}$ *if* τ *is a schematic type.*

(iii) $U, V \in \mathcal{NF}(\mathcal{T})$.

Proof: We prove the assertions simultaneously by induction over the construction of $\widehat{\Sigma}$, U and V.

Case TVAR**:**

(i) Since $\widehat{\Sigma} = \Sigma$, the assertion holds trivially.

(ii) The first part of the assertion follows directly by rule start. For the second part, $\Gamma \vdash_\Sigma \mathsf{hol} : s$ with $s = \mathsf{type}$ follows from rule $\mathsf{declstart}$.

(iii) Obvious.

Case TAPPL**:** By definition, γ is n-ary and $K = \underbrace{\mathsf{hol} \to \cdots \to \mathsf{hol}}_{n \text{ times}} \to \mathsf{hol}$.

(i) By definition, either $\Sigma' = \Sigma$ or $\Sigma' = \Sigma, \gamma{:}K$. In the first case the assertion follows trivially. In the second case, clearly $\Sigma \subseteq \Sigma'$. To show the validity of Σ' we only have to show that $\vdash_\Sigma K : \mathsf{type}$. This follows by n applications of the rule $(\mathsf{type}, \mathsf{type})$ to $\vdash_\Sigma \mathsf{hol} : \mathsf{type}$, which in turn follows from the rule $\mathsf{declstart}$.

(ii) Since $\gamma{:}K \in \Gamma$ or $\gamma{:}K \in \Sigma'$, we obtain $\Gamma \vdash_{\Sigma'} \gamma : K$ by rule start or $\mathsf{declstart}$, respectively. By induction hypothesis, for $0 < i \leq n$, $\Gamma \vdash_{\Sigma_i} T_i : \mathsf{hol}$. Hence, $\Gamma \vdash_{\Sigma'} \gamma T_1 \ldots T_n : \mathsf{hol}$ by n applications of rule appl.
$\Gamma \vdash_{\Sigma'} \mathsf{hol} : s$ with $s = \mathsf{type}$ follows from rule $\mathsf{declstart}$.

(iii) By induction hypothesis $T_1, \ldots, T_n$ in $\beta\eta$-lnf. Then, the assertion holds, since γ is fully applied in $\gamma T_1 \ldots T_n$ and hol is trivially in $\beta\eta$-lnf.

Cases TFUNC**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) By $\mathsf{declstart}$, $\Gamma \vdash_{\Sigma''} \Rightarrow \; : \mathsf{hol} \to \mathsf{hol} \to \mathsf{hol}$. Thus, the first part of the assertion follows from the induction hypothesis by twofold application of rule appl. The second part is as in case VAR.

(iii) Obvious.

Cases TPOLY**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) By induction hypothesis, $\Gamma' \vdash_{\Sigma'} T : \mathsf{hol}$, since Γ' is valid by application of $\mathsf{ctxdecl}$. By $\mathsf{declstart}$, $\Gamma' \vdash_{\Sigma'} \mathsf{obj} : \mathsf{hol} \to \mathsf{type}$. Thus, $\Gamma \vdash_{\Sigma'} \mathsf{obj}\, T : \mathsf{type}$ by rule appl. Then, the first part of the assertion follows by n-fold application of rule $(\mathsf{type},\mathsf{type})$.
$\Gamma \vdash_{\Sigma'} \mathsf{type} : s$ for $s = \mathsf{kind}$ by rule axiom.

(iii) Obvious. ∎

Corollary 4.1 *$\mathcal{E}$ is well defined on* ΩMEGA *types.*

Next, we prove the correctness of $\mathcal{E}$ for terms.

Lemma 4.3 (Correctness for Terms) *Let $t \in \mathbf{T}$ be a valid term, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, t) = (\widehat{\Sigma}, U, V)$. Then the following hold:*

(i) $\Sigma \subseteq \widehat{\Sigma}$ *and* $\widehat{\Sigma}$ *is valid.*

(ii) $\Gamma \vdash_{\widehat{\Sigma}} U : V$ *and* $\Gamma \vdash_{\widehat{\Sigma}} V : \mathsf{type}$.

(iii) $U, V \in \mathcal{NF}(\mathcal{T})$.

Proof: Similarly to the proof of Lemma 4.2, the assertions are shown simultaneously by induction over the construction of $\widehat{\Sigma}$, U, and V. The proof is spelled out in Appendix A.2. ∎

Corollary 4.2 *$\mathcal{E}$ is well defined on* ΩMEGA *terms.*

An important property of the encoding is that it preserves the types of terms, that is, if t is a term of type τ and $\mathcal{E}(\Sigma, \Gamma, t) = (\Sigma', u, T)$ for signatures Σ, Σ' and a context Γ, then $\ulcorner t \urcorner = u$ and $\ulcorner \tau \urcorner = T$. We formulate this property more precisely in the following lemma:

Lemma 4.4 (Type Preservation) *Let $t \in \mathbf{T}$ be a term of type τ, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, t) = (\Sigma', u, T)$. Then, the following hold:*

(i) If τ a simple type, then $T = \mathsf{obj}\, T'$ and $\mathcal{E}(\Sigma, \Gamma, \tau) = (\Sigma'', T', \mathsf{hol})$ for a $\Sigma'' \subseteq \Sigma'$.

(ii) If τ a schematic type $\mathcal{E}(\Sigma, \Gamma, \tau) = (\Sigma'', T, \mathsf{type})$ for a $\Sigma'' \subseteq \Sigma'$.

Proof: Using Lemma 4.3, close inspection of the rules SYM, VAR, APPL, ABSTR and POLY shows that this is indeed the case. ∎

Since, a formula in ΩMEGA has type $\forall\alpha_1.\ldots.\forall\alpha_n.o$ for an $n \geq 0$ by definition, we obtain the following corollary:

Corollary 4.3 *Let φ be a formula, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, \varphi) = (\Sigma', u, T)$. Then*

$$o : \mathsf{hol} \in \Sigma' \text{ and } T = \Pi\overline{\alpha_n : \mathsf{type}}.\mathsf{obj}\, o.$$

for some $n \geq 0$.

Lemma 4.3 and Corollary 4.3 state the correctness of the encoding of the first constituents, the formulae of a PDS. The correctness of the encoding of the remaining constituents is stated in the following:

Lemma 4.5 (Correctness for Derivations) *Let ω be a PDS node, a justification sequence, or a PDS. Furthermore, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, \omega) = (\widehat{\Sigma}, U, V)$. Then the following hold:*

(i) $\Sigma \subseteq \widehat{\Sigma}$ *and* $\widehat{\Sigma}$ *is valid.*

(ii) $\Gamma \vdash_{\widehat{\Sigma}} U : V$ *and* $\Gamma \vdash_{\widehat{\Sigma}} V : \mathsf{type}$.

(iii) $U, V \in \mathcal{NF}(\mathcal{T})$.

Proof: Like in the proofs of Lemma 4.2 and Lemma 4.3, we prove the assertions simultaneously by induction over the construction of $\widehat{\Sigma}$, U, and V. The complete proof is shown in Appendix A.2. ∎

Corollary 4.4 *$\mathcal{E}$ is well defined on PDS nodes, justification sequences and complete PDSs.*

Now, we can state and prove our main theorem:

Theorem 4.1 (Correctness of the Encoding) *Let $\omega \in \Omega$, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, \omega) = (\widehat{\Sigma}, U, V)$. Then the following hold:*

(i) *$\mathcal{E}$ is well defined.*

(ii) *$\Sigma \subseteq \widehat{\Sigma}$ and $\widehat{\Sigma}$ is valid.*

(iii) *$\Gamma \vdash_{\widehat{\Sigma}} U : V$ and $\Gamma \vdash_{\widehat{\Sigma}} V : s$ for some $s \in \mathbb{S}$.*

(iv) *$U, V \in \mathcal{NF}(\mathfrak{T})$*

Proof: (i) follows from Corollary 4.1, Corollary 4.2 and Corollary 4.4. (ii), (iii) and (iv) follow directly from Lemma 4.2, Lemma 4.3 and Lemma 4.5. ∎

4.3.2 Adequacy of the Encoding

Now, we state and prove the adequacy theorem for the encoding of ΩMEGA by splitting it into three parts concerning types, terms and derivations.

Theorem 4.2 (Adequacy of the Encoding for Types) *The encoding $\mathcal{E}$ is a compositional bijection between*

types $\tau \in \mathcal{S}$ with free variables in $\mathcal{V}' \subset \mathcal{V}$

and

terms $T \in \mathcal{NF}(\mathfrak{T})$ with $\Gamma \vdash_\Sigma T : K$, where $\Sigma^{HOL} \subseteq \Sigma$, $\alpha : \mathsf{hol} \in \Gamma$ for every $\alpha \in \mathcal{V}'$ and

$$K = \begin{cases} \mathsf{hol} & \text{if } \tau \text{ a simple type} \\ \mathsf{type} & \text{if } \tau \text{ a schematic type} \end{cases}$$

Proof: Obviously, $\mathcal{E}$ is injective and maps ΩMEGA types into terms in long $\beta\eta$-normal form with the required constraint by Lemma 4.2. Surjectivity is shown by giving the inverse function $\llcorner\cdot\lrcorner$ to $\ulcorner\cdot\urcorner$ such that $\llcorner\ulcorner\tau\urcorner\lrcorner = \tau$ for all $\tau \in \mathcal{S}$. Compositionality is finally proved by showing $\ulcorner\tau[\tau'/\alpha]\urcorner = \ulcorner\tau\urcorner[\ulcorner\tau'\urcorner/\ulcorner\alpha\urcorner]$ for arbitrary $\tau, \tau' \in \mathcal{S}$ and $\alpha \in \mathcal{V}$ by induction over the structure of τ. ∎

Theorem 4.3 (Adequacy of the Encoding for Terms) *The encoding $\mathcal{E}$ is a compositional bijection between*

terms $t \in \mathbf{T}$ with free variables in $\mathbf{V}' \subset \mathbf{V}$, where t has type τ with free variables in $\mathcal{V}' \subset \mathcal{V}$

and

terms $u \in \mathcal{NF}(\mathfrak{T})$ with $\Gamma \vdash_\Sigma u : T$, where $\Sigma^{HOL} \subseteq \Sigma$, $x : \mathsf{obj}\ulcorner\tau'\urcorner \in \Gamma$ for every x of type τ' in $\mathbf{V}'$ and $\alpha : \mathsf{hol} \in \Gamma$ for every $\alpha \in \mathcal{V}'$ and

$$T = \begin{cases} \mathsf{obj}\ulcorner\tau\urcorner & \text{if } \tau \text{ a simple type} \\ \ulcorner\tau\urcorner & \text{if } \tau \text{ a schematic type} \end{cases}$$

Proof: Obviously, $\mathcal{E}$ is injective and maps ΩMEGA terms into terms in long $\beta\eta$-normal form with the required constraint by Lemma 4.3 and Lemma 4.4. Surjectivity is shown by giving the inverse function $\llcorner\cdot\lrcorner$ to $\ulcorner\cdot\urcorner$ such that $\llcorner\ulcorner t\urcorner\lrcorner = t$ for all $t \in \mathbf{T}$. Similarly to the proof of Theorem 3.4, compositionality is shown by induction over the structure of t. ■

Theorem 4.4 (Adequacy of the Encoding for Derivations) *The encoding $\mathcal{E}$ is a compositional bijection between*

> ΩMEGA *objects ω depending on a set of assumption formulae Φ, where ω is a PDS node, a justification sequence or a PDS and has free variables in $F = \mathbf{V}' \cup \mathcal{V}'$ (where $\mathbf{V}' \subset \mathbf{V}$ and $\mathcal{V}' \subset \mathcal{V}$)*

and

> *terms $D \in \mathcal{NF}(\mathcal{T})$ with $\Gamma \vdash_\Sigma D : C$ and $\Gamma \vdash_\Sigma C : \mathsf{type}$, where $\Sigma^{HOL} \subseteq \Sigma$, $h : \mathsf{pf}\ulcorner\tau\urcorner\ulcorner\varphi\urcorner \in \Gamma$ for every assumption $\varphi \in \Phi$ of type τ, $x : \mathsf{obj}\ulcorner\tau'\urcorner \in \Gamma$ for every x of type τ' in $\mathbf{V}'$ and $\alpha : \mathsf{hol} \in \Gamma$ for every $\alpha \in \mathcal{V}'$.*

Proof: Close inspection of the definition of $\mathcal{E}$ shows that it is indeed injective and maps ΩMEGA objects into terms in long $\beta\eta$-normal form with the required constraint by Lemma 4.5 and Corollary 4.3. Surjectivity is shown by giving the inverse function $\llcorner\cdot\lrcorner$ to $\ulcorner\cdot\urcorner$ such that $\llcorner\ulcorner\omega\urcorner\lrcorner = \omega$. Similarly to the proof of Theorem 3.5, compositionality is shown by induction over the structure of ω. ■

Having defined TWEGA as the representation of proofs in *P.rex*, we shall now turn our attention to the process of planning the explanation of a proof. In the following chapter, we shall review ACT-R, a theory of human cognition that combines the abilities for user modeling and planning in a uniform framework and is therefore particularly well suited as a basis for a user-adaptive dialog planner.

Chapter 5

The Cognitive Architecture ACT-R

Unification of theories is one of the ultimate goals of any science, or, as Allen Newell puts it, it is "an apple pie of science" [Newell, 1990, p. 17]. He suggests that cognitive scientists should strive to attain *unified theories of cognition*, that is, theories that gain their power by formulating a single structure of mechanisms that operate together to produce the full range of human cognition. Unified theories of cognition try to conceptualize mental structures and processes of the cognitive apparatus and reflect the constraints on the structure of the cognitive apparatus. For Newell, it is most important that—like the mind—a single system produces all aspects of behavior.

Although it is almost certainly too early in the history of cognitive science to already formulate such a theory, it is nevertheless a worthwhile goal to strive for one. By describing unified theories of cognition as theories of the *cognitive architecture*, that is, the fixed structure that realizes the cognitive apparatus, we commit ourselves to a set of basic structures and primitive operators. The combination and cooperation of only these primitive operators manipulating the basic structures effects complex cognitive behavior.

In the literature, several cognitive architectures have been described, the most important ones being ACT-R [Anderson and Lebiere, 1998], Soar [Newell, 1990], EPIC [Meyer and Kieras, 1997a] and 3CAPS [Carpenter and Just, 1995]. In this chapter, we shall review ACT-R, a theory of the human cognitive architecture, which is implemented as a production system. Anderson and Lebiere [1998] described ACT-R 4.0, the latest in a series of formalizations of ACT-R. Throughout the thesis, we mean ACT-R 4.0 whenever we speak of ACT-R. First, in Section 5.1, we shall review production systems as the basis to implement cognitive architectures. Next, in Section 5.2, we shall give an overview of ACT-R. Then, Section 5.3 and Section 5.4 are devoted to describe the declarative and procedural memory, respectively. In Section 5.5, finally, we shall describe which features of ACT-R are used in *P.rex* and which are neglected.

5.1 Production Systems

The implementation of most cognitive architectures is based on *production systems*. A production system consists of the following four components (see, e.g., [Rich and Knight, 1991; Russell and Norvig, 1995]):

1. A *data base*, which contains any information that is needed for the particular

task. Some parts of the data base may be permanent whereas other parts may pertain only to the solution of the current problem.

2. A *rule base*, which contains the *production rules* (or short: *productions*). Each production consists of a *condition*, which determines the applicability of the rules, and an *action*, which describes the operation to be performed if the rule is applied.

3. A *control strategy*, which specifies the order, in which the productions will be applied to the knowledge base, and a way to resolve conflicts, which arise when several productions are applicable.

4. A *rule applier*, which actually applies the productions.

Production systems operate in a cycle, which can be divided into three phases: First, in the *match phase*, the system computes the set of applicable productions by comparing the condition of each rule with the data base. Second, in the *conflict resolution phase*, the system decides which of the applicable productions should be executed. Third, in the final step of each cycle, the *act phase*, the chosen production is actually executed, the data base is updated accordingly and the cycle starts anew.

5.2 Overview of ACT-R

ACT-R [Anderson and Lebiere, 1998] is a theory of the nature of human knowledge, a theory of how this knowledge is deployed, and a theory of how this knowledge is acquired. In psychology, it is widely accepted that knowledge in general can be divided into declarative and procedural knowledge. *Declarative knowledge* corresponds to the things that we are aware we know and can usually describe to others, such as "three plus four is seven." *Procedural knowledge* is knowledge that we display in our behavior but that we are not conscious of.[1] ACT-R adopts this distinction and provides two types of knowledge bases or *memories* to store permanent knowledge: the representations of declarative and procedural knowledge are explicitly separated into the *declarative memory* and the *procedural memory*. To properly model human cognition, it is necessary to represent the current purpose and organize behavior in response to that purpose. To this end, ACT-R employs a third memory, the *goal stack*, which is a transitory memory that encodes the hierarchy of intentions, which guide the behavior.

Figure 5.1 depicts the architecture of ACT-R. The three memories—goal stack, declarative memory and procedural memory—are organized through the goal on top of the goal stack, the *current goal*, which represents the focus of attention.

Procedural knowledge is represented in productions, whose conditions and actions are defined in terms of declarative structures. A production can only apply if its conditions are satisfied by the current goal and the knowledge currently available

[1] Note that this definition of declarative and procedural knowledge in psychology considerably differs from the definition of declarative and procedural knowledge in computer science and artificial intelligence. In psychology, declarative knowledge is knowledge that we can express and procedural knowledge is knowledge about actions that we can perform but cannot express (i.e., we cannot describe how we perform it). In computer science and artificial intelligence, declarative knowledge is any knowledge that is viewed as *data* to a program, whereas procedural knowledge is any knowledge that is viewed as a *program*. However, both declarative and procedural knowledge can be *represented* either declaratively (i.e., the knowledge is specified, but it is not specified how this knowledge is used) or procedurally (i.e., the knowledge contains information about how it is used). Note the distinction between the knowledge *per se* and its *representation*. For example, in logic programming languages such as Prolog, logical assertions, that is, declarative knowledge, is procedurally represented. (Cf., e.g., [Rich and Knight, 1991] for a detailed discussion.) In this thesis, whenever we speak of declarative and procedural knowledge, we use the terms in the psychologists' connotation.

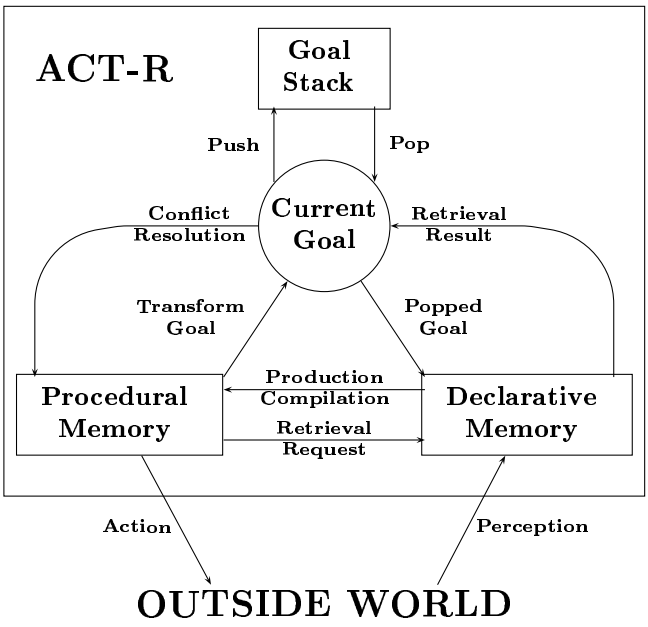


Figure 5.1. Flow of information among the various modules of ACT-R, taken from [Anderson and Lebiere, 1998].

in the declarative memory. The application of a production usually modifies the goal stack by transforming the top goal, by popping it or by pushing new goals. It also may result in an observable event or it may pose a retrieval request from the declarative memory. The retrieval result can be returned to the current goal. The set of productions that match the current goal is called the *conflict set*. A *conflict resolution* heuristic derived from a rational analysis of human cognition determines which production in the conflict set will eventually be applied (cf. [Anderson, 1990] for the details of the rational analysis). New declarative knowledge is acquired as popped goals or as perceptions from the environment. New procedural knowledge can be acquired from declarative knowledge by a process called *production compilation*.

ACT-R can be described as a symbolic system, in which discrete items of the declarative memory interact with productions in discrete cycles. ACT-R also has a subsymbolic level, in which continuously varying quantities are processed to produce much of the qualitative structure of human cognition. The subsymbolic level is the basis, on which the conflict resolution process operates.

The following two sections describe the declarative and procedural knowledge, respectively, both at the symbolic and the subsymbolic level.

5.3 Declarative Knowledge

Declarative knowledge is represented in terms of *chunks* in the declarative memory. Chunks encode small independent patterns of information as sets of slots with associated values. In the implementation of ACT-R 4.0, they are realized as declarative data structures. The following is an example for a chunk C_1 = `factFsubsetG` encoding the fact that $F \subseteq G$, where `subset-fact` is a concept and C_2 = `F` and C_3 = `G` are contextual chunks associated to C_1:

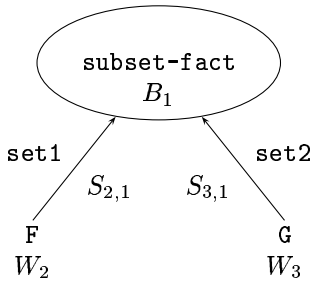


Figure 5.2. Network representation of the chunk `factFsubsetG` (adapted from [Anderson and Lebiere, 1998]).

```
factFsubsetG
  isa    subset-fact
  set1   F
  set2   G
```

Figure 5.2 depicts the same chunk graphically. In addition to the data structure of the chunk, the figure also displays the subsymbolic parameters that are associated with the data structure. We shall elaborate on these parameters, which are summarized in Table 5.1 on page 77, in the following two sections.

5.3.1 Activation of Chunks

Chunks are annotated with continuous activations that influence their retrieval. The activation A_i of a chunk C_i consists of its base-level activation and the weighted activations of contextual chunks:

$$A_i = B_i + \sum_j W_j S_{j,i} \tag{5.1}$$

where B_i is the base-level activation of C_i, W_j is the source activation of a contextual chunk C_j, and $S_{j,i}$ is the strength of the association of C_i with C_j. We shall not go into detail of how the components of A_i are formalized, but only sketch their definitions briefly (cf. [Anderson and Lebiere, 1998] for a thorough presentation).

The base-level activation B_i represents how recently and frequently the chunk C_i has been accessed. B_i is defined such that it grows with repeated use of C_i and decreases logarithmically when C_i is not used. Thus, ACT-R models the fact that used declarative knowledge becomes better accessible and that unused declarative knowledge fades away over time. By adding a noise term, ACT-R introduces a stochastic element into the calculation of the base-level activation.

The source activations W_j reflect the amount of attention given to the elements of C_i. ACT-R assumes that there is a fixed capacity for source activation and that each element has an equal amount, that is, if the capacity is assumed to be 1 and if there are n elements, then $W_j = 1/n$ for each element.

The strength $S_{j,i}$ of the association of C_i with C_j is a measure of how often C_i was needed when C_j was an element of the goal. It can be thought of an estimation of the likelihood of C_j being a source present if C_i is retrieved. $S_{j,i}$ is defined as the logarithmic deviation of the probability of C_i in the context of C_j from its base probability.

Note that the definition of the activation establishes a spreading activation to adjacent chunks, but not further. That is, multi-link spread is not supported.

5.3.2 Chunk Retrieval

The only way declarative knowledge can be accessed is by productions retrieving the corresponding chunks. The constraint on the capacity of the human working memory is approached by defining a retrieval threshold τ, where only those chunks C_i can be matched whose activation A_i is higher than τ. Chunks with an activation less than τ are considered as forgotten.

Hence, failure of matching a chunk that falls below the retrieval threshold result in errors of omission. Errors of commission, in contrast, are produced when a chunk is retrieved that only partially matches the retrieving production. Intuitively speaking, ACT-R prefers a high activation in a partial match to a low activation in a perfect match. The matching between a production P_p and a chunk C_i is modeled by the match score $M_{i,p}$:

$$M_{i,p} = A_i - D_{i,p} \tag{5.2}$$

where A_i is the activation of C_i as defined in Equation 5.1. $D_{i,p}$ is the degree of mismatch between C_i and P_p, that is, it reflects the number of slots that fail to match. Again, ACT-R adds some stochasticity by including approximately normally distributed noise in the match score.

The time $t_{i,p}$ a production P_p needs to retrieve a chunk C_i is defined as:[2]

$$t_{i,p} = Fe^{-f(M_{i,p}+S_p)} \tag{5.3}$$

where S_p is the strength of the production (cf. Section 5.4) and F and f are scaling factors, on which we shall not elaborate. Hence, the higher the activation of the chunk and the strength of the production, the faster the production matches the chunk. Since the activation must be greater than the retrieval threshold τ, τ constrains the time maximally available to match a production with a chunk. Therefore, the time spent for a failing retrieval is:

$$t_{i,p}^{\max} = Fe^{-f(\tau+S_p)} \tag{5.4}$$

5.3.3 Learning Chunks

New chunks are learned when they enter the declarative memory. There are two sources for chunks: They can be either encoded directly from the environment as a perception, or created during a production's firing. According to the theory of ACT-R, only goal chunks can be created. They enter the declarative memory when they are popped from the goal stack. When an already existing chunk is recreated as a goal chunk ACT-R avoids its duplication in the declarative memory. When the goal is popped it is merged with the existing chunk by combining their subsymbolic parameters.

5.4 Procedural Knowledge

The operational knowledge of ACT-R is formalized in terms of *productions*. Intuitively, productions are supposed to correspond to steps of the cognitive process. As mentioned earlier, a production consists of a condition and an action. In ACT-R, the condition in turn consists of a *goal condition* and some *chunk retrieval*. In the goal condition, some tests on the goal state are made. Only if these tests are successful the chunk retrieval is performed, where chunk patterns are matched to the declarative memory to retrieve information. If the retrieval is successful the production can fire and the action is performed. The actions that can be initiated

[2] In this context, *time* does not mean the CPU time needed to calculate the match, but the time a human would need for the match according to the cognitive model.

by ACT-R productions are goal transformations: the state of the current goal can be changed or new subgoals can be created and pushed onto the goal stack. These subgoals are then fulfilled by firing more productions or by performing some motor program. A *subgoal return* mechanism allows for passing results from a subgoal to the supergoal. In the implementation of ACT-R 4.0, productions are represented procedurally: when a production is defined its action side is immediately compiled into a Lisp function.

An example for a production is the following:

IF it is known that $x \in S_1$ and $S_1 \subseteq S_2$
THEN conclude that $x \in S_2$.

Similarly to the base-level activation of chunks, the strength S_p of a production P_p is defined such that it grows with repeated use of the production and decreases logarithmically when the production is not used.

Note that we give productions only paraphrased in English but not in ACT-R syntax. We shall give the concrete syntax of productions in Appendix C.2.1.

5.4.1 Conflict Resolution

If the goal test does not filter out a single production a *conflict resolution* process is initiated that chooses a production. When a single production has been chosen usually several ways of retrieving chunks (each way is called an *instantiation*) are possible and the conflict resolution process has to choose a single instantiation. Both choices are determined by subsymbolic processes.

Productions that satisfy the current goal are placed into the *conflict set*, that is, the only test is if the goal matches the production's goal condition. Chunk retrievals required for the condition are only attempted if a production has been selected from the conflict set.

The productions in the conflict set are ordered in terms of their net utility E:

$$E = PG - C \tag{5.5}$$

where P is the expected probability that the goal will be achieved by the production's firing, G is the value of the goal and C is the expected cost of achieving the goal. As in the case of the chunk retrieval, stochastic behavior is achieved by including approximately normally distributed noise in the net utility.

The scale for measuring cost in ACT-R is time. Hence, C can be seen as an estimation of the time needed to achieve the goal. G is the amount of time ACT-R should be willing to spend on the goal. Hence, the term PG corresponds to the expected gain and the term C corresponds to the expected cost.

Intuitively, Equation 5.5 can be read as a rule for trading off the probable cost against the value of the goal.

Both probability and cost consist of subterms that reflect quantities associated with the current production and quantities associated with subsequently applied rules.

The probability P is defined as:

$$P = qr \tag{5.6}$$

where q is the probability of the production working successfully and r is the probability of achieving the goal if the production works successfully. Both q and r are estimated by relating the number of successful events (i.e., production firing for q and achieving the goal for r) to the overall number of events (cf. [Anderson and Lebiere, 1998] for details).

The cost C are defined as:

$$C = a + b \tag{5.7}$$

where a is the amount of time that the production will take and b is an estimate of the amount of time from when the production completes until the goal is achieved. Both a and b are estimated by relating the effort spent when applying the production to the overall number of applications of the production. For a, effort means the retrieval time and action time needed by the production. For b, effort means the cost for subsequent production cycles until the goal succeeds or fails. (Again, cf. [Anderson and Lebiere, 1998] for details.)

Note that the parameters q and a refer to the production and any subgoal it might set.

Even if a subgoal is achieved there remains uncertainty about whether the supergoal will be achieved and there remains some potential cost, and this has to be taken into consideration. The value G' of a subgoal is defined as:

$$G' = rG - b \tag{5.8}$$

Hence, the value assigned to a subgoal depends on the context in which it occurs. ACT-R will value a goal less the more deeply it is embedded into uncertain subgoals. That is, ACT-R will more likely abandon deeply embedded subgoals.

Separating probability parameters (i.e., q and r) from cost parameters (i.e., a and b) allows the system to be differentially sensitive to probability and cost as a function of the value of a goal. Separating quantities associated with the current production (i.e., q and a) from the quantities associated with future rules (i.e., r and b) allows ACT-R to appropriately discount the value of a subgoal.

When a production is selected, there is no guarantee that its non-goal chunks can be successfully matched. If the matching fails the next production in the conflict set with the highest utility E is chosen. If no production is found the current goal is popped with failure. When a goal is popped with failure it can return a failure value to a slot of the supergoal. This allows higher goal levels to detect and deal with subgoal failure.

To sum up, in ACT-R the choice of a production is determined as follows:

1. The conflict set is determined by testing the match of the productions with the current goal.

2. If the conflict set is empty the goal is popped with failure and this production cycle is concluded. Otherwise, the production P_p with the highest utility is chosen.

3. The actual instantiation of P_p is determined via the activations of the corresponding chunks. If no instantiation is possible (because of τ), P_p is removed from the conflict set and the algorithm resumes in step 2, otherwise the instantiation of P_p is applied and this production cycle is concluded.

The equations and parameters of the conflict resolution mechanism are summarized in Table 5.1 on page 77.

5.4.2 Production Compilation

The theory of ACT-R models the learning of procedural knowledge by the processing of chunks of a certain type, called *dependency*. A dependency goal reflects the intention to understand a problem solving step. When such a goal is fulfilled, and hence popped from the goal stack, a production that performs the corresponding problem solving step is automatically compiled. Hence, this learning mechanism is called *production compilation*.

To encode goal condition, chunk retrieval and action of the production to be compiled, the following slots are defined for dependency chunks:

- The `goal` slot describes what the goal was before the problem solving step, that is, the goal condition of the production to be compiled.
- The `constraint` slot reflects the chunk retrieval patterns.
- The `modified` slot encodes what the changed goal looks like, that is, the goal transformations of the production to be compiled.
- The `stack` slot describes any changes of the goal stack such as popping the goal or pushing a subgoal.

When a production is compiled patterns that occur only once are considered as specific patterns and are, hence, treated as constants. Patterns that occur twice or several times, in contrast, are considered as general patterns and are treated as variables. If this default variabilization is not appropriate the following slots of dependency chunks can be used to tailor the variabilization to the case in point:

- The `generals` slot contains the patterns considered as general patterns. They are treated as variables.
- The `specifics` slot contains the patterns considered as specific patterns. They are treated as constants.
- The `don't-cares` slot contains any patterns that are to be omitted in the compiled production.
- The `differents` slot describes which patterns should not occur for the compiled production to be applicable.

To elucidate the production compilation process, consider the following example:

Example 5.1
Assume our aim is to calculate the largest set of which a given element is a member. Furthermore, assume that $a \in F$, $a \in G$ and $F \subset G$ have been derived earlier, that is, let the following chunks be in the declarative memory:

```
factAelementF                    factFsubsetG
  isa       element-fact           isa    subset-fact
  element   a                      set1   F
  set       F                      set2   G
factAelementG
  isa       element-fact
  element   a
  set       G
```

The problem solving step that achieves `factAelementG` from `factAelementF` and `factFsubsetG` can be described by the following dependency chunk (which we assume is the current goal):

```
Dependency
  isa           dependency
  goal          factAelementF
  modified      factAelementG
  constraints   factFsubsetG
```

Then, when the current goal is popped the following production is compiled:

```
IF      =goal>
          isa       element-fact
          element   x
          set       S1
        =subset>
          isa       subset-fact
          set1      S1
          set2      S2
THEN    =goal>
          set   S2
```

where x, S_1 and S_2 are variables.

Note that this production changes the goal chunk by changing its `set` slot. The production can be paraphrased as:

IF it is known that $x \in S_1$ and $S_1 \subset S_2$
THEN conclude that $x \in S_2$.

Clearly, repeated application of this production rule calculates the largest set a given element is a member of. □

5.5 The Use of ACT-R in *P.rex*

Since ACT-R combines the abilities for user modeling and planning in a uniform framework, it is well suited to serve as a basis for a proof explanation system such as *P.rex* that has to have a model of the user (in particular of his mathematical knowledge and skills) in order to plan an appropriate presentation. However, ACT-R models human cognition in much more detail than necessary for our purposes. Therefore, we can neglect some of its features, as we shall discuss in this section.

When we base our proof explanation system *P.rex* on ACT-R, we model an idealized teacher who explains mathematical proofs to his student. We assume that the teacher has a perfect understanding of the mathematical theories he explains and, in particular, of the way to explain proofs of theorems in these theories.

Explaining proofs to a student has to take his knowledge and skills into account. Therefore, we have to model which mathematical facts and methods the user knows and which he does not know. We do this by making use of the learning of chunks C_i, the learning of activations A_i of chunks and the dependency of the chunk retrieval on the retrieval threshold τ.

In contrast to a tutorial system, teaching mathematical skills is beyond the scope of *P.rex*. Hence, we do not let the user of the system solve problems or exercises and, hence, we need neither analyze his solutions and nor model the mistakes he might make. Therefore, we dispense with partial matching, that is, we can always assume that $D_{i,p} = 0$ in Equation 5.2.

When we assume an ideal teacher, we assume in particular that he has perfectly adapted explanation skills. Since we model his explanation skills by giving appropriate production rules, assuming perfectly adapted skills means that the choice of productions for application is perfect and need not adapt. That is, the chosen productions always succeed in fulfilling the ultimate goal. Therefore, we switch off the learning of production parameters (i.e., q, r, a and b) but fix the parameters at their initially given values. Since the main reason for adding noise to ACT-R is to allow for adaptation to a changing environment, we do not use any noise either. Nevertheless, we draw on learning new explanation skills by production compilation.

The theory of ACT-R sometimes includes restrictions that are not enforced by the implementation of ACT-R 4.0. For example, according to the theory, lists of

chunks are not allowed as values of chunks (except for dependency chunks), productions should not include more than a few chunks and only goal chunks can be created by productions. It is always possible to meet these restrictions by defining an extra chunk slot for chaining chunks, by splitting a production into several cooperative productions and by pushing newly created chunks onto the goal stack and have them popped immediately by additional productions, respectively. This does not result in qualitatively different behavior, but the predictions of the cognitive model with respect to time and number of production cycles needed to produce that behavior are no longer valid.

Thus, if only qualitative behavior is important and quantitative behavior can be ignored, as is the case in *P.rex*, it makes sense to deviate from the previously mentioned restrictions in favor of attaining a more efficient implementation of the cognitive model.

In the following chapter, we shall give a thorough presentation of *P.rex*'s dialog planner, which is implemented in ACT-R.

Table 5.1. Equations and parameters in ACT-R.

Chunk Retrieval

$A_i = B_i + \sum_j W_j S_{j,i}$ Equation (5.1)

- A_i activation of chunk C_i
- B_i base-level activation of chunk C_i
- W_j activation of source chunk C_j
- $S_{j,i}$ strength of the association of chunk C_i with chunk C_j

$M_{i,p} = A_i - D_{i,p}$ Equation (5.2)

- $M_{i,p}$ match score of production P_p with chunk C_i
- $D_{i,p}$ degree of mismatch between production P_p and chunk C_i

$t_{i,p} = Fe^{-f(M_{i,p}+S_p)}$ Equation (5.3)

- $t_{i,p}$ time production P_p needs to retrieve chunk C_i
- S_p strength of production P_p
- F, f scaling factors

$t_{i,p}^{\max} = Fe^{-f(\tau+S_p)}$ Equation (5.4)

- $t_{i,p}^{\max}$ time maximally available for production P_p to retrieve chunk C_i
- τ retrieval threshold: only a chunk C_i with $A_i > \tau$ can be matched

Conflict Resolution

$E = PG - C$ Equation (5.5)

- E net utility of a production
- P expected probability that the goal will be achieved by the production's firing
- G value of the goal
- C expected cost of achieving the goal

$P = qr$ Equation (5.6)

- q probability of the production working successfully
- r probability of achieving the goal if the production works successfully

$C = a + b$ Equation (5.7)

- a amount of time the production takes
- b estimate of the amount of time from when the production completes until the goal is achieved

$G' = rG - b$ Equation (5.8)

- G' value of a subgoal

Chapter 6

The Dialog Planner

An important task that any natural language generation system has to fulfill is determining the content of the utterances to be produced. To comply with the needs of interactive explanation, pure content determination must be augmented by dialog management facilities to allow also for user turns in the dialog. To account for such dialogs, we designed an accordingly enhanced content determination component, the *dialog planner*. It plays a central role in *P.rex*.

In Section 6.1, we shall briefly review approaches to content determination and give an overview of our dialog planner. Section 6.2 is devoted to the representation of discourses in *P.rex*. Next, in Section 6.3, we shall introduce the basics of content determination in *P.rex*. Then, in Section 6.4, we shall enrich the content determination by user-adaptive and context-sensitive planning decisions. In Section 6.5, we shall describe how the dialog planner allows for user interaction. We shall conclude this chapter with a final discussion.

6.1 Approaches to Planning Discourses

A *discourse* is generally seen as a structured collection of utterances such as texts or dialogs. For a discourse to be successful, it is crucial that the hearer knows the communicative role of each portion of it. If the hearer knows how the speaker intends each clause to relate to each other clause the discourse is *coherent* [Hovy, 1993]. Since a system generating natural language will hardly be accepted by the users unless it produces coherent discourse, the content planner has to take into account some theory of discourse.

In Section 6.1.1, we shall review two approaches to discourse that describe discourses from different perspectives. Subsequently, we shall examine several approaches to the implementation of discourse theories. The Sections 6.1.2, 6.1.3 and 6.1.4 are devoted to static, dynamic and hybrid approaches to planning of discourses, respectively. Then, in Section 6.1.5, we shall review the macro-planner of *PROVERB*, the most sophisticated proof presentation system to date that uses natural language generation techniques. It can be considered as a predecessor to *P.rex*. In Section 6.1.6, finally, we shall give an overview of the dialog planner of *P.rex*.

6.1.1 Discourse Theories

In the field of natural language processing, two competing approaches, which Hovy [1993] calls formalist and functionalist approaches, describe discourses from different perspectives. First, according to *formalist theories*, the discourse exhibits

an internal structure, where structural segments capsulate semantic units that are closely related. Second, in *functionalist theories*, the discourse exhibits an internal structure, where the segments are defined by their communicative purpose. In the following, we shall review two major theories that influenced our work.

In formalist theories, the discourse is seen as a collection of segments, which are defined in terms of structure (e.g., [Kamp, 1981; Reichman, 1985]). Concerned with natural language analysis, Grosz and Sidner [1986] developed a discourse theory that distinguishes three separate, but interrelated components, namely the linguistic structure, the intentional structure and the attentional state.

The *linguistic structure* describes the segmentation of the discourse. It consists of *discourse segments* of various sizes with utterances as basic elements and an embedding relation between discourse segments. Cue words and phrases indicate segment boundaries. These boundary markers are classified according to whether they indicate changes in the intentional structure or in the attentional state.

Each discourse segment has a purpose, called *discourse segment purpose*. The *intentional structure* captures how the purposes of the discourse segments relate to one another. As Grosz and Sidner emphasized, the range of intentions that can serve as discourse segment purposes is open-ended. Since the participants in a conversation therefore can never know the whole set of intentions, they must recognize the relevant structural relationships between intentions. Although the number of intentions is potentially infinite, there are only a small number of relations relevant to discourse structure that can hold between them. It is this small set of relations that defines whether a discourse is coherent or not. The original theory was formulated with only two relations. First, a dominance relation describes that a discourse segment purpose P_1 dominates another discourse segment purpose P_2, or, vice versa, that the satisfaction of P_2 contributes to the satisfaction of P_1. Second, a satisfaction precedence relation defines a partial order on discourse segment purposes by specifying which discourse segment purposes must be satisfied before which other ones.

The *attentional state*, finally, is an abstraction of the focus of attention of the participants as the discourse unfolds. Without including the intentional structure as a whole, the attentional state records objects, properties and relations that are salient at each point of the discourse.

The major feature of Grosz and Sidner's theory is that it approaches the discourse in terms of its structure. The functions of the individual segments play only a minor role.

Functionalist theories advocate the combination of linguistic structure and intentional structure and define the segments by their communicative purposes. For natural language analysis, Mann and Thompson [1987] formulated a discourse theory called *Rhetorical Structure Theory (RST)*. The theory states that the relations that hold between segments of normal English text can be represented by a finite set of relations. The relations are used recursively, connecting segments of various sizes down to single clauses. A paragraph is considered to be coherent only if all parts fit into a single overarching relation. Most relations are associated with characteristic cue words or phrases, which inform the hearer how to relate the segments. Thus, the role of each part of the discourse can be determined with respect to the whole.

The major feature of RST is that it describes the discourse in terms of function without elaborating on its structure. Moreover, no notion of attentional states of the interlocutors is covered by the theory.

Similarly to Mann and Thompson, Hovy [1993] argues in favor of a single discourse tree. He considers a discourse as a structured collection of clauses, which are grouped into segments by their semantic relatedness. The discourse structure is expressed by the nesting of segments within each other according to specific relationships (i.e., RST relations). Hence, a discourse can be represented by a *discourse*

structure tree, in which each node governs the segment beneath it. At the top level, the discourse is governed by a single root node. At the leaves, the basic segments are single grammatical clauses. The discourse segment purposes are the communicative goals of the speaker and are represented at each node of the discourse structure tree. As Hovy points out, such a discourse structure tree can be used for the generation of natural language as well.

Comparing these discourse theories, we can conclude the following: As a formalist theory, Grosz and Sidner's theory is a theory of the structure of discourse. It describes the structural relationships among discourse segments, that is, the embedding of the segments and their order. In the intentional structure, the theory captures only a weak notion of the functional relationships among the discourse segments.

Mann and Thompson's RST, in contrast, is a functional theory, that is, it makes strong claims about the functional relationships between the discourse segments. It also describes the embedding of discourse segments, but does not determine their order. The same is true for Hovy's theory, which is closely related to RST.

Note that the complementary characteristics of formalist and functional theories manifest themselves in the use of the theories in practical systems. Whereas formalist theories are widely used as the basis for systems for natural language analysis, functional theories are implemented in most natural language generations systems.

In the following, we shall examine several approaches to the implementation of discourse theories in NLG systems.

6.1.2 Planning with Schemata

Early systems concerned with multi-sentence text simply ignored the issue of discourse structure. One of the first systems to take into account discourse structure was Text [McKeown, 1985], a system producing descriptions of complex objects. It used predefined *schemata* as templates that organize the content and order of clauses in a paragraph. An example for a schema taken from [McKeown, 1985] is depicted in Figure 6.1.

Identification

(1) Identification (class & attribute/function)
(2) {Analogy/Constituency/Attributive/Renaming/Amplification}*
(3) Particular-illustration/Evidence+
(4) {Amplification/Analogy/Attributive}
(5) {Particular-illustration/Evidence}

Example:
Eltville (Germany) (1) An important village of the Rheingau region. (2) The vineyards make wines that are emphatically of the Rheingau style, (3) with a considerable weight for a white wine. (4) Taubenberg, Sonnenberg and Langenstuck are among vineyards of note.

Figure 6.1. The IDENTIFICATION schema in Text taken from [McKeown, 1985]. Note that "{·}" and "/" mean optionality and alternative, respectively. Moreover, "+" indicates that an object can appear one or more times, and "*" indicates that the item is optional and may appear zero to n times for some n.

The schemata can be employed recursively to progressively build up the whole text. They are particularly well suited for the generation of stereotypical portions of a discourse. Their essential shortcoming lies in the lack of the representation of the purpose of each part in the schema with respect to whole. Because of this

deficiency, the system can neither replan any portion of the discourse in case that a portion should not communicate successfully, nor motivate why it said what it said. That is, schemata cannot be used flexibly; they represent *static* chunks of text.

A *dynamic* approach developed to overcome this problem is reviewed in the following section.

6.1.3 Planning with Discourse Relations

To overcome the main drawback of schemata, namely that the purpose of segments and the relations between segments in the schema are not represented, *dynamic* approaches to assemble coherent discourse were developed. *Rhetorical Structure Theory (RST)* [Mann and Thompson, 1987] became an influential theory for natural language generation and has been used as the basis for assorted systems (e.g., [Hovy, 1991; Reithinger, 1991]). As described in Section 6.1.1, the theory decomposes texts into ever smaller segments, which are connected with one another via RST relations.

Most RST relations contain two parts. Whereas the *Nucleus* contains the major material, the *Satellite* contains ancillary material that supports the Nucleus. The Satellite is incomprehensible without a Nucleus, but a Satellite can be replaced by a completely different one to better support the Nucleus. Most relations have characteristic cue words or phrases, which inform the hearer about how to relate the segments, such that the role played by each clause can be determined with respect to the whole. An example relation is given in Figure 6.2.

relation name:	EVIDENCE
constraints on N:	R might not believe N to a degree satisfactory to W
constraints on S:	The reader believes S or will find it credible.
constraints on the N + S combination:	
	R's comprehending S increases R's belief of N
the effect:	R's belief of N is increased
locus of the effect:	N

Figure 6.2. The definition of the relation EVIDENCE, taken from [Mann and Thompson, 1987]. N stands for the Nucleus, S for the Satellite. W and R denote the writer and reader, respectively.

An example for a dynamic planner is Hovy's paragraph structure planner, which was applied to several domains such as a multi-modal database information display system [Hovy, 1991]. In this planner, RST relations play a twofold role. First, they are used as inner nodes of the paragraph structure tree, whereas the leaves of the paragraph structure tree are input units. Second, RST relations are formulated as plan operators (cf. Figure 6.3 for an example) where Nucleus and Satellite requirements, depending on the hearer's knowledge, are treated as semantic preconditions on the material to be conveyed. Possible paths of expansion of the Nucleus or Satellites are given in *growth points* of subgoals, that is, plan operators that are allowed by coherence to apply to the Nucleus or Satellites. The plan operators recursively relate some units of the input or another relation (cast as Nucleus) to other units of the input or another relation (cast as Satellite). By this top-down refinement, which serves both content selection and organization, the planner constructs a paragraph structure tree. Since the tree captures the internal organization and rhetorical dependencies between clauses in the text, it allows for powerful reasoning about the text.

Note that the plan provided by the paragraph structurer simultaneously serves as a discourse structure and as a plan for achieving the desired communicative goal.

```
Name: SEQUENCE

Results:
  ((BMB SPEAKER HEARER (SEQUENCE-OF ?PART ?NEXT)))

Nucleus requirements/subgoals:
  ((BMB SPEAKER HEARER (TOPIC ?PART)))

Satellite requirements/subgoals:
  ((BMB SPEAKER HEARER (TOPIC ?NEXT)))

Nucleus+Satellite requirements/subgoals:
  ((NEXT-ACTION ?PART ?NEXT))

Nucleus growth points:
  ((BMB SPEAKER HEARER (CIRCUMSTANCE-OF ?PART ?CIR))
   (BMB SPEAKER HEARER (ATTRIBUTE-OF ?PART ?VAL))
   (BMB SPEAKER HEARER (PURPOSE-OF ?PART ?PURP)))

Satellite growth points:
  ((BMB SPEAKER HEARER (ATTRIBUTE-OF ?NEXT ?VAL))
   (BMB SPEAKER HEARER (DETAILS-OF ?NEXT ?DETS))
   (BMB SPEAKER HEARER (SEQUENCE-OF ?NEXT ?FOLL)))

Order: (NUCLEUS SATELLITE)
Relation-phrases: ("" "then" "next")
Activation-question:
  "Could ~A be presented as start-point, mid-point, or end-point
  of some succession of items along some dimension? -- that is,
  should the hearer know that ~A is part of a sequence?"
```

The contents of the RST relation/plan Sequence can be paraphrased as follows: the plan, when used successfully, guarantees that both speaker and hearer will mutually believe that the relationship `SEQUENCE-OF` holds between two input entities (that is to say, that one entity follows another in temporal, ordinal, or spatial sequence). That is the contents of the Results field. To ensure proper ordering and focus, one input entity is bound to the variable `?PART` in the Nucleus requirements field and the other to the variable `?NEXT` in the Satellite requirements field. No other semantic requirements hold on the input entities individually. There is, however, the requirement that they be semantically related by some kind of sequential link (in the current domain, the temporal relation `NEXT-ACTION`), as stated in the Nucleus+Satellite requirements field; that is, that `?PART` does in fact precede `?NEXT`. Suggestions for including additional input material related to the nucleus are contained in the Nucleus growth points field: these call for circumstantially related material (time, location, etc.), attributes (size, color, etc.) and purpose. They are stated in terms of mutual beliefs in order to act as subgoals that the planner must try to achieve. A similar set is associated with the Satellite. The typical order of expression in the text is Nucleus first and the Satellite, using either no cue word, "then", or "next".

Figure 6.3. The RST plan operator Sequence taken from [Hovy, 1993]. The term `(BMB x y P)` stands for *P follows from x's beliefs about what x and y mutually believe.*

That is, the discourse structure is simultaneously a linguistic construct and a plan of action.

The major disadvantage of the dynamic planning approach is that it is not feasible in practice to plan longer texts without some explicit representation of the structure of spaces of text that are longer than single paragraphs. Since schemata can easily represent longer spaces of text, hybrid approaches lead the way to overcome this problem.

6.1.4 Hybrid Planning

Although schemata do not explicitly represent the purposes of discourse segments and hence forgo the ability to reason about the discourse, they are useful to include stereotypical discourse elements and to plan longer text plans. Discourse structure operators such as RST operators, in contrast, enable powerful reasoning about the discourse by producing explicit discourse structure trees. However, these operators do not produce longer texts satisfactorily.

An appropriate way to combine the complementary strengths of both approaches without committing to their weaknesses is to consider schemata as fossilized subtrees of the discourse structure that represent formulaic texts. Whereas the application of discourse structure operators includes only single nodes in the discourse structure tree, the application of a schema includes the corresponding subtree. Since such a subtree includes information about purpose and discourse relation, it allows us to reason about the function and interrelation of each portion of the discourse. Vice versa, the effect of a discourse structure operators can be seen as that of a mini-schema. Hence, we do not only obtain the strength of both approaches, we also gain homogenity in the representation.

6.1.5 *PROVERB*: Hierarchical Planning and Local Navigation

The proof presentation system *PROVERB* [Huang, 1994a; Huang and Fiedler, 1997] can be considered as the first serious attempt at a comprehensive computational model that produces adequate natural language text from machine-found proofs. Since it can be seen as the predecessor of *P.rex*, we shall examine *PROVERB*'s macro-planner in some detail in this section.

The macro-planner of *PROVERB* accepts as input a natural deduction style proof, and produces speech acts, called *proof communicative acts (PCAs)*, which are structured into hierarchical attentional spaces. To do so, it uses a strategy that combines *hierarchical planning* and *local navigation*.

The previous sections acquainted us with various approaches to planning the structure of a discourse. However, there is psychological evidence that language has an unplanned, spontaneous aspect as well [Ochs, 1979]. Based on this observation, Sibun [1990] implemented a system for generating descriptions of objects with a strong domain structure, such as houses, ships and families. While a hierarchical planner recursively breaks generation tasks into subtasks, local organization navigates the domain object following the local focus of attention.

PROVERB combines both of these approaches within a uniform planning framework [Huang, 1994b]. *Top-down* operators for *hierarchical planning* split the task of presenting a particular proof into subtasks of presenting subproofs. *Bottom-up* operators for *local navigation* simulate the spontaneous aspect, where the next conclusion to be presented is chosen under the guidance of a local focus mechanism.

The two kinds of plan operators are treated differently in *PROVERB*. Since top-down operators embody explicit communicative norms, they are given a higher

priority. Only when none of them is applicable a bottom-up operator will be chosen. The cooperation of top-down and bottom-up operators naturally leads to the organization of the produced PCAs in a hierarchical structure of attentional spaces.

As discourse plan, *PROVERB* produces a sequence of PCAs. The discourse history consists of the proof tree to be presented with the nodes marked as conveyed or unconveyed and the sequence of PCAs produced so far.

In the following, we shall review the definition of PCAs and the planning framework of *PROVERB*. All examples for PCAs and plan operators are taken from [Huang, 1994a].

Proof Communicative Acts

Proof communicative acts (PCAs) are the primitive actions planned by the macroplanner of *PROVERB*. As speech acts, they can be defined in terms of the communicative goals they fulfill as well as their possible verbalization. An example of a PCA conveying the derivation of a formula is

$$\texttt{(Derive Reasons:}\ \ (a \in F, F \subseteq G)\ \texttt{Derived-Formula:}\ \ a \in G \quad \texttt{Method: def-subset)}$$

Depending on the reference choices, a possible verbalization is

> "Since a is an element of F and F is a subset of G, a is an element of G by the definition of subset."

There are also PCAs that predicate actions planned for further presentation and thereby update the global attentional structure. For instance, the PCA

$$\texttt{(Begin-Cases Goal:}\ \ \textit{Formula}\ \texttt{Assumptions:}\ \ (A\ B))$$

creates two attentional spaces with A and B as the assumptions, and *Formula* as the goal by producing the verbalization:

> "To prove *Formula*, let us consider the two cases by assuming A and B."

[Huang, 1994a] defines 14 PCAs in total. [Huang and Fiedler, 1997] adds two further PCAs to mark the begin and end of attentional spaces, respectively.

Hierarchical Planning

Hierarchical planning operators represent communication strategies concerning how the task of presenting a proof can be split into subtasks of presenting subproofs, and how the subproofs can be mapped onto some linear order. Since these operators proceed from the root of the proof tree toward the leaves, they are also called *top-down* operators.

As an example, Huang [1994a] gives an operator that handles the goal of presenting a proof by case analysis. The operator applies to a proof tree where the subproof rooted at $?L_4$ leads to $F \vee G$, while subproofs rooted at $?L_2$ and $?L_3$ are the two cases proving Q by assuming F and G, respectively. The applicability condition encodes the two scenarios of case analysis, where either $?L_1$ is to be presented next or $?L_4$ has just been presented. In both circumstances this operator first presents the part leading to $F \vee G$, and then proceeds with the two cases. It also inserts certain PCAs to mediate between parts of proofs.

Case-Implicit

- Proof:

$$\dfrac{\dfrac{\cdots}{\vdots}\quad\dfrac{F}{\vdots}\quad\dfrac{G}{\vdots}}{\dfrac{?L_4 : F \vee G \qquad ?L_2 : Q \qquad ?L_3 : Q}{?L_1 : \Delta \vdash Q}\ \vee E}$$

- Applicability Condition: ((task $?L_1$) $\vee$ (local-focus $?L_4$))
  $\wedge$ (not-conveyed ($?L_2$ $?L_3$))
- Acts:
  1. if $?L_4$ has not been conveyed, then present $?L_4$ (subgoal 1)
  2. produce the PCA `(Case-First` F`)`
  3. present $?L_2$ (subgoal 2)
  4. produce the PCA `(Case-Second` G`)`
  5. present $?L_3$ (subgoal 3)
  6. mark $?L_1$ as conveyed
- features: (top-down compulsory implicit)

The features indicate that this is a higher priority operator (compulsory) and should be chosen when a more implicit style is preferred by the user.

[Huang, 1994a] defines 16 top-down operators in total.

Local Navigation

The *local navigation* operators simulate the spontaneous part of proof presentation. Instead of splitting presentation goals into subgoals, they follow the local derivation relation to find a proof step to be presented next. Since these operators proceed from the leaves of the proof tree toward the root, they are also called *bottom-up* operators.

The node to be presented next is suggested by the mechanism of local focus. In *PROVERB*, the *local focus* is the last derived step, while *focal centers* are semantic objects mentioned in the local focus. Although logically any proof node that uses the local focus as a premise could be chosen for the next step, usually the one with the greatest semantic overlap with the focal centers is preferred. As Huang puts it, if one has proved a property about some semantic object, one will tend to continue to talk about this particular object, before turning to a new one. As an example, [Huang, 1994a] gives the situation where the following proof is to be presented:

$$\dfrac{\dfrac{[1]:\ P(a,b)}{[2]:\ Q(a,b)}\qquad\dfrac{[1]:\ P(a,b),\ [3]:\ S(c)}{[4]:\ R(b,c)}}{[5]:\ Q(a,b) \wedge R(b,c)}$$

Assume that node [1] is the local focus, $\{a, b\}$ is the set of focal centers, [3] is a previously presented node and node [5] is the current task. [2] is chosen as the next node to be presented, since it does not (re)introduce any new semantic objects and its overlap with the focal centers (i.e., $\{a, b\}$) is larger than the overlap of [4] with the focal centers (i.e., $\{b\}$).

When a node is chosen by the local focus mechanism, a bottom-up operator is invoked to present it. The following is an example for a bottom-up operator:

Derive-Bottom-Up

- Proof:
$$\frac{N_1, \ldots, N_n}{N_{n+1}} M$$
- Applicability Condition: N_{n+1} is chosen as the next node, $N_1, \ldots, N_n$ are conveyed
- Acts: produce the PCA
`(Derive Reasons:` $(N_1, \ldots, N_n)$ `Derived-Formula:` N_{n+1}
`Method:` M`)`
- features: (bottom-up general explicit detailed)

The features indicate that this is a low priority operator (general) and should be chosen when a more explicit and detailed style is preferred by the user.

[Huang, 1994a] defines seven bottom-up operators in total.

Discussion

The macro-planner of *PROVERB* is a schema-based text planner that combines hierarchical planning with local navigation to simulate the unplanned aspect of the production of utterances.

Since the system follows the static planning approach, it suffers from the typical shortcoming of schema-based text planners: The plan operators do not represent the purpose of each part of the text plan with respect to the whole. Thus, flexible replanning of unsuccessfully communicated parts of the text is not possible.

Moreover, the text plan consists of a sequence of PCAs that is only structured by markers for the begin and the end of attentional spaces. No interrelation among PCAs is made explicit, although such interrelations have been used while planning the sequence of PCAs. The discourse history does not compensate for this drawback either, since it consists only of the proof tree, which is enhanced by marking conveyed nodes, and of the sequence of PCAs, such that the interrelations cannot be recovered without major effort.

For these reasons, it is not easily possible to extend the macro-planner of *PROVERB* to an interactive dialog planner that can cope with discourse parts that failed to communicate successfully.

A second major disadvantage of *PROVERB*'s macro-planner is that it has no user model that would allow the system to adapt its presentation to the knowledge and skills of the respective audience. The only influence the user can have is to implicitly restrict the set of applicable plan operators when he chooses an explicit vs. implicit and an abstract vs. detailed presentation. Note, however, that *abstract* presentation in *PROVERB* means that some steps may be omitted, but not that the proof is presented at a higher level of logical abstraction.

Therefore, we decided to design a new dialog planner for *P.rex* from scratch. In the following section, we shall give an overview of that dialog planner before we shall present it in detail in the remainder of this chapter.

6.1.6 Dialog Planning in *P.rex*: An Overview

The task of the dialog planner of *P.rex* is to construct a *discourse structure tree* as the representation of the dialog. We base the definition of our discourse structure tree on Hovy's approach [1993]. Therefore, it reflects the segmentation of the discourse in its subtree relation. In its leaves, it stores the individual utterances, the *speech acts*. To account for the notion of salience, we add the concept of attentional spaces to the discourse structure tree. Later, we shall use the attentional spaces to

model how salient an already conveyed piece of information is to decide whether the user has to be reminded of this piece of information.

The dialog planner invokes *plan operators* that add single nodes to the discourse structure tree and *schemata* that add whole subtrees to the discourse structure tree. Thus, it implements a hybrid planning approach as discussed in Section 6.1.4. Apart from monologs, the dialog planner allows for commands and interruptions entered by the user and for clarification dialogs.

Assorted natural language generation systems take into account the intended audience's knowledge in the generation of explanations (see e.g. [Cawsey, 1990; Paris, 1991a; Wahlster *et al.*, 1993]). Most of them adapt to the addressee by choosing between different discourse strategies. Since proofs are inherently rich in inferences, the explanation of proofs must also consider which inferences the audience can make [Zukerman and McConachy, 1993; Horacek, 1997b; 1999]. However, because of the constraints of the human memory, inferences are not chainable without costs. Explicit representation of the addressee's cognitive states proves to be useful in choosing the information to convey [Walker and Rambow, 1994]. As discussed in Chapter 5, the cognitive architecture ACT-R is particularly well suited as a basis for the dialog planner for the following reasons: First, it combines the abilities for user modeling and planning in a uniform framework. Next, the conflict resolution mechanism, which chooses a production when several productions are applicable, allows for the definition of more general productions to be applied when no more specific ones are available. Moreover, the subgoal return mechanism allows the planner to pass results from fulfilled subgoals to the supergoal. Furthermore, new productions can be learned by the system via the production compilation process. Finally, the chunk retrieval mechanism takes into account which information can be retrieved best and which information cannot be retrieved at all.

Hence, the dialog planner of *P.rex* is implemented in ACT-R. In particular, the nodes of the discourse structure tree are defined as chunks in the declarative memory, and the plan operators and schemata are realized as productions. Note that we model an idealized teacher who explains mathematical proofs to his student. We assume that the teacher has a perfect understanding of the mathematical theories he explains and, in particular, of the way to explain proofs of theorems in these theories. Since the perfect explanation knowledge is encoded by productions, we can safely turn off parameter learning for productions.

We assume that the proof is available at several levels of abstraction. The dialog planner adapts to the user by explaining the proof at a level of abstraction and a degree of explicitness that it assumes to be the most appropriate for the current user. Moreover, it combines two different presentation strategies depending on how familiar with the current subject the user is. If the user is more familiar with the subject, the dialog planner chooses a textbook-style presentation strategy where it merely derives the conclusion from the premises. However, in case the user is less familiar with the subject under consideration, the system uses a classroom-style explanation strategy where it gives more information about what is being done and why. In any style, the system allows the user to interrupt anytime if he is not satisfied with the current explanation and adapts its presentation accordingly.

Finally, the dialog planner also models that the user forgets information. This is done by ACT-R via the chunk retrieval mechanism. If the base activation of a chunk and the activations of its contextual chunks are too low the chunk's activation falls below the retrieval threshold (cf. Section 5.3.2). Since chunks with activations below the retrieval threshold cannot be accessed anymore, such forgotten information has to be derived anew by the dialog planner.

In the following section, we shall define the discourse structure tree as the representation of discourses in *P.rex*. Then, in subsequent sections, we shall define plan operators and schemata, which manipulate the discourse structure tree.

6.2 The Representation of Dialog Plans

Successful communication between an NLG system and its user presupposes that the content to be conveyed is appropriately structured to ensure a coherent semantic organization of the utterances. For an explanation system, it is also important to accept user feedback and follow-up questions. To be able to clarify misunderstood explanations, the system needs to represent the different parts of the explanation as well as the relations between them. In this section, we shall introduce how the discourse and the utterances are represented in *P.rex*.

Based on RST [Mann and Thompson, 1987], Hovy [1993] described a discourse by a nesting of *discourse segments*. According to his discourse theory, each segment essentially contains the communicative goal the speaker wants to fulfill with this segment and either one to several discourse segments with intersegment discourse relations or the semantic material to be communicated. In Section 6.2.2, we shall present our representation of discourses, which adapts Hovy's discourse segments, combines it with some aspects of a structural approach to discourse adapted from Grosz and Sidner's theory [1986] and extends it to account for certain types of dialogs as well. But first, we shall define in the following section how the semantic material to be communicated is represented in *P.rex*.

6.2.1 Speech Acts

Speech acts are the primitive actions planned by the dialog planner. They represent frozen rhetorical relations between exchangeable semantic entities. The semantic entities are represented as arguments to the rhetorical relation in the speech act. Each speech act can always be realized by a single sentence.[1] Some speech acts are preferably realized as formatting directives, which affect the layout of the presentation. We use speech acts in *P.rex* not only to represent utterances that are produced by the system, but also to represent utterances from the user in the discourse. In the following, we introduce a taxonomy of speech acts in our domain along with examples for some types of speech acts. All speech act types are defined formally in Appendix B.

We distinguish two major classes of speech acts. First, *mathematical communicative acts (MCAs)* are employed to present or explain mathematical concepts or derivations. MCAs suffice for those parts of the discourse, where the initiative is taken by the education system. The taxonomy of MCAs is shown in Figure 6.4. Second, *interpersonal communicative acts (ICAs)* serve the dialog, where both the system and the user alternately take over the active role. Figure 6.5 displays the taxonomy of ICAs.

Note that, for pragmatic reasons, we sometimes introduce on the lowest level of the taxonomy also speech acts that inherently carry content (such as `Obvious-Step`, `what-is?` and `too-difficult`) instead of keeping the speech acts separate from the content they convey.

Mathematical Communicative Acts

Mathematical communicative acts (MCAs) are speech acts that are employed to present or explain mathematical concepts or derivations. Our class of MCAs was originally derived from *PROVERB*'s PCAs [Huang, 1994a], but has been substantially reorganized and extended. In particular, we sometimes merged several closely related PCAs into a single MCA and sometimes split a PCA into several MCAs to separate their derivational and explanatory functions. Finally, compared to the

[1] Although it is possible to allow for speech acts that are realized by a sequence of sentences, no such speech acts occur in *P.rex*.

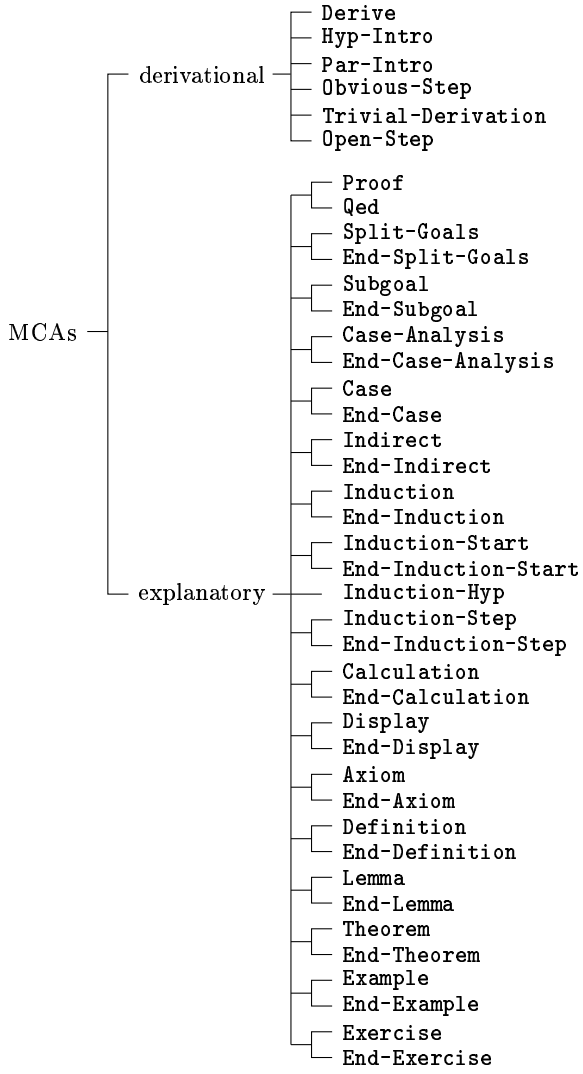


Figure 6.4. The taxonomy of MCAs.

hierarchy of PCAs, we organized the MCAs in a finer grained taxonomy. We distinguish two classes of MCAs (cf. Figure 6.4):

- *Derivational MCAs* convey steps of the derivation, which are logically necessary. Failing to produce a derivational MCA makes the presentation logically incorrect. We identify six types of MCAs in this class.

  An MCA of the first type, `Derive`, derives a fact from some premises, called reasons, by some inference method. The following is an example for a `Derive` MCA given with a possible verbalization:

  `(Derive :Reasons` $(a \in F, F \subseteq G)$ `:Conclusion` $a \in G$
  `:Method` Def$\subseteq$`)`

  "Since a is an element of F and F is a subset of G, a is an element of G by the definition of subset."

  An MCA of the second type, `Hyp-Intro`, introduces the hypothesis of a hypothetical judgment. We again give an example with a possible verbalization:

`(Hyp-Intro :Hypothesis` $F \subseteq G$`)`
"Let F be a subset of G."

An MCA of the third type, `Par-Intro`, introduces the parameter of a parametric judgment:

`(Par-Intro :Parameter` G `:Type` Set`)`
"Let G be a set."

MCAs of two further types, `Obvious-Step` and `Trivial-Derivation`, indicate that a proof step or a derivation is obvious or trivial, respectively. An MCA of type `Open-Step`, finally, conveys the information that it is still open how to derive a fact.

- *Explanatory MCAs* comment on the steps of a derivation or give information about the structure of a derivation. This information is logically unnecessary, that is, omission leaves the derivation logically correct. However, inclusion of explanatory MCAs makes it much easier for the addressee to understand the derivations, since these comments keep him oriented. Explanatory MCAs usually come in pairs: an introductory or opening explanation is paired with a closing explanation. Together, the opening and the closing explanations usually mark the boundaries of a focus space. For example, an MCA of type `Case-Analysis` introduces a case analysis:

  `(Case-Analysis :Goal` $x < a$ `:Cases (`$a < b, b < a$`))`
  "To prove $x < a$, let us consider the two cases where $a < b$ and $b < a$."

  `End-Case-Analysis` is the corresponding closing MCA type:

  `(End-Case-Analysis :Goal` $x < a$`)`
  "This completes the case analysis."[2]

  There also exist MCA types `Case` and `End-Case` to explain the individual cases:

  `(Case :Number 1 :Hypothesis` $a < b$`)`
  "Case 1: $a < b$" or
  "Let us consider the first case where $a < b$."
  `(End-Case :Number 1 :Hypothesis` $a < b$`)`
  "This completes the first case." or
  silence if the second case follows right away.

  Similar explanatory MCAs exist not only to comment on other structures that are often found in proofs such as induction and indirect proofs, but also to include calculations or diagrams in the proofs. For example, a `Calculation` MCA is interpreted as the start of a focus space, in which derivations are displayed in a chain of equations or inequations. `End-Calculation` marks the end of that focus space. The `Calculation/End-Calculation` MCA pair is also an example, where the realization as a formatting directive is preferred to a direct verbalization.

  There are also bracketing explanatory MCAs to mark the focus spaces for large-scale structures such as theorems, proofs or definitions. For example, the presentation of a proof is always preceeded by a `Proof` MCA and followed by a `Qed` MCA which may be realized as "***Proof:***" and "■", respectively.

[2] Note that the example verbalization is not appropriate for nested case analyses. When nested case analyses occur the verbalization must make clear which case analysis is closed, as in "Hence, the case analysis proves that $x < a$."

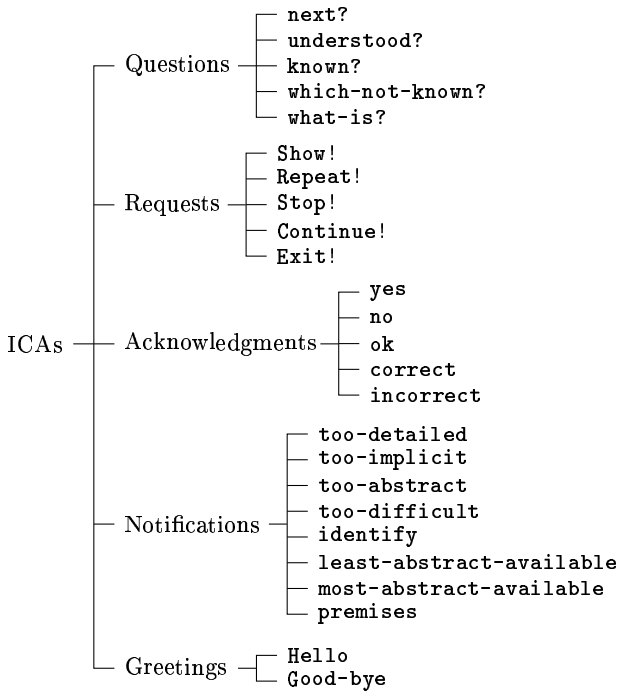


Figure 6.5. The taxonomy of ICAs.

Interpersonal Communicative Acts

MCAs, which only convey information to the dialog partner without prompting any interaction, suffice to present mathematical facts and derivations in a monolog. To allow for dialogs we also need *interpersonal communicative acts (ICAs)*, which are employed for mixed-initiative, interpersonal communication. In our taxonomy we distinguish five classes of ICAs (cf. Figure 6.5):

- *Questions* are used to ask the user what to do next (`next?`), whether he understood a presentation or a part thereof (`understood?`), whether he knows a fact or concept (`known?`), and what concept or fact he does not know (`which-not-known?`). Moreover, the user can ask the system, what a concept or entity is (`what-is?`).

- *Requests* are used to ask the user or the system to perform a task. Whereas `Show!` means that a derivation should be shown, `Repeat!` means that a the previous presentation of a derivation should be repeated. `Stop!` and `Continue!` mean that a task should be stopped or continued, respectively. `Exit!` means that *P.rex* is to be exited.

- *Acknowledgments* such as `yes` or `no` are used to answer easy questions. Further acknowledgments such as `ok`, `correct` and `incorrect` inform the user that the system will perform his request or that the user answered correctly or incorrectly, respectively.

- *Notifications* can be used by the user and the system to inform the interlocutor about something. The notifications `too-detailed`, `too-implicit`, `too-abstract` and `too-difficult` mean that the explanation of a proof step was too detailed, too implicit, too abstract or too difficult, respectively. Next,

`identify` is used to assign an object to its conceptual class, as in "F is a set." The notifications `least-abstract-available` or `most-abstract-available` indicate that a derivation was explained at the least or most abstract level available, respectively. Finally, `premises` can be used to pick some premises by their ordinal number in the list of premises of a proof step, as in "the second premise."

- *Greetings*, finally, are used to start or end a session.

Note that the user never enters speech acts directly into the system. Instead, the user's utterances are interpreted by the analyzer (cf. Section 7.3) and mapped into the corresponding speech acts.

ICAs are especially important to allow for clarification dialogs. If the system failed to successfully communicate a derivation, it starts a clarification dialog to detect the reason for the failure. Then, it can replan the previously failed part of the presentation and double-check that the user understood the derivation. We shall come back to this issue in Section 6.5.

Note that our taxonomy of speech acts relates to the metafunctions of language as defined in systemic-functional grammars [Halliday, 1994]. Derivational MCAs relate to the ideational metafunction, which is the function of the propositional content of language. Explanatory MCAs resemble the textual metafunction, which is the function of making a use of language appropriate to its particular context of use. ICAs, finally, correspond to the interpersonal metafunction, which is the function concerned with the relationships between the interlocutors.

6.2.2 Discourse Structure Trees

In the previous section, we introduced speech acts as the primitive actions planned by the dialog planner. Now, the question arises how to organize speech acts in a discourse structure that represents a coherent discourse. We shall give one possible answer in this section.

According to Grosz and Sidner [1986], we can always identify a purpose that is foundational to the discourse, called the *discourse purpose*. The discourse purpose can be seen as providing the reason the discourse is being performed (rather than some other action) and the reason the particular content of the discourse is being conveyed (rather than some other content). Similarly, we can single out a purpose for each discourse segment, called the *discourse segment purpose*. Intuitively speaking, the discourse segment purpose specifies how this segment contributes to achieving the overall discourse purpose.

To obtain a representation of discourses, [Hovy, 1993] defines a *discourse segment* as a triple (*name*, *purpose*, *content*), where the *name* is a unique identifier for the segment; the *purpose* consists of one or more communicative goals the speaker has with respect to the hearer's mental state (i.e., the discourse segment purpose as introduced in [Grosz and Sidner, 1986]); the *content* is either an ordered list of discourse segments together with one or more intersegment discourse relations that hold between them, or a single discourse segment, or the semantic material to be communicated. Note that this definition results in a tree of nested discourse segments. Based on the definition of the discourse segment, the *discourse structure* is then defined as a discourse segment that is not contained in any other discourse segment and all of whose leaves contain semantic material to be communicated.

Following Hovy's approach, we describe a discourse by a *discourse structure tree* that consists of *discourse structure nodes*, where each node corresponds to a segment of the discourse. The nesting of segments is captured by the subtree relation, such that the direct children of a node N correspond to direct subsegments of the segment

corresponding to N. Hence, the root of a discourse structure tree represents the whole discourse. The speech acts, which correspond to minimal discourse segments, are represented in the leaves. We achieve a possible linearization of the speech acts by traversing the discourse structure tree depth-first from left to right. Comparing our approach with Grosz and Sidner's theory [1986] (cf. Section 6.1), we can say that we model their dominance relation by the subtree relation and their satisfaction precedence relation by the ordering of the child nodes, as Hovy did in his approach.

As discussed in Section 6.1.3, RST relations play a twofold role in Hovy's approach. First, they are used as the relationships between nodes in the discourse structure tree. Second, they are formulated as plan operators that assemble discourse structure trees. Because of that dual role, the certainty that only coherent discourses are produced is based solely on RST relations. We deviate from Hovy's approach in this aspect by separating the relations between nodes from the plan operators. The different relations that may occur between segments and their direct subsegments are reflected by the *role* a node plays with respect to its parent node. The *plan operators* construct the discourse structure trees. They will be discussed in detail in subsequent sections of this chapter. Because of this separation, certainty that only coherent discourses are planned must be based on both the discourse structure tree and the plan operators.

The communicative goal of a discourse segment is represented in the *purpose* of the discourse structure node. A purpose consists of two parts. The *purpose intention* captures the intention that underlies the communicative goal, for example, to explain something, to inform about something or to ask something. What is to be explained, informed about or asked is captured in the *purpose content*. Whether or not the purpose of a segment is assumed to be known to the addressee is annotated in the *status* of the corresponding node.

To adopt Grosz and Sidner's theory of attentional spaces in a discourse [1986], we distinguish *basic nodes*, which correspond to ordinary segments, from *focus space nodes*, which correspond to segments that are associated with an attentional space. If a subtree is rooted by a focus space node an attentional space is considered as enclosing the discourse segment that corresponds to the root node.

Note that the definition of appropriate roles and purpose intentions enable us to represent certain types of dialogs as well.

Before we define discourse structure nodes formally, let us first define purpose intentions, roles and status.

Definition 6.1 The set $\mathbb{I} = \mathbb{I}_P \cup \mathbb{I}_U$ is the set of *purpose intentions*, where $\mathbb{I}_P = \{Explain, Present, Omit, Reverbalize, Clarify, Repeat\}$ is the set of *planning intentions* and $\mathbb{I}_U = \{Inform, Ask, Answer, Request, Acknowledge\}$ is the set of *utterance intentions*. ■

The distinction between planning and utterance intentions reflects different complexities of intentions. Whereas a goal that includes an utterance intention can be fulfilled by producing a single speech act, a goal that includes a planning intention has to be decomposed into subgoals to be fulfilled. Intuitively speaking, an utterance intention is restricted to a single proof step, whereas a planning intention encloses a larger subproof. As a consequence, planning intentions are progressively refined by the dialog planner and therefore occur only in inner nodes of a discourse structure tree. Utterance intentions, in contrast, stop the further refinement of a branch and hence occur only in the leaves of a discourse structure tree. They always trigger the production of a speech act.

The individual purpose intentions have the following meanings:

Planning Intentions

- *Explain* stands for a classroom-style explanation of a subproof, where the students are taught how to proceed in proving a theorem, that is, the emphasis lies on the process of finding a proof.
- *Present* stands for a textbook-style presentation of a subproof, where the correctness of the proof is the main concern.
- *Omit* stands for the omission of a subproof. A subproof can be omitted if it has been shown before, or if it is obvious or easily inferable by the user of the system.
- *Reverbalize* means the reverbalization of a proof step that was not successfully communicated previously. During the reverbalization of the step, any reasoning that was left implicit before is made explicit.
- *Clarify* means a clarification dialog, which is entered if the user interrupts the system, but his remark is not specific enough to allow the system to plan a response.
- *Repeat* means the repetition of a previous presentation of a proof.

Note that *Explain* and *Present* represent different strategies how a proof can be shown. In the remainder, we often use *Show* to denote either one of the planning intentions *Explain* or *Present*. Moreover, note that, in contrast to the other planning intentions, the dialog planner does not refine goals that include an *Omit* intention. Instead, *Omit* makes the dialog planner stop a branch without producing a speech act and therefore occurs only in leaves. The rationale behind *Omit* is that some subproofs may be omitted for pragmatic reasons, although they are necessary for the logical correctness of the overall proof. *Clarify*, finally, allows for a special type of dialogs, the clarification dialog.

Utterance Intentions

- *Inform* means the production of a speech act that expresses or comments on a single proof step (i.e., an MCA).
- *Ask*, *Answer*, *Request* and *Acknowledge* mean the production of speech acts that express a question, an answer, a request and an acknowledgment, respectively (i.e., ICAs).

The function of a discourse segment with respect to its dominating segment is captured by the following definition:

Definition 6.2 $R = \{verbalizes, contributes, commands, interrupts, clarifies, opens, closes\}$ is the set of *roles*. ∎

Consider the case where a derivation is to be shown. In general, the derivation $\mathcal{D}$ of a conclusion C consists of the application of an inference rule R to premises $P_1, \ldots, P_n$, leading to C and the derivations $\mathcal{D}_i$ of P_i for each $1 \leq i \leq n$ (cf. Figure 6.6). Therefore, to show the derivation $\mathcal{D}$, we usually have to show the derivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ as well as the last step of $\mathcal{D}$, that is, the step that leads to C from $P_1, \ldots, P_n$ by application of R.

Now, the role *verbalizes* captures that a discourse segment that expresses the last step of the derivation $\mathcal{D}$ is necessary to understand $\mathcal{D}$. The discourse segments that correspond to the subderivations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ contribute to the understanding of $\mathcal{D}$, too. Therefore, they are annotated with the role *contributes*.

The necessity of the distinction between the roles *verbalizes* and *contributes* becomes clearer when we consider the representation of derivations in Twega (cf.

$$\mathcal{D}\left\{\begin{array}{c} \begin{array}{ccc} \mathcal{D}_1 & \cdots & \mathcal{D}_n \\ P_1 & \cdots & P_n \end{array} \\ \hline C \end{array} R\right.$$

Figure 6.6. The relationship between a derivation and its subderivations.

Chapter 3). Since a derivation $\mathcal{D}$ is represented by a proof term D, the last step of $\mathcal{D}$ cannot easily be represented without the derivations of the premises of the last step, which are represented by subterms of D. Let us elucidate that problem with the following example:

Example 6.1
Recall from Example 3.2 on page 39 that the derivation of $\vdash A \wedge B \supset B \wedge A$ in the ND calculus

$$\dfrac{\dfrac{\dfrac{[\vdash A \wedge B]^u}{\vdash B} \wedge E_r \quad \dfrac{[\vdash A \wedge B]^u}{\vdash A} \wedge E_l}{\vdash B \wedge A} \wedge I}{\vdash A \wedge B \supset B \wedge A} \supset I^u$$

is represented by the proof term

impi (and A B) (and B A) (λu:nd (and A B).andi B A (ander A B u) (andel A B u)).

Note that (and A B) and (and B A) are parameters to the inference rule impi. The subterm λu:nd (and A B).andi B A (ander A B u) (andel A B u) represents the derivation of the premise $\vdash B \wedge A$ of the last step of the derivation. The premise itself cannot be represented without its derivation. □

Since we cannot represent a proof step without the derivations of its premises, we have to make clear when we identify a proof term with the derivation it represents and when we identify a proof term with the last step of the derivation it represents. The role *verbalizes* indicates that only the last step is meant, whereas *contributes* indicates that the whole derivation is meant.

We can summarize the distinction between *contributes* and *verbalizes* as follows: Whereas *contributes* describes the composition of the structure, *verbalizes* refers to the semantic content of the structure.

The role *commands* indicates that the user wishes *P.rex* to fulfill a new task, whereas *interrupts* means that the user interrupts the system—for example, to tell the system that he does not understand a step in the derivation. The role *clarifies* captures that a clarification dialog contributes to the fulfillment of the dominating purpose. We shall examine commands, interruptions and clarification dialogs in more detail in Section 6.5.

The roles *opens* and *closes* only occur in the first respective last child of a focus space node. They mean that the subsegment (usually a single utterance) marks the begin and the end of the focus space, respectively. Moreover, the segment that opens a focus space usually has the rhetorical function of a motivation for the derivation shown in the focus space, whereas the closing segment serves as a verbal indicator for the end of the focus space.

Clearly, in comparison to RST relations, our roles capture only a weak notion of the functions of discourse segments.

Whether the purpose of a discourse segment is assumed to be known by the user is captured by the following definition:

Definition 6.3 $S = \{known, unknown, inferable\}$ is the set of *status*. ■

The status *known* or *unknown* represents the assumption of the system that the user does or does not know the purpose of the corresponding segment, respectively. The status *inferable* indicates that the purpose is easily inferable by the user. As soon as a segment has been conveyed its status is set to *known*. If the dialog shows later that the communication of the purpose of a segment was not successful its status can be reset to *unknown*.

Now, let us define discourse structure nodes formally.

Definition 6.4 A *discourse structure node* D is a 6-tuple (P, C, G, ρ, s, A), where

(i) the *purpose* P is a pair (P_I, P_C) where $P_I \in \mathbb{I}$ is the *purpose intention* and P_C is the *purpose content*. P_C is a list of judgments, a single judgment, a speech act, or the association of a judgment with a speech act.

(ii) C is the content of the node. It is either an ordered list of discourse structure nodes or a single speech act.

(iii) G is the parent discourse structure node which dominates D, that is, D is in the content of G.

(iv) $\rho \in R$ is the role of D with respect to G.

(v) $s \in S$ is the status of D.

(vi) A is a tuple, called an *annotation*. ■

An association of a judgment with a speech act indicates that the judgment is conveyed by uttering the speech act.

Notation: We often write $(P_I\ P_C)$ to denote a purpose. We often write $(P_I\ X\ by\ Y)$ if P_C is an association of a judgment X with a speech act Y.

The annotations A are used for bookkeeping during the construction of discourse structure trees. We distinguish two types of discourse structure nodes, namely basic nodes and focus space nodes, which differ in their annotations. But first, we introduce the following notation:

Notation: We use ϵ for the empty value.

Definition 6.5 A *basic node* D is a discourse structure node, where P_C is a single judgment, a speech act or an association of both, and the annotation is a triple $A = (u_C, J, u_P)$, where the following hold:

(i) If the node has not been conveyed, u_C is the one-element list containing only P_C, otherwise $u_C = \epsilon$.

(ii) If P_C is a judgment or an association of a judgment and a speech act, then $J = R\ c_1 \ldots c_m P_1 \ldots P_n$ is the justification chosen to convey the judgment, where R is an inference rule, c_i, $1 \leq i \leq m$, are parameters and P_i, $1 \leq i \leq n$, are premises. If P_C is a speech act $J = \epsilon$.

(iii) If P_C is a judgment or an association of a judgment with a speech act $u_P \subseteq \{P_0, \ldots, P_n\}$ are the unconveyed premises of J. If P_C is a speech act $u_P = \epsilon$. ■

Basic nodes represent ordinary discourse segments. The dialog planner uses u_C and u_P to keep track of the tasks that are still to do. In J, the dialog planner records which justification is used in the verbalization of a judgment. Thus, in case the communication is not successful, the planner can ensure a different verbalization when replanning the presentation.

Definition 6.6 A *focus space node* D is a discourse structure node, where P_C is a list of judgments and the annotation is a pair $A = (u_C, d)$, where

(i) $u_C \subseteq P_C$ is a list of unconveyed judgments.

(ii) d is the list of facts that have been derived in the focus space corresponding to D. ■

Focus space nodes are necessary to structure the presentation. As in the case of basic nodes, the planner uses u_C to keep track of the tasks that remain to be done. All facts that have been derived in a focus space are dynamically recorded in d. Hence, d represents the facts that are most salient in a focus space. Even though d is not needed theoretically its rationale is to achieve a more efficient implementation.

Since not all combinations of value assignments in discourse structure nodes make sense, we restrict ourselves to *valid* discourse structure nodes.

Definition 6.7 A discourse structure node $D = ((P_I, P_C), C, G, \rho, s, A)$ is *valid* if and only if the following hold:

(i) D is a basic node or a focus space node.

(ii) If $P_I \in \mathbb{I}_P$ then P_C is a list of judgments or a single judgment and C is a list of discourse structure nodes.

(iii) If $P_I = \mathit{Omit}$ then P_C is a single judgment and C is the empty list.

(iv) If $P_I \in \mathbb{I}_U$ then P_C is a single judgment, a speech act or an association of both. If P_C is a speech act or an association of a judgment with a speech act, then C is the same speech act.

(v) If $\rho \in \{\mathit{opens}, \mathit{closes}\}$ the parent node G is a focus space node.

(vi) If D is a focus space node, then the first node in C has role *opens* or *contributes*, the last node in C has role *closes* or *contributes* and all other nodes in C have *contributes* as roles.

(vii) The status $s = \mathit{known}$ if and only if all nodes in C have status *known* or C is a speech act or $P_I = \mathit{Omit}$. ■

In (i), we stress that there are no other discourse structure nodes than basic nodes and focus space nodes. As (ii) indicates, when the purpose intention is a planning intention, the purpose content must be one or several judgments, and the content of the node must be a list of discourse structure nodes. These content nodes contribute to the fulfillment of the purpose of the discourse structure node. In (iii), we consider the special case where the purpose intention is *Omit*. Then, the purpose content is a single judgment and the content is empty, as the derivation of the judgment is omitted. In (iv), we ensure that the purpose content of an utterance intention is a judgment, a speech act or the association of both. Moreover, if a speech act is involved in the purpose content, this speech act is also the content of the discourse structure node, because it is this speech act that fulfills the purpose of the discourse structure node. In (v) and (vi), we require that discourse structure nodes with roles *opens* or *closes* can occur only in the content of a focus space node and only as the first or last node, respectively, since these roles mark the begin or end of a focus space, respectively. All other content nodes of focus space nodes must have the role *contributes*. That means, in particular, that no commands, interruptions or clarification dialogs may be adjoined in focus space nodes. Finally, (vii) guarantees that a discourse structure node has status *known* if its content is a speech act (which

fulfills the purpose—see (iv)), or its purpose intention is *Omit* (which means that a derivation is omitted[3]—see (iii)), or all its content nodes are known.

We allow only valid discourse structure nodes in discourse structure trees, as the following definition states.

Definition 6.8 A *discourse structure tree* is a tree of valid discourse structure nodes. ■

Let us first give an example for a discourse structure tree that represents a monolog:

Example 6.2
Consider the following judgment:

$$\Gamma \vdash \lambda H{:}\mathsf{nd}(a \in U).\mathsf{Def}\cup\ H : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V)$$

where Def∪ means the definition of ∪. This hypothetical judgment corresponds to the following ND-style proof:

$$\frac{\vdash a \in U}{\vdash a \in U \cup V}\,\mathsf{Def}\cup$$

The proof can be explained by the following discourse segment:

"Let $a \in U$. Then $a \in U \cup V$ by the definition of ∪."

This verbalization can be produced by the following speech acts:

$$\left[\begin{array}{l} S_1 : \texttt{(Hyp-Intro :Hypothesis } \mathsf{nd}(a \in U)\texttt{)} \\ S_2 : \texttt{(Derive :Reasons (}\mathsf{nd}(a \in U)\texttt{)) :Conclusion } \mathsf{nd}(a \in U \cup V) \\ \qquad\qquad \texttt{:Method Def}\cup\texttt{)} \end{array}\right.$$

The bar on the left indicates that a focus space encompasses both speech acts. The corresponding discourse structure tree is shown in Figure 6.7. For the sake of

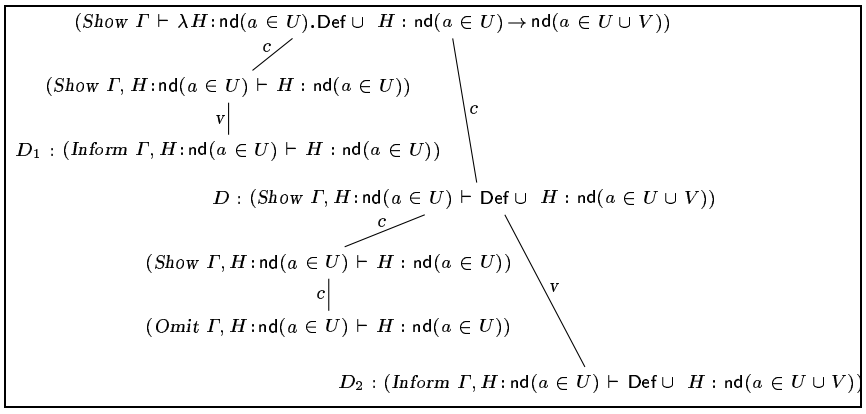


Figure 6.7. Example discourse structure tree. The box indicates a focus space. *c* stands for *contributes*, *v* stands for *verbalizes*.

readability, we display only the purposes of the nodes and the roles. The speech act S_1 is the content of node D_1 and the speech act S_2 is the content of the node D_2.

The box indicates that the root node is a focus space node and that the focus space encloses the whole discourse structure tree. □

[3]Note that the dialog planner must ensure that a derivation is only omitted if its conclusion is known or easily inferable (cf. Section 6.4.2).

Aside from monologs, *P.rex* allows for commands and interruptions entered by the user and for clarification dialogs. A discourse structure subtree whose root is marked with the role *commands* or *interrupts* represents a command or an interruption, respectively, and is always contributed by the user. Thus, the system cannot initiate a command or an interruption. A clarification dialog is characterized by the role *clarifies*. It consists of two segments: a question asked by the system and the user's answer. Hence, it can always be determined, which segments of the discourse are contributed by which participant: only commands, interruptions and answers in clarification dialogs originate from the user; all other discourse segments are produced by the system.

Let us extend the previous example to show a dialog and how it is represented as a discourse structure tree:

Example 6.2a (continued)
Assume that the explanation produced by the system did not satisfy the user, such that he enters a dialog. To elucidate the representation of dialogs in discourse structure trees, it suffices to examine the following simplistic dialog:

> ***P.rex*:** [...] Then $a \in U \cup V$ by the definition of $\cup$.
>
> **User:** This step is too difficult.
>
> ***P.rex*:** Do you understand the premises?
>
> **User:** Yes.
>
> ***P.rex*:** [*produces a different explanation*]

Since the explanation produced by the system is not satisfactory, the user interrupts the system complaining that he does not understand the last step. To find out where the problem exactly is, the system enters a clarification dialog asking the user whether he understood the premises of the last step. When the user answers with yes, the system has enough information to replan the explanation of the last step. The discourse structure tree that represents this dialog is shown in Figure 6.8. □

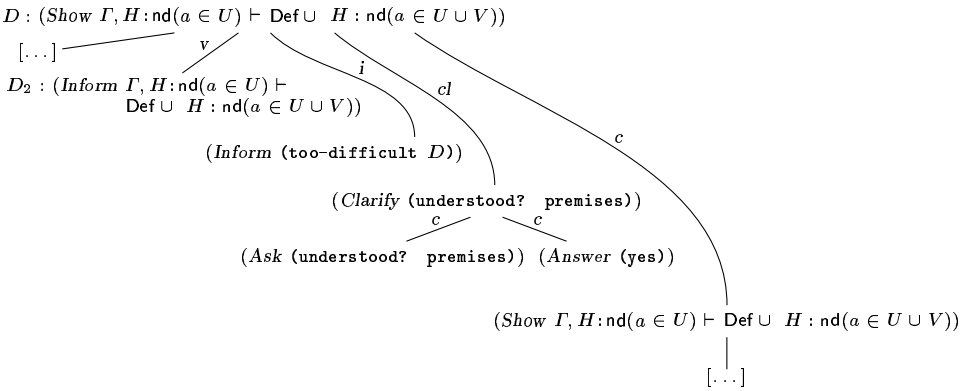


Figure 6.8. A discourse structure tree representing a dialog. *c* stands for *contributes*, *v* for *verbalizes*, *i* for *interrupts* and *cl* for *clarifies*.

To build a proof explanation system that allows for the user's interaction as well as the system's reasoning about its own explanations, we need a powerful representation of the discourse. To obtain such a representation, we adopted Hovy's discourse structure trees [1993] and adapted them to account for some aspects of Grosz and Sidner's theory [1986] and for certain types of dialogs.

Most prominently, in *P.rex*, we allow for two types of discourses besides monologs: interruptions by the user and clarification dialogs initiated by the system. Monologs, interruptions and clarification dialogs are represented in one common structure, the discourse structure tree.

Moreover, we model Grosz and Sidner's theory of the attentional structure of discourses [1986] by defining special nodes that reflect the focus spaces. The focus spaces will be used later for reference and format decisions (cf. Sections 6.4.4 and 7.1).

The definition of discourse structure trees alone does not guarantee the production of coherent dialogs. It is in cooperation with the plan operators that the discourse structure tree strives for a coherent discourse. The plan operators are the subject of the remainder of this chapter.

In *P.rex*, discourse structure nodes are implemented as chunks. The formal definition of the respective chunk types is given in Appendix C.1.2. During the construction of a discourse structure tree, the nodes are created and added to the declarative memory. Since the nodes persist in the memory, the discourse structure tree can be used later as a dialog history. Hence, a discourse structure tree serves both as dialog plan and as dialog history.

In the following section, we shall show in detail how the dialog planner constructs discourse structure trees.

6.3 Plan Operators

Operational knowledge concerning the presentation is encoded as productions in ACT-R. Each production either fulfills the current goal directly or splits it into subgoals. The goals to be fulfilled are, for example, the goal to show a subproof (as captured in a discourse structure node) or the goal to choose one of several possible justifications for a proof step.

In this section, using ND proofs as examples we shall show which productions are appropriate to plan a discourse that explains a proof and how these productions interact to construct a discourse structure tree. Recall from Section 3.3 that we can represent diverse logics in TWEGA, such as first-order or higher-order logics, modal logics, or temporal logics. The productions in *P.rex* are independent of the represented logic, some of them, however, depend on inference rules of the represented calculus. We shall first introduce the general planning principle in Section 6.3.1. Then, we give the actual productions that are used by *P.rex* in Section 6.3.2. In Section 6.3.3, finally, we shall define schemata as descriptions of standardized parts of the presentation in mathematics.

In the following we give derivations as TWEGA terms. Sometimes, we also add an ND-style description to improve readability, but keep in mind that the dialog planner operates on TWEGA terms directly. We consider only TWEGA terms $\mathcal{D}$ and φ with $\Gamma \vdash \mathcal{D} : \varphi$ for some context Γ, where according to the judgments-as-types principle (cf. Section 3.4) φ stands for a judgment in a calculus that is embedded in TWEGA, and $\mathcal{D}$ stands for a derivation of φ in this calculus. We call φ *judgment term*[4] and $\mathcal{D}$ its *proof term*. Moreover, we call the subterm of a judgment term that represents a calculus-level formula *formula term* and the formula's type *type term*. We often write $\Gamma \vdash \varphi$ if the proof term is not further considered.

Example 6.3
Let us consider the signature Σ^{ND} from Table 3.6 on page 36. Furthermore, let

[4]Do not confuse the calculus-level *judgment term* φ with the meta-level *judgment* $\Gamma \vdash \mathcal{D} : \varphi$ in TWEGA!

$\Gamma = A{:}o, B{:}o, H_1{:}\mathsf{nd}A, H_2{:}\mathsf{nd}B$. Then, we can derive the following judgments:

$$\Gamma \vdash \mathsf{and}\ A\ B : \mathsf{o} \quad \text{and} \quad \Gamma \vdash \mathsf{andi}\ A\ B\ H_1\ H_2 : \mathsf{nd}(\mathsf{and}\ A\ B)$$

In these judgments, $\mathsf{nd}(\mathsf{and}\ A\ B)$ is the judgment term and $\mathsf{andi}\ A\ B\ H_1\ H_2$ is its proof term. Moreover, $\mathsf{and}\ A\ B$ is the formula term and o is the type term of the formula term. □

In the remainder of this chapter, we often identify an ND expression ψ with the TWEGA representation of its judgment $\ulcorner \vdash \psi \urcorner$.

6.3.1 The Dialog Planning Principle

To elucidate the general principle of how proofs are explained, let us for now neglect discourse structure and concentrate only on the speech acts to be produced.

General Productions

The patterns that occur in general in proofs are applications of inference rules and introductions and discharges of hypotheses and parameters. The most prominent pattern is the application of an inference rule, in ND notation

$$\frac{\vdash \varphi_1 \quad \cdots \quad \vdash \varphi_n}{\vdash \psi} R$$

Inference rules in TWEGA are declared in the signature and have the form

$$R : \Pi a_1 : A_1 . \ldots . \Pi a_m : A_m . P_1 \to \ldots \to P_n \to C \in \Sigma$$

Here, $a_1, \ldots, a_m$ are implicit arguments of type $A_1, \ldots, A_m$, respectively, that may occur in the inference rule. $P_1, \ldots, P_n$ are the premises of the inference rule, C is its conclusion. Thus, an application of an inference rule has the form

$$\Gamma \vdash R c_1 \ldots c_m \mathcal{D}_1 \ldots \mathcal{D}_n : \psi$$

where the c_i with $\Gamma \vdash c_i : A_i$ are the instances of a_i for $1 \leq i \leq m$, $\psi = C[c_1/a_1, \ldots, c_m/a_m]$ is the conclusion, and $\varphi_i = P_i[c_1/a_1, \ldots, c_m/a_m]$ are the premises with $\Gamma \vdash \mathcal{D}_i : \varphi_i$ (i.e., $\mathcal{D}_i$ are the proof terms of the premises) for $1 \leq i \leq n$.

If we want to show such a proof step, it makes sense to show its premises first. To do so, we need a production that ensures that all premises are shown before the proof step. To this end, we introduce the following production:

(N1) IF the current goal is to show $\Gamma \vdash \psi$
and R is the most abstract known rule justifying the current goal
and $\Phi = \{\varphi_i | \Gamma \vdash \varphi_i \text{ is unknown for } 1 \leq i \leq n\} \neq \emptyset$
THEN for each $\varphi_i \in \Phi$ push the goal to show $\Gamma \vdash \varphi_i$.

This production is an example for a production that splits the current goal into several subgoals.

If the premises $\varphi_1, \ldots, \varphi_n$ have already been shown, the proof step can be presented by the following production:

(N2) IF the current goal is to show $\Gamma \vdash \psi$
and R is the most abstract known rule justifying the current goal
and $\Gamma \vdash \varphi_1, \ldots, \Gamma \vdash \varphi_n$ are known
THEN produce MCA
`(Derive :Reasons` $(\varphi_1, \ldots, \varphi_n)$ `:Conclusion` ψ `:Method` R`)`
and pop the current goal (thereby storing $\Gamma \vdash \psi$ in the declarative memory).

This production is an example for a production that directly fulfills the current goal, in this case by producing the MCA. Hence, the current goal can be popped from the goal stack. Note that the conditions of (N1) and (N2) differ only in the knowledge of the premises φ_i for rule R. (N1) introduces the subgoals to show the unknown premises in Φ. As soon as those are derived, (N2) can apply and derive the conclusion.

As discussed in Section 3.4, introduction and discharge of hypothesis and parameters are encoded in TWEGA by hypothetical and parametric judgments, respectively. Hypothetical judgments have the form

$$\Gamma \vdash \lambda p{:}\varphi.\mathcal{D} : \Pi p{:}\varphi.\psi$$

where $\mathcal{D}$ is a proof term and φ and ψ are judgment terms. The following production verbalizes the introduction of the hypothesis and pushes the goal to show the conclusion of the hypothetical judgment:

(N3) IF the current goal is to show $\Gamma \vdash \Pi p{:}\varphi.\psi$
THEN produce MCA (`Hyp-Intro :Hypothesis` φ),
store $\Gamma, p{:}\varphi \vdash \varphi$ in the declarative memory,
pop the current goal,
and push the goal to show $\Gamma, p{:}\varphi \vdash \psi$.

This production splits the current goal into the subgoals to introduce the hypothesis (therefore, we can store the judgment expressing the hypothesis in the declarative memory) and to show the conclusion. Hence, the current goal will be fulfilled and can be popped from the goal stack.

Parametric judgments have a form that is similar to hypothetical judgments, namely

$$\Gamma \vdash \lambda a{:}A.\mathcal{D} : \Pi a{:}A.\psi$$

where A is a type term or a formula term, $\mathcal{D}$ is a proof term and ψ is a judgment term. Similarly to (N3), the following production verbalizes the introduction of the new parameter and pushes the goal to show the conclusion of the parametric judgments:

(N4) IF the current goal is to show $\Gamma \vdash \Pi a{:}A.\psi$
THEN produce MCA (`Par-Intro :Parameter` a `:Type` A),
store $\Gamma, a{:}A \vdash a : A$ in the declarative memory,
pop the current goal,
and push the goal to show $\Gamma, a{:}A \vdash \psi$.

Similarly to the previous production, this production splits the current goal into the subgoals to introduce the parameter (therefore, we can store the judgment expressing the declaration of the parameter in the declarative memory) and to show the conclusion. Hence, the current goal will be fulfilled and can be popped from the goal stack.

Since these four productions already cover all patterns that occur in proofs, they suffice in principle to present any proof. However, the presentations they produce are often very tedious to follow and hard to comprehend, because they do not include any explaining comments or motivations. Hence, further specialized productions that overcome these problems are desirable. Nevertheless, the productions (N1)–(N4) guarantee that a presentation can always be produced.

To examine how the productions work together in the presentation of a proof, let us consider the following example:

Example 6.4
Let Σ^{ND} be the signature encoding the ND calculus as given in Table 3.6 on page 36.

We add the following declarations to Σ^{ND} (for the sake of readability, we omit implicit arguments to inference rules and show instead only their premises and conclusions):

$$\begin{array}{ll} \in & : \mathsf{i} \to (\mathsf{i} \to \mathsf{o}) \to \mathsf{o} \\ \cup & : (\mathsf{i} \to \mathsf{o}) \to (\mathsf{i} \to \mathsf{o}) \to (\mathsf{i} \to \mathsf{o}) \\ \mathsf{Def}\cup_1 & : \mathsf{nd}(x \in S) \to \mathsf{nd}(x \in S \cup S') \\ \mathsf{Def}\cup_2 & : \mathsf{nd}(x \in S') \to \mathsf{nd}(x \in S \cup S') \end{array}$$

Recall that the inference rule $\vee E$ is encoded by

$$\mathsf{ore} : \mathsf{nd}(A \vee B) \to (\mathsf{nd}A \to \mathsf{nd}C) \to (\mathsf{nd}B \to \mathsf{nd}C) \to \mathsf{nd}C \in \Sigma.$$

Now, let us consider the following judgment:

$$\begin{array}{l} \Gamma \vdash \mathsf{ore}\ H \\ \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def}\cup_1\ H_1) \\ \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def}\cup_2\ H_2) \\ \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

where $\Gamma = a{:}\mathsf{i}, U{:}\mathsf{i}\to\mathsf{o}, V{:}\mathsf{i}\to\mathsf{o}, H{:}\mathsf{nd}(a \in U \vee a \in V)$. This judgment corresponds to the following ND proof:

$$\dfrac{\vdash a \in U \vee a \in V \quad \dfrac{[\vdash a \in U]^{H_1}}{\vdash a \in U \cup V}\mathrm{Def}\cup_1 \quad \dfrac{[\vdash a \in V]^{H_2}}{\vdash a \in U \cup V}\mathrm{Def}\cup_2}{\vdash a \in U \cup V}\vee E^{H_1,H_2}$$

To examine how the presentation of this proof is planned, let us assume the following situation:

- The goal stack has only one goal, namely:

  $$\begin{array}{lll} \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def}\cup_1\ H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def}\cup_2\ H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

  Since the top goal on the goal stack is always the current goal, G1 is the current goal.

- Considering all justifications that prove the current goal, $\vee E$ is the most abstract rule known to the user. (In fact, $\vee E$ is the only rule that justifies the current goal in our proof. We assume that the user is familiar with this rule.)

- Each declaration in Γ has been shown earlier, that is, in particular, the user already knows the first premise to the rule $\vee E$, the so-called *major premise*:

  $$\Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V).$$

- The hypothetical judgments that are the second and third premises to the rule $\vee E$, called *cases*, namely

  $$\begin{array}{l} \Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def}\cup_1\ H_1 : \mathsf{nd}(a \in U) \to \mathsf{nd}(a \in U \cup V) \quad \text{and} \\ \Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def}\cup_2\ H_2 : \mathsf{nd}(a \in V) \to \mathsf{nd}(a \in U \cup V), \end{array}$$

  are unknown to the user.

To fulfill the current goal G1 in this situation, we cannot apply (N2), since the second and third premises, the hypothetical judgments, are still unknown. But we can apply (N1), which pushes the goals to show the hypothetical judgments (but not the first premise, $\Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V)$, which is already known). Thus, the application of (N1) results in the following goal stack:

$$\begin{array}{lll} \text{G1.1:} & \textit{Show} & \Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 \\ & & \quad : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V) \\ \text{G1.2:} & \textit{Show} & \Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 \\ & & \quad : \mathsf{nd}(a \in V) \rightarrow \mathsf{nd}(a \in U \cup V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

Thus, G1.1 becomes our current goal. Since it includes a hypothetical judgment, we apply (N3), which produces the MCA

```
(Hyp-Intro :Hypothesis nd(a ∈ U))
```

and stores $\Gamma, H_1 : \mathsf{nd}(a \in U) \vdash H_1 : \mathsf{nd}(a \in U)$ as known in the declarative memory. Moreover, (N3) replaces G1.1 on the goal stack by G1.1.1 as follows:

$$\begin{array}{lll} \text{G1.1.1:} & \textit{Show} & \Gamma, H_1 : \mathsf{nd}(a \in U) \vdash \mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U \cup V) \\ \text{G1.2:} & \textit{Show} & \Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 \\ & & \quad : \mathsf{nd}(a \in V) \rightarrow \mathsf{nd}(a \in U \cup V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

Now, G1.1.1 is the current goal. Since $\Gamma, H_1 : \mathsf{nd}(a \in U) \vdash H_1 : \mathsf{nd}(a \in U)$ is the only premise of the current goal and is known because of the previous step, we can apply (N2). This production produces the MCA

```
(Derive :Reasons (nd(a ∈ U)) :Conclusion nd(a ∈ U ∪ V)
        :Method Def∪)
```

and pops the current goal from the goal stack:

$$\begin{array}{lll} \text{G1.2:} & \textit{Show} & \Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 \\ & & \quad : \mathsf{nd}(a \in V) \rightarrow \mathsf{nd}(a \in U \cup V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

Now, we have shown the first case and G1.2, the goal to show the second case, becomes the current goal. It is fulfilled similarly to G1.1 by the productions (N3) and (N2) producing the following MCAs:

```
(Hyp-Intro :Hypothesis nd(a ∈ V))
(Derive :Reasons (nd(a ∈ V)) :Conclusion nd(a ∈ U ∪ V)
        :Method Def∪)
```

Hence, we obtain the following goal stack:

$$\begin{array}{lll} \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

This goal stack is identical to our initial goal stack. But since the cases have been presented in the meantime and are therefore now known to the user, we can apply (N2), which produces the MCA

(Derive :Reasons (nd($a \in U \vee a \in V$), nd($a \in U \cup V$), nd($a \in U \cup V$))
:Conclusion nd($a \in U \cup V$) :Method $\vee E$).

and leaves us with an empty goal stack after popping the current goal.

To sum up, the following MCAs are produced to explain our proof:

(Hyp-Intro :Hypothesis nd($a \in U$))
(Derive :Reasons (nd($a \in U$)) :Conclusion nd($a \in U \cup V$)
:Method Def$\cup$)
(Hyp-Intro :Hypothesis nd($a \in V$))
(Derive :Reasons (nd($a \in V$)) :Conclusion nd($a \in U \cup V$)
:Method Def$\cup$)
(Derive :Reasons (nd($a \in U \vee a \in V$), nd($a \in U \cup V$), nd($a \in U \cup V$))
:Conclusion nd($a \in U \cup V$) :Method $\vee E$)

The following is a possible verbalization:

> Let $a \in U$. Then, $a \in U \cup V$ by the definition of $\cup$. Let $a \in V$. Then, $a \in U \cup V$ by the definition of $\cup$. Therefore, $a \in U \cup V$ by $\vee E$.

□

Specific Productions

Clearly, even though every step of the proof has been shown, a presentation as given in Example 6.4 is sometimes not satisfying. Since the inference rule $\vee E$ corresponds to the case analysis in mathematics, we often prefer an explanation such as the following:

> To prove $a \in U \cup V$, let us consider the cases where $a \in U$ and $a \in V$, respectively.
>
> Case 1: Let $a \in U$. Then, $a \in U \cup V$ by the definition of $\cup$.
>
> Case 2: Let $a \in V$. Then, $a \in U \cup V$ by the definition of $\cup$.
>
> This completes our case analysis.

We can obtain such an explanation by introducing a specialized production that handles only case analyses:

(N5) IF The current goal is to show $\Gamma \vdash \psi$
and $\vee E$ is the most abstract known rule justifying the current goal
and the major premise $\Gamma \vdash \varphi_1 \vee \varphi_2$ is known
and the cases $\Gamma \vdash \Pi p_1 : \varphi_1 . \psi$ and $\Gamma \vdash \Pi p_2 : \varphi_2 . \psi$ are unknown
THEN pop the current goal
and push the goals
to produce MCA (Case-Analysis :Goal ψ :Cases (φ_1, φ_2)),
to produce MCA (Case :number 1),
to show $\Gamma \vdash \Pi p_1 : \varphi_1 . \psi$,
to produce MCA (Case :number 2),
to show $\Gamma \vdash \Pi p_2 : \varphi 2 . \psi$,
and to produce MCA (End-Case-Analysis :Goal ψ).

This production introduces as new subgoals the cases of the case analysis and motivates them by producing the corresponding explanatory MCAs.

Comparing the preconditions of the productions (N1) and (N5), it is easy to see that both are applicable to goal G1 in the initial situation of Example 6.4. But since (N1) is less specific than (N5), the latter should be preferred to the former.

The proofs we investigated so far indicate that, in general, more specific productions should be preferred to more general ones, since more specific rules treat common communicative standards used in mathematical presentations. To ensure this, more specific productions are assigned lower costs, that is, in particular, $C_{(\mathrm{N5})} < C_{(\mathrm{N1})}$ (cf. Equation 5.5 in Section 5.4.1). If a general rule should be preferred in the presentation to a specific one, it is also possible to achieve that by assigning higher costs to the specific production.

Moreover, we suppose that each user knows all ND rules. This is reasonable, since ND rules are the least abstract possible logical rules in proofs. Hence, for each production p that is defined such that its goal is justified by an ND rule in the proof, the probability P_p that the application of p leads to the goal to explain that proof step equals one. Therefore, since $\vee E$ is such an ND rule, $P_{(\mathrm{N5})} = 1$.

Since (N5) cannot produce the bracketing MCAs directly (the presentation of the individual cases has to be placed in between), we need another production that actually produces the speech acts.

(N6) IF the current goal is to produce a speech act
THEN produce it and pop the current goal.

To examine how the presentation is planned with these additional productions, we consider again our example:

Example 6.4a (continued)
Let us go back to the initial situation from Example 6.4. This time both productions (N1) and (N5) are applicable. Recall that the conflict resolution mechanism chooses the production with the highest utility E (cf. Equation 5.5). Since $P_{(\mathrm{N5})} = 1$ and $P_p \leq 1$ for all productions p, we obtain

$$P_{(\mathrm{N5})} \geq P_{(\mathrm{N1})}.$$

Since the application of (N1) or (N5) would serve the same goal, $G_{(\mathrm{N5})} = G_{(\mathrm{N1})}$. Since (N5) is more specific than (N1), $C_{(\mathrm{N5})} < C_{(\mathrm{N1})}$. Thus

$$E_{(\mathrm{N5})} = P_{(\mathrm{N5})}G_{(\mathrm{N5})} - C_{(\mathrm{N5})} > P_{(\mathrm{N1})}G_{(\mathrm{N1})} - C_{(\mathrm{N1})} = E_{(\mathrm{N1})}$$

Therefore, the dialog planner chooses (N5) for the explanation, thus resulting in the following goal stack:

G1.0:	*Produce*	`(Case-Analysis :Goal` $\mathsf{nd}(a \in U \cup V)$ `:Cases (`$\mathsf{nd}(a \in U), \mathsf{nd}(a \in V)$`))`
G1.1a	*Produce*	`(Case :number 1)`
G1.1:	*Show*	$\Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def}_{\cup_1} H_1 : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V)$
G1.2a	*Produce*	`(Case :number 2)`
G1.2:	*Show*	$\Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def}_{\cup_2} H_2 : \mathsf{nd}(a \in V) \rightarrow \mathsf{nd}(a \in U \cup V)$
G1.3:	*Produce*	`(End-Case-Analysis :Goal` $\mathsf{nd}(a \in U \cup V)$`)`

Note that the initial goal G1 is popped from the goal stack before the new goals are pushed onto the goal stack.

Now, (N6) pops first G1.0 and then G1.1a and produces the corresponding MCAs. Then G1.1 becomes the current goal. It is handled as previously by the productions (N3) and (N2) leaving G1.2a on top of the goal stack. This goal is also popped by (N6) producing the corresponding MCA and leaving G1.2 as the current goal. Again, (N3) and (N2) apply and leave G1.3 as the only goal on the goal stack. A final application of (N6) produces the corresponding MCA and leaves the goal stack empty.

Hence, the following MCAs are produced:

```
(Case-Analysis :Goal nd(a ∈ U ∪ V) :Cases (nd(a ∈ U), nd(a ∈ V)))
(Case :number 1)
(Hyp-Intro :Hypothesis nd(a ∈ U))
(Derive :Reasons (nd(a ∈ U)) :Conclusion nd(a ∈ U ∪ V)
        :Method Def∪)
(Case :number 2)
(Hyp-Intro :Hypothesis nd(a ∈ V))
(Derive :Reasons (nd(a ∈ V)) :Conclusion nd(a ∈ U ∪ V)
        :Method Def∪)
(End-Case-Analysis :Goal nd(a ∈ U ∪ V))
```

A possible verbalization of these speech acts is the following:

> To prove $a \in U \cup V$, let us consider the cases where $a \in U$ and $a \in V$, respectively.
>
> Case 1: Let $a \in U$. Then, $a \in U \cup V$ by the definition of $\cup$.
>
> Case 2: Let $a \in V$. Then, $a \in U \cup V$ by the definition of $\cup$.
>
> This completes our case analysis.

□

6.3.2 Planning Discourse Structure Trees

In the previous section, we examined how we can define productions that present a proof by producing a sequence of speech acts. However, the dialog planner should not produce just a mere sequence of speech acts but a discourse structure tree. In this section, we shall modify the productions from the previous section and introduce additional productions to allow the dialog planner to build a discourse structure tree as defined in Section 6.2.2. Moreover, using the example from the previous section, we shall examine how these productions cooperate to present a proof.

Let us now examine what has to be changed in the productions to construct discourse structure trees. The goals on the goal stack are discourse structure nodes. Whenever a production pushes a new discourse structure node onto the goal stack, it also has to ensure that this new node is inserted in the discourse structure tree that is under construction.

In contrast to the approach shown in the previous section when we introduced the general principle, we do not produce any speech acts when the purpose of a goal is a planning intention. Instead, we produce MCAs only if the purpose is an utterance intention.

Recall from Section 5.4.1 that ACT-R's conflict resolution process chooses the production to apply by calculating its utility. Since we switched off production parameter learning (cf. Section 6.1.6), the productions maintain their initial parameter values. The default values are $q = r = 1.0$, $a = 0.05$ and $b = 1.0$.

We often define first general productions that apply in many situations and later more specific productions for special situations. Since these specific productions should be preferred to the general ones when both are applicable, we usually set the cost parameter a for the specific production to a lower value than for the general production. The rationale behind this is that it can be expected that the cost a to fulfill the goal by a specific production is lower than the cost a to fulfill the same goal by a general production, since the discourse is more structured by the specific production than by the general one. Therefore, fulfilling the subgoals set by the specific production takes less effort than fulfilling the subgoals set by the general

one. (Recall from Section 5.4.1 that a includes the cost of applying the production and to fulfill any subgoals it sets.)

Let us again consider the proof step

$$\Gamma \vdash R\ c_1 \ldots c_m \mathcal{D}_1 \ldots \mathcal{D}_n : \psi$$

where $\Gamma \vdash \mathcal{D}_i : \varphi_i$. A discourse segment that shows this proof step must include for each premise a segment for the derivation of that premise as well. To ensure that the premises are shown before the proof step, the discourse structure nodes that represent the premises must be pushed onto the goal stack and processed before the proof step is shown. Hence, the naive production (N1) is replaced by the following production:

(P1) IF the current goal is the basic node $G = (\mathit{Show}\ \Gamma \vdash \psi)$
and R is the most abstract known rule justifying the current goal
and there is a $1 \leq i \leq n$ such that $\Gamma \vdash \varphi_i$ has not yet been conveyed
THEN push the basic node $G_i = (\mathit{Show}\ \Gamma \vdash \varphi_i)$ onto the goal stack
and append it to the content of G.

This production extends the discourse structure tree in node G as shown in Figure 6.9. Note that while (N1) pushed the goals to show all unknown premises at once, (P1) pushes only one goal, namely the goal to show one unconveyed premise. Which premise is chosen depends on a global flag. Among the options are the first unconveyed premise and the unconveyed premise with the shortest derivation.

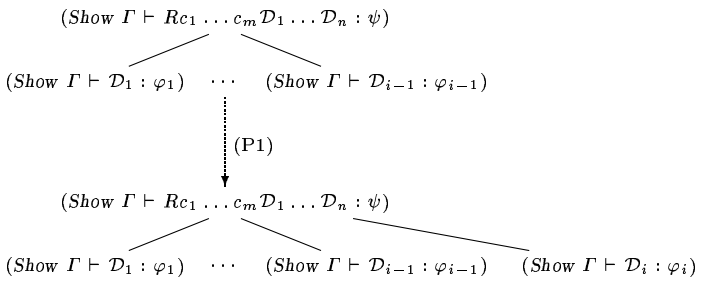


Figure 6.9. Extension of the discourse structure tree by production (P1).

Moreover, note that successive applications of (P1) ensure that for every premise a goal is pushed to show the premise. (N1), in contrast, pushes only goals to show unknown premises. The production (O1), which will be defined later in this section, will ensure that only unknown premises will eventually be derived.

When all premises have finally been conveyed, the proof step itself may be conveyed. This is triggered by the following production, which replaces (N2):

(P2) IF the current goal is $G = (\mathit{Show}\ \Gamma \vdash \psi)$
and R is the most abstract known rule justifying the current goal
and all $\Gamma \vdash \varphi_i$ for $1 \leq i \leq n$ have been conveyed
THEN push the goal $G' = (\mathit{Inform}\ \Gamma \vdash \psi)$
and append it to the content of G.
G will inherit the status of G' by the subgoal return mechanism.

This production extends the discourse structure tree in node G as shown in Figure 6.10. Note that G inherits the status of G' by ACT-R's subgoal return mechanism (cf. Section 5.4).

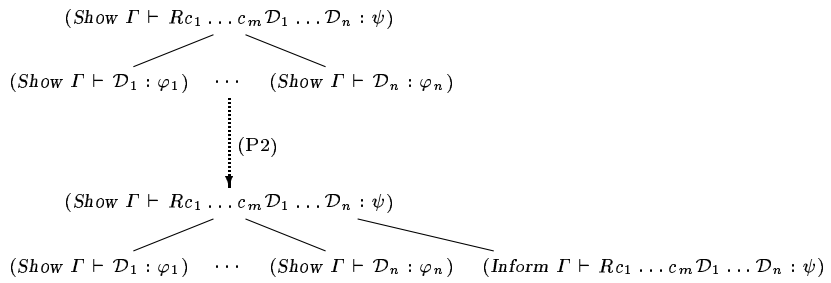


Figure 6.10. Extension of the discourse structure tree by production (P2).

While (N2) directly produced a corresponding MCA, (P2) pushes an *Inform* goal. The following production then verbalizes the proof step by producing the corresponding MCA and sets the status to *known*:

(I1) IF the current goal is $G = (\mathit{Inform}\ \Gamma \vdash R\ c_1 \ldots c_m \mathcal{D}_1 \ldots \mathcal{D}_n : \psi)$,
where $\Gamma \vdash \mathcal{D}_i : \varphi_i$ for $1 \leq i \leq n$
THEN produce the MCA
`(Derive :Reasons (`$\varphi_1, \ldots, \varphi_n$`) :Conclusion` ψ `:Method` R`)`,
set the content of G to the MCA,
and set the status of G to *known*.

Note that (P1) and (P2) only test whether the premises have been conveyed and not whether the premises are known to the user. But it may happen that a proof step has not yet been conveyed, but the fact derived by this step is already known to the user, for example, because the same fact was already derived earlier in the proof. Then, the fact should not be derived again. The following production prevents the dialog planner from deriving a known fact several times:

(O1) IF the current goal is $G = (\mathit{Show}\ \Gamma \vdash \psi)$,
and $\Gamma \vdash \psi$ is known to the user
THEN set the content of G to the discourse structure node $G' = (\mathit{Omit}\ \Gamma \vdash \psi)$,
and set the status of G and G' to *known*.

Since G' is not pushed onto the goal stack, this branch of the discourse structure tree is not further extended. Note that (P1) or (P2) may be applicable even when (O1) is applicable. To ensure that the dialog planner chooses (O1), we assign lower costs to it, namely we set $a_{(\mathrm{O1})} = 0.003$. This is reasonable, since the goal is immediately fulfilled and no subgoals are introduced.

Since we produce MCAs only indirectly via *Inform* goals, we can handle hypothetical and parametric judgments in a single production. Hence, we replace (N3) and (N4) by the following production:

(P3) IF the current goal is the basic node $G = (\mathit{Show}\ \Gamma \vdash \Pi z{:}\zeta.\psi)$,
where ζ is a type term, a formula term or a judgment term
THEN append to the content of G the discourse structure subtree rooted by the focus space node G_0 from Figure 6.11,
and push the goals G_0, G_2 and G_1.
G_0 will inherit the status of G_2 and G will inherit the status of G_0 by the subgoal return mechanism.

This production extends the discourse structure tree in node G as shown in Figure 6.11. Note that the box indicates a focus space that encompasses the whole subtree. The introduction of this focus space is necessary to represent the scope of the hypothesis of the hypothetical judgment or the parameter of the parametric

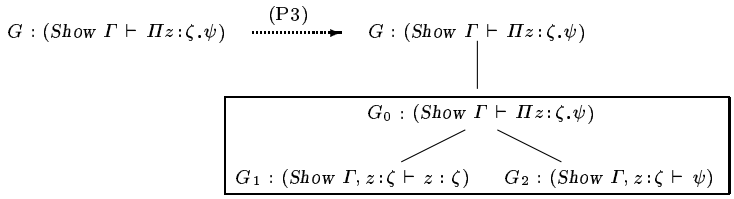


Figure 6.11. Extension of the discourse structure tree by production (P3).

judgment, respectively. Since the production pushes G_0, G_2 and G_1 in that order, it leaves G_1 on top of the stack.

To verbalize the introduction of a hypothesis or a parameter, we need two further productions similar to (I1) that produce the corresponding MCA and set the status to *known*:

(I2) IF the current goal is $G = (\mathit{Inform}\ \Gamma \vdash p : \varphi)$,
where $p{:}\varphi \in \Gamma$ and φ is a judgment term
THEN produce the MCA `(Hyp-Intro :Hypothesis` φ`)`,
set the content of G to the MCA,
and set the status of G to *known*.

(I3) IF the current goal is $G = (\mathit{Inform}\ \Gamma \vdash a : A)$,
where $a{:}A \in \Gamma$ and A is a type term or a formula term
THEN produce the MCA `(Par-Intro :Parameter` a `:Type` A`)`,
set the content of G to the MCA,
and set the status of G to *known*.

Note that none of the productions we introduced so far in this section ever pops a goal from the goal stack. Hence, we need a production that pops goals with status *known*:

(P4) IF the current goal is a discourse structure node with status *known*
THEN pop it.

To examine how these productions plan a discourse structure tree, let us come back to Example 6.4 from page 103.

Example 6.4b (continued)
This time, we assume that the initial situation is the following:

- The goal stack has only one goal, namely:

$$\begin{aligned} \text{G1:}\quad \mathit{Show}\quad \Gamma \vdash \mathsf{ore}\ H \\ (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ : \mathsf{nd}(a \in U \cup V) \end{aligned}$$

- Considering all justifications that prove the current goal, $\vee E$ is the most abstract rule known to the user.

- Each declaration in Γ has been shown earlier, that is, in particular, the user already knows the major premise to the rule $\vee E$, namely

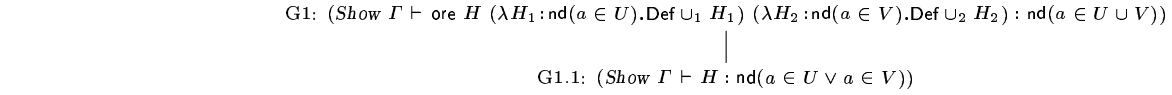
$$\Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V).$$

- The hypothetical judgments that are the second and third premises (the cases) to the rule $\vee E$, namely

  $$\Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U) \to \mathsf{nd}(a \in U \cup V) \quad \text{and}$$
  $$\Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 : \mathsf{nd}(a \in V) \to \mathsf{nd}(a \in U \cup V),$$

  are unknown to the user.

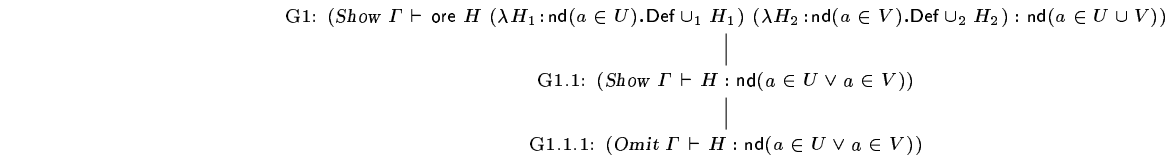
In this situation, the only applicable production is (P1). Assume, the global flag indicates that from all unconveyed premises, the first one is always chosen. Then, application of (P1) results in the following discourse structure tree



and the following goal stack:

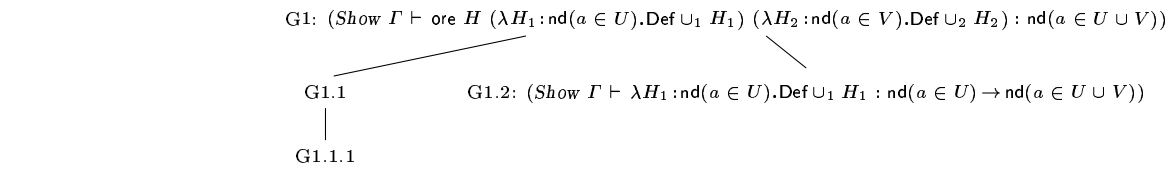
$$\begin{array}{lll} \text{G1.1:} & \textit{Show} & \Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \qquad : \mathsf{nd}(a \in U \cup V) \end{array}$$

Hence, G1.1 becomes the new current goal. Now, since the proof term in G1.1 has no premises at all, (P2) is applicable. But since the user already knows $\Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V)$, (O1) is applicable too. Since $a_{(\mathrm{O1})} < a_{(\mathrm{P2})}$, the conflict resolution mechanism of ACT-R chooses (O1). Its application extends the discourse structure tree by the new node G1.1.1:



The goal stack remains unchanged. Note that (O1) sets the status of G1.1 to *known*. Hence, (P4) applies and pops G1.1 from the goal stack leaving G1 on top of the goal stack.

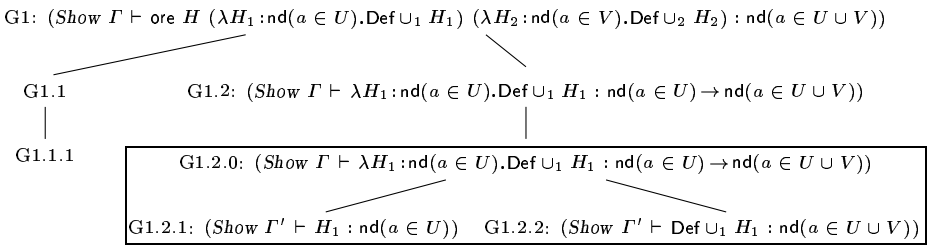
Now, (P1) is again the only applicable production. This time, it results in the following discourse structure tree



and the following goal stack:

$$\begin{array}{lll} \text{G1.2} & \textit{Show} & \Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U) \to \mathsf{nd}(a \in U \cup V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \qquad : \mathsf{nd}(a \in U \cup V) \end{array}$$

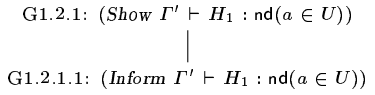
Hence, G1.2 becomes the new current goal. Since its purpose content is a hypothetical judgment, the only applicable production is (P3). Its application adjoins a subtree to the discourse structure tree in node G1.2:



where $\Gamma' = \Gamma, H_1 : \mathsf{nd}(a \in U)$. Recall that the box indicates a focus space, that is, G1.2.0 is a focus space node. The goal stack is now as follows:

G1.2.1	*Show*	$\Gamma' \vdash H_1 : \mathsf{nd}(a \in U)$
G1.2.2	*Show*	$\Gamma' \vdash \mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U \cup V)$
G1.2.0	*Show*	$\Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V)$
G1.2	*Show*	$\Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V)$
G1	*Show*	$\Gamma \vdash \mathsf{ore}\ H$ $(\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1)$ $(\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2)$ $: \mathsf{nd}(a \in U \cup V)$

Now, G1.2.1 becomes the current goal. Since its proof term has no premises, (P2) can be applied. It extends the discourse structure tree in node G1.2.1 as follows:

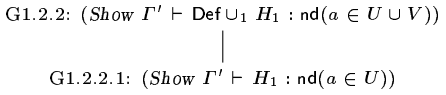


and pushes the node G1.2.1.1 onto the goal stack. Now, since $H_1 : \mathsf{nd}(a \in U) \in \Gamma'$, (I2) applies, produces the MCA

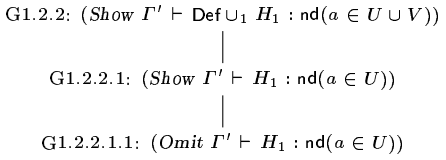
`(Hyp-Intro :Hypothesis` $\mathsf{nd}(a \in U)$`)`

and sets the status of G1.2.1.1 to *known*. Hence, (P4) can be applied and G1.2.1.1 is popped from the goal stack. Thus, G1.2.1.1 passes its status by the subgoal return mechanism to G1.2.1, whose status is therefore set to *known* as well, and G1.2.1 is then popped by (P4) leaving G1.2.2 as the new current goal.

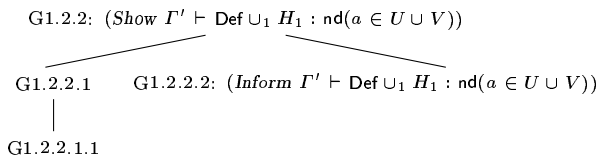
Next, (P1) applies extending the discourse structure tree in node G1.2.2 by



and pushing G1.2.2.1 onto the goal stack. Since $\Gamma' \vdash H_1 : \mathsf{nd}(a \in U)$ is now known to the user because of the last produced MCA, (O1) applies and changes the discourse structure subtree rooted by G1.2.2 to



Furthermore, (O1) sets the status of G1.2.2.1 to *known*, such that (P4) can apply, which pops G1.2.2.1 from the goal stack, again leaving G1.2.2 as the new current goal. This time, all premises have been conveyed and (P2) applies resulting in



and pushing G1.2.2.2 onto the goal stack:

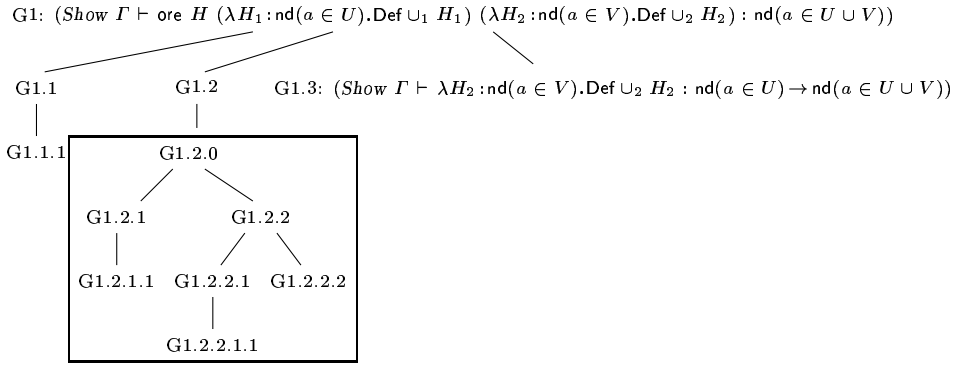
$$\begin{array}{lll} \text{G1.2.2.2} & \textit{Inform} & \Gamma' \vdash \mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U \cup V) \\ \text{G1.2.2} & \textit{Show} & \Gamma' \vdash \mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U \cup V) \\ \text{G1} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

This goal is then fulfilled by (I1), which produces the MCA

$$\texttt{(Derive :Reasons (}\mathsf{nd}(a \in U)\texttt{) :Conclusion }\mathsf{nd}(a \in U \cup V) \\ \texttt{:Method Def}\cup\texttt{)}$$

and sets the status of G.1.2.2.2 to *known*. Hence, (P4) can apply and pops G1.2.2.2, thereby passing the status *known* to G.1.2.2 and initiating a cascade of popping goals and passing the status *known* to the next goal on the stack, such that G1.2.2, G1.2.0 and G1.2 are successively popped from the goal stack by (P4).

Thus, G1 becomes the current goal. Now, one further premise is unknown and (P1) applies resulting in the following discourse structure tree:



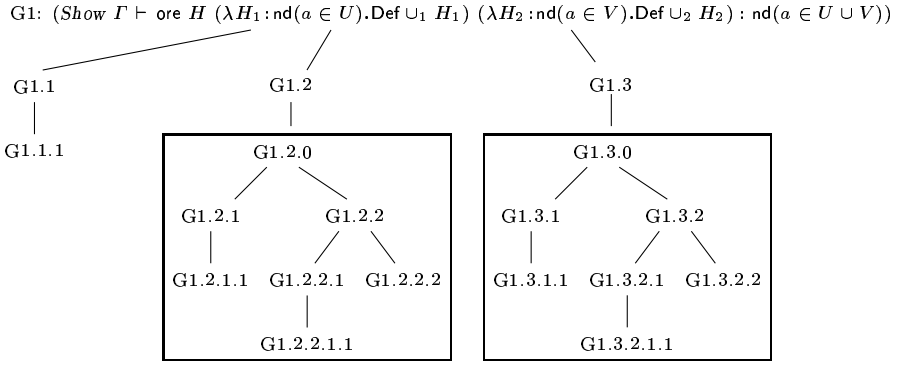
and the goal stack

$$\begin{array}{lll} \text{G1.3} & \textit{Show} & \Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

The new current goal, G1.3, is processed analogously to G1.2 producing the MCAs

$$\texttt{(Hyp-Intro :Hypothesis }\mathsf{nd}(a \in V)\texttt{)} \\ \texttt{(Derive :Reasons (}\mathsf{nd}(a \in V)\texttt{) :Conclusion }\mathsf{nd}(a \in U \cup V) \\ \texttt{:Method Def}\cup\texttt{)}$$

and finally resulting in the following discourse structure tree



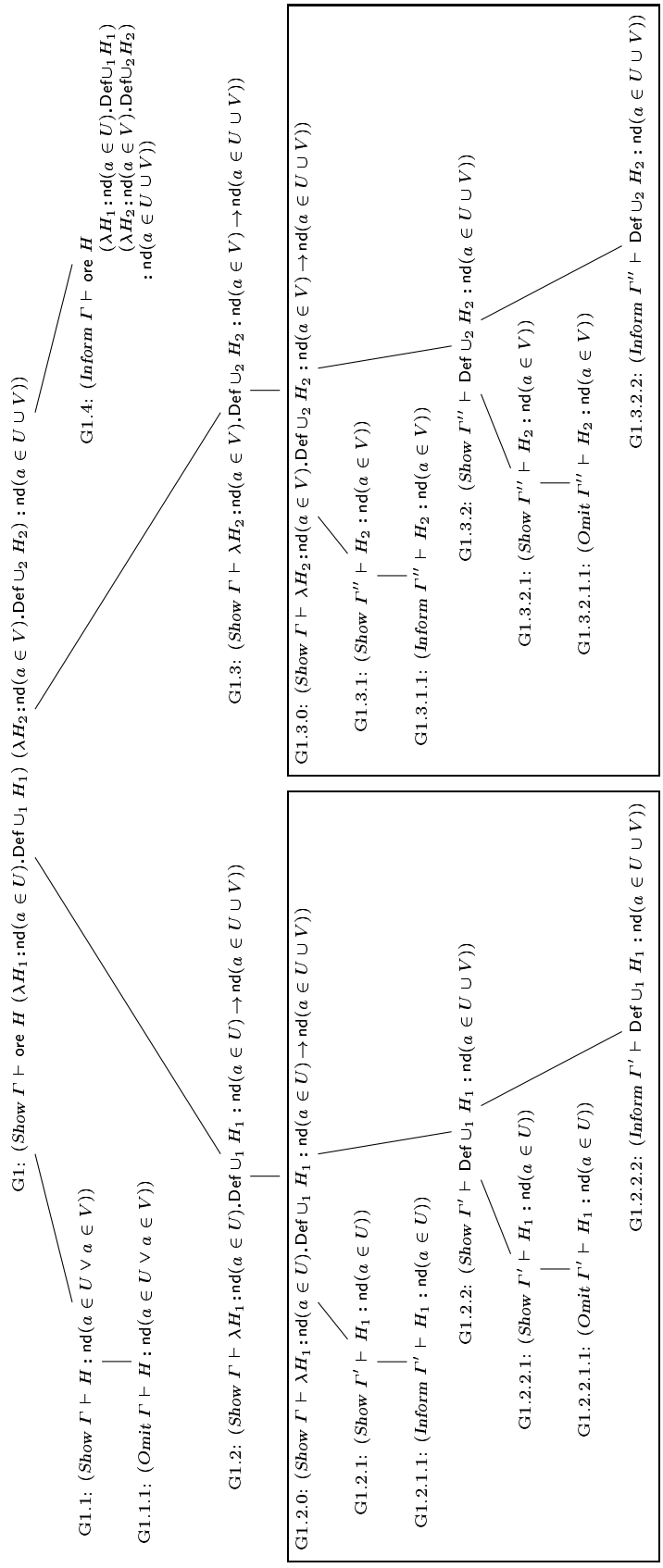


Figure 6.12. The final discourse structure tree from Example 6.4b. $\Gamma' = \Gamma, H_1 : \mathsf{nd}(a \in U)$ and $\Gamma'' = \Gamma, H_2 : \mathsf{nd}(a \in V)$.

(cf. Figure 6.12 for the purposes of the nodes of the subtree rooted by node G1.3). Afterward, the goal stack is as follows:

$$\begin{array}{lll} \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \quad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \quad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & : \mathsf{nd}(a \in U \cup V) \end{array}$$

Now, G1 is again our current goal. This time all premises are conveyed, so (P2) applies resulting in the discourse structure tree displayed in Figure 6.12 and the following goal stack:

$$\begin{array}{lll} \text{G1.4} & \textit{Inform} & \Gamma \vdash \mathsf{ore}\ H \\ & & \quad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \quad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & : \mathsf{nd}(a \in U \cup V) \\ \text{G1:} & \textit{Show} & \Gamma \vdash \mathsf{ore}\ H \\ & & \quad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & & \quad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & & : \mathsf{nd}(a \in U \cup V) \end{array}$$

This goal is then fulfilled by applying (I1), which produces the MCA

```
(Derive :Reasons (nd(a ∈ U ∨ a ∈ V), nd(a ∈ U ∪ V), nd(a ∈ U ∪ V))
        :Conclusion nd(a ∈ U ∪ V) :Method ∨E)
```

and (P4) applies twice leaving an empty goal stack.

Hence, the discourse structure given in Figure 6.12 is the final discourse structure tree. During its generation, the following speech acts are produced (the bars on the left indicate the focus spaces):

```
⌈(Hyp-Intro :Hypothesis nd(a ∈ U))
|(Derive :Reasons (nd(a ∈ U)) :Conclusion nd(a ∈ U ∪ V)
|        :Method Def∪)
⌈(Hyp-Intro :Hypothesis nd(a ∈ V))
|(Derive :Reasons (nd(a ∈ V)) :Conclusion nd(a ∈ U ∪ V)
|        :Method Def∪)
(Derive :Reasons (nd(a ∈ U ∨ a ∈ V), nd(a ∈ U ∪ V), nd(a ∈ U ∪ V))
        :Conclusion nd(a ∈ U ∪ V) :Method ∨E)
```

We have already seen the following possible verbalization:

> Let $a \in U$. Then, $a \in U \cup V$ by the definition of $\cup$. Let $a \in V$. Then, $a \in U \cup V$ by the definition of $\cup$. Therefore, $a \in U \cup V$ by $\vee E$.

□

Recall from Section 6.3.1 that the $\vee E$ rule should be presented as a case analysis, as it is common practice in mathematics. We shall examine how this is done in *P.rex* in the next section.

6.3.3 Schemata

The presentation of derivations is standardized in mathematics. For example, the individual cases of a case analysis are usually explicitly introduced, or an induction argument is in general presented by showing the base case before the step case. We can capture such schematic presentations by defining schemata that are associated

to inference rules. A *schema* is a predefined discourse structure subtree that is inserted in the discourse structure tree by one to several productions.

In this section, using the example of the case analysis, we shall show how a schema is inserted, first by a single production and then, to grasp a more general notion of case analyses, by the cooperation of three productions. Finally, we shall conclude this section with the definition of a formal language to define schemata for inference rules.

A First Approach to Case Analysis

Recall that the ND rule $\vee E$ corresponds to a case analysis with two cases. Such a case analysis is handled by the following production:

(C1) IF the current goal is $G = (\mathit{Show}\ \Gamma \vdash \psi)$
and $\vee E$ is the most abstract known rule justifying the current goal
and the major premise $\Gamma \vdash \varphi_1 \vee \varphi_2$ has not yet been conveyed
and the cases $\Gamma \vdash \Pi p_1 : \varphi_1 . \psi$ and $\Gamma \vdash \Pi p_2 : \varphi_2 . \psi$ have not yet been conveyed
THEN pop the current goal,
append to the content of G the node $G' = (\mathit{Show}\ \Gamma \vdash \varphi_1 \vee \varphi_2)$, the discourse structure tree with root G'' as given in Figure 6.13 and the node $G''' = (\mathit{Omit}\ \Gamma \vdash \psi)$,
and push the goals $G''', G'', G_4, G_3, G_{3.3}, G_{3.2}, G_{3.1}, G_2, G_{2.3}, G_{2.2}, G_{2.1}, G_1$ and G'.
G_3 inherits the status of $G_{3.3}$, G_2 inherits the status of $G_{2.3}$, G'' inherits the status of G_4 and G inherits the status of G''' by the subgoal return mechanism.

Since this is a production specific to case analyses, we assign lower costs to it, namely $a_{(\mathrm{C1})} = 0.01$. In Figure 6.13, *(Inform X by Y)* means that the judgment

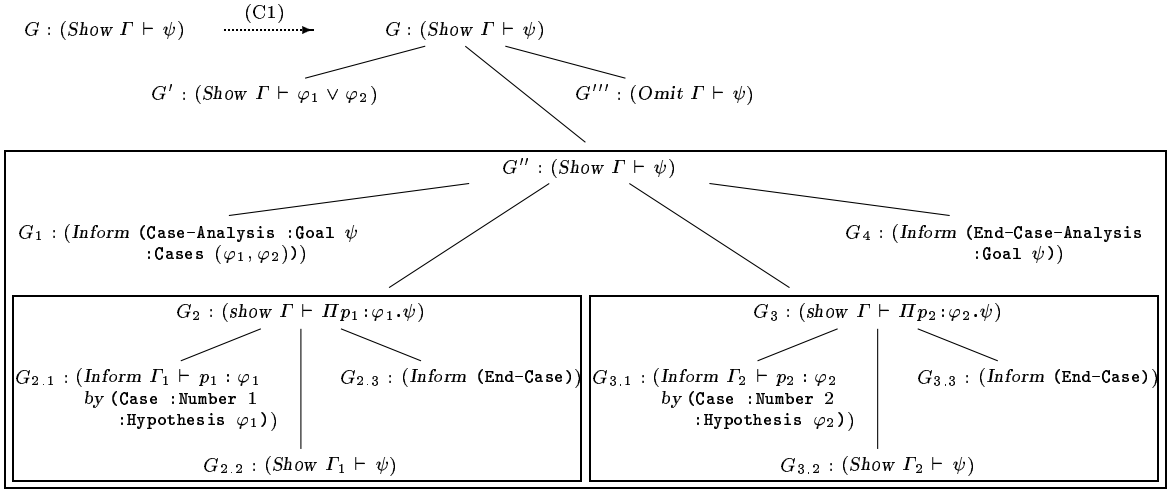


Figure 6.13. Extension of the discourse structure tree by production (C1). $\Gamma_1 = \Gamma, p_1 : \varphi_2$ and $\Gamma_2 = \Gamma, p_2 : \varphi_2$.

X is verbalized by the speech act Y. Therefore, the judgment can be considered as known to the user after the verbalization. This is realized by the following production:

(I4) IF the current goal is $G = (\mathit{Inform}\ \Gamma \vdash \varphi\ \mathit{by}\ S)$
THEN produce the speech act S
set the content of G to S,
and set the status of G to *known*.

Production (C1) also introduces subgoals that have speech acts as purpose contents. These subgoals are fulfilled by the following production:

(I5) IF the current goal is $G = (\mathit{Inform}\ S)$ and S is a speech act
THEN produce the speech act S
set the content of G to S
and set the status of G to *known*.

Let us once more consider our example proof.

Example 6.4c (continued)
Again, we assume that the initial situation is the following:

- The goal stack has only one goal, namely:

$$\begin{aligned} \text{G1:}\quad \mathit{Show}\quad & \Gamma \vdash \mathsf{ore}\ H \\ & \quad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1) \\ & \quad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2) \\ & : \mathsf{nd}(a \in U \cup V) \end{aligned}$$

- Considering all justifications that prove the current goal, $\vee E$ is the most abstract rule known to the user.
- Each declaration in Γ has been shown earlier, that is, in particular, the user already knows the proposition of the major premise to the rule $\vee E$, namely

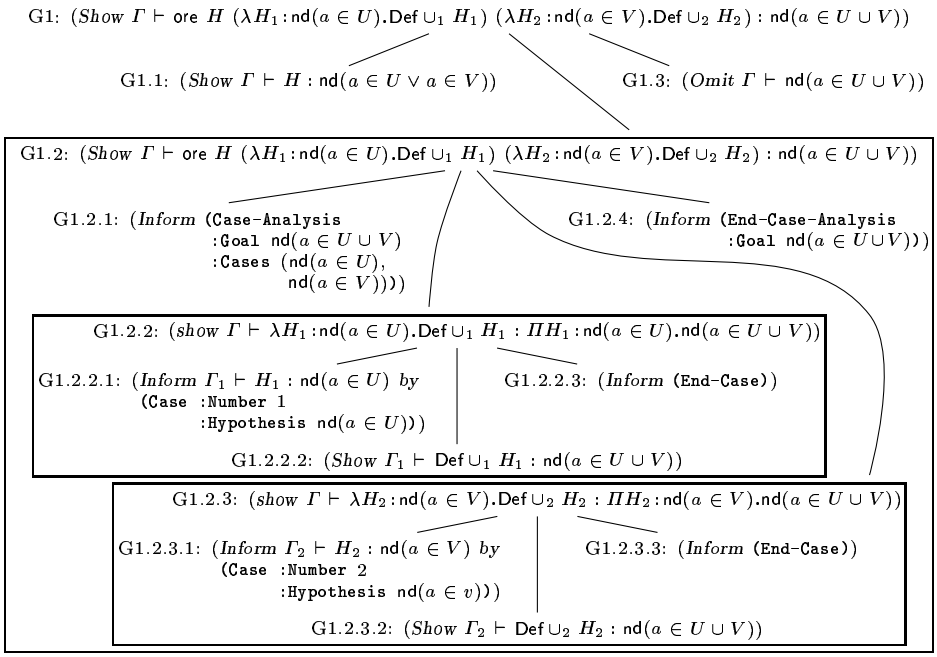
$$\Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V).$$

- The cases of the rule $\vee E$, namely

$$\Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U) \rightarrow \mathsf{nd}(a \in U \cup V) \quad \text{and}$$
$$\Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 : \mathsf{nd}(a \in V) \rightarrow \mathsf{nd}(a \in U \cup V),$$

  are unknown to the user.

In this situation, the productions (P1) and (C1) are both applicable. Since the costs of (C1) are lower, the conflict resolution mechanism of ACT-R chooses (C1) for application. It results in the following discourse structure tree



and the goal stack G1, G1.3, G1.2, G1.2.4, G1.2.3, G1.2.3.3, G1.2.3.2, G1.2.3.1, G1.2.2, G1.2.2.3, G1.2.2.2, G1.2.2.1, G1.2.1, G1.1 (the leftmost goal is the top goal). These goals are then processed canonically. We will not go into detail here, but turn the attention to the crucial parts only.

Note that the major premise is presented first, since G1.1 lies on top of the goal stack. Next, the case analysis is introduced by processing G1.2.1 by (I5), before the individual cases are presented. Moreover, note that the goals to introduce the cases, G1.2.2.1 and G1.2.3.1, are fulfilled by (I4), which verbalizes the hypotheses of the cases and sets their status to *known*. Hence, during the fulfillment of the goals G1.2.2.2 and G1.2.3.2, the respective hypotheses are not verbalized again, but omitted by (O1). The case analysis is finally concluded by processing G1.2.4.

Since the goals G1.2.2, G1.2.3 and G1.2.1 inherit the status *known* from their respective last child, they are immediately popped by (P4). G1.3 is then fulfilled by (O1) and popped by (P4) returning the status *known* to G1. G1 is finally popped by (P4). The complete discourse structure is displayed in Figure 6.14.

Altogether, the following speech acts are produced:

```
(Case-Analysis :Goal nd(a ∈ U ∪ V) :Cases (nd(a ∈ U), nd(a ∈ V)))
  (Case :Number 1 :Hypothesis nd(a ∈ U))
  (Derive :Reasons (nd(a ∈ U)) :Conclusion nd(a ∈ U ∪ V)
          :Method Def∪)
  (End-Case)
  (Case :Number 2 :Hypothesis nd(a ∈ V))
  (Derive :Reasons (nd(a ∈ V)) :Conclusion nd(a ∈ U ∪ V)
          :Method Def∪)
  (End-Case)
(End-Case-Analysis :Goal nd(a ∈ U ∪ V))
```

A possible verbalization is:

> To prove $a \in U \cup V$, let us consider the cases where $a \in U$ and $a \in V$, respectively.
>
> Case 1: Let $a \in U$. Then, $a \in U \cup V$ by the definition of $\cup$.
>
> Case 2: Let $a \in V$. Then, $a \in U \cup V$ by the definition of $\cup$.
>
> This completes our case analysis.

□

A Second Approach to Case Analysis

The main drawback of the production (C1) is that it is only applicable for case analyses with exactly two cases. But in mathematics, a case analysis usually has a varying number of cases. We can realize the presentation of case analyses with arbitrarily many cases by employing three cooperative productions. The first production ensures that the major premise is presented first and that the case analysis is bracketed by an opening and a closing explanatory MCA:

(C2) IF the current goal is $G = (\mathit{Show}\ \Gamma \vdash \psi)$
and $\vee E$ is the most abstract known rule justifying the current goal
and the major premise $\Gamma \vdash \varphi_1 \vee \cdots \vee \varphi_n$ has not yet been conveyed
and no case $\Gamma \vdash \Pi p_i{:}\varphi_i.\psi$ for $1 \leq i \leq n$ has been conveyed

THEN append to the content of G the node $G' = (\mathit{Show}\ \Gamma \vdash \varphi_1 \vee \cdots \vee \varphi_n)$, the discourse structure tree with root G'' as given in Figure 6.15 and the node

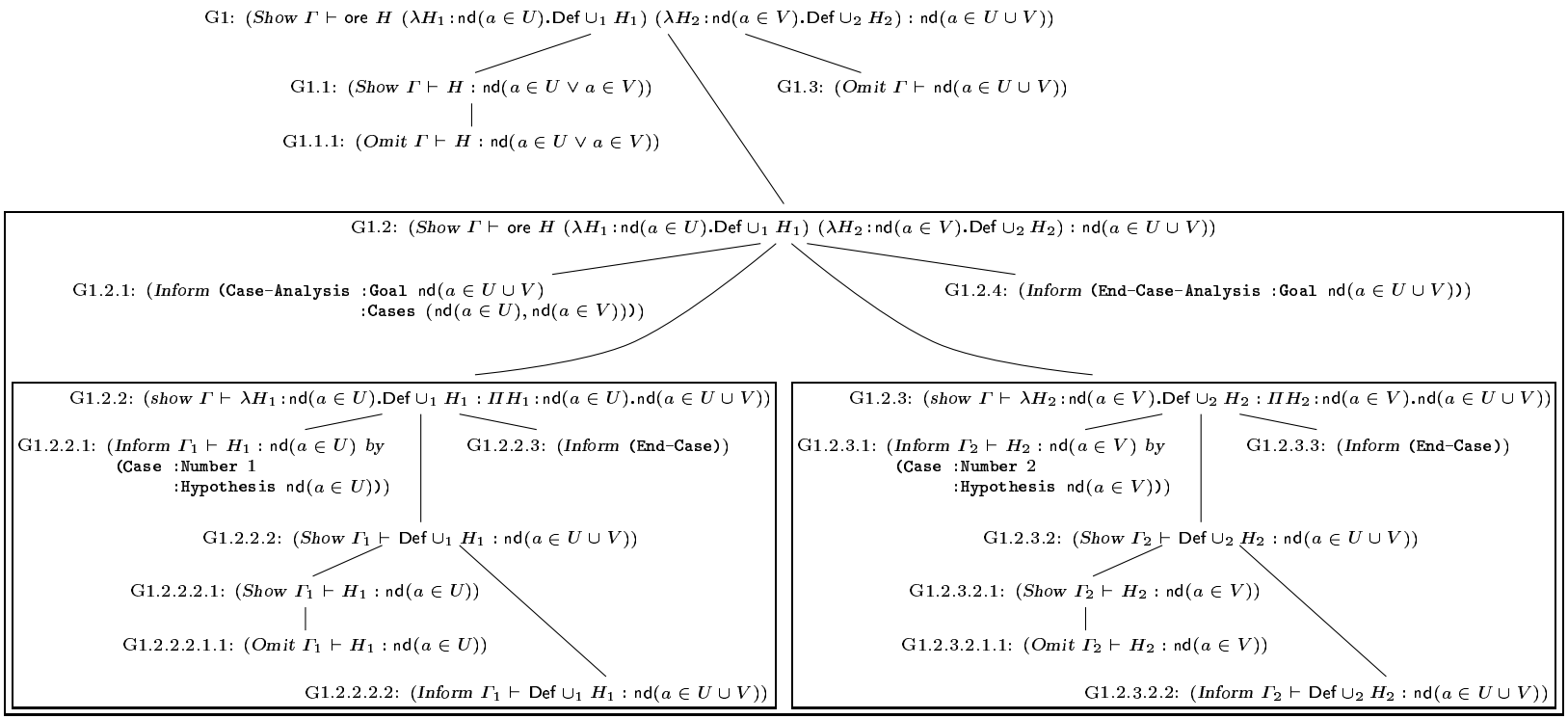


Figure 6.14. The final discourse structure tree from Example 6.4c. $\Gamma_1 = \Gamma, H_1 : \mathsf{nd}(a \in U)$ and $\Gamma_2 = \Gamma, H_2 : \mathsf{nd}(a \in V)$.

$G''' = (Omit\ \Gamma \vdash \psi)$,
and push the goals G''', G'', G_3, G_2, G_1 and G'.
G'' inherits the status of G_3 and G inherits the status of G''' by the subgoal return mechanism.

The effect of this production on the discourse structure tree is shown in Figure 6.15.

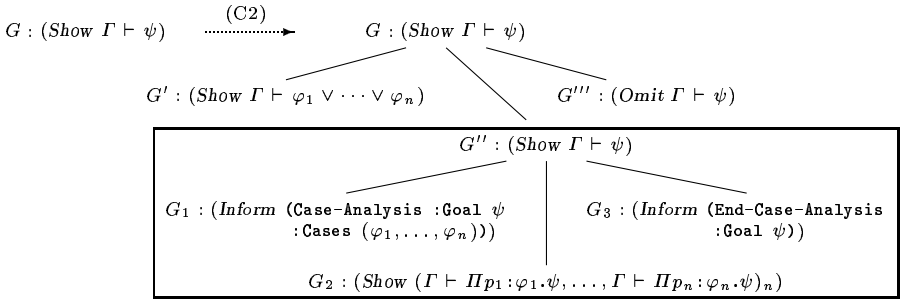


Figure 6.15. Extension of the discourse structure tree by production (C2). The index n of the list $(\Gamma \vdash \Pi p_1 : \varphi_1.\psi, \ldots, \Gamma \vdash \Pi p_n : \varphi_n.\psi)_n$ indicates that there are n cases in total.

The second production then presents the individual cases, each with an opening and a closing explanatory MCA:

(C3) IF the current goal is $G = (Show\ (\Gamma \vdash \Pi p_1 : \varphi_1.\psi, \ldots, \Gamma \vdash \Pi p_k : \varphi_k.\psi)_n)$
THEN remove $\Gamma \vdash \Pi p_1 : \varphi_1.\psi$ from the purpose content of G,
append to the content of G the discourse structure tree with root G' as given in Figure 6.16,
and push the goals G', G_3, G_2 and G_1.
G' inherits the status of G_3 as soon as G_3 is popped from the goal stack.

The index n in $(\Gamma_1 \vdash \psi, \ldots, \Gamma_k \vdash \psi)_n$ indicates that the initial length of the judgment list was n when the node G was created, that is, there are n cases in total. Note that the production uses n to calculate the number of the case under

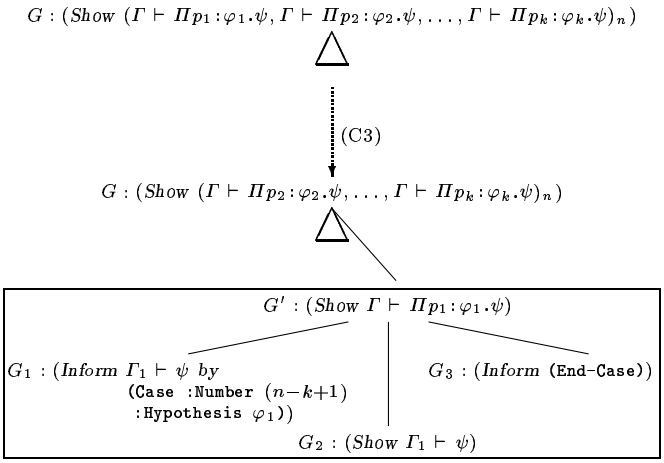


Figure 6.16. Extension of the discourse structure tree by production (C3). $\Gamma_1 = \Gamma, p_1 : \varphi_1$. The triangles represent the content of G before the application of (C3).

consideration (cf. Figure 6.16).

The production (C3) successively removes judgments from the current goal. The third production then removes a node with an empty judgment list from the discourse structure tree by replacing the node by its contents:

(L1) IF the current goal is $G = (Show\ ()_n)$ with content $G_1, \ldots, G_n$,
and G' with the content $G'_1, \ldots, G'_k, G, G'_{k+1}, \ldots, G'_m$ is the parent of G
THEN pop G from the goal stack
and replace G in the contents of its G' by its own content $G_1, \ldots, G_n$.

The effect of this production on the discourse structure tree is displayed in Figure 6.17.

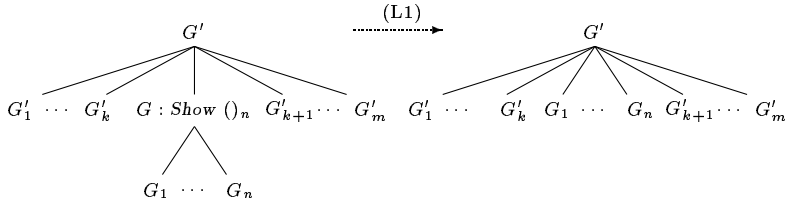
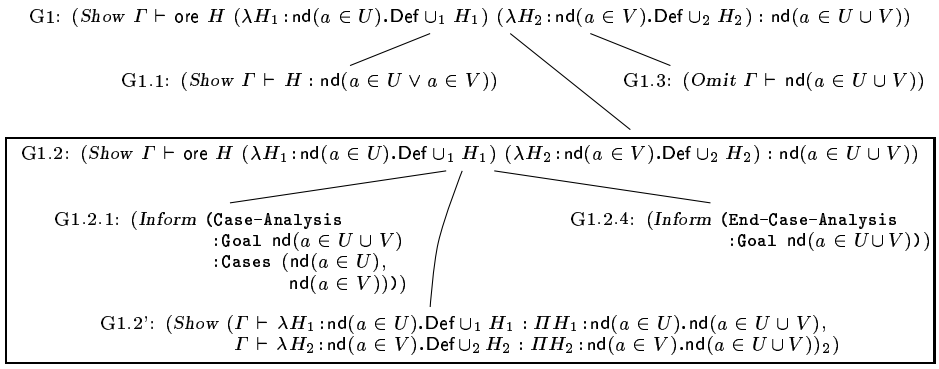


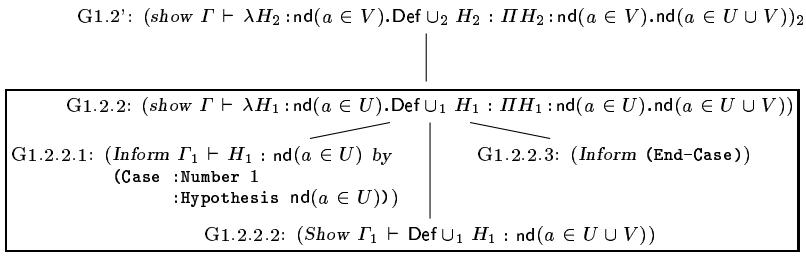
Figure 6.17. Modification of the discourse structure tree by production (L1).

For one last time, let us consider our example to examine how these productions work together to present a case analysis.

Example 6.4d (continued)
Let us again assume the start situation from Example 6.4c. This time, the production (C2) applies to goal G1 resulting in the discourse structure tree



and the goal stack G1, G1.3, G1.2, G1.2.4, G1.2', G1.2.1, G1.1. The top goals G1.1 and G1.2.1 are processed as in Example 6.4c. Then, (C3) applies to G1.2' inserting the following discourse structure subtree in node G1.2'



and resulting in the goal stack G1, G1.3, G1.2, G1.2.4, G1.2', G1.2.2, G1.2.2.3, G1.2.2.2, G1.2.2.1. Note that (C3) also deletes the first judgment from the judgment list of G1.2'.

Next the goals G1.2.2.1, G1.2.2.2, G1.2.2.3 and G1.2.2 are processed as in Example 6.4c. Thus, G1.2' becomes again the top goal:

$$\text{G1.2'} \quad \textit{Show} \quad (\Gamma \vdash \lambda H_2 : \mathtt{nd}(a \in V).\mathsf{Def} \cup_2 H_2 \\ : \Pi H_2 : \mathtt{nd}(a \in V).\mathtt{nd}(a \in U \cup V))_2$$

Hence, (C3) applies again deleting the second judgment from the judgment list and inserting another discourse structure subtree (with root G1.2.3), which is built like the tree rooted by G1.2.2. This subtree is then processed analogously, leaving again G1.2' on top of the goal stack:

$$\text{G1.2'} \quad \textit{Show} \quad ()_2$$

Now, (L1) applies and deletes G1.2' from the discourse structure tree. Finally, G1.2.4, G1.2, G1.3 and G1 are processed as in Example 6.4c.

Hence, a discourse structure tree that is identical to the tree in Figure 6.14 from Example 6.4c is build. Consequently, the same speech acts are produced, too:

```
(Case-Analysis :Goal nd(a ∈ U ∪ V) :Cases (nd(a ∈ U), nd(a ∈ V)))
  (Case :Number 1 :Hypothesis nd(a ∈ U))
  (Derive :Reasons (nd(a ∈ U)) :Conclusion nd(a ∈ U ∪ V)
          :Method Def∪)
  (End-Case)
  (Case :Number 2 :Hypothesis nd(a ∈ V))
  (Derive :Reasons (nd(a ∈ V)) :Conclusion nd(a ∈ U ∪ V)
          :Method Def∪)
  (End-Case)
(End-Case-Analysis :Goal nd(a ∈ U ∪ V))
```

□

As the previous two examples elucidate, the productions (C2), (C3) and (L1) result in the same discourse structure tree and thus in the same speech acts as (C1). Even though the former need more production cycles, they are more general than the latter, since they also handle case analyses that have more than two cases.

Defining Schemata

To describe abstractly the schemata that are to be inserted in the discourse structure tree, we define a formal language. This *schema language* allows us to formulate schema definitions in terms of the desired presentation without the need to consider productions. The operational semantics of the schema language ensures that schema inserting productions such as the productions (C1), (C2) and (C3) are automatically built from the schema definition. Only the production (L1), which is not restricted to case analyses, is predefined.

Syntax The syntax of the schema language is given by the following grammar:

schema	::=	`(schema~define` *rule-name strategies schema-def*`)`
rule-name	::=	*string*
strategies	::=	`ALL` | *purpose* | `(`*purpose*`+)`
purpose	::=	*symbol*
schema-def	::=	`(`*premise-roles premise-order mc-begin mc-end* *premise-def*`*)`
premise-roles	::=	*premise-role*`+ [(LIST` *premise-role*`)]` | `(LIST` *premise-role*`)`
premise-role	::=	*symbol* | `(HYP` *symbol*`)` | `(PAR` *symbol*`)` | `(IGNORE` *symbol*`)` | `MAJOR-PREMISE`

premise-order	::=	*numberlist* | SIZE | RSIZE | PRESORTED | NIL
premise-def	::=	(*symbol mc-begin mc-end*)
mc-begin	::=	NIL | *mc-def* | (LIST *mc-def**) | IGNORE
mc-end	::=	NIL | *mc-def* | (LIST *mc-def**) | IGNORE
mc-def	::=	(*symbol* [*:symbol arg*]*)
arg	::=	?HYP | ?PAR | ?TYPE | ?PREM | ?CONC | ?i

Here, *mc-begin* and *mc-end* stand for speech acts that give explanatory comments on the goal or subgoals to be presented. In accordance with [Zukerman, 1991], we call such explanatory comments *meta-comments*, since they are not necessary for the logical validity. In the definition of schemata, we distinguish between the *main* meta-comments, which comment on the presentation of the goal and the *premise* meta-comments, which comment on the presentation of the premises of the goal.

Definition 6.9 A schema is *valid* if the following hold:

- *purpose* must be a planning intention, such as *Explain* or *Present*.
- *premise-roles* is NIL if and only if *premise-order* is NIL if and only if *premise-defs* is empty.
- The symbol *premise-role* in (LIST *premise-role*) must not be MAJOR-PREMISE.
- In *premise-def*, *symbol* must be a premise role symbol.
- *premise-roles* and *premise-defs* must correspond to each other, respectively. That is, they must have the same length, and the ith *premise-role* and the ith *premise-def* must coincide in the symbol.
- In *mc-def*, *symbol* must be a speech act type and *:symbol* must be an argument keyword for that speech act type.
- Main meta-comments may have ?CONC as an argument. They may have ?HYP as an argument if at least one of the premise roles is of type (HYP *symbol*).
- Premise meta-comments may have ?PREM as an argument. They may have ?HYP as an argument if the premise role is of type (HYP *symbol*). They may have ?PAR and ?TYPE as arguments if the premise role is of type (PAR *symbol*). They may have ?i as an argument if the premise role is of type (LIST *premise-role*). ■

In the remainder, whenever we speak of schemata, we mean valid schemata.

Before we move on to the operational semantics of schemata, let us first describe the meaning of the components of such a definition informally.

- The *rule-name* is the name of the inference rule that is to be presented by the schema.
- The schema will be applicable to any current goal with a purpose intention as given in *strategies*. It will pass this purpose intention to the new subgoals.
- If one of the main meta-comments is not NIL, the presentation of the current goal will be enclosed in a focus space and no explicit verbalization of the proof step is given. The rationale behind this is that the main meta-comments give the explanation of the proof step. However, IGNORE means that the corresponding meta-comment is not expressed. Main meta-comments may have ?HYP or ?CONC as arguments. ?HYP means all the hypotheses that are introduced by any premises of the current goal. ?CONC stands for the conclusion of the current goal.

- If both main meta-comments are `NIL` the proof step is explicitly mentioned.
- The premise roles define the premises that are considered by the schema:
  - The premise with role `MAJOR-PREMISE` is always presented first, even before the first main meta-comment.
  - If one of the premise meta-comments is not `NIL` the presentation of that premise is enclosed in a focus space.
  - A `HYP` role means that the respective premise must be a hypothetical judgment. Then, `?HYP` can be used in its meta-comments to denote the hypothesis.
  - A `PAR` role means that the respective premise must be a parametric judgment. Then, `?PAR` and `?TYPE` can be used in its meta-comments to denote the parameter and its type, respectively.
  - An `IGNORE` role means that the derivation of the respective premise is ignored for the presentation of the current goal.
  - A `LIST` role describes all remaining premises collectively, regardless of their number. In particular, these premises will all be handled the same way. `?i` can be used in the premise meta-comments as a counter for these premises.
  - In any premise meta-comment `?PREM` can be used to denote that premise.

  Note that the actual rule may have more premises than defined by *premise-roles*. Excess premises will simply be ignored. Hence, the schema definition

  `(schema~define "`R`" ALL (nil nil nil nil))`

  produces only the speech act `(Derive :Conclusion` ψ `:Method` R`)` for the presentation of such a derivation without showing the premises. Similarly, the schema definition

  `(schema~define "`R`" ALL (nil nil IGNORE nil))`

  prevents the dialog planner from showing any derivation that ends with an application of rule R, that is, any judgment $\Gamma \vdash R\ P_1 \ldots P_n : \psi$ is ignored.
- The premise order determines in which order the premises will be presented.
  - `PRESORTED` means the premises are presented in the order given in the proof term.
  - `SIZE` or `RSIZE` mean the premises are presented according to the size of the derivation ordered from the smallest to the largest or vice versa, respectively. The size of a derivation is defined as the number of nodes in the proof tree at the highest level of abstraction.
  - *numberlist* is a list of numbers that indicate the place of the premises in the proof term. For example, (2 3 1) means the second premise is presented first, then the third and finally the first one.
- Note that the meaning of *arg* in main meta-comments may differ from its meaning in meta-comments for premises:
  - `?HYP` means all hypotheses in main meta-comments and the hypothesis of the premise in premise meta-comments.

- `?PAR` means the parameter of the premise in premise meta-comments and `?TYPE` means the type of the parameter.
- `?CONC` means the conclusion of the main rule in main meta-comments.
- `?PREM` means the premise in premise meta-comments.
- `?i` is a counter for the premises for premise roles of type (`LIST` *symbol*).

Operational Semantics A schema has the general form

$$\begin{array}{l}\texttt{(schema\~{}define "}R\texttt{"}\ p \\ \qquad\qquad ((r_1 \ldots r_n)\ \omega\ M_{0,1}\ M_{0,2} \\ \qquad\qquad\ (r'_1\ \overline{M_{1,1}}\ \overline{M_{1,2}}) \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\ (r'_n\ \overline{M_{n,1}}\ \overline{M_{n,2}})))\end{array}$$

where R is an inference rule, p a purpose intention and ω an order. Moreover, for $1 \leq i \leq n$, r_i is a premise role and r'_i is its symbol, that is, without `HYP`, `PAR`, `IGNORE` or `LIST`. Furthermore, $M_{0,1}$ and $M_{0,2}$ are the opening and closing main meta-comments and $\overline{M_{i,1}}$ and $\overline{M_{i,2}}$ the opening and closing premise meta-comments. In the following, we shall write $\omega(A_1, \ldots, A_k)$ for the list $A'_1, \ldots, A'_k$ that results from the reordering of $A_1, \ldots, A_k$ according to ω. A schema as given previously is expanded to the following production:

IF the current goal is $G = (p\ \Gamma \vdash R\ \overline{P_1} \ldots \overline{P_{n'}} : \psi)$
and G has not yet been conveyed

THEN do the following:

1. Let
$$\begin{array}{lcl} P_1, \ldots, P_{n'} & = & \omega(\overline{P_1} \ldots \overline{P_{n'}}) \\ M_{1,1}, \ldots, M_{1,n} & = & \omega(\overline{M_{1,1}}, \ldots, \overline{M_{n,1}}) \\ M_{1,2}, \ldots, M_{n,1} & = & \omega(\overline{M_{1,2}}, \ldots, \overline{M_{n,2}}) \end{array}$$

2. For $0 \leq i \leq n$ and $j \in \{1, 2\}$ let
$$S_{i,j} = \begin{cases} (\textit{Inform}\ M_{i,j}) & \text{if } \texttt{NIL} \neq M_{i,j} \neq \texttt{IGNORE} \\ \text{the empty node } \epsilon & \text{otherwise} \end{cases}$$

3. For $1 \leq i < n$ and for $i = n$ if $r_n \neq$ (`LIST` r''_n) let
$$G_i = \begin{cases} (p\ \Gamma \vdash P_i : \varphi_i) & \text{if } S_{i,1} = S_{i,2} = \epsilon \\ \boxed{\begin{array}{c}(p\ \Gamma \vdash P_i : \varphi_i) \\ \diagup \quad | \quad \diagdown \\ S_{i,1} \quad G'_i : (p\ \Gamma \vdash P_i : \varphi_i) \quad S_{i,2}\end{array}} & \text{if } S_{i,1} \neq \epsilon \text{ or } S_{i,2} \neq \epsilon \end{cases}$$

4. For $i = n$ if $r_n =$ (`LIST` r''_n) let
$$G_n = (p\ (\Gamma \vdash P_n : \varphi_n, \ldots, \Gamma \vdash P_{n'} : \varphi_{n'})_l)$$
where $l = n' - n + 1$.

5. Let
$$G_{n+1} = \begin{cases} (\textit{Inform}\ \Gamma \vdash \psi) & \text{if } M_{0,1} = M_{0,2} = \texttt{NIL} \\ (\textit{Omit}\ \Gamma \vdash \psi) & \text{otherwise} \end{cases}$$

6. If $S_{0,1} \neq \epsilon$ or $S_{0,2} \neq \epsilon$, then do the following:
   - If there is a $1 \leq k \leq n$ such that $r_k =$ `MAJOR-PREMISE`, then let
$$G' = \boxed{\begin{array}{c}(p\ \Gamma \vdash \psi) \\ \diagup \quad \diagup \quad \diagup \quad \diagdown \quad \diagdown \quad \diagdown \\ S_{0,1} \quad G_1 \cdots G_{k-1} \quad G_{k+1} \cdots G_n \quad S_{0,2}\end{array}}$$
   add G_k, G' and G_{n+1} to the contents of G;
   push $G_{n+1}, G', S_{0,2}$,

for each $n \geq i \geq 1, i \neq k$ if $S_{i,1} = S_{i,2} = \epsilon$ push G_i,
otherwise push $G_i, S_{i,2}, G'_i, S_{i,1}$;
push $S_{0,1}$;
if $S_{k,1} = S_{k,2} = \epsilon$ push G_k,
otherwise push $G_k, S_{k,2}, G'_k, S_{k,1}$.

- Otherwise, let

$$G' = \boxed{\begin{array}{c}(p\ \Gamma \vdash \psi) \\ \diagup\ \diagup\ \diagdown\ \diagdown \\ S_{0,1}\ \ G_1\ \cdots\ G_n\ \ S_{0,2}\end{array}}$$

and add G' and G_{n+1} to the content of G;
push $G_{n+1}, G', S_{0,2}$,
for each $n \geq i \geq 1$ if $S_{i,1} = S_{i,2} = \epsilon$ push G_i,
otherwise push $G_i, S_{i,2}, G'_i, S_{i,1}$;
push $S_{0,1}$.

7. Otherwise do the following:
   - If there is a $1 \leq k \leq n$ such that $r_k =$ `MAJOR-PREMISE`, then
     add $G_k, G_1, \ldots, G_{k-1}, G_{k+1}, \ldots, G_n, G_{n+1}$ to the contents of G;
     push G_{n+1},
     for each $n \geq i \geq 1, i \neq k$ if $S_{i,1} = S_{i,2} = \epsilon$ push G_i,
     otherwise push G_i, $S_{i,2}$, G'_i, $S_{i,1}$;
     if $S_{k,1} = S_{k,2} = \epsilon$ push G_k,
     otherwise push $G_k, S_{k,2}, G'_k, S_{k,1}$.
   - Otherwise
     add $G_1, \ldots, G_n, G_{n+1}$ to the contents of G;
     push G_{n+1},
     for each $n \geq i \geq 1$ if $S_{i,1} = S_{i,2} = \epsilon$ push G_i,
     otherwise push G_i, $S_{i,2}$, G'_i, $S_{i,1}$.
8. Each root of an inserted subtree that is pushed onto the goal stack inherits the status of the last node of its contents by the subgoal return mechanism.

Note that Step (1) ensures that the premises and their meta-comments are in the right order with respect to ω. Then, Step (2) builds the discourse structure nodes for all defined meta-comments. Moreover, Step (3) builds for each premise that is not of type (`LIST` *premise-role*) a discourse structure subtree (which is a focus space if and only if one of the corresponding premise meta-comments is defined). As a supplement, Step (4) builds a single node for all premises that are collected by (`LIST` *premise-role*). Step (5) serves the goal to give or omit an explicit verbalization of the proof step. The explicit verbalization is given when there are no main meta-comments that comment on the proof step, otherwise it is omitted. Next, Step (6) collects the main meta-comments and the premises in a focus space tree, whereas Step (7) adds the subtrees that show the premises to the goal.

Hence, for each premise of the proof step a new discourse structure node is added to the discourse structure tree. For each premise with a meta-comment this new node is in the scope of a new focus space. If at least one main meta-comment is defined, the subtrees for minor premises are placed in another focus space and no explicit verbalization of the proof step except in the main meta-comments is given. If no main meta-comment is defined the proof step is explicitly mentioned.

If $r_n =$ (`LIST` r''_n) for some premise role r''_n the following additional production is produced from the schema definition as well:

IF the current goal is $G = (p\ (\Gamma \vdash \mathcal{D}_1 : \psi_1, \ldots, \Gamma \vdash \mathcal{D}_m : \psi_m)_l)$ for some $m \geq 1$ and some $l \geq m$ and
$\Gamma \vdash \mathcal{D}_1 : \psi_1$ has not yet been conveyed

THEN do the following:

1. for $j \in \{1, 2\}$ let

$$S_j = \begin{cases} (\mathit{Inform}\ M_{n,j}) & \text{if } \texttt{NIL} \neq M_{n,j} \neq \texttt{IGNORE} \\ \text{the empty node } \epsilon & \text{otherwise} \end{cases}$$

2. let

$$G' = \begin{cases} (p\ \Gamma \vdash \mathcal{D}_1 : \psi_1) & \text{if } S_1 = S_2 = \epsilon \\ \boxed{\begin{array}{c} (p\ \Gamma \vdash \mathcal{D}_1 : \psi_1) \\ \diagup \quad | \quad \diagdown \\ S_1 \quad G'' : (p\ \Gamma \vdash \mathcal{D}_1 : \psi_1) \quad S_2 \end{array}} & \text{if } S_1 \neq \epsilon \text{ or } S_2 \neq \epsilon \end{cases}$$

3. remove $\Gamma \vdash \mathcal{D}_1 : \psi_1$ from the purpose content of G
4. add G' to the content of G
5. if $S_1 = S_2 = \epsilon$ push G',
   otherwise push G'_i, S_2, G'', S_1;
   G' inherits the status of G'' by the subgoal return mechanism.

Note that this production processes only the first judgment in the purpose content of the current goal. Step (1) builds the discourse structure nodes for the premise meta-comments. Step (2) builds the discourse structure subtree for the judgment.

Example 6.5
It is not difficult albeit rather tedious to see that the schema definition

```
(schema~define "ORE" all
               ((major-premise (hyp case-1) (hyp case-2)) presorted
                (case-analysis :goal ?conc :cases ?hyp)
                (end-case-analysis :goal ?conc)
                (major-premise nil nil)
                (case-1 (case :number 1 :hypothesis ?hyp)
                        (end-case))
                (case-2 (case :number 2 :hypothesis ?hyp)
                        (end-case))))
```

expands to the production (C1) and that the schema definition

```
(schema~define "ORE" all
               ((major-premise (list (hyp cases))) presorted
                (case-analysis :goal ?conc :cases ?hyp)
                (end-case-analysis :goal ?conc)
                (major-premise nil nil)
                (cases (case :number ?i :hypothesis ?hyp)
                       (end-case))))
```

expands to the productions (C2) and (C3). □

Note that the production (L1) is not produced by a schema. Instead, it is provided separately.

To sum up, schemata provide a powerful tool to define the presentation of a proof step. In particular, by defining for one inference rule several schemata that differ in the purpose intention, several presentation strategies can be implemented. Clearly, new strategies can be added by defining corresponding schemata.

In this section, we described how the dialog planner invokes plan operators to plan the presentation of a proof. In the following section, we shall examine ways to adapt the presentation to the user and to take into account the context of the presentation.

6.4 User Adaptivity and Context Sensitivity

In the previous section, we introduced productions that the dialog planner uses to traverse the proof tree and to build up a discourse structure for presenting the proof.

But since these productions do not allow for much flexibility in the presentation, the proofs are still a little mechanic. To achieve a broad acceptance from diverse users, it is indispensable both to tailor the utterances to the intended audience and to flexibly embed the utterances in the context. In the following, we shall examine how the dialog planner adapts the presentation to the assumed needs and skills of the user and to the context, in which the presentation occurs.

The first two sections are devoted to user adaptivity. In Section 6.4.1 we shall show how the dialog planner chooses the most abstract justification of a proof step that is assumed to be known to the user. In Section 6.4.2 we shall then see how the system handles trivial and user-inferable subproofs.

The remaining sections discuss context sensitivity. Section 6.4.3 will show how the dialog planner context-sensitively omits certain explanations. Then, in Section 6.4.4 we shall examine the choice of the reference to premises that have been introduced earlier in the proof. Section 6.4.5, finally, is devoted to the mechanism that switches between different presentation strategies.

6.4.1 Levels of Abstraction

In the previous sections, many productions were applicable only when the most abstract rule known to the user had been determined. In this section, we now introduce a set of productions that together determine which rule from a given sequence of justifications is the most abstract one known to the user. We assume that the justifications are ordered from most abstract to least abstract. This is a valid assumption, since the expansion of abstract rules defines an order on the level of abstraction.

First, we need a production that pushes the goal to find the most abstract justification:

(J1) IF the current goal is $G = (\mathit{Show}\ \Gamma \vdash \psi)$,
$\mathcal{D}_1, \ldots, \mathcal{D}_n$ are all proof terms that justify G, ordered from most abstract to least abstract,
and the most abstract known justification has not been determined
THEN push the goal $G' = (\mathit{Choose}\ (\mathcal{D}_1, \ldots, \mathcal{D}_n))$.
G will inherit a justification from G' by the subgoal return mechanism.

Note that G' is not a discourse structure node but a chunk defined for choosing a proof term.

The next production chooses the first justification if it is known:

(J2) IF the current goal is $G = (\mathit{Choose}\ (\mathcal{D}_1, \ldots, \mathcal{D}_n))$,
and $\mathcal{D}_1 = R\ P_1 \ldots P_m$,
and R is a known rule
THEN mark $R\ P_1 \ldots P_m$ as chosen.

Note that, since the justifications are ordered by levels of abstraction, the most abstract rule in the sequence of justifications is chosen. We set the costs $a_{(\mathrm{J2})}$ to 0.01.

The next production removes the first justification:

(J3) IF the current goal is $G = (\mathit{Choose}\ (\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n))$
THEN set G to $(\mathit{Choose}\ (\mathcal{D}_2, \ldots, \mathcal{D}_n))$.

Note that (J3) is always applicable when (J2) is applicable. Moreover, note that $a_{(\mathrm{J2})} < a_{(\mathrm{J3})}$. The conflict resolution mechanism of ACT-R ensures that (J2) is applied whenever applicable, even though (J3) is also applicable. Hence, since the justifications are ordered from most abstract to least abstract, the orchestration of

the productions (J2) and (J3) ensures that always the most abstract justification known to the user is chosen.

Only if none of the justifications is known to the user the least abstract one (i.e., an ND level justification in our examples) should be chosen:

(J4) IF the current goal $G = (Choose\ ())$
and $\mathcal{D} = R\ P_1 \ldots P_m$ was the last deleted justification in G
THEN mark $R\ P_1 \ldots P_m$ as chosen.

Hence, if all justifications are unknown to the user, the least abstract one is chosen to ensure some explanation of the proof step.

The following production pops the goal to choose a known justification, if one has been chosen:

(J5) IF the current goal is $G = (Choose\ (\mathcal{D}_1, \ldots, \mathcal{D}_n))$,
and $\mathcal{D}$ is marked as chosen
THEN pop G returning $\mathcal{D}$.

6.4.2 Omission of Subproofs

P.rex should in general avoid to state or even derive facts that are clear to the addressee, since their presentation does not give any new information but distracts from the crucial focus of the line of reasoning. (Situations where repetitions are useful are discussed in Sections 6.5.2 and 6.5.3.) To this end, we introduce productions that omit the derivations of facts that are trivial or easy to infer.

We have already seen in Section 6.3.2 the following production that omits subproofs that are already known to the user.

(O1) IF the current goal is $G = (Show\ \Gamma \vdash \psi)$,
and $\Gamma \vdash \psi$ is known to the user
THEN set the content of G to the discourse structure node $G' = (Omit\ \Gamma \vdash \psi)$,
and set the status of G and G' to *known*.

Recall that we set $a_{(O1)} = 0.003$ to ensure that this production is preferred.

A similar production omits derivations of facts that are easily inferable by the user:

(O2) IF the current goal is $G = (Show\ \Gamma \vdash \psi)$,
and $\Gamma \vdash \psi$ is easily inferable by the user
THEN set the content of G to the discourse structure node $G' = (Omit\ \Gamma \vdash \psi)$,
and set the status of G and G' to *known*.

Since this production should be preferred, its cost parameter $a_{(O2)}$ is set to 0.003.

What is in fact easily inferable by the user depends on the mathematical theory under consideration and the expertise of the user. We give only a straightforward example from the theories of numbers, such as integers or real numbers.

Example 6.6
Let the current goal be $G = (Show\ \Gamma \vdash a < x)$. If $a < b$ and the user already knows that $x > b$, then we consider $a < x$ as easily inferable. This is captured in the following production:

IF the current goal is $G = (Show\ \Gamma \vdash a < x)$
and $\Gamma \vdash a < b$ is known
and $\Gamma \vdash x > b$ is known
THEN consider $\Gamma \vdash a < x$ as easily inferable by the user. □

6.4.3 Omission of Explanatory Comments

Explanatory comments are very useful when larger structures in the proof are to be elucidated. But when the structures are small such that the closing comment follows soon after the opening comment, the closing comment is often more disturbing than helpful. Thus, we introduce the following production, which omits such closing comments:

(O3) IF the current goal is to produce a speech act that closes a focus space and the focus space is small
THEN omit the speech act by setting the status of the current goal to *known*.

To prefer this production to (I5) we set its cost parameter $a_{(O3)} = 0.01$.

If the closing comment is realized as a layout directive and is hence needed for a balanced layout, like in the case of theorems, proofs or definitions, it should not be omitted. This is enforced by the following production:

(I6) IF the current goal is to produce a speech act that closes a theorem, a proof, a definition, an example or an exercise
THEN produce the speech act and set the status of the current goal to *known*.

To prefer this production to (O3) we set its cost parameter $a_{(I6)} = 0.005$.

6.4.4 Reference Choice for the Premises of a Proof Step

Subgoals that have been derived within a proof must often be referred to later on. The reference can be made explicitly by mentioning them or implicitly by only hinting at them.

Definition 6.10 We distinguish between three *methods* to refer to the premises of a proof step:

- The reference method *implicit* means that the system only hints at the premise, for example, by uttering "Then, $a \in V$."
- The reference method *explicit* means that the premise is explicitly mentioned, as in "Since $a \in U$, $a \in V$."
- The reference method *link* means that the premise is explicitly mentioned and in addition furnished with a hyperlink to the place where it was initially derived. ■

Do not confuse the omission of a subproof, where the whole derivation of a fact is omitted, with the omission of a reference expression that refers to a fact that was previously derived or mentioned. The following productions determine the reference method for referring to premises.

(R1) IF the current goal is to produce for the discourse structure node N the MCA `(Derive :Reasons (`$\varphi_1, \ldots, \varphi_n$`) :Conclusion` ψ `:Method` R`)`
and the reference method for the premises $\varphi_k, \ldots, \varphi_n$ has not been chosen
THEN push the goal to determine the reference method for φ_k in discourse structure node N.

(R2) IF the current goal is to determine the reference method for φ in discourse structure node N
and φ was derived in the focus space to which N belongs
THEN pick *implicit* as reference method and pop the current goal.

(R3) IF the current goal is to determine the reference method for φ in discourse structure node N
and φ was derived in a focus space that dominates the focus space to which N belongs
THEN pick *explicit* as reference method and pop the current goal.

(R4) IF the current goal is to determine the reference method for φ in discourse structure node N
THEN pick *link* as reference method and pop the current goal.

We set $a_{(\mathrm{R4})} = 0.07$, which is greater than the default value 0.05, such that (R4) will be applied only when no other production is applicable.

Example 6.7
Consider the discourse structure tree in Figure 6.14 from Example 6.4c on page 120.

When the speech act in the discourse structure node G1.2.2.2.2 = (*Inform* $\Gamma_1 \vdash \mathsf{Def}\cup_1 H_1 : \mathsf{nd}(a \in U \cup V)$) is produced, the system has to refer to the premise of this proof step, namely the hypothesis $\Gamma_1 \vdash H_1 : \mathsf{nd}(a \in U)$. Since this hypothesis was introduced in the same focus space, namely in node G1.2.2, it is only hinted at implicitly by uttering "Then, $a \in U \cup V$ by the definition of $\cup$." □

Even though it is possible to determine the reference choice by the previously defined productions, we decided for efficiency reasons to implement it by a small Lisp function that is invoked whenever a `Derive` speech act is produced.

6.4.5 Presentation Strategies

As Wick and Thompson [1992] argued, a human expert, when asked to account for complex reasoning, rarely does so exclusively in terms of the *process* used to solve the problem. This problem solving process corresponds to the expert's *line of reasoning*. Instead, an expert tends to reconstruct a story that accounts for the *solution* of the problem. This story reflects the expert's *line of explanation*. Similarly, an author, when writing a textbook, knows where the reasoning is leading and can therefore present an elegant method of moving from the assumptions to the conclusion. However, this prior knowledge is seldom available during the original problem solving, requiring reasoning that is often more obscure and indirect. Therefore, Wick and Thompson advocated a decoupling of the line of explanation from the line of reasoning, which correspond to the two main focus alternatives, namely solution and process, respectively. Since they dealt with expert systems, which usually only represent knowledge for the problem solving process, they argued in favor of a reconstructive approach that recovered the line of explanation.

In mathematics, the situation is similar. The authors of mathematical textbooks usually describe theories and proofs in terms of the solution, but rarely in terms of the process of constructing theories or proofs. We call this presentation style the *textbook style*. To allow for textbook-style presentations, most current automated theorem provers actually construct and output a proof instead of only providing the user with a trace of the proof finding process.

Although acknowledging that the textbook style is well suited for securing the validity of proofs, Leron [1983; 1985] attacked the textbook style as being unsuitable to communicate mathematics. He argued that it obscures many ideas and connections, which are essential to any real understanding of the proof. Instead, he proposed a structured presentation of proofs where the structure reflects the composition of the parts to build the whole (cf. also [Melis and Leron, 1999]). The presentation moves on from the general idea to more and more detailed subproofs until the bottom is reached. Thus, the structural presentation makes explicit the

global idea and the ideas behind the parts of the proof. These ideas are crucial for a real understanding of the proof. Since the structural presentation is the method by which teachers usually explain proofs to their students in class, we call it the *classroom style* of presenting proofs.

To elucidate the distinction between textbook-style and classroom-style presentations, let us consider the following example:

Example 6.8
The limit of a function $f : A \to B$ can be defined by

$\lim_{x \to c} f(x) = L$ if and only if
for all $\varepsilon > 0$ there exists a $\delta > 0$ such that if $x \in A$ and $0 < |x - c| < \delta$
then $|f(x) - L| < \varepsilon$.

A textbook-style presentation of the proof of $\lim_{x \to c} x^2 = c^2$ starts with choosing $\delta = \inf\{1, \frac{\varepsilon}{2|c|+1}\}$ and then shows that if $0 < |x - c| < \delta$ then $|x^2 - c^2| < \varepsilon$ to prove the proposition.

A classroom-style explanation of the same proof, in contrast, proceeds as follows: It is first shown that $|x^2 - c^2| \leq (2|c|+1)|x-c|$ if $|x-c| < 1$. Also $(2|c|+1)|x-c| < \varepsilon$ if $|x - c| < \frac{\varepsilon}{2|c|+1}$. Therefore, δ can be chosen as $\inf\{1, \frac{\varepsilon}{2|c|+1}\}$.

Hence, in the textbook-style presentation an instantiation of δ is chosen at first and then the property is shown to be true. In the classroom-style presentation, in contrast, it is shown how an appropriate instantiation for δ can be found. □

By distinguishing the purpose intentions *Explain* and *Present*, the dialog planner allows for the two presentation strategies: Whereas *Explain* stands for the classroom-style explanation, *Present* means a textbook-style presentation. Recall from Section 6.3.3 that the schema language requires that the schemata be assigned to a purpose intention. Hence, it is possible to devise different schemata for the same inference rule for the two presentation strategies. Clearly, the system is easily extensible to account for further presentation strategies simply by defining new purpose intentions and corresponding schemata.

For a proof explanation system, it seems reasonable to allow for switching between the presentation strategies. When a proof that belongs to a mathematical theory T is to be shown by a classroom-style explanation, all inference rules that belong to T should be presented by a classroom-style explanation. However, any inference rules that belong to a theory that underlies T should be shown in a textbook-style presentation. The rationale behind this is that theory T is considered as the topic of the current session. Therefore, the explanation focuses on the inference rules in T. Any underlying theories are assumed to be familiar to the user and should therefore only be presented in textbook style without an elaborate explanation. Only if the user intervenes because he cannot understand the textbook-style presentation the system should replan the corresponding proof steps to present them in a classroom-style explanation. We shall elaborate on this issue in Section 6.5, when we discuss user interactions.

To ensure that the appropriate presentation strategy is chosen, we introduce the following two productions:

(S1) IF the current goal is $G = (\mathit{Explain}\ \Gamma \vdash R\ P_1 \ldots P_n : \psi)$
and T is the theory the current proof belongs to
and R belongs to a theory underlying T
THEN set the purpose intention of G to *Present*.

(S2) IF the current goal is $G = (\mathit{Present}\ \Gamma \vdash R\ P_1 \ldots P_n : \psi)$
and T is the theory the current proof belongs to

and R belongs T
and the global presentation strategy is *Explain*
THEN set the purpose intention of G to *Explain*.

To ensure that both productions (S1) and (S2) are applied before the presentation is started, we assign them with lower cost: $a_{(S1)} = a_{(S2)} = 0.001$. Let us now examine how *P.rex* reacts to user interaction.

6.5 User Interaction

As discussed in Section 2.1.2, the ability for user interaction is an important feature of explanation systems. Moore and Swartout [1991] presented a context-sensitive explanation facility for expert systems that, one the one hand, allows the user to ask follow-up questions and, on the other hand, actively seeks feedback from the user to determine whether the explanations are satisfactory. Mooney and colleagues [1991] emphasized that the user must be able to interrupt the explanation system at any time.

In *P.rex*, the user can interact with the system at any time. When the system is idle—for example, after starting it or after completion of an explanation—it waits for the user to tell it the next task. During an explanation, *P.rex* checks after each production cycle whether the user wishes to interrupt the current explanation—for example, to complain about the current explanation. However, a complaint might be so unspecific that several new discourse goals are possible. Then, the system enters a clarification dialog to single out a unique discourse goal.

Each interaction is analyzed by the analyzer (cf. Section 7.3) and passed on the dialog planner as a speech act, which is included in the current discourse structure tree to represent the user's utterance. From the speech act, specialized productions then extract one or several new discourse goals to be fulfilled subsequently.

In the following section, we shall present the different types of messages a user can direct to *P.rex*. Then, in the subsequent two sections, we shall examine how the system reacts to those messages. Section 6.5.4 will elucidate the system's behavior with two example dialogs. In Section 6.5.5, finally, we shall show how a discourse is started.

6.5.1 Messages from the User

We allow for three types of user interaction in *P.rex*: A *command* tells the system to fulfill a certain task, such as explaining a proof. An *interruption* interrupts the system during an explanation to inform it that the explanation is not satisfactory or that the user wants to move on to a different task. In clarification dialogs, finally, the user is prompted to give *answers* to questions that *P.rex* asks when it cannot identify a unique task to fulfill. A user message is first analyzed by the analyzer (cf. Section 7.3) by mapping it into a speech act. The speech act is then passed on to the dialog planner, which includes it into the discourse structure tree.

In the following, we shall present each type of user interaction in more detail. We shall give the messages as speech acts as rendered by the analyzer.

Commands

Whenever *P.rex* is idle, the user is supposed to enter a command that specifies the next discourse goal the system should fulfill. The following speech acts represent the commands that are possible in *P.rex*:

`(Show! :Proof` P `:Strategy` S`)`
Show the proof P using the presentation strategy S. Note that P stands for the name of the proof.

`(Repeat! :Proof` P`)`
Repeat the presentation of the previously shown proof P. Again, P stands for the name of the proof.

`(what-is? :Object` A`)`
Explain a concept or an entity. Here, A can be a variable declared in a context or a constant declared in the signature.

`(Exit!)`
Exit *P.rex*.

Interruptions

The user can interrupt *P.rex* anytime to enter a new command or to complain about the current explanation. In addition to the command speech acts from the previous paragraph, the following speech acts are allowed as messages to interrupt the system:

`(Obvious-Step :Conclusion` C`)`
The step leading to C is obvious and should not be mentioned. For example, the user can inform the system that the step deriving A_5 from $A_1 \wedge \cdots \wedge A_{10}$ is obvious. Note that the conclusion suffices to allow *P.rex* to identify the appropriate proof step.

`(Trivial-Derivation :Reasons` $P_1, \ldots, P_n$ `:Conclusion` C`)`
The whole subderivation that derives C from the premises $P_1, \ldots, P_n$ is trivial and should be omitted. For example, the user can inform the system that the derivation of $a \neq 0$ from $a < 0$ is trivial. Note that it is necessary to give both the premises and the conclusion to allow *P.rex* to identify the appropriate subderivation.

The following two speech acts allow the user to navigate through the different levels of abstraction in a proof. We call a higher level of abstraction *abstract* and a lower level of abstraction *detailed*.

`(too-detailed :Conclusion` C`)`
The explanation of the derivation leading to C is too detailed, that is, the derivation should be explained at a more abstract level.

`(too-abstract :Conclusion` C`)`
The explanation of the step leading to C is too abstract, that is, the step should be explained at a more detailed level.

`(too-implicit :Conclusion` C`)`
The explanation of the step leading to C is too implicit, that is, the step should be explained more explicitly. For example, in "That implies that $x > 0$." the reference to the premises is made implicitly by using the word "that". The premises are mentioned explicitly in "Since $x \neq 0$ and $x \in \mathbb{N}$, we conclude that $x > 0$."

`(too-difficult :Conclusion` C`)`
The explanation of the step leading to C is too difficult.

`(Continue!)`
Continue with the current explanation.

`(Stop!)`
Discard the current explanation.

Answers

In clarification dialogs, the system asks the user questions that he is supposed to answer. *P.rex* only asks simple questions that can be answered with the speech acts `(yes)` or `(no)`, and questions where the user must pick one or several options from a set given by *P.rex*. In the latter case, the analyzer can attribute to each option a node from the current discourse structure tree.

Let us now examine, how *P.rex* reacts to the user's interactions. We shall start with commands before we shall move on to interruptions.

6.5.2 Commands

Whenever the user enters a command $\mathcal{S}$ while the system is idle, the dialog planner generates a new discourse structure node C and includes it in the discourse structure tree by appending it to the content of the root. To represent the user's command in the discourse structure tree, the new node C has purpose (*Request* $\mathcal{S}$), content $\mathcal{S}$ and role *commands*.

To trigger a reaction to the user's command, C is also pushed onto the goal stack. Productions specialized to the different commands are then invoked to plan the system's reaction. We shall now give a detailed description of these productions.

The Reaction to $\mathcal{S}$ = `(Show! :Proof` P `:Strategy` S`)`

Since the user wants the system to show the proof P using strategy S, a corresponding goal is to be pushed onto the goal stack. This is done by the following production:

(Rc1) IF the current goal is (*Request* `(Show! :Proof` P `:Strategy` S`)`)
with parent D
THEN push the basic node $G = (S\ P)$
and append it to the content of D.

Since the new goal G has purpose $(S\ P)$, the proof P is presented using the strategy S by invoking appropriate productions as shown in Section 6.3.

The Reaction to $\mathcal{S}$ = `(Repeat! :Proof` P`)`

Now, the user demands to repeat the presentation of proof P. The following production pushes the corresponding goal onto the goal stack:

(Rc2) IF the current goal is (*Request* `(Repeat! :Proof` P`)`)
with parent D
THEN push the basic node $G = (Repeat\ P)$
and append it to the content of D.

Subsequently, a previous presentation of proof P is repeated by invoking productions that traverse the corresponding discourse structure tree and again verbalize the speech acts in it. We omit these productions here, they are included in Appendix C.2.2.

The Reaction to $\mathcal{S}$ = `(what-is? :Object` A`)`

The user asks the system about some object. Hence, a goal to explain that object is to be pushed onto the goal stack. This is ensured by the following production:

(Rc3) IF the current goal is (*Request* `(what-is? :Object` A`)`)
with parent D
THEN push the basic node $G = (\textit{Explain}\ A)$
and append it to the content of D.

Note that *Request* indicates that the question `what-is?` plays the role of the request to explain the object A. If A was introduced by a declaration, it is explained by stating the declaration. If A corresponds to a definition, it is explained by stating the definition. This is ensured by the following production.

(P5) IF the current goal is $G = (\textit{Explain}\ A)$ with $\Gamma \vdash A : \psi$,
where ψ is a formula term or a type term
THEN push the basic node $G' = (\textit{Inform}$ `(Identify :object` A `:class` ψ`))`
and append it to the content of G.

The Reaction to $\mathcal{S}$ = `(Exit!)`

The user wants the system to exit. The system's reaction is mediated by the following production:

(Rc4) IF the current goal is (*Request* `(Exit!)`)
with parent D
THEN push the goal G' = (exit)
and push the basic node $G = (\textit{Inform}$ `(Good-bye)`)
and append G to the content of D.

Since G has purpose (*Inform* `(Good-bye)`), the production (I5) (cf. page 118) applies in the next cycle, such that the system expresses the speech act (i.e., it says "Good bye!"). Then, another production is invoked that applies to G' and ensures that *P.rex* exits.

6.5.3 Interruptions

Similarly to the case of commands in Section 6.5.2, a new discourse structure node I is generated and included in the discourse structure tree when the user interrupts the system by entering a message $\mathcal{S}$. To represent the user's interruption in the discourse structure tree, the new node I has purpose (*Request* $\mathcal{S}$), content $\mathcal{S}$ and role *interruption*. Even though most messages that interrupt *P.rex* are notifications, we use *Request* to indicate that the messages play the role of requests. Recall that the user can also enter a command speech act to interrupt the system. In contrast to the case of commands entered while the system was idle, however, I is not inserted at the root of the tree, but either at the node the message $\mathcal{S}$ refers to, or—if $\mathcal{S}$ does not refer to a node—at the current node.

To trigger a reaction to the user's interruption, I is also pushed onto the goal stack. Productions specialized to the different interruptions are then invoked to plan the system's reaction. We shall now give a detailed description of these productions.

The Reaction to $\mathcal{S}$ = `(Obvious-Step :Conclusion` C`)`

The user informs the system that a proof step is obvious. The system should learn to omit such steps in the future. To this end, the following production pushes an appropriate dependency goal onto the goal stack:

(Ri1) IF the current goal is G : (*Request* (Obvious-Step :Conclusion C))
and inference rule R was used to explain G
THEN push the dependency goal Dependency shown in Figure 6.18.

Dependency is a dependency goal ensuring that a production is learned that will omit in the future any proof steps justified by rule R. It is shown in Figure 6.18.

```
Dependency
  isa           dependency
  goal          before
  modified      after
  constraints   ruleR
  specifics     (inform omit
                 known unknown
                 empty)
  dont-cares    (nil)

before
  isa           basic-node
  purpose       inform
  status        unknown
  rule          ruleR
  uncon-prems   empty

after
  isa           basic-node
  purpose       omit
  status        known
  unconveyed    empty

ruleR
  isa           rule
  name          R
```

Figure 6.18. The dependency goal and adjacent chunks to learn to omit a step. The slots uncon-prems and unconveyed correspond to u_P and u_C from Definition 6.5, respectively. Similarly, the slot rule corresponds to the rule in the justification J from Definition 6.5.

Recall from Section 5.4.2 that popping a dependency goal from the goal stack initiates the compilation of a new production that performs the problem solving step encoded in the dependency goal. Here, the before chunk stands for the goal before the application of the new production, whereas after reflects the situation afterwards. Hence, when Dependency is popped from the goal stack the following production is compiled:

```
IF    =goal>
        isa           basic-node
        purpose       inform
        status        unknown
        rule          =rule
        uncon-prems   empty
      =rule>
        isa           rule
        name          R
THEN  =goal>
        purpose       omit
        status        known
        unconveyed    empty
```

This production can be expressed more intuitively as follows:

IF the current goal is (*Inform* $\Gamma \vdash \varphi$)
and the most abstract rule justifying φ is R
and all premises are known
and the status of G is *unknown*
THEN set the purpose intention of G to *Omit*
and set the status of G to *known*.

Since this new production sets the purpose intention of the current goal to *Omit* and its status to *known* without triggering any verbalization, the corresponding step is de facto omitted.

The Reaction to $\mathcal{S}$ = (`Trivial-Derivation :Reasons` $P_1, \ldots, P_n$ `:Conclusion` C)

Now, the user tells the system, that the derivation of the conclusion C from the premises $P_1, \ldots, P_n$ is trivial. The system should learn to avoid the detailed presentation of such a derivation in the future. The following production marks the conclusion C of the derivation as inferable:

(Ri2) IF the current goal is
G : (*Request* `(Trivial-Derivation :Reasons` $P_1, \ldots, P_n$ `:Conclusion` C`)`)
and the premises $P_1, \ldots, P_n$ are known
THEN set the status of the fact C to *inferable*.

In subsequent situations, then, production (O2) from page 130 omits the derivation of C, since C is marked as inferable.

The Reaction to $\mathcal{S}$ = (`too-detailed :Conclusion` C)

When the user complains that the derivation of a conclusion C was too detailed, the dialog planner checks whether there is a higher level of abstraction on which C can be shown. If so, the corresponding higher level inference rule is marked as known, so that it is available for future explanations. Then, the explanation of the derivation of C is re-planned. Otherwise, the dialog planner paraphrases the used inference rule. This reaction of the system is depicted in Figure 6.19. Note that every arrow corresponds to a production. How the inference rule is paraphrased will be shown later in this section. An example dialog where the user complained that the original explanation of a proof was too detailed will be given in Example 6.9 in Section 6.5.4.

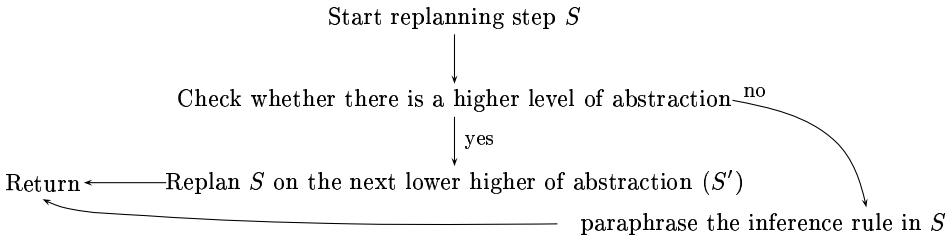


Figure 6.19. The reaction of the dialog planner if a step S was too detailed.

The Reaction to $\mathcal{S}$ = (`too-abstract :Conclusion` C)

When the user complains that the derivation of a conclusion C was too abstract, the dialog planner checks whether there is a lower level of abstraction on which C can be shown. If so, the inference rule used in the previous explanation is marked as unknown, so that it is not available for future explanations any more. Then, the explanation of the derivation of C is re-planned. Otherwise, the dialog planner paraphrases the used inference rule. This behavior of the system is depicted in Figure 6.20. Note that every arrow corresponds to a production. How the inference rule is paraphrased will be shown later in this section.

The Reaction to $\mathcal{S}$ = (`too-implicit :Conclusion` C)

When the user complains that the derivation of a conclusion C was too implicit, the dialog planner reverbalizes that step with explicit premises. This is encoded by the following production:

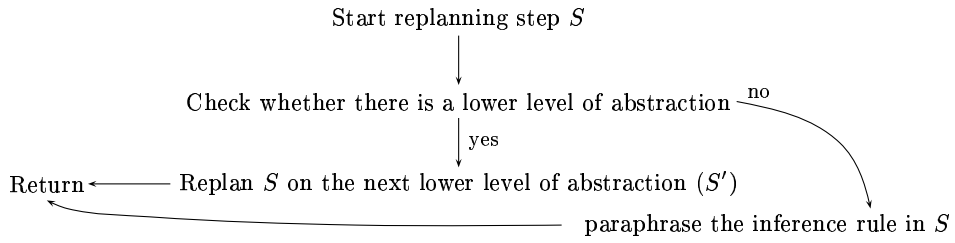


Figure 6.20. The reaction of the dialog planner if a step S was too abstract.

(Ri3) IF the current goal is
G : (*Request* `(too-implicit :Conclusion` C`)`)
THEN push the goal G' : (*Reverbalize* C).

The Reaction to $\mathcal{S}$ = `(too-difficult :Conclusion` C`)`

When the user complains that the derivation of a conclusion C was too difficult, the dialog planner enters a complex clarification dialog to find out which part of the explanation failed to remedy this failure. The control of the behavior of the dialog planner is displayed in Figure 6.21, where every arrow corresponds to a production. Note that the dialog planner first tries to find out whether the previous explanation was too implicit or too abstract and then reacts accordingly.

To elucidate the diagram in Figure 6.21, an example dialog where the user complained that the original explanation of a proof was too difficult will be given in Example 6.9a in Section 6.5.4.

Paraphrasing Inference Rules

When the system is confronted with user messages `too-detailed`, `too-abstract` or `too-difficult`, and no alternative explanation can be generated, the dialog planner paraphrases the proof step under consideration as shown in Figure 6.22. Again, every arrow corresponds to a production. The paraphrase consists in notifying the user either that the step is hypothesis, or that no more respectively less abstract explanation can be generated.

The Reaction to $\mathcal{S}$ = `(Continue!)`

When the user tells the system to continue, the dialog planner ignores the interruption and continues with its previous task. Note that neither the interruption nor the speech act is included in the discourse structure tree.

The Reaction to $\mathcal{S}$ = `(Stop!)`

With this message, the user indicates that he wants the system to stop the current task. The system's reaction is mediated by the following production:

(Ri4) IF the current goal is (*Request* `(Stop!)`)
with parent D
THEN pop all goals from the goal stack
and push the goal to read a user command.

By applying this production the dialog planner empties goal stack and starts over.

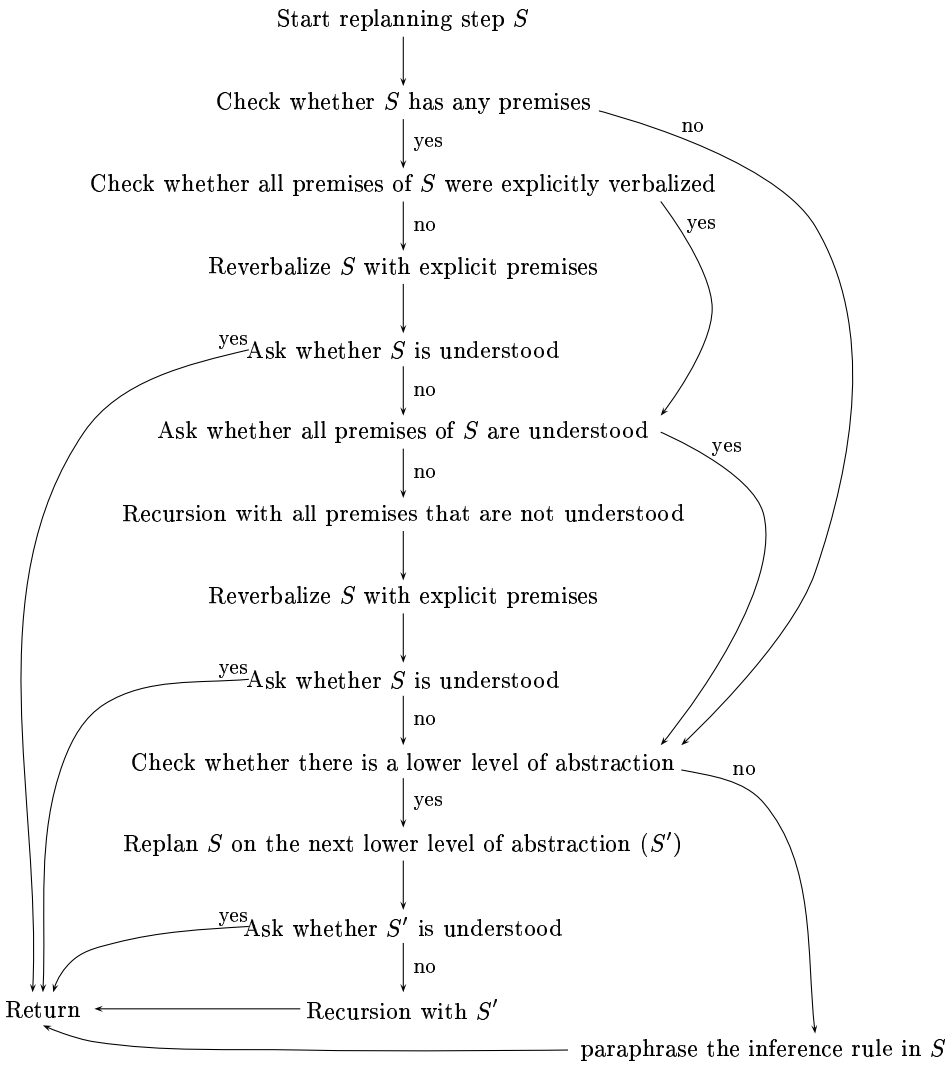


Figure 6.21. The reaction of the dialog planner if a step S was too difficult.

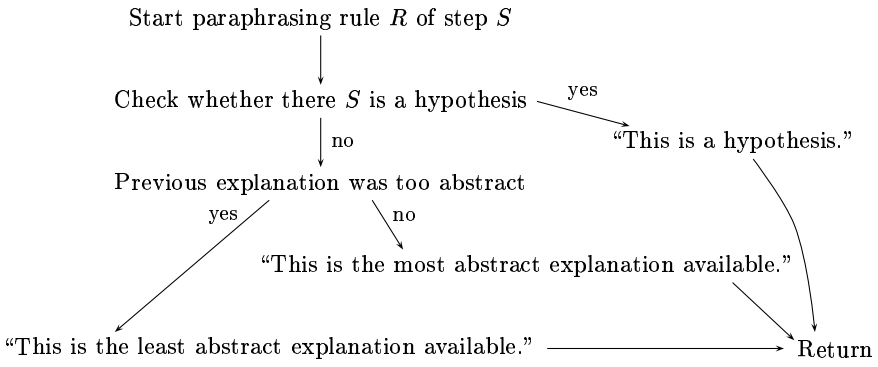


Figure 6.22. The paraphrasing of an inference rule.

6.5.4 Example Dialogs

Let us examine more closely how *P.rex* reacts to user interaction with the help of two example dialogs. In first dialog, the user complains that the explanation is too detailed. In the second dialog, in contrast, the user complains that the explanation is too difficult.

The following example will elucidate the reaction of the dialog planner as shown in Figure 6.19.

Example 6.9
Let Σ^{ND} be the signature encoding the ND calculus as given in Table 3.6 on page 36. As in Example 6.4, we add the following declarations to Σ^{ND} (for the sake of readability, we omit implicit arguments to inference rules and show instead only their premises and conclusions):

$$\begin{array}{ll} \in & : \mathsf{i} \to (\mathsf{i} \to \mathsf{o}) \to \mathsf{o} \\ \cup & : (\mathsf{i} \to \mathsf{o}) \to (\mathsf{i} \to \mathsf{o}) \to (\mathsf{i} \to \mathsf{o}) \\ \mathsf{Def}\cup_1 & : \mathsf{nd}(x \in S) \to \mathsf{nd}(x \in S \cup S') \\ \mathsf{Def}\cup_2 & : \mathsf{nd}(x \in S') \to \mathsf{nd}(x \in S \cup S') \end{array}$$

Recall that the inference rule $\vee E$ is encoded by

$$\mathsf{ore} : \mathsf{nd}(A \vee B) \to (\mathsf{nd}A \to \mathsf{nd}C) \to (\mathsf{nd}B \to \mathsf{nd}C) \to \mathsf{nd}C \in \Sigma.$$

We also add the following constant definition to our signature:

$$\begin{array}{lll} \cup\mathsf{Lemma} & = & \lambda u : \mathsf{nd}(x \in S \vee x \in S').\mathsf{ore}\ u \\ & & \qquad (\lambda u_1 : \mathsf{nd}(x \in S).\mathsf{Def}\cup_1\ u_1) \\ & & \qquad (\lambda u_2 : \mathsf{nd}(x \in S').\mathsf{Def}\cup_2\ u_2) \\ & & : \mathsf{nd}(x \in S \vee x \in S') \to \mathsf{nd}(x \in S \cup S') \end{array}$$

Hence, in terms of the ND calculus, the corresponding inference rule $\cup$-Lemma

$$\frac{\vdash x \in S \vee x \in S'}{\vdash x \in S \cup S'} \cup\text{-Lemma}$$

is a derived rule with the expansion

$$\frac{\vdash x \in S \vee x \in S' \quad \dfrac{[\vdash x \in S]^{u_1}}{\vdash x \in S \cup S'}\mathrm{Def}\cup_1 \quad \dfrac{[\vdash x \in S']^{u_2}}{\vdash x \in S \cup S'}\mathrm{Def}\cup_2}{\vdash x \in S \cup S'} \vee E^{u_1,u_2}$$

Now, let us consider the following situation:

- The top goal on the goal stack is

  $$G : \Gamma \vdash \cup\mathsf{Lemma}(H) : \mathsf{nd}(a \in U \cup V)$$

  with expansion

  $$\begin{array}{l} \Gamma \vdash \mathsf{ore}\ H \\ \qquad (\lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def}\cup_1\ H_1) \\ \qquad (\lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def}\cup_2\ H_2) \\ \quad : \mathsf{nd}(a \in U \cup V) \end{array}$$

  where $\Gamma = a : \mathsf{i}, U : \mathsf{i} \to \mathsf{o}, V : \mathsf{i} \to \mathsf{o}, H : \mathsf{nd}(a \in U \vee a \in V)$.

  The next goal on the stack is

  $$G' : \Gamma' \vdash \cup\mathsf{Lemma}(H') : \mathsf{nd}(a \in F \cup G)$$

with expansion

$$\begin{array}{l}\Gamma' \vdash \mathsf{ore}\ H' \\ \qquad (\lambda H_1' : \mathsf{nd}(a \in F).\mathsf{Def} \cup_1 H_1') \\ \qquad (\lambda H_2' : \mathsf{nd}(a \in G).\mathsf{Def} \cup_2 H_2') \\ \quad : \mathsf{nd}(a \in F \cup G)\end{array}$$

where $\Gamma' = a{:}\mathsf{i}, F{:}\mathsf{i} \to \mathsf{o}, G{:}\mathsf{i} \to \mathsf{o}, H' : \mathsf{nd}(a \in F \vee a \in G)$.

- The rules $\vee E$ and Def$\cup$ are known, the rule $\cup$-Lemma is unknown.
- Each declaration in Γ and Γ' has been shown earlier. In particular, the user already knows that

$$\begin{array}{ll}\Gamma \vdash H : \mathsf{nd}(a \in U \vee a \in V) & \text{and} \\ \Gamma' \vdash H' : \mathsf{nd}(a \in F \vee a \in G). & \end{array}$$

- The cases of both case analyses, that is,

$$\begin{array}{l}\Gamma \vdash \lambda H_1 : \mathsf{nd}(a \in U).\mathsf{Def} \cup_1 H_1 : \mathsf{nd}(a \in U) \to \mathsf{nd}(a \in U \cup V), \\ \Gamma \vdash \lambda H_2 : \mathsf{nd}(a \in V).\mathsf{Def} \cup_2 H_2 : \mathsf{nd}(a \in V) \to \mathsf{nd}(a \in U \cup V), \\ \Gamma' \vdash \lambda H_1' : \mathsf{nd}(a \in F).\mathsf{Def} \cup_1 H_1' : \mathsf{nd}(a \in F) \to \mathsf{nd}(a \in F \cup G), \quad \text{and} \\ \Gamma' \vdash \lambda H_2' : \mathsf{nd}(a \in G).\mathsf{Def} \cup_2 H_2' : \mathsf{nd}(a \in G) \to \mathsf{nd}(a \in F \cup G)\end{array}$$

  are unknown.

Note that G is the current goal. Since $\cup$-Lemma is unknown to the user, $\vee E$ is the most abstract known rule justifying the current goal. Thus, *P.rex* explains G with the case analysis as in Example 6.4d, that is, the following speech acts are produced (we omit the discourse structure trees in this example):

```
(Case-Analysis :Goal nd(a ∈ U ∪ V) :Cases (nd(a ∈ U), nd(a ∈ V)))
  (Case :Number 1 :Hypothesis nd(a ∈ U))
  (Derive :Reasons (nd(a ∈ U)) :Conclusion nd(a ∈ U ∪ V)
          :Method Def∪)
  (End-Case)
  (Case :Number 2 :Hypothesis nd(a ∈ V))
  (Derive :Reasons (nd(a ∈ V)) :Conclusion nd(a ∈ U ∪ V)
          :Method Def∪)
  (End-Case)
(End-Case-Analysis :Goal nd(a ∈ U ∪ V))
```

Suppose now that the user interrupts the system throwing in that the presentation is too detailed. Then, the analyzer passes the speech act `(too-detailed :Conclusion` $a \in U \cup V$`)` to the dialog planner. Since the inference rule $\cup$-Lemma, which is more abstract than $\vee E$, also justifies the conclusion, the $\cup$-Lemma is marked as known and the goal G is again pushed onto the goal stack. Then, (P2) is the only applicable production. Since $\cup$-Lemma is more abstract than $\vee E$ and both are known, it is chosen to instantiate (P2). Hence, the dialog planner produces the MCA

```
(Derive :Reasons (a ∈ U ∨ a ∈ V) :Conclusion a ∈ U ∪ V
        :Method ∪-Lemma)
```

Since $\cup$-Lemma is now marked as known to the user, it is also used for presentation in subsequent situations. In particular, when G' is to be fulfilled, $\cup$-Lemma is the most abstract rule known to the user, such that (P2) is the only applicable production. The whole dialog takes place as follows:

***P.rex*:** In order to prove that $a \in U \cup V$ let us consider the following cases.

Case 1: Let $a \in U$. Then $a \in U \cup V$ by the definition of $\cup$.

Case 2: Let $a \in V$. That implies that $a \in U \cup V$ by the definition of $\cup$.

User: This derivation is too detailed.

***P.rex*:** Since $a \in U$ or $a \in V$, $a \in U \cup V$ by the $\cup$-Lemma. Since $a \in F$ or $a \in G$, $a \in F \cup G$ by the $\cup$-Lemma.

□

Let us now turn our attention to the case where the user complains that the explanation is too difficult. To elucidate the behavior of the dialog planner as depicted in Figure 6.21, let us examine the following example:

Example 6.9a (continued)
Let us again assume the initial situation from Example 6.9, but now, the user is assumed to know $\cup$-Lemma. This time, the only applicable production is (P2). Since $\cup$-Lemma is more abstract than $\vee E$ and both are known, it is chosen to instantiate (P2). Hence, the dialog planner produces the MCA

`(Derive :Reasons (`$a \in U \vee a \in V$`) :Conclusion` $a \in U \cup V$
`:Method` $\cup$-Lemma`)`

Suppose now that the user points to this utterance and interrupts *P.rex* throwing in that this step was too difficult. The analyzer translates the user's interaction to the speech act `(too-difficult :Conclusion` $a \in U \cup V$`)`. Now, the dialog planner enters the clarification dialog as displayed in Figure 6.21. Since all premises were explicitly mentioned, the system does not reverbalize the step, but asks whether all premises are understood, what the user affirms. Hence, the system checks whether there is a lower level of abstraction, at which the step can be presented. Since this is the case, *P.rex* replans the explanation of the step by marking the inference rule $\cup$-Lemma as unknown and pushing the goal G again onto the goal stack. Now, (P2) is not applicable, since $\cup$-Lemma is unknown, but (P1) and (C2) are applicable. Hence, the system proceeds as in Example 6.4d and verbalizes the case analysis. Since $\cup$-Lemma is unknown, it is not used in subsequent situations either. Therefore, goal G' is also explained using the case analysis. The whole dialog takes place as follows:

***P.rex*:** Since $a \in U$ or $a \in V$, $a \in U \cup V$ by the $\cup$-Lemma.

User: This step is too difficult.

***P.rex*:** Do you understand the premises?

User: Yes.

***P.rex*:** In order to prove that $a \in U \cup V$ let us consider the following cases.

Case 1: Let $a \in U$. That leads to $a \in U \cup V$ by the definition of $\cup$.

Case 2: Let $a \in V$. Then $a \in U \cup V$ by the definition of $\cup$.

Do you understand this step?

User: Yes.

***P.rex*:** In order to prove that $a \in F \cup G$ let us consider the following cases.

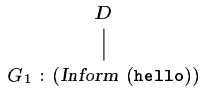
Case 1: Let $a \in F$. Therefore $a \in F \cup G$ by the definition of $\cup$.

Case 2: Let $a \in G$. Then $a \in F \cup G$ by the definition of $\cup$.

□

6.5.5 The Initiation of a Discourse

When *P.rex* is launched, the dialog planner starts a new discourse. To do so, before the production cycle starts, the dialog planner pushes the goal G_0 to read a user command. Then, it creates the following initial discourse structure tree



and pushes G_1 onto the goal stack. Note that D is the root of the discourse structure tree. Then, the dialog planner starts the production cycle. Since G_1 is on top of the goal stack, it is the current goal. Hence, the dialog planner invokes production (I5) and, thus, says "Hello." Then, the goal G_0 to read a user command becomes the current goal, that is, *P.rex* waits for the user to tell it the next task.

6.6 Discussion

The dialog planner of *P.rex* is a hybrid planner, which combines planning with discourse relations as captured in productions with planning with schemata. To represent the dialog, it constructs a discourse structure tree that combines the RST-like approach of Hovy [1993] with the concept of attentional spaces from Grosz and Sidner [1986].

Note that the attentional spaces are employed to model the salience of the information. Based on the information in which focus space a fact was derived, the dialog planner decides how the fact can be referred to. Do not confuse this with the forgetting of information modeled by ACT-R via the retrieval threshold. A fact that cannot be retrieved anymore is derived anew by the dialog planner.

Beside adapting the degree of explicitness taking into account the attentional spaces, the dialog planner also adapts to the user by explaining the proof at the highest level of abstraction that it assumes to be known to the user and by omitting steps that it assumes the user can easily infer. Moreover, it combines two different presentation strategies depending on how familiar with the current subject the user is. Whereas a textbook-style presentation strategy is used for those parts of the proof where the user is more knowledgeable, a classroom-style explanation strategy is used for the unfamiliar parts.

The system allows the user to interrupt anytime if he is not satisfied with the current explanation and reacts to the interruption by accordingly replanning the parts of the proof the user complained. The dialog planner uses the structural information that is explicitly represented in the discourse structure trees to identify which parts of the explanation failed to convey successfully. Those parts are then replanned.

With the current productions, *P.rex* plans its explanations locally, that is, it examines isolated steps of the derivation and decides how to present them. However,

to implement global planning it is easily possible to define productions that examine a whole subproof by considering not only the top symbol of the proof term, which corresponds to the last step of a derivation, but also its subterms, which correspond to subderivations (cf. Example 6.1 on page 96 for the relationship between subderivations and subterms in TWEGA). Such global planning should prove useful in situations where one step cannot be well presented without taking into account its surroundings. For example, when nested case analyses occur in a formal proof they are often preferably presented as one large flat case analysis. Only global planning can detect that a case analysis contains further nested case analyses and present them as one large case analysis.

Initially, *P.rex* is equipped with 91 domain independent productions, that is, they are independent of the represented logic and the mathematical theory under consideration. These productions are listed in Appendix C.2.2. Further domain dependent productions are added by the definition of schemata tailored to the mathematical theory under consideration. Moreover the system can learn new productions via the production compilation process.

In this chapter, we discussed the dialog planner, which provides a discourse plan as a representation of the dialog. In the following chapter, we shall describe further components of *P.rex* that ensure that the information in the discourse plan is conveyed to the user and that his interactions are accepted and analyzed.

Chapter 7

Front End Components

As discussed in Chapter 2 and, in more detail, in Chapter 6, the dialog planner is the central component of *P.rex*. It plans and structures the overall discourse between the system and the user, essentially by placing speech acts in a discourse structure tree. However, the speech acts still need to be conveyed to the user and the user must be allowed to make utterances himself, which in turn are to be transformed into speech acts to be included in the discourse structure tree.

In *P.rex*, the dialog planner pipes the speech acts to the generation components, which verbalize them by producing appropriate sentences. Section 7.1 is devoted to the generation components. The sentences are then passed on to the user interface. The user interface, which we shall present in Section 7.2, displays the system's utterances and accepts the user's utterances. The user's utterances are passed to the analyzer, which interprets them in terms of speech acts. The analyzer, which is the subject of Section 7.3, pipes the speech acts to the dialog planner for inclusion in the discourse structure tree and further processing.

7.1 The Generation Components

As discussed in great detail in Chapter 6, the dialog planner produces a dialog plan based on decisions that concern the presentation. More detailed linguistic decisions are made by the *sentence planner*. It makes reference choices, chooses between linguistic resources for domain concepts, and combines and reorganizes such resources into paragraphs and sentences. To include a sentence planner in *P.rex*, we adapted *PROVERB*'s micro-planner [Huang and Fiedler, 1997; Fiedler, 1996] and enriched its linguistic resources to support the extended set of speech acts used by *P.rex*. In this section, we shall briefly sketch the sentence planner.

To allow for all previously mentioned operations, the sentence planner is based on an intermediate representation called *text structure*, which was initially proposed by Meteer [1992]. The text structure reflects linguistic constraints, while abstracting away from syntactic detail. Similarly to the discourse structure tree, which reflects the constituency of the segments of the discourse, the text structure is a tree that reflects the constituency of words and phrases in a sentence.

To specify how subtrees of the text structure may be combined, we consider two orthogonal dimensions of semantic categories. Each text structure node is classed with respect to the *ideational* dimension in terms of the upper model, and the *textual* dimension in terms of textual semantic categories.

The *upper model* [Bateman *et al.*, 1990] is a domain-independent property inheritance network of concepts that are hierarchically organized according to how they can be linguistically expressed. Figure 7.1 shows a fragment of the upper model in

P.rex. For every concept used in the speech acts, a domain concept is inserted to the upper model, primarily including the rhetorical relations, and the logic predicate and function symbols.

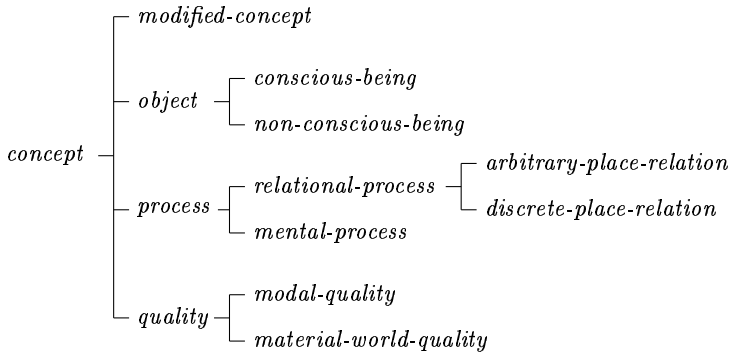


Figure 7.1. A fragment of the upper model in *P.rex*.

The hierarchy of *textual semantic categories* [Panaget, 1994] is also a domain-independent property inheritance network. The concepts are organized in a hierarchy based on their textual realization. For example, the concept clause-modifier-rankingI is realized as an adverb, clause-modifier-rankingII as a prepositional phrase, and clause-modifier-embedded as an adverbial clause. Figure 7.2 shows a fragment of the hierarchy of textual semantic categories.

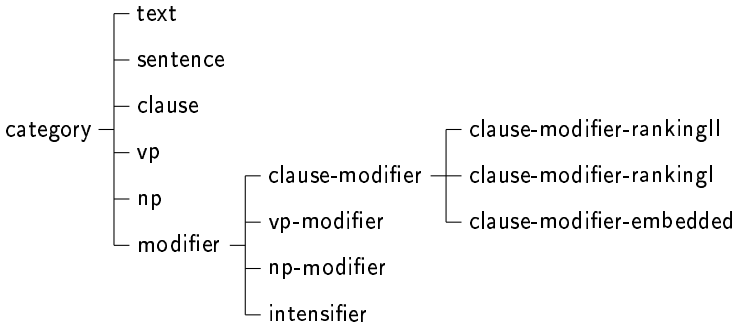


Figure 7.2. A fragment of the hierarchy of textual semantic categories in *P.rex*.

The content of the text structure nodes can be an application object (such as a speech acts or a formula) or an upper model object. Starting from a single-node text structure containing a list of speech acts, the sentence planner progressively maps application objects into appropriate linguistic realizations. This is done in two stages: First, the sentence planner decides for the current text structure leaf node to which of possibly several allowed upper model object its application object is mapped. Then, taking into account the set of allowed textual semantic categories for the leaf node, a reified instance of a text structure subtree is chosen to expand it. Thus, the text structure evolves by expanding leaves top-down and from left to right. While doing so, the sentence planner draws on various rules, including aggregation to remove redundancies, insertion of cue words to increase coherence, lexical choices, sentence scoping and layout. A more detailed discussion of the sentence planner is given in [Huang and Fiedler, 1997]. A thorough description can be found in [Fiedler, 1996].

A text structure constructed in this way is the output of the sentence planner, and is transformed into the input formalism of TAG-GEN [Kilger and Finkler, 1995], the linguistic realizer we use in *P.rex*. TAG-GEN, which is based on the grammar formalism of tree adjoining grammars [Joshi, 1985], then produces the surface sentences taking into account the morphology of the individual words. The surface sentences are finally displayed to the user via the user interface, which is the subject of the following section.

7.2 The User Interface

Having described the generation components of *P.rex*, we now turn our attention to the component that mediates between the system and the user, the *user interface*. The task of the user interface is twofold. First, it exposes to the user the system's utterances, which it receives from the linguistic realizer. Second, it accepts the user's interactions and either passes them on to the analyzer or executes them directly.

When we design the user interface, we must bear in mind that *P.rex* is devised as a generic system that can be connected to different theorem provers. Many automated theorem provers (e.g., Otter [McCune, 1994], SPASS [Weidenbach, 1997]) have only simple file interfaces and are not necessarily optimized for human interaction. When used with such systems *P.rex* needs an elaborate user interface to allow the user the convenient interaction with the proof explanation system. In contrast, other theorem provers, especially interactive systems (e.g., ΩMEGA [Benzmüller *et al.*, 1997], *Theorema* [Buchberger, 1997]), have elaborate graphical user interfaces, which allow the user to interact with the system easily. In such an environment, a generic interface that can be integrated easily into the existing graphical user interface is to be preferred.

Therefore, we equip *P.rex* with a *generic interface* that can be connected to the existing user interface of a theorem prover. The generic interface will be defined in Section 7.2.1. Moreover, for the direct use by the human user, *P.rex* is distributed with an Emacs interface, which is built on top of the generic interface. We shall describe the Emacs interface in Section 7.2.2. We also used the generic interface to couple *P.rex* with the proof development system ΩMEGA [Benzmüller *et al.*, 1997] via a socket connection.

7.2.1 The Generic Interface

We define the generic interface in terms of the *output* produced by *P.rex* and the *input* it allows the user to enter. Thereby, we distinguish between two types of input: Whereas input of the first type, an *instruction*, is directly processed by the interface, input of the second type, a *dialog turn*, is passed on to the analyzer for further interpretation. Any input of the second type is considered as quasi–natural-language input and therefore also added in natural language in the output, such that the output contains the complete dialog. That is, the generic interface outputs both the system's utterances and the user's utterances. In addition, when entering a command or an interruption, the user can refer to an utterance made by the system previously.

Let us first define the output of the generic interface.

Definition 7.1 The *output* of the generic interface is a triple (S, id, s), where $S \in \{\texttt{system}, \texttt{user}\}$ is the speaker, id is the identifier of a discourse structure node and s is the uttered sentence. ∎

Since the generic interface outputs both the system's and the user's utterances, S can be used to distinguish the utterances from both interlocutors—for example,

by using different typefaces or colors. The identifier *id* stands for the discourse structure node that contains the speech act that represents the utterance. The sentence s, finally, is the output string of the linguistic realizer.

Technically, the output of the generic interface is a single string, whose syntax is given by the following grammar:

output	::=	*speaker id sentence*
speaker	::=	`system` | `user`
id	::=	`NIL` | *symbol*

where *sentence* is an output string of the linguistic realizer and *symbol* is the identifier of a discourse structure node.

Now, let us define the input to the generic interface.

Definition 7.2 The *input* to the generic interface is a triple (A, a, s), where a is the action of type A taken by the user and s the arguments to a. ■

The action a that can be taken is either an instruction or a dialog turn. *Instructions* are technical commands that are considered as outside the dialog, such as the instruction to load a TWEGA signature, in which a proof is represented. The instructions are appropriately processed by the generic interface directly. *Dialog turns* in contrast, are considered part of the dialog. Each dialog turn entered by the user is passed on to the analyzer without further examination. This has the advantage that the replacement of the current analyzer by a more powerful one does not effect the generic interface. The dialog turns will be defined formally as input to the analyzer in Section 7.3.

Whether the action a is an instruction or a dialog turn is annotated by its type A. The arguments s to a are defined with respect to a. This is done in Appendix D.1 for instructions and in Section 7.3 for dialog turns. To refer to an utterance made previously by the system, the user can include in s the identifier of the corresponding discourse structure node as given in the output.

Technically, the input to the generic interface is a single string, whose syntax is given by the following grammar:

input	::=	*type-action arguments*
type-action	::=	`instruction` *instruction* | `dialog` *dialog_turn*
instruction	::=	`NewProof` | `ReadSgnEntry` | `ReadInfoEntry` | `ReadJudgment` | `Reset` | `SetTheory` | `Load`

where *arguments* is a string of arguments to the action and *dialog_turn* is a dialog turn as will be defined in Section 7.3.

7.2.2 The Emacs Interface

The distribution of *P.rex* features an Emacs interface, which is built on top of the generic interface. It uses an Emacs buffer where the dialog is displayed and where the input can be entered by the user. In this section, we shall give an overview of the Emacs interface.

The Emacs interface accepts the output of the generic interface, say (S, id, s), and displays the sentence s in the Emacs buffer. To account for the speaker S, it uses different colors for the sentences, for example, black for the system and brown for the user (cf. Figure 7.3). Moreover, it stores the relationship between the sentence s and the identifier *id* of the discourse structure node that contains the speech act that represents s. Furthermore, it keeps track of where in the buffer the sentences are displayed.

The Emacs interface parses the sentence s for any PML directives and interprets them appropriately. As a sublanguage of the Hypertext Markup Language HTML,

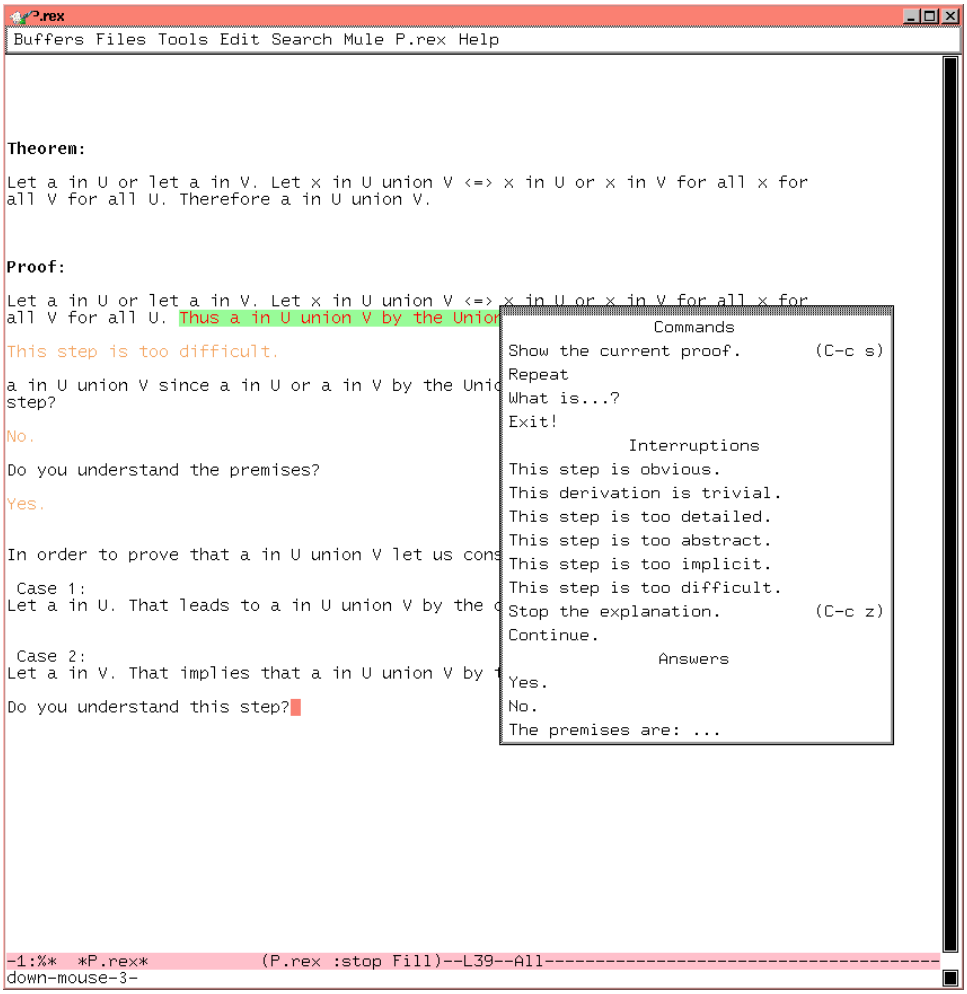


Figure 7.3. A screen shot of the Emacs interface to *P.rex*.

the *P.rex Markup Language (PML)* allows for the inclusion of layout information in the sentence. For example, `<B>` starts writing the text in boldface, `</I>` ends writing the text in italics and `<FONT color=red>` starts writing the text colored red. PML is formally defined in Appendix D.2.

Since the current analyzer of *P.rex* is not powerful enough to understand natural language, the user cannot enter free text or speech. Therefore, he is not allowed to type directly into the Emacs buffer. Instead, the Emacs interface allows the user to interact with the system by mouse events or key strokes. Instructions can be chosen by using a pop-up menu. Dialog turns can be entered via a context menu. The menu entries for dialog turns are given in natural language (e.g., "This step is too difficult." or "This derivation is trivial.", cf. Figure 7.3). All instructions and some commands can also be invoked by key strokes. The menu choices and key strokes are transformed appropriately and piped into the input of the generic interface.

When the user chooses a dialog turn that includes a reference expression he has to point to the sentence that verbalizes the appropriate step. Since the Emacs interface stored the identifier of the discourse structure node that contains the speech act that verbalizes the sentence, it can add this identifier to the input string to be passed to the generic interface. For example, let `DS123` be the discourse structure node with

speech act (Derive :Reasons (nd($a \in U \vee a \in V$)) :Conclusion nd($a \in U \cup V$) :Method $\cup$-Lemma), which was verbalized earlier as "Thus, $a \in U \cup V$ by the $\cup$-Lemma." When the user points with the mouse to that sentence and simultaneously chooses "This step is too difficult." from the context menu by clicking with the right mouse button, the Emacs interface pipes the string "dialog too-difficult DS123" into the input of the generic interface.

Figure 7.3 shows a screen shot of the Emacs interface. Note that the system's utterances are typed in black, whereas the user's utterances are brown. The sentence to which the mouse is pointing is highlighted using red letters on a green background. Since the user was pressing the right mouse button while pointing at the sentence, the context menu is displayed to allow him to choose an utterance. Note that the context menu items, which are displayed in natural language, are ordered according to their type (that is, whether they are commands, interruptions or answers). The pop-up menu with the instructions can be accessed via the menu bar entry "P.rex" at the top of the Emacs window.

Note that the Emacs interface described in this section is one possible solution, which is tailored to the current output and input capabilities of *P.rex*. In particular, if the analyzer were to be replaced by a more powerful one, the Emacs interface might need to be adapted as well.

7.3 The Analyzer

Since the development of a natural language analysis component is far beyond the scope of this work, we are content for the time being with a simplistic analyzer that produces speech acts from a small set of allowed input objects. In this section, we shall describe this analyzer.

The analyzer receives its input from the generic interface and transforms it to speech acts.

Definition 7.3 Let (dialog, a, s) be the input to the generic interface. Then, (a, s) is the *input* to the analyzer. ■

Recall from Section 7.2.1 that a is a *dialog turn*. Possible dialog turns are *commands*, *interruptions* and *answers*, that is, the action taken is mapped by the analyzer to the command, interruption and answer speech acts as defined in Section 6.5.1.

Technically, the input to the analyzer is a single string, whose syntax is given by the following grammar:

input	::=	*dialog_turn arguments*
dialog_turn	::=	*command* | *interruption* | *answer*
command	::=	Show! | Repeat! | what-is? | Exit!
interruption	::=	Obvious-Step | Trivial-Derivation | too-detailed | too-abstract | too-implicit | too-difficult | Stop! | Continue!
answer	::=	yes | no | premises

where *arguments* is a string of arguments to the action.

In the following, we use ϵ to denote empty arguments. The mapping from the analyzer's input to its output is given as follows:

Commands

Show!

Let (Show!, s) be the input, where $s = (P, S)$ with a proof name P and a strategy $S \in \{Explain, Present\}$. Then, the analyzer returns the speech act (Show! :Proof P :Strategy S).

`Repeat!`
: Let (`Repeat!`, P) be the input, where P is a proof name. Then, the analyzer returns the speech act (`Repeat! :Proof` P).

`what-is?`
: Let (`what-is?`, s) be the input, where $s = (A, id)$ for an object A and the identifier id of a discourse structure node N. Then, the analyzer returns the speech act (`what-is? :Object` A `:ds-node` N).

`Exit!`
: Let (`Exit!`, ϵ) be the input. Then, the analyzer returns the speech act (`Exit!`).

Interruptions

`Obvious-Step`, `Trivial-Derivation`, `too-detailed`, `too-abstract`, `too-implicit`, `too-difficult`
: Let (S_A, id) be the input to the analyzer, where $S_A \in$ {`Obvious-Step`, `Trivial-Derivation`, `too-detailed`, `too-abstract`, `too-implicit`, `too-difficult`} and id is the identifier of a discourse structure node N that contains a speech act. Furthermore, let C be the conclusion that was verbalized by the speech act contained in N. Then, the analyzer returns the speech act (S_A `:Conclusion` C).

`Continue!`, `Stop!`
: Let (S_A, ϵ) be the input where $S_A \in$ {`Continue!`, `Stop!`}. Then, the analyzer returns the speech act (S_A).

Answers

`yes`, `no`
: Let (S_A, ϵ) be the input where $S_A \in$ {`yes`, `no`}. Then, the analyzer returns the speech act (S_A).

`premises`
: Let (`premises`, l) be the input where l is a list of numbers. Then, the analyzer returns the speech act (`premises :items` l).

If the input can be successfully analyzed, the corresponding speech act is passed on to the dialog planner for the inclusion in the discourse structure tree and further processing. Moreover, it also piped to the sentence planner for verbalization and subsequent inclusion as a user's utterance in the output of the generic interface.

Chapter 8

Conclusion

This thesis is about a computational model of user-adaptive proof explanation, which is implemented in the generic proof explanation system *P.rex*.

The powerful logical framework TWEGA defines the interface via which theorem provers can be connected: The calculus of the prover, the input proof to be explained as well as relevant information from the mathematical theories that relate to the proof are represented in TWEGA. The representation of the proofs is according to the judgment-as-types paradigm. This paradigm not only guarantees the correctness of the proofs with respect to the calculus in which they were found, but is also well suited for the subsequent explanation because of its explicit representation of hypothetical and parametric judgments. Moreover, constant definitions in TWEGA allow for the simultaneous representation of several levels of abstraction of one proof, an important prerequisite for user-adaptive explanation.

The dialog planning is based on the cognitive architecture ACT-R, which allows us to combine user modeling and planning in a uniform framework. ACT-R is a production system that relates procedural knowledge in a production rule base to declarative knowledge in a declarative memory. We model the knowledge of the user in the declarative memory of ACT-R. In the production rule base, we model the system's knowledge of how to explain proofs.

Besides the user model, we also represent the discourse structure tree, which serves as both a dialog plan and a dialog history, in the declarative memory of ACT-R. The discourse structure tree combines the RST-like approach of [Hovy, 1993] with the attentional spaces of [Grosz and Sidner, 1986] and, thus, allows us to model both the segmentation of the discourse and the focus of attention in a uniform representation. Since the attentional spaces model salience, they are employed to decide on the degree of explicitness and to choose appropriate referring expressions. Explicit representation of the discourse purpose allows for presentations using different styles, such as a textbook-like style or a classroom-like style. Moreover, discourse structure trees also account for restricted types of dialogs as well, namely certain types of interruptions and clarification dialogs. This is a necessary prerequisite to enable the system to represent user interactions and to appropriately react to them.

The plan operators that modify the discourse structure tree are realized as productions. Meeting the characteristic of a hybrid planner, we distinguish between two types of plan operators: simple plan operators, which add single nodes to the discourse structure tree, and schemas, which add whole subtrees to the discourse structure trees. Using special plan operators, the dialog planner adapts to the user by explaining the proof at a level of abstraction and a degree of explicitness that is, according to the user model, the most appropriate for the current user. Moreover, depending on how familiar with the current subject the user is, it combines two

different presentation strategies (textbook-like and classroom-like styles), which are represented by overlapping sets of plan operators.

The system also allows for user interaction. The interactions are translated into new discourse goals, which the dialog planner tries to fulfill. In particular, the system is capable of dialog-driven user adaptation, that is, the user can intervene if he is not satisfied with an explanation and the system replans the complained parts of the proof. Then, similar proof parts are explained accordingly in future situations.

P.rex has been tested with proofs from group theory and the theory of first-order predicate logic. An in-depth case study with proofs from limit theory is under way.

Possible Extensions

P.rex has the potential to further improvements and extensions. The modular architecture of *P.rex* and the fixed interfaces between the modules ensure that the possible improvements and extensions discussed subsequently can be realized without major re-implementation.

Since *P.rex* cannot modify the input proofs, it depends on them in the quality of the explanations. In particular, proofs that are input only at a low level of abstraction can also be explained only at that low level. It is not possible to find a generic approach to achieve a logical abstraction of the input proof independent of the attached theorem prover, since such a logical abstraction can be defined only with respect to the calculus in which the proof was found. However, it is possible to define some conceptual notion of abstraction, which is independent of the calculus, but relative to the mathematical theory under consideration. This approach has been pursued in ΩMEGA [Benzmüller *et al.*, 1997].

P.rex allows for the use of mathematical formulae in natural language sentences. However, many concepts and ideas are much easier to understand when they are depicted graphically. The inclusion of graphs and diagrams is standard routine in mathematics communication. Therefore, the ability of multi-modal presentations is desirable for a proof explanation system as well. As [Wahlster *et al.*, 1993] points out, a three-staged architecture as realized in *P.rex* is also appropriate for multi-modal generation. To do so, a multi-modal extension of speech acts and productions specialized for planning graphics must be defined. Moreover, the sentence planner and linguistic realizer must be either appropriately extended or supported by graphics generation components. Finally, the user interface must be extended to support graphics as well.

To extend *P.rex* to a real dialog system, a more powerful analyzer that understands natural language is needed, and a speech-based user interface is desirable. Systems that transform natural language text into spoken language are already available. It should be only a minor problem to connect one to *P.rex*. However, an extension that only reads *P.rex* output to the user does not seem appropriate. Instead, *P.rex* should be extended to plan the speech separately from the text to imitate a teacher who does not read loud what we writes on the board, but who explains his writing using different words.

P.rex is a system to explain only proofs. However, an extension to also explain definitions and theorems and the structure of mathematical theories is desirable. An appropriately extended system should answer questions as why a concept is defined as it is if alternative definitions are possible. To generate such explanations additional knowledge about the mathematical theories is needed—knowledge that is usually not represented by theorem provers. Instead, the proposed extension aims at a tutorial system for mathematics that draws on a comprehensive database containing the mathematical theory under consideration.

Since ACT-R is a powerful theory of human cognition, it has the potential to allow for an extension of *P.rex* towards a tutorial system for mathematics. In *P.rex*, we already trace the user's cognitive states during an explanation. Clearly, it is also possible to trace a student's cognitive states when he is proving a theorem or solving a mathematical exercise. Moreover, ACT-R allows for the definition of an error model that captures misconceptions in problem solving methods. In addition, the functionality of partial matching in ACT-R can be employed to account for incorrect applications of problem solving methods by the student.

Availability

P.rex is implemented in Allegro Common Lisp with CLOS and has been tested on dozens of input proofs. The implementation (currently Version 1.0) can be accessed via the *P.rex* homepage at `http://www.ags.uni-sb.de/~prex`.

Appendix

Appendix A

Representing Proofs

This chapter is devoted to technical details concerning the representation of proofs in TWEGA that have not been presented in the main part of this thesis. In the first section, we shall give the concrete syntax of TWEGA. In the second section, we shall give the proofs of the lemmata and theorems that we omitted in Chapter 4.

A.1 The Concrete Syntax of TWEGA

Whereas the abstract syntax is a means that allows us to easily talk about objects, the concrete syntax determines how these objects must be written so that the system can understand them. The abstract syntax of TWEGA was given in Section 3.3.1 as follows:

$$\begin{array}{lll} \textit{Terms} & \mathcal{T} ::= \mathcal{V} \mid \mathcal{C} \mid \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T} \\ \textit{Signatures} & \textit{Sig} ::= \cdot \mid \textit{Sig}, \mathcal{C}{:}\mathcal{T} \mid \textit{Sig}, \mathcal{C}{=}\mathcal{T}{:}\mathcal{T} \end{array}$$

where $\mathcal{V}$ and $\mathcal{C}$ are infinite collections of variables and constants, respectively.

The concrete syntax of TWEGA is given by the following grammar. It defines the non-terminals `sig`, `decl`, `term` and uses the terminal `type`.

`sig`	::=		empty signature
	|	`decl sig`	signature entry
`decl`	::=	`name (num) : term`	constant declaration $\mathcal{C}{:}\mathcal{T}$
	|	`name (num) = term : term`	constant definition $\mathcal{C}{=}\mathcal{T}{:}\mathcal{T}$
`term`	::=	`type`	type
	|	`name`	variable $\mathcal{V}$ or constant $\mathcal{C}$
	|	`term term`	application $\mathcal{T}\mathcal{T}$
	|	`[name : term] term`	λ-abstraction $\lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T}$
	|	`{name : term} term`	Π-abstraction $\Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}$
	|	`term -> term`	$\mathcal{T}\to\mathcal{T}$ (i.e., $\Pi\_{:}\mathcal{T}.\mathcal{T}$)

As usual in type theory, `A -> B` is treated as an abbreviation for `{x : A} B`, where `x` does not occur in `B`.

The non-terminal `name` stands for a name string that must not contain colons or whitespaces. To distinguish names from reserved characters, the following strings are not allowed as name strings: '`{`', '`[`', '`(`', '`)`', '`]`', '`}`'. Furthermore, the names '`->`' and '`type`' are predefined and therefore also disallowed.

The non-terminal `num` stands for a natural number (including 0). It denotes the *implicit arguments* of the constant declaration or definition (cf. 3.4.1). Techni-

cally speaking, implicit arguments are those arguments that can be typed by type reconstruction. From a pragmatic point of view, implicit arguments do not carry intuitive information and are therefore omitted in the verbalization.

The following agreements, which are ordered by precedence, disambiguate the syntax:

1. Application is left associative and has the highest precedence.
2. '->' is right associative and has the second highest precedence.
3. ':' is left associative.
4. '{}' and '[]' are weak prefix operators.

Example A.1
As an example for a signature we give the TWEGA representation the ND calculus (cf. Section 3.4.1) enriched by the derived inference rule $\wedge Comm$ and a theorem stating the commutativity of the conjunction (cf. Section 4.1.3).

```
i (0) : type
o (0) : type
and (0) : o -> o -> o
imp (0) : o -> o -> o
or (0) : o -> o -> o
true (0) : o
forall (0) : (i -> o) -> o
exists (0) : (i -> o) -> o
nd (0) : o -> type
andi (2) : {A : o} {B : o} nd A -> nd B -> nd (and A B)
andel (2) : {A : o} {B : o} nd (and A B) -> nd A
ander (2) : {A : o} {B : o} nd (and A B) -> nd B
impi (2) : {A : o} {B : o} (nd A -> nd B) -> nd (imp A B)
impe (2) : {A : o} {B : o} nd (imp A B) -> nd A -> nd B
oril (2) : {A : o} {B : o} nd A -> nd (or A B)
orir (2) : {B : o} {A : o} nd B -> nd (or A B)
ore (3) : {A : o} {B : o} {C : o} nd (or A B) ->
          (nd A -> nd C) -> (nd B -> nd C) -> nd C
truei (0) : nd true
foralli (1) : {A : i -> o} ({a : i} nd (A a)) -> nd (forall A)
foralle (1) : {A : i -> o} nd (forall A) -> {T : i} nd (A T)
existsi (1) : {A : i -> o} {T : i} nd (A T) -> nd (exists A)
existse (2) : {A : i -> o} {C : o} nd (exists A) ->
              ({a : i} nd (A a) -> nd C) -> nd C
andcomm (2) = [A : o] [B : o] [u : nd (and A B)] andi B A (ander A B u)
                                                          (andel A B u)
              : {A : o} {B : o} nd (and A B) -> nd (and B A)
thm (2) = [P : o] [Q : o] impi (and P Q) (and Q P)
                               [u : nd (and P Q)] andcomm P Q u
          : {P : o} {Q : o} nd (imp (and P Q) (and Q P))
```

□

A.2 Correctness Proofs

In this section, we shall give the formal proofs of the lemmata and theorems we omitted in Chapter 4.

First, we shall prove the correctness of the encoding of terms:

Lemma 4.3 (Correctness for Terms) *Let $t \in \mathbf{T}$ be a valid term, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, t) = (\widehat{\Sigma}, U, V)$. Then the following hold:*

(i) $\Sigma \subseteq \widehat{\Sigma}$ *and* $\widehat{\Sigma}$ *is valid.*

(ii) $\Gamma \vdash_{\widehat{\Sigma}} U : V$ *and* $\Gamma \vdash_{\widehat{\Sigma}} V : \mathsf{type}$.

(iii) $U, V \in \mathcal{NF}(\mathfrak{T})$.

Proof: Similarly to the proof of Lemma 4.2, the assertions are shown simultaneously by induction over the construction of $\widehat{\Sigma}$, U, and V.

Case SYM**:**

(i) By definition, either $\Sigma''' = \Sigma''$ or $\Sigma''' = \Sigma'', c{:}T$. In the first case the assertion follows directly by Lemma 4.2. In the second case, clearly $\Sigma''' \subseteq \Sigma''$ and Σ'' is valid by Lemma 4.2. To show the validity of Σ''' we only have to show that $\vdash_{\Sigma''} T : s$ for an $s \in \mathcal{S}$. This follows directly by Lemma 4.2, since T does not contain any free variables.

(ii) The second part of the assertion follows from the second premise of SYM by applying Lemma 4.2.

Now, we prove the first part of the assertion.

By rule declstart or start, respectively, we obtain

$$\Gamma \vdash_{\Sigma'''} c : T \tag{A.1}$$

Recall that $T = \lambda\overline{\alpha_n : \mathsf{hol}}.\mathsf{obj}\, S$. For $n = 0$, $T = \mathsf{obj}\, S = \mathsf{obj}\, T'$ and we are done.

Now, let us consider the case where $n > 0$.

Since $\tau' = \tau''[\tau_1/\alpha_1] \ldots [\tau_n/\alpha_n]$ we have $T' = S[T'_1/\alpha_1] \ldots [T'_n/\alpha_n]$ for some $T'_1, \ldots, T'_n$. The definition of the function $\mathcal{I}$ ensures, that $T_i = T'_i$ for $1 \leq i \leq n$. Hence, $T' = S[T_1/\alpha_1] \ldots [T_n/\alpha_n]$. Therefore, and since $\Gamma \vdash_{\Sigma'''} T : \mathsf{type}$ by Lemma 4.2, the judgments

$$\Gamma \vdash_{\Sigma'''} T_i : \mathsf{hol} \qquad \text{for } 1 \leq i \leq n \tag{A.2}$$

must be derivable. Hence, n applications of rule appl to (A.1) and (A.2) lead to

$$\Gamma \vdash_{\Sigma'''} cT_1 \ldots T_n : \mathsf{obj}\, T'$$

(iii) By induction hypothesis, $T' \in \mathcal{NF}(\mathfrak{T})$, and thus, $T_1, \ldots, T_n \in \mathcal{NF}(\mathfrak{T})$ as well. Therefore, on the one hand $\mathsf{obj}\, T' \in \mathcal{NF}(\mathfrak{T})$, and on the other hand $cT_1 \ldots T_n \in \mathcal{NF}(\mathfrak{T})$, since c is fully applied.

Case VAR**:**

(i) Since $\widehat{\Sigma} = \Sigma$, the assertion holds trivially.

(ii) The first part of the assertion follows directly by rule start.
By definition of VAR, $x{:}\mathsf{obj}\,T \in \Gamma$. Close inspection of the rules for $\mathcal{E}$ shows that this declaration could only be added to the context by rule ABSTR.
Application of Lemma 4.2 to the first premise of rule ABSTR gives us $\Gamma \vdash_\Sigma T : \mathsf{hol}$. By $\mathsf{declstart}$, we obtain $\Gamma \vdash_\Sigma \mathsf{obj} : \mathsf{hol} \to \mathsf{type}$. Hence, application of rule appl establishes that $\Gamma \vdash_\Sigma \mathsf{obj}\,T : \mathsf{type}$.

(iii) The assertion follows directly from case ABSTR.

Case APPL**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) By $\mathsf{declstart}$, $\Gamma \vdash_{\Sigma''} \mathsf{ap} : \Pi\alpha{:}\mathsf{hol}.\Pi\beta{:}\mathsf{hol}.\mathsf{obj}\,(\alpha \Rightarrow \beta) \to \mathsf{obj}\,\alpha \to \mathsf{obj}\,\beta$. By induction hypothesis, $\Gamma \vdash_{\Sigma''} u : \mathsf{obj}\,(T' \Rightarrow T)$ and $\Gamma \vdash_{\Sigma''} u' : \mathsf{obj}\,T'$. Thus, there must be judgments $\Gamma \vdash_{\Sigma''} T : \mathsf{hol}$ and $\Gamma \vdash_{\Sigma''} T' : \mathsf{hol}$. Hence, fourfold application of rule appl gives us $\Gamma \vdash_{\Sigma''} \mathsf{ap}\,T'\,T\,u\,u' : \mathsf{obj}\,T$.
Since $\Gamma \vdash_{\Sigma''} \mathsf{obj} : \mathsf{hol} \to \mathsf{type}$ and $\Gamma \vdash_{\Sigma''} T : \mathsf{hol}$ as just argued, $\Gamma \vdash_{\Sigma''} \mathsf{obj}\,T : \mathsf{type}$ by rule appl.

(iii) Obvious, since $T', T, u, u' \in \mathcal{NF}(\mathfrak{T})$ by induction hypothesis and $\mathsf{ap}\,T'\,T\,u\,u'$ and $\mathsf{obj}\,T$ are fully applied.

Case ABSTR**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) By $\mathsf{declstart}$, $\Gamma \vdash_{\Sigma''} \Lambda : \Pi\alpha{:}\mathsf{hol}.\Pi\beta{:}\mathsf{hol}.(\mathsf{obj}\,\alpha \to \mathsf{obj}\,\beta) \to \mathsf{obj}\,(\alpha \Rightarrow \beta)$.
By induction hypothesis $\Gamma \vdash_{\Sigma''} T' : \mathsf{hol}$ and $\Gamma, x{:}\mathsf{obj}\,T' \vdash_{\Sigma''} u : \mathsf{obj}\,T$, since $\Gamma, x{:}\mathsf{obj}\,T'$ is a valid context by $\mathsf{ctxdecl}$. Since ΩMEGA does not allow for dependent types, x cannot occur in T. Hence, the judgment $\Gamma \vdash_{\Sigma''} T : \mathsf{hol}$ must be derivable. Application of rule abstr then gives us $\Gamma \vdash_{\Sigma''} \lambda x{:}\mathsf{obj}\,T'.u : \mathsf{obj}\,T' \to \mathsf{obj}\,T$. Thus, threefold application of rule appl establishes that $\Gamma \vdash_{\Sigma''} \Lambda T'\,T\,\lambda x{:}\mathsf{obj}\,T'.u : \mathsf{obj}\,(T' \Rightarrow T)$.
By $\mathsf{declstart}$, $\Gamma \vdash_{\Sigma''} \mathsf{obj} : \mathsf{hol} \to \mathsf{type}$ and $\Gamma \vdash_{\Sigma''} \Rightarrow : \mathsf{hol} \to \mathsf{hol} \to \mathsf{hol}$. As we argued previously, $\Gamma \vdash_{\Sigma''} T' : \mathsf{hol}$ and $\Gamma \vdash_{\Sigma''} T : \mathsf{hol}$. Hence, threefold application of rule appl gives us $\Gamma \vdash_{\Sigma''} \mathsf{obj}\,(T' \Rightarrow T) : \mathsf{type}$.

(iii) By induction hypothesis, $T', T, u \in \mathcal{NF}(\mathfrak{T})$. Hence, $\mathsf{obj}\,T, \mathsf{obj}\,T' \in \mathcal{NF}(\mathfrak{T})$ and $\lambda x{:}\mathsf{obj}\,T'.u \in \mathcal{NF}(\mathfrak{T})$. Therefore, $\Lambda\,T'\,T\,\lambda x{:}\mathsf{obj}\,T'.u \in \mathcal{NF}(\mathfrak{T})$ as well.

Case POLY**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) The assertion follows by n-fold application of rule abstr and rule (type,type) to the induction hypothesis.

(iii) Obvious. ∎

Next, we shall show the proof of the correctness of the encoding of derivations:

Lemma 4.5 (Correctness for Derivations) *Let ω be a PDS node, a justification sequence, or a PDS. Furthermore, let Σ be a valid signature with $\Sigma^{HOL} \subseteq \Sigma$ and Γ be a valid context, and let $\mathcal{E}(\Sigma, \Gamma, \omega) = (\widehat{\Sigma}, U, V)$. Then the following hold:*

(i) $\Sigma \subseteq \widehat{\Sigma}$ *and* $\widehat{\Sigma}$ *is valid.*

(ii) $\Gamma \vdash_{\widehat{\Sigma}} U : V$ *and* $\Gamma \vdash_{\widehat{\Sigma}} V : \mathsf{type}$.

(iii) $U, V \in \mathcal{NF}(\mathfrak{T})$.

Proof: Like in the proofs of Lemma 4.2 and Lemma 4.3, we prove the assertions simultaneously by induction over the construction of $\widehat{\Sigma}$, U, and V.

Case HJNODE**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) We have to show $\Gamma \vdash_{\Sigma_n} \lambda\overline{h_n : \mathsf{pf}\, T_n\, u_n}.l : \Pi\overline{h_n : \mathsf{pf}\, T_n\, u_n}.\mathsf{pf}\, T\, u$ and $\Gamma \vdash_{\Sigma_n} \Pi\overline{h_n : \mathsf{pf}\, T_n\, u_n}.\mathsf{pf}\, T\, u : \mathsf{type}$. Instead, we show the following stronger property:

Let for all $1 \leq i \leq n$

$$\begin{array}{lll} \Gamma_n & = & \Gamma \\ \Gamma_{i-1} & = & \Gamma_i, h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i} \\ \lambda_0 & = & l \\ \lambda_i & = & \lambda h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i}.\lambda_{i-1} \\ \Pi_0 & = & \mathsf{pf}\, T\, u \\ \Pi_i & = & \Pi h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i}.\Pi_{i-1} \end{array}$$

Then, for all $0 \leq i \leq n$ *the following hold:*

(a) Γ_i *is a valid context.*

(b) $\Gamma_i \vdash_{\Sigma_n} \lambda_i : \Pi_i$

(c) $\Gamma_i \vdash_{\Sigma_n} \Pi_i : \mathsf{type}$

Note that the assertion follows from this property with $i = n$.

To prove the stronger property, note that $\Gamma_i \vdash_{\Sigma_n} \mathsf{pf} : \Pi\alpha : \mathsf{type}.\alpha \to \mathsf{type}$ for all $0 \leq i \leq n$ by rule declstart.

First, we show (a). Since $\Gamma_n = \Gamma$, its validity holds trivially. The application of Corollary 4.3 to the premises of HJNODE establishes that $\Gamma \vdash_{\Sigma_n} u_j : T_j$ and $\Gamma \vdash_{\Sigma_n} T_j : \mathsf{type}$ for $1 \leq j \leq n$, and hence, $\Gamma \vdash_{\Sigma_n} \mathsf{pf}\, T_j\, u_j : \mathsf{type}$ by applying rule appl twice. Since this conclusion holds for all $1 \leq j \leq n$, it is in particular true for $j = n + 1 - i$ and Γ_{i-1} is valid for $1 \leq i \leq n$ by ctxdecl. This concludes the proof of (a).

Next, we show (b) and (c) simultaneously by induction over i.

$i = 0$**:**

Since $l : \mathsf{pf}\, T\, u \in \Gamma \subseteq \Gamma_0$, (b) follows directly by rule start.

Now, we show (c).

Since $l : \mathsf{pf}\, T\, u \in \Gamma \subseteq \Gamma_0$ and Γ is a valid context, there must be a $\Gamma' \subseteq \Gamma$ and an $s \in \mathcal{S}$, such that $\Gamma' \vdash_{\Sigma_n} \mathsf{pf}\, T\, u : s$. Hence, the judgment $\Gamma_0 \vdash_{\Sigma_n} \mathsf{pf}\, T\, u : s$ must be derivable. Since $\Gamma_0 \vdash_{\Sigma_n} \mathsf{pf} : \Pi\alpha : \mathsf{type}.\alpha \to \mathsf{type}$, the judgments $\Gamma_0 \vdash_{\Sigma_n} u : T$ and $\Gamma_0 \vdash_{\Sigma_n} T : \mathsf{type}$ must be derivable. Thus, rule appl applied twice to these judgments leads to $\Gamma_0 \vdash_{\Sigma_n} \mathsf{pf}\, T\, u : \mathsf{type}$, that is, $s = \mathsf{type}$.

$(i-1) \to i$**:**

First, we show (c).

By induction hypothesis,

$$\Gamma_i, h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i} \vdash_{\Sigma_n} \Pi_{i-1} : \mathsf{type}$$

From the proof of (a) we know that $\Gamma_i \vdash_{\Sigma_n} \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i} : \mathsf{type}$. Hence, applying rule (type,type) gives us

$$\Gamma_i \vdash_{\Sigma_n} \Pi h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i}.\Pi_{i-1} : \mathsf{type}$$

This completes the proof of (c).

Next, we show (b).
By induction hypothesis,

$$\Gamma_i, h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i} \vdash_{\Sigma_n} \lambda_{i-1} : \Pi_{i-1}$$

By (c), we know that $\Gamma_i \vdash_{\Sigma_n} \Pi h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i}.\Pi_{i-1} : \mathsf{type}$.
Hence, applying rule abstr gives us

$$\begin{aligned}\Gamma_i \vdash_{\Sigma_n} \lambda h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i}.\lambda_{i-1} \\ : \Pi h_{n+1-i} : \mathsf{pf}\, T_{n+1-i}\, u_{n+1-i}.\Pi_{i-1}\end{aligned}$$

This completes the proof of (b).

(iii) Obviously, D and C are in long $\beta\eta$-normal form.

Case NODE**:**

(i) The assertion follows directly from the induction hypothesis.

(ii) The proof is similar to the one in case HJNODE. The only difference is that we define $\lambda_0 = D'$ in the stronger property and that $l : \mathsf{pf}\, T\, u \notin \Gamma$. Therefore, the base step of the induction to prove (b) and (c) follows directly from the induction hypothesis of the main proof applied to the $(n+1)$st premise of rule NODE. The rest of the proof remains unchanged.

(iii) Obvious.

Case JUST**:**

(i) By induction hypothesis and Lemma 4.3, $\Sigma \subseteq \Sigma_{n+m+1}$ and Σ_{n+m+1} is valid. Clearly, $\Sigma_{n+m+1} \subseteq \Sigma'$. To show the validity of Σ', we only have to show that

$$\vdash_{\Sigma_{n+m+1}} C : \mathsf{type}$$

Lemma 4.3 applied to the $(n + m + 1)$st premise of rule JUST leads to $\Gamma \vdash_{\Sigma_{n+m+1}} u : T$ and $\Gamma \vdash_{\Sigma_{n+m+1}} T : \mathsf{type}$. Rule declstart gives us $\Gamma \vdash_{\Sigma_{n+m+1}} \mathsf{pf} : \Pi\alpha : \mathsf{type}.\alpha \to \mathsf{type}$. Then, applying appl twice establishes that $\Gamma \vdash_{\Sigma_{n+m+1}} \mathsf{pf}\, T\, u : \mathsf{type}$.
By induction hypothesis, $\Gamma \vdash_{\Sigma_{n+m+1}} C_i : \mathsf{type}$ for $1 \le i \le n$. Application of Lemma 4.3 gives us $\Gamma \vdash_{\Sigma_{n+m+1}} T_i : \mathsf{type}$ for $1 \le i \le m$.
Thus, applying rule (type,type) $n + m$ times leads to

$$\Gamma \vdash_{\Sigma_{n+m+1}} \Pi\overline{y_n : C_n}.\Pi\overline{x_m : T_m}.\mathsf{pf}\, T\, u : \mathsf{type}$$

and $\vdash_{\Sigma_{n+m+1}} C : \mathsf{type}$ follows by the definition of the function $\mathcal{A}_\Pi$.

(ii) From (i), we know that $\Gamma \vdash_{\Sigma'} \mathsf{pf}\, T\, u : \mathsf{type}$. Therefore, $\Gamma \vdash_{\Sigma'} C : \mathsf{type}$ must be derivable, where

$$C = (\mathsf{pf}\, T\, u)[D_1/y_1] \ldots [D_n/y_n][u_1/x_1] \ldots [u_m/x_m]$$

We now show $\Gamma \vdash_{\Sigma'} cz_1 \ldots z_l D_1 \ldots D_n u_1 \ldots u_m : C$.
By declstart, $\Gamma \vdash_{\Sigma'} c : \Pi\overline{z_l : S_l}.\Pi\overline{y_n : C_n}.\Pi\overline{x_m : T_m}.\mathsf{pf}\, T\, u$. Since $\mathcal{A}_\Pi(\Sigma'', \Gamma, C') = \Pi\overline{z_l : S_l}.C'$, we know that $z_i : S_i \in \Gamma$ for $1 \le i \le l$ and therefore $\Gamma \vdash_{\Sigma'} z_i : S_i$. Furthermore, by induction hypothesis, $\Gamma \vdash_{\Sigma'} D_i : C_i$ for $1 \le i \le n$, and by Lemma 4.3, $\Gamma \vdash_{\Sigma'} u_i : T_i$ for $1 \le i \le m$. Hence, $l + n + m$ applications of appl establish that $\Gamma \vdash_{\Sigma'} cz_1 \ldots z_l D_1 \ldots D_n u_1 \ldots u_m : C$.

(iii) Obviously, D and C are in long $\beta\eta$-normal form.

Case JUST-SEQ**:**

(i) By induction hypothesis and Lemma 4.3, $\Sigma \subseteq \Sigma_{n+m+1}$ and Σ_{n+m+1} is valid. Clearly, $\Sigma_{n+m+1} \subseteq \Sigma'$.

Thus, we only have to show $\vdash_{\Sigma_{n+m+1}} C'' : \mathsf{type}$ and $\vdash_{\Sigma_{n+m+1}} D'' : C''$ to derive the validity of Σ' by sigdefn.

First, we prove the following property:

Let for $1 \leq i \leq n+m$

$$\begin{array}{llllll} a_0 & = D' & ; & a_i & = \begin{cases} x_{m+1-i} & \text{for } 1 \leq i \leq m \\ y_{m+n+1-i} & \text{for } m < i \leq m+n \end{cases} \\ A_0 & = C & ; & A_i & = \begin{cases} C_{m+1-i} & \text{for } 1 \leq i \leq m \\ T_{m+n+1-i} & \text{for } m < i \leq m+n \end{cases} \\ \Gamma_{n+m} & = \Gamma & ; & \Gamma_{i-1} & = \Gamma_i, a_i : A_i \\ \lambda_0 & = a_0 & ; & \lambda_i & = \lambda a_i : A_i . \lambda_{i-1} \\ \Pi_0 & = A_0 & ; & \Pi_i & = \Pi a_i : A_i . \Pi_{i-1} \end{array}$$

Then, for all $0 \leq i \leq m+n$ *the following hold:*

(a) Γ_i *is a valid context.*

(b) $\Gamma_i \vdash_{\Sigma_{n+m+1}} \lambda_i : \Pi_i$.

(c) $\Gamma_i \vdash_{\Sigma_{n+m+1}} \Pi_i : \mathsf{type}$.

To prove (a), note that, since $\Gamma_{m+n} = \Gamma$, its validity holds trivially. By induction hypothesis, $\Gamma \vdash_{\Sigma_{n+m+1}} C_i : \mathsf{type}$ for $1 \leq i \leq n$. By Lemma 4.3, $\Gamma \vdash_{\Sigma_{n+m+1}} T_i : \mathsf{type}$. Hence, $\Gamma \vdash_{\Sigma_{n+m+1}} A_i : \mathsf{type}$ and Γ_{i-1} is valid for $0 \leq i < n+m$.

Next, we prove (b) and (c) simultaneously by induction over i.

$i = 0$**:**

$\Gamma_0 \vdash_{\Sigma_{n+m+1}} D' : C$ and $\Gamma_0 \vdash_{\Sigma_{n+m+1}} C : \mathsf{type}$ follow from the induction hypothesis of the main proof applied to the $(n+m+1)$st premise of rule JUST-SEQ.

$(i-1) \to i$**:**

First, we show (c).

By induction hypothesis, $\Gamma_i, a_i : A_i \vdash_{\Sigma_{n+m+1}} \Pi_{i-1} : \mathsf{type}$. In the proof of (a), we showed that $\Gamma \vdash_{\Sigma_{n+m+1}} A_i : \mathsf{type}$, and hence, $\Gamma_i \vdash_{\Sigma_{n+m+1}} A_i : \mathsf{type}$. Application of rule (type,type) then leads to $\Gamma_i \vdash_{\Sigma_{n+m+1}} \Pi a_i : A_i . \Pi_{i-1} : \mathsf{type}$. This completes the proofs of (c).

Next, we show (b).

By induction hypothesis, $\Gamma_i, a_i : A_i \vdash_{\Sigma_{n+m+1}} \lambda_{i-1} : \Pi_{i-1}$. By (c), we have $\Gamma_i \vdash_{\Sigma_{n+m+1}} \Pi a_i : A_i . \Pi_{i-1} : \mathsf{type}$. Hence, application of rule abstr gives us $\Gamma_i, \vdash_{\Sigma_{n+m+1}} \lambda a_i : A_i . \lambda_{i-1} : \Pi a_i : A_i . \Pi_{i-1}$. This completes the proof of (b).

From this property, we can conclude $\Gamma \vdash_{\Sigma_{n+m+1}} \lambda_{m+n} : \Pi_{m+n}$ and $\Gamma \vdash_{\Sigma_{n+m+1}} \Pi_{m+n} : \mathsf{type}$. Note that $D'' = \mathcal{A}_\lambda(\Sigma_{n+m+1}, \Gamma, \lambda_{m+n}) = \lambda \overline{z_l : S_l} . \lambda_{m+n}$. By definition of $\mathcal{A}_\lambda$, we obtain

$$\vdash_{\Sigma_{n+m+1}} \lambda \overline{z_l : S_l} . \lambda_{m+n} : \Pi \overline{z_l : S_l} . \Pi_{m+n}$$

and

$$\vdash_{\Sigma_{n+m+1}} \Pi \overline{z_l : S_l} . \Pi_{m+n} : \mathsf{type}$$

(ii) First, we show $\Gamma \vdash_{\Sigma'} D : C$.
By declstart, $\Gamma \vdash_{\Sigma'} c : \Pi\overline{z_l : S_l}.\Pi\overline{y_n : C_n}.\Pi\overline{x_m : T_m}.C'$. Clearly, $z_i : S_i \in \Gamma$ for $1 \leq i \leq l$. Hence, $\Gamma \vdash_{\Sigma'} z_i : S_i$. Furthermore, by induction hypothesis, $\Gamma \vdash_{\Sigma'} D_i : C_i$ for $1 \leq i \leq n$, and by Lemma 4.3, $\Gamma \vdash_{\Sigma'} u_i : T_i$ for $1 \leq i \leq m$. Hence, $l+n+m$ applications of appl establish that $\Gamma \vdash_{\Sigma'} cz_1 \ldots z_l D_1 \ldots D_n u_1 \ldots u_m : C'[D_1/y_1] \ldots [D_n/y_n][u_1/x_1] \ldots [u_m/x_m]$.
Next, we show $\Gamma \vdash_{\Sigma'} C$: type.
From the induction hypothesis applied to the $(n+m+1)$st premise we know that $\Gamma' \vdash_{\Sigma_n} C'$: type. Then, clearly $\Gamma \vdash_{\Sigma'} C$: type must also be derivable.

(iii) Obviously, D and C are in long $\beta\eta$-normal form.

Case PDS**:**

(i) By Lemma 4.2 and induction hypothesis, $\Sigma \subseteq \Sigma_{m+n+1}$ and Σ_{m+n+1} is valid. Clearly, $\Sigma_{m+n+1} \subseteq \Sigma'$. This leaves us to show $\vdash_{\Sigma_{m+n+1}} C'$: type and $\vdash_{\Sigma_{m+n+1}} D' : C'$.
First, we prove the following property:
Let for $1 \leq i \leq n$

$$
\begin{array}{lll}
a_0 & = D & \\
a_i & = \begin{cases} \gamma_{m+1-i} & \text{for } 1 \leq i \leq m \\ c_{m+n+1-i} & \text{for } m < i \leq m+n \end{cases} & \\
A_0 & = C & \\
A_i & = \begin{cases} \mathsf{hol} & \text{for } 1 \leq i \leq m \\ \mathsf{obj}\, T_{m+n+1-i} & \text{for } m < i \leq m+n \end{cases} & \\
\Gamma_{m+n} & = \Gamma & \\
\Gamma_{i-1} & = \Gamma_i, a_i : A_i &
\end{array}
$$

Then, Γ_i *is a valid context for all* $0 \leq i \leq n+m$.
To prove the property, note that, since $\Gamma_{m+n} = \Gamma$, its validity holds trivially. By Lemma 4.2, $\Gamma \vdash_{\Sigma_{n+m+1}} A_i$: type for $1 \leq i \leq m$. Furthermore, $\vdash_{\Sigma_{n+m+1}}$ obj : hol $\to$ type by declstart. Hence, by Lemma 4.2 and rule appl $\Gamma \vdash_{\Sigma_{n+m+1}} A_i$: type for $m < i \leq m+n$. Then, Γ_{i-1} is valid for $1 \leq i \leq m+n$ by ctxdecl.
From this result, we can conclude in particular that $\Gamma' = \Gamma_0$ is valid. By induction hypothesis, $\Gamma' \vdash_{\Sigma_{m+n+1}} D : C$ and $\Gamma' \vdash_{\Sigma_{m+n+1}} C$: type. Hence, $\vdash_{\Sigma_{m+n+1}} \lambda\overline{x_l : S_l}.D : \Pi\overline{x_l : S_l}.C$ and $\vdash_{\Sigma_{m+n+1}} \Pi\overline{x_l : S_l}.C$: type by definition of $\mathcal{A}_\lambda$. Application of rule sigdefn finally establishes that Σ' is valid.

(ii) The assertion follows directly by induction hypothesis

(iii) Obvious, since $D, C \in \mathcal{NF}(\mathfrak{T})$ by induction hypothesis. ∎

Appendix B

Speech Acts

In Chapter 6.2.1 we introduced the notion of speech acts in *P.rex* and gave an informal overview of our taxonomy of speech acts. This taxonomy consists of two major classes, *mathematical communicative acts (MCAs)* and *interpersonal communicative acts (ICAs)*, which are depicted in the Figures B.1 and B.2, respectively.

In this chapter, we shall define all speech acts formally and explain their intuitive meaning, often by giving possible verbalizations. But first, we introduce the following notation:

Notation: We write `(S (C) :s1 f1 ...:sn fn)` for the definition of a speech act of class `S`, which is a subclass of `C` and has slots `:s1`,...,`:sn` and fillers `f1`,...,`fn`. In fillers, we use $t, t', t_1, t_2, \ldots$ to denote terms and $i, i_1, \ldots, i_n$ to denote natural numbers.

The top classes of speech acts are the following:

- `(SA ())` is the class of all speech acts.
- `(MCA (SA))` is the class of MCAs.
- `(ICA (SA))` is the class of ICAs.

B.1 Mathematical Communicative Acts

MCAs are employed to present or explain mathematical concepts or derivations. We distinguish two subclasses of MCAs:

- `(derivational (MCA))` is the class of derivational MCAs.
- `(explanatory (MCA))` is the class of explanatory MCAs.

B.1.1 Derivational MCAs

Derivational MCAs convey information that is necessary to establish logical correctness. The first derivational MCAs stands for a derivation step:

- `(Derive (derivational)`
  `:Reasons` $(t_1, \ldots, t_n)$ `:Method` $\langle name \rangle$ `:Conclusion` t`)`

  "Since $t_1, \ldots, t_n$, t by $\langle name \rangle$."

The next two MCAs introduce an hypothesis or a parameter, respectively, in the derivation:

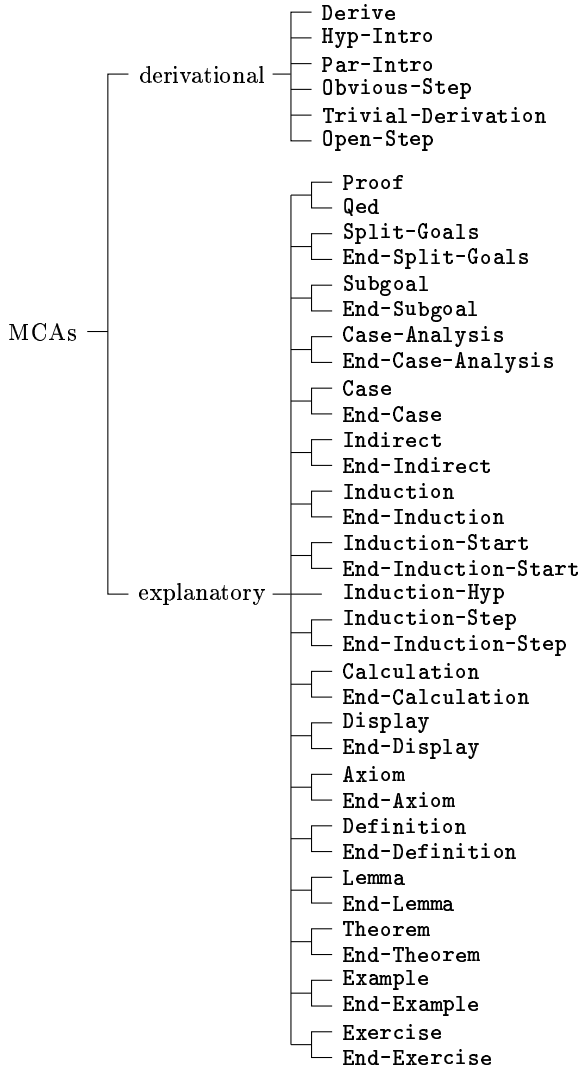


Figure B.1. The taxonomy of MCAs.

- `(Hyp-Intro (derivational) :Hypothesis` t`)`
  "Let t."
- `(Par-Intro (derivational) :Parameter` t `:Type` t'`)`
  "Let t be a t'."

Obvious steps and trivial derivations call for special derivational MCAs:

- `(Obvious-Step (derivational) :Conclusion` t`)`
  "Obviously, t."
- `(Trivial-Derivation (derivational) :Conclusion` t`)`
  "t is trivially true."

Finally, we define a derivational MCA for unproved assertions:

- `(Open-Step (derivational) :Conclusion` t`)`
  "It is an open question how to prove t."

B.1.2 Explanatory MCAs

Explanatory MCAs give auxiliary information that helps to understand a derivation. Most explanatory MCAs come in pairs where an opening explanation is associated with a closing explanation. This is grasped by the following two subclasses:

- `(open (explanatory))` is the class of opening MCAs.
- `(close (explanatory))` is the class of closing MCAs.

We first define explanatory MCAs that structure a proof. The first two MCAs encapsulate a proof:

- `(Proof (open))`
  "***Proof:***"
- `(Qed (close))`
  "■"

The next four MCAs structure a proof part where a goal is shown by splitting it into subgoals, which are presented individually:

- `(Split-Goals (open) :Goal` t `:Subgoals` $t_1, \ldots, t_n$`)`
  "To prove t we prove $t_1, \ldots, t_n$."
- `(End-Split-Goals (close) :Goal` t`)`
  "Therefore t."
- `(Subgoal (open) :Number` i `:Goal` t`)`
  "{First, Second, Next}, we prove t."
- `(End-Subgoal (open) :Number` i `:Goal` t`)`
  "This completes the proof of t."

The next four MCAs structure a case analysis:

- `(Case-Analysis (open) :Goal` t `:Cases` $t_1, \ldots, t_n$`)`
  "To prove t, we consider the following cases:"
- `(End-Case-Analysis (close) :Goal` t`)`
  "This completes the case analysis."
- `(Case (open) :Number` i `:Hypothesis` t`)`
  "Case i: t"
- `(End-Case (close) :Number` i `:Hypothesis` t`)`
  "This completes the ith case."

The following two MCAs encapsulate an indirect proof:

- `(Indirect (open) :Goal` t`)`
  "We show t indirectly."
- `(End-Indirect (close) :Goal` t`)`
  "Therefore t."

The next seven MCAs structure an induction proof:

- `(Induction (open) :Goal` t `:Type` $\langle type \rangle$`)`
  "We show t by $\langle type \rangle$ induction."
- `(End-Induction (close) :Goal` t `:Type` $\langle type \rangle$`)`
  "This completes the induction proof."
- `(Induction-Start (open))`
  "Base case:"
- `(End-Induction-Start (close))`
  "This completes the base case."
- `(Induction-Hyp (explanatory) :Hypothesis` t`)`
  "Induction hypothesis: t."
- `(Induction-Step (open))`
  "Step case:"
- `(End-Induction-Step (close))`
  "This completes the step case."

The following four MCAs encapsulate calculations and figures, respectively:

- `(Calculation (open))` marks the start of a calculation.
- `(End-Calculation (close))` marks the end of a calculation.
- `(Display (open))` marks the start of a displayed figure.
- `(End-Display (close))` marks the end of a displayed figure.

The remaining explanatory MCAs structure a mathematical lesson:

- `(Axiom (open))`
  "**Axiom:**"
- `(End-Axiom (close))` marks the end of an axiom.
- `(Definition (open))`
  "**Definition:**"
- `(End-Definition (close))`
  "■"
- `(Lemma (open))`
  "**Lemma:**"
- `(End-Lemma (close))` marks the end of a lemma.
- `(Theorem (open))`
  "**Theorem:**"
- `(End-Theorem (close))` marks the end of a theorem.
- `(Example (open))`
  "*Example:*"

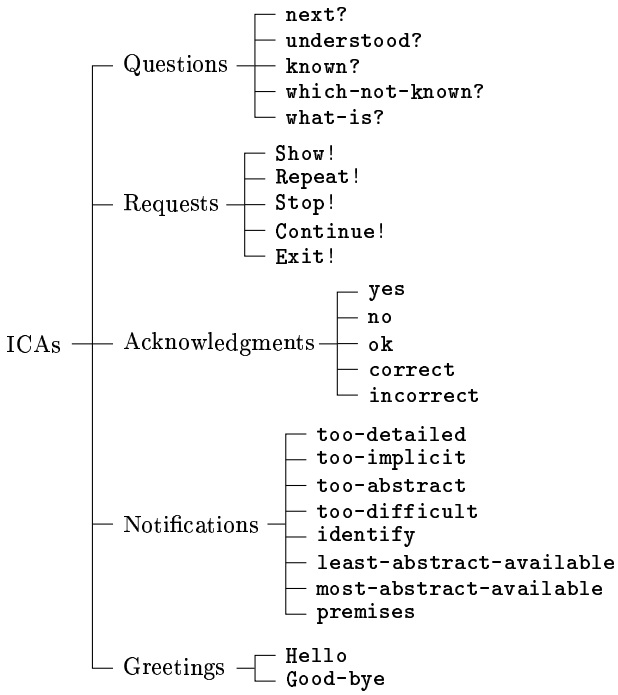


Figure B.2. The taxonomy of ICAs.

- `(End-Example (close))`
  "□"
- `(Exercise (open))`
  "**Exercise:**"
- `(End-Exercise (close))` marks the end of an exercise.

B.2 Interpersonal Communicative Acts

ICAs are used in situations, where the system and the user alternately take over the active role in the discourse. We distinguish five classes of ICAs:

- `(Question (ICA))` is the class of questions.
- `(Request (ICA))` is the class of requests.
- `(Acknowledgment (ICA))` is the class of acknowledgments.
- `(Notification (ICA))` is the class of notifications.
- `(Greeting (ICA))` is the class of greetings.

B.2.1 Questions

- `(next? (Question))`
  "What next?"

- `(understood? (Question) :Goal` t`)`

  "Do you understand the step t?"

- `(known? (Question) :Goal` t`)`

  "Do you know t?"

- `(which-not-known? (Question) :Goal` t`)`

  "Which t don't you know?"

- `(what-is (Question) :Goal` t `:ds-node` t'`)`

  "What is t?"

B.2.2 Requests

- `(Show! (Request) :Proof` t `:Strategy` ⟨*strategy*⟩`)`

  "Show the proof t!"

- `(Repeat! (Request) :Proof` t`)`

  "Repeat the proof t!"

- `(Stop! (Request))`

  "Stop!"

- `(Exit! (Request) )`

  "Exit!"

- `(Continue! (Request) )`

  "Continue!"

B.2.3 Acknowledgments

- `(yes (Acknowledgment))`

  "Yes."

- `(no (Acknowledgment))`

  "No."

- `(ok (Acknowledgment))`

  "OK."

- `(correct (Acknowledgment))`

  "This is correct."

- `(incorrect (Acknowledgment))`

  "This is not correct."

B.2.4 Notifications

- `(too-detailed (Notification) :Conclusion` t`)`
  "The step t is too detailed."
- `(too-difficult (Notification) :Conclusion` t`)`
  "The step t is too difficult."
- `(too-implicit (Notification) :Conclusion` t`)`
  "The step t is too implicit."
- `(too-abstract (Notification) :Conclusion` t`)`
  "The step t is too abstract."
- `(hypothesis (Notification))`
  "This is a hypothesis."
- `(identify (Notification) :object` t `:class` t'`)`
  "t is t'."
- `(least-abstract-available (Notification))`
  "This is the least abstract explanation available."
- `(most-abstract-available (Notification))`
  "This is the most abstract explanation available."
- `(premises (Notification) :items` $i_1, \ldots, i_n$`)`
  "The premises are $i_1, \ldots, i_n$."

B.2.5 Greetings

- `(Hello (Greeting))`
  "Hello."
- `(Good-Bye (Greeting))`
  "Good bye."

Appendix C

Cognitive Knowledge Bases

In this chapter, we shall present the declarative and procedural knowledge that is predefined in *P.rex*.

C.1 Declarative Knowledge

In this section, we shall describe the predefined declarative knowledge of *P.rex*. First, in Section C.1.1 we shall present the syntax to define chunk types. Then, in Section C.1.2, we shall define the chunk types in *P.rex*. In Section C.1.3, finally, we shall introduce all predefined chunks.

C.1.1 Syntax

The syntax to define chunk types is as follows:

chunktypedef ::= `(chunk-type`
{*chunktype* | (*chunktype* `(:include` *chunktype*))}
{*slot* | (*slot value*)}*)

The non-terminal symbols *chunktype*, *slot* and *value* denote chunk types, slots and values, respectively.

Note that '`type (:include parenttype)`' means that the chunk type `type` inherits the slots of `parenttype`. However, ACT-R allows only single inheritance in chunk types, that is, `parenttype` must not inherit any slots and no chunk type may include `type` to inherit its slots. Moreover, '`(slot value)`' means that the slot `slot` has the default value `value`.

C.1.2 Chunk Types

In this section, we shall give the chunk types that are defined in *P.rex*.

```
(chunk-type empty) ;;; a chunk type for a special chunk for empty slots

;#  discourse structure trees

(chunk-type DS ;;; a chunk type for nodes of discourse structure trees
           purpose     ; the intention of the node (a PURPOSE chunk)
           to-convey   ; the information to be conveyed:
                       ; iff type is 'basic a proof node chunk or speech act
                       ; iff type is 'focus-space a list of proof node chunks
           content     ; an ordered list of DS chunks (iff planning purpose), or
                       ; the EMPTY chunk (iff purpose is OMIT), or
                       ; a speech act (iff utterance purpose)
           content-of  ; the parent node (a DS chunk)
```

```
            role        ; the relation to the parent node:
                        ; 'contributes (iff planning purpose)
                        ; 'verbalizes  (iff utterance purpose)
                        ; 'opens or 'closes (if utterance purpose and
                        ;                    parent is of 'focus-space type)
            (status 'unknown)      ; status w.r.t. the user ('known or 'unkown)
            ;; bookkeeping slots
            unconveyed  ; unconveyed chunks in 'to-convey'
                        ; iff Basic-DS a proof node chunk or a speech act
                        ; iff Focus-Space-DS a list of proof node chunks
            )
(chunk-type (basic-DS (:include DS)) ;;; basic DS nodes
            ;; bookkeeping slots
            rule        ; only 'basic type: inference rule of the most abstract known
                        ; justification of the proof node in to-convey
            premises    ; only 'basic type: list of premises of the most abstract known
                        ; justification of the proof node in to-convey
            uncon-prems ; only 'basic type: list of unconveyed premises of the unconveyed
                        ; chunk (a list of proof node chunks)
            )
(chunk-type (focus-space-DS (:include DS)) ;;; focus space DS nodes
            ;; bookkeeping slot to speed up the implementation
            derived     ; only iff 'focus-space type: chronologically ordered list of
                        ; formulae that have been derived in this focus space
            )

(chunk-type purpose ;;; chunk types for purposes
            class ; 'planning or 'utterance for planning and utterance purposes, resp.
            )

(chunk-type proof ;;; a chunk type for proofs
            proof ; a proof
            name  ; a string
            )
(chunk-type proof-node ;;; a chunk type for proof nodes
            node  ; the proof node
            fact  ; its fact (a FACT chunk)
            justs ; its justifications (a list of JUST chunks)
            )
(chunk-type proof-node-list ;;; a chunk type for proof node lists (used in DS trees)
            purpose    ; purpose
            nodes      ; list of proof nodes
            total      ; total number of proof nodes
            type       ; the type of the proof nodes
            content    ; as in DS
            content-of ; as in DS
            )
(chunk-type assoc ;;; association of a proof node with a speech act
            proof-node ; a proof-node chunk
            speech-act ; a speech act
            )

(chunk-type fact ;;; a chunk type for facts
            (antecedent empty) ; list of FORMULA chunks or EMPTY
            succedent          ; a FORMULA chunk
            (status 'unknown)  ; status w.r.t. the user ('known or 'unkown)
            )
(chunk-type formula ;;; a chunk type for formulae
            wff ; the term
            )
(chunk-type rule ;;; a chunk type for rules
            name              ; the name of the rule as string
            (status 'unknown) ; status w.r.t. the user ('known or 'unkown)
            )
(chunk-type find-just ;;; a chunk type for finding a known justification
            justs    ; a list of justifications
            previous ; the next more abstract justification
```

```
            rule     ; a "rule" chunk
            premises ; a list of proof nodes: the premises used by RULE
            )

(chunk-type identify ;;; a chunk type to identify objects
            object ; the object to identify (a twa+proof-node)
            )

(chunk-type format ;;; a chunk type for format directives
            type ; the format type: 'begin-focus or 'end-focus
            )

;#  chunk types for user interaction

(chunk-type interaction ;;; a chunk type for interaction goals
            type    ; a string for goal type
            content ; a speech act
            )
(chunk-type failed-presentation ;;; a chunk type to react to a failed presentation
            node             ; the DS node whose presentation failed
            lower            ; a DS node that shows node at a lower level of abstraction
            problem          ; a string indicating the problem why presentation failed
            remedy           ; a string indicating how to repair the presentation
            (fact-unknown t) ; a boolean indication that the fact should be set to unknown
            answer           ; the answer of the user to possible questions
            )
(chunk-type command ;;; a chunk type for commands
            type ; the command type ("exit", "stop")
            )
```

C.1.3 Predefined Chunks

The following chunks are predefined in *P.rex*:

```
; special chunk for empty slots
(Empty ISA empty)

; planning and utterance purposes
(explain       ISA purpose class 'planning)
(present       ISA purpose class 'planning)
(omit          ISA purpose class 'planning)
(reverbalize   ISA purpose class 'planning)
(clarify       ISA purpose class 'planning)
(repeat        ISA purpose class 'planning)
(inform        ISA purpose class 'utterance)
(ask           ISA purpose class 'utterance)
(answer        ISA purpose class 'utterance)
(request       ISA purpose class 'utterance)
(acknowledge   ISA purpose class 'utterance)

; interaction chunks
(read-interaction ISA interaction)
(read-interruption ISA interaction type "interruption")

; known inference rules
(assertion ISA rule name "ASSERTION" status 'known)
(hyp ISA rule name "HYP" status 'known)
(dec ISA rule name "DEC" status 'known)
(open ISA rule name "OPEN" status 'known)
```

C.2 Procedural Knowledge

In this section, we shall describe the predefined procedural knowledge of *P.rex*. First, in Section C.2.1, we shall give the syntax to define productions in ACT-R.

Then, in Section C.2.2, we shall introduce all predefined production, which represent the plan operators of the dialog planner (cf. Chapter 6).

C.2.1 Syntax

The syntax to define productions is given by the following grammar:

production	::=	`(P` *rhs** `==>` *lhs**`)`
lhs	::=	*chunkdef* | *lhscommand*
rhs	::=	*chunkdef* | *rhscommand*
chunkdef	::=	`=`*chunk*`>`
		[`ISA` *chunktype*]
		[[`-`] *slot* {*value* | `=`*variable* | `=`*chunk*}]*
lhscommand	::=	`!eval!` *expression*
		| `!bind!` `=`*variable expression*
rhscommand	::=	*lhscommand*
		| `!push!` `=`*chunk*
		| `!pop!`
		| `!focus-on!` `=`*chunk*
		| `!delete!` `=`*chunk*
		| `!stop!`
		| `!restart!`
		| `!output!` *formatstring expression**

Note that the non-terminal symbols *chunktype*, *chunk*, *slot* and *variable* stand for the names of chunk types, chunks, slots and variables, respectively.

The symbol *value* stands for the value a slot can assume. It can be a chunk, a Lisp symbol or string, or a number. Also lists thereof are allowed.

The non-terminal symbol *expression* can be any valid Lisp expression. Any symbol beginning with '`=`' such as `=symbol` is interpreted as a Lisp variable, where its first appearance in the production definition denotes its binding. Hence, if `=symbol` first occurs in a chunk slot, it is bound to the filler of this slot. Otherwise, the first appearance must be as the variable in a `!bind!` command, which explicitly binds the variable to the value of the subsequent Lisp expression. Note that variables cannot be bound to `NIL` on the left-hand side of a production. In *P.rex*, we use the special chunk `Empty` as the filler of an empty slot.

The first appearance of a chunk definition starting with `=chunk>` must be followed by a type identification of the form '`ISA chunk-type`'. In subsequent extensions or modifications of the same chunk `=chunk>`, the type identification can be omitted. The chunk definition may contain any number of slot descriptions, which are interpreted as conditions for the chunk retrieval on the left-hand side of a production, and as initializations or modifications of the chunk on the right-hand side of a production. On the left-hand side, the condition may start with a '`-`', which is interpreted as the negation of the condition.

Note that the first chunk on the left-hand side is always matched against the current goal, that is, the chunk on top of the goal stack.

The commands have the following meanings:

`!eval!` *expression*
: Evaluate the Lisp expression *expression*. On the left-hand side of a production, the result must be non-`NIL` for the production to be applicable.

`!bind!` `=`*variable expression*
: Bind the variable `=`*variable* to the result of the evaluation of the Lisp expression *expression*. On the left-hand side of a production, the result must be non-`NIL` for the production to be applicable.

`!push!` =*chunk*
: Push the chunk =*chunk* onto the goal stack.

`!pop!` Pop the top goal from the goal stack.

`!focus-on!` =*chunk*
: Replace the top goal on the goal stack by chunk =*chunk*. This corresponds to `!pop!` followed by `!push!` =*chunk*.

`!delete!` =*chunk*
: Delete the chunk =*chunk* from declarative memory.

`!stop!`
: Stop the production system after completing the current cycle.

`!restart!`
: Pop all goals from the goal stack and restore the top-most goal.

`!output!` *formatstring expressions*
: Print a message. The syntax is as for the Lisp function `format`.

C.2.2 Productions

The following productions are predefined in *P.rex*.

```
;### Planning the Explanation of Proofs

;## General Productions

;# Discourse Structure Nodes

;;; The following production marks DS chunks whose informations have been
;;; conveyed as known.
(P mark-conveyed-DS-as-known
   "IF   Goal is a DS with no unconveyed chunks,
    THEN mark it as known."
   =goal>
      ISA DS
      unconveyed empty
    - status 'known
==>
   =goal>
      status 'known
   )
; This production should be preferred to all other ones. We therefore set lower
; costs.
(spp mark-conveyed-DS-as-known
     :a 0.003)

;;; The following production pops basic-DS chunks that are known by the user.
(P pop-known-DS
   "IF   Goal is a DS that is known,
    THEN pop it."
   =goal>
      ISA DS
      role =role
      status 'known
   !eval! (not (find (cadr =role) '(commands interrupts)))
==>
   !pop!
   )
; This production should be preferred to all other ones except pop-known-focus-space-DS.
; We therefore set lower costs.
(spp pop-known-DS
     :a 0.002)
```

```
;;; The following production pops focus-space-DS chunks that are known by the user.
;;; It ensures that the last verbalized fact is carried to the governing
;;; focus-space-DS's derived-list.
(P pop-known-focus-space-DS
   "IF   Goal is a focus space DS that is known,
    THEN append the last derived fact to its 'derived' slot and pop it."
   =goal>
      ISA focus-space-DS
      status 'known
      derived =derived
      content-of =pa
    - content-of empty
==>
   !eval! (dst~fact-derived-in-dominating-focus-space (car (last =derived)) =pa)
   !pop!
   )
; This production should be preferred to all other ones. We therefore set lower costs.
(spp pop-known-focus-space-DS
     :a 0.001)

;;; The following production pops DS chunks that want to convey known facts
(P omit-DS-known-non-hyp-fact
   "IF   Goal is a basic DS with a non-hypothesis proof node whose fact is known,
    THEN fulfill the goal by omitting it and mark as conveyed and known."
   =goal>
      ISA basic-DS
      to-convey =proof-node
      rule =rule
   =rule>
      ISA rule
    - name "HYP"
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      status 'known
==>
   =omit>
      ISA basic-DS
      purpose omit
      to-convey  =proof-node
      unconveyed empty
      content    empty
      content-of =goal
      role       'contributes
      status     'known
   =goal>
      unconveyed empty
      content    =omit
      status     'known
   )
; This production should be preferred to all other ones except pop-known-DS. We
; therefore set the costs slightly higher than those of pop-known-DS.
(spp omit-DS-known-non-hyp-fact
     :a 0.003)

(P omit-DS-known-hyp-fact
   "IF   Goal is a basic DS with a hypothesis proof node whose fact has been derived
         in a dominating focus space,
    THEN fulfill the goal by omitting it and mark as conveyed and known."
   =goal>
      ISA basic-DS
      to-convey =proof-node
      rule =rule
   =rule>
```

```
      ISA rule
      name "HYP"
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      status 'known
   !eval! (dst~fact-derived-in-dominating-focus-space-p =fact =goal)
==>
   =omit>
      ISA basic-DS
      purpose omit
      to-convey  =proof-node
      unconveyed empty
      content    empty
      content-of =goal
      role       'contributes
      status     'known
   =goal>
      unconveyed empty
      content    =omit
      status     'known
   )
; This production should be preferred to all other ones except pop-known-DS. We
; therefore the costs slightly higher than those of pop-known-DS.
(spp omit-DS-known-hyp-fact
     :a 0.003)

(P omit-DS-inferable-fact
   "IF   Goal is a basic DS with a proof node with status 'inferable
    THEN fulfill Goal by omitting it and set the status to 'known"
   =goal>
      ISA DS
      to-convey =proof-node
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      status 'inferable
==>
   =omit>
      ISA basic-DS
      purpose omit
      to-convey  =proof-node
      unconveyed empty
      content    empty
      content-of =goal
      role       'contributes
      status     'known
   =goal>
      unconveyed empty
      content =omit
      status 'known
)
; This production should be preferred to all other ones except pop-known-DS. We
; therefore the costs slightly higher than those of pop-known-DS.
(spp omit-DS-inferable-fact
     :a 0.003)

;#  Proof Node Chunks

;;; The following production pushes proof nodes on the goal stack to eventually
;;; extract facts and justifications.
(P push-proof-node
   "IF   Goal is a DS with a proof node to convey whose fact and justification are not
```

```
         yet made explicit,
    THEN push the proof node onto the goal stack for completion."
   =goal>
      ISA DS
      to-convey =proof-node
   =proof-node>
      ISA proof-node
      node  =node
      fact  nil
      justs nil
==>
   !push! =proof-node
   )

(P prepare-assoc
   "IF   Goal is to convey an association of a proof node and a speech act and the
         fact of the proof node has not been extracted yet,
    THEN push the goal to extract the fact."
   =goal>
      ISA basic-DS
      to-convey  =assoc
      unconveyed =assoc
   =assoc>
      ISA assoc
      proof-node =proof-node
   =proof-node>
      ISA proof-node
      fact nil
==>
   !push! =proof-node
   )

;;; The following production pops proof nodes whose facts have been extracted.
(P pop-proof-node
   "IF   Goal is a proof node whose fact has been extracted,
    THEN pop the proof node."
   =goal>
      ISA proof-node
    - fact  nil
==>
   !pop!
   )

;;; The following two productions extract the fact and the list of justifications of
;;; a proof-node.
(P extract-fact-and-justs
   "IF   Goal is a proof node to convey whose fact and justification are not yet made
         explicit,
    THEN extracts its fact and justifications and make them explicit."
   =goal>
      ISA proof-node
      node  =node
      fact  nil
      justs nil
==>
   !bind! =abstract (actr~abstract =node)
   !bind! =ante (actr~nil-2-empty (actr~find-or-create-antecedent-chunks =abstract))
   !bind! =succ (actr~find-or-create-succedent-chunk =abstract)
   =fact>
      ISA fact
      antecedent =ante
      succedent  =succ
   !bind! =justs (actr~all-justs =abstract)
   =goal>
      fact  =fact
      justs =justs
   )
```

```
(P extract-existing-fact-and-justs
   "IF   Goal is a proof node to convey whose fact and justification are not yet made
         explicit, and whose fact is already known,
    THEN extracts its fact and justifications and make them explicit."
   =goal>
      ISA proof-node
      node  =node
      fact  nil
      justs nil
   !bind! =abstract (actr~abstract =node)
   !bind! =ante (actr~nil-2-empty (actr~find-or-create-antecedent-chunks =abstract))
   !bind! =succ (actr~find-or-create-succedent-chunk =abstract)
   =fact>
      ISA fact
      antecedent =ante
      succedent  =succ
==>
   !bind! =justs (actr~all-justs =abstract)
   =goal>
      fact  =fact
      justs =justs
   )
; This production should be preferred to extract-fact-and-justs. We therefore set
; lower costs.
(spp extract-existing-fact-and-justs
     :a 0.01)

;#  Choosing the Most Abstract Rule and Corresponding Premises

;;; The following production pushes the goal to find the most abstract justification.
(P get-justification
   "IF   Goal is a basic DS with a proof node whose most abstract known justification
         has
         not been determined,
    THEN push the goal to choose the most abstract known justification."
   =goal>
      ISA basic-DS
      to-convey =proof-node
      rule     nil
      premises nil
   =proof-node>
      ISA proof-node
      justs =justs
==>
   =find-most-abstract>
      ISA find-just
      justs =justs
      rule =method
      premises =premises
   =goal>
      rule     =method
      premises =premises
      uncon-prems =premises
   !push! =find-most-abstract
   )

;;;; The following three productions choose the most abstract known rule, if the list
;;;; of justifications in the "justs" slot of the "find-just" goal is ordered from
;;;; most abstract to least abstract.

;;; The following production chooses the rule of the first justification in the list,
;;; if it is known.
(P first-justification-is-known
   "IF   Goal is to find a known justification and the first justification's method is
         known,
    THEN extract its method and premises."
```

```
   =goal>
      ISA find-just
      justs =justs
      rule nil
   !bind! =first (car =justs)
   !bind! =method (actr~method =first)
   =rule>
      ISA rule
      name   =method
      status 'known
==>
   !bind! =premises (actr~nil-2-empty (actr~premises =first))
   !output! "~&;;; Choosing Rule ~A with premises ~A" =rule =premises
   =goal>
      rule =rule
      premises =premises
   )
; This production should be preferred to remove-first-justification. We therefore set
; a higher value or lower costs.
(spp first-justification-is-known
     :a 0.01)

;;; The following production chooses the rule of the first justification in the list,
;;; it this an  assertion with a known method.
;;; (The first premise of an assertion is its method.)
(P first-justification-is-known-assertion
   "IF   GOAL is to find a known assertion justification and the first justification's
         method is known,
    THEN extract its method and premises."
   =goal>
      ISA find-just
      justs =justs
      rule nil
   !bind! =first (car =justs)
   !eval! (actr~assertion-p (actr~method =first))
   !bind! =premises+ass (actr~premises =first)
   !bind! =method (actr~assertion-method (car =premises+ass))
   =rule>
      ISA rule
      name   =method
      status 'known
==>
   !bind! =premises (cdr =premises+ass)
   !output! "~&;;; Choosing Rule ~A with premises ~A" =rule =premises
   =goal>
      rule =rule
      premises =premises
   )
(spp first-justification-is-known-assertion
     :a 0.005)

;;; The following production removes the first justification from the list.
(P remove-first-justification
   "IF   Goal is to find a known justification,
    THEN remove the first justification."
   =goal>
      ISA find-just
      justs =justs
      rule nil
==>
   !bind! =first (car =justs)
   !bind! =rest (cdr =justs)
   =goal>
      justs =rest
      previous =first
   )
```

```
(P no-justification-found
   "IF   Goal is to find a known justification but none could be found,
    THEN pick the least abstract justification."
   =goal>
      ISA find-just
      justs nil
      previous =previous
      rule nil
   !bind! =method (actr~method =previous)
==>
   =rule>
      ISA rule
      name =method
      status 'unknown
   !bind! =premises (actr~nil-2-empty (actr~premises =previous))
   !output! "~&;;; Choosing Rule ~A with premises ~A" =rule =premises
   =goal>
      rule =rule
      premises =premises
   )

;;; The following production pops the goal to find a known justification, if a rule is
;;; chosen
(P pop-find-justification
   "IF   Goal is to find a known justification and the justification is already found,
    THEN pop Goal."
   =goal>
      ISA find-just
    - rule nil
==>
   !pop!
   )

;#  Produce Speech Acts

;;; The following production produces a speech act and sets the unconveyed slot of the
;;; governing DS chunk to EMPTY. Since no fact is derived by this production, NIL is
;;; added to the derived-list of the governing focus-space-DS
(P produce-SA
   "IF   Goal is a basic DS with a speech act to convey,
    THEN produce the speech act and pop Goal."
   =goal>
      ISA basic-DS
      to-convey  =speech-act
      unconveyed =speech-act
   !eval! (typep =speech-act 'sa+sa)
==>
   !eval! (sa~produce =speech-act)
   !eval! (dst~fact-derived-in-dominating-focus-space nil =goal)
   =goal>
      unconveyed empty
      status  'known
      content =speech-act
   )

(P omit-closing-sa
   "IF   Goal is a basic DS with a closing speech act to convey, and the corresponding
         focus space is small,
    THEN omit the speech act by setting the status of Goal to 'known."
   =goal>
      ISA basic-DS
      to-convey  =speech-act
      unconveyed =speech-act
      content-of =focus
   =focus>
      ISA focus-space-DS
```

```
   !eval! (sa~p =speech-act 'close)
   !eval! (dst~small-focus-space-p =focus)
==>
   =goal>
      purpose omit
      status  'known
      content nil
   )
(spp omit-closing-sa
     :a 0.01)

(P never-omit-closing-sa
   "IF   Goal is a basic DS with a closing speech act to convey that should never be
         omitted,
    THEN produce the speech act."
   =goal>
      ISA basic-DS
      to-convey  =speech-act
      unconveyed =speech-act
      content-of =focus
   =focus>
      ISA focus-space-DS

   !eval! (sa~p =speech-act '(qed end-definition end-theorem end-example end-exercise))
==>
   !eval! (sa~produce =speech-act)
   !eval! (dst~fact-derived-in-dominating-focus-space nil =goal)
   =goal>
      unconveyed empty
      status  'known
      content =speech-act
   )
(spp never-omit-closing-sa
     :a 0.005)

(P process-format
   "IF   Goal is a format chunk
    THEN produce the corresponding speech act."
   =goal>
      ISA format
      type =format
==>
   !eval! (sa~produce (sa~make-sa (eval =format))) ; get rid of quotes
   !pop!
   )

(P process-assoc
   "IF   Goal is a basic DS to convey a fact by a speech act,
    THEN mark the fact as known and set to-convey and unconveyed to the speech act"
   =goal>
      ISA basic-DS
      to-convey  =assoc
      unconveyed =assoc
   =assoc>
      ISA assoc
      proof-node =proof-node
      speech-act =speech-act
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
==>
   =fact>
      status 'known
   =goal>
      to-convey  =speech-act
```

```
      unconveyed =speech-act
   )

;;; The following production verbalizes a DS by producing the corresponding speech act.
;;; It ensures that the conveyed fact is marked as known and added to the derived-list
;;; of the governing focus-space-DS.
(P produce-verbalization
   "IF   Goal is a basic DS with a speech act to convey,
    THEN produce the speech act and pop the goal."
   =goal>
      ISA basic-DS
      to-convey  =proof-node
      unconveyed =speech-act
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
   !eval! (typep =speech-act 'sa+sa)
==>
   !eval! (sa~produce =speech-act)
   =fact>
      status 'known
   !eval! (dst~fact-derived-in-dominating-focus-space =fact =goal)
   =goal>
      unconveyed empty
      status  'known
      content =speech-act
   )
(spp produce-verbalization
     :a 0.005)

(P produce-derive
   "IF   Goal is a basic DS with the intention to inform about a proof node,
    THEN set unconveyed to a 'derive' SA choosing appropriate reference methods for
         the premises."
   =goal>
      ISA basic-DS
      purpose    inform
      to-convey  =proof-node
      unconveyed =proof-node
      rule       =rule
      premises   =premises
   =rule>
      ISA rule
      name =rule-name
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      succedent =succ
   =succ>
      ISA formula
      wff =wff
==>
   !bind! =args (list :reasons
                      (mapcar #'(lambda (pre) (ref~reference-succedent pre =goal))
                              (actr~empty-2-nil =premises))
                      :method (prex::ref~reference-rule (sa~make-rule =rule-name))
                      :conclusion =wff
                      :ds-node =goal)
   !bind! =speech-act (sa~make-sa 'derive =args)
   =goal>
      unconveyed =speech-act
   )
; This production should not be preferred to any omitting productions that are learned.
```

```
; Therefore we set higher costs.
(spp produce-derive
     :a 1)

(P produce-open
   "IF   Goal is a basic DS with the intention to inform about an open proof node,
    THEN set unconveyed to an 'open' SA."
   =goal>
      ISA basic-DS
      purpose    inform
      to-convey  =proof-node
      unconveyed =proof-node
      rule       =rule
   =rule>
      ISA rule
      name "OPEN"
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      succedent =succ
   =succ>
      ISA formula
      wff =wff
==>
   !bind! =args (list :conclusion =wff
                      :ds-node =goal)
   !bind! =speech-act (sa~make-sa 'open-step =args)
   =goal>
      unconveyed =speech-act
   )
(spp produce-open
     :a 0.001)

(P produce-hyp-intro
   "IF   Goal is a basic DS with the intention to inform about a hypothesis proof node,
    THEN set unconveyed to a 'Hyp-Intro' SA."
   =goal>
      ISA basic-DS
      purpose    inform
      to-convey  =proof-node
      unconveyed =proof-node
      rule       =rule
      premises   =premises
   =rule>
      ISA rule
      name "HYP"
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      succedent =succ
   =succ>
      ISA formula
      wff =wff
==>
   !bind! =args (list :hypothesis =wff :ds-node =goal)
   !bind! =speech-act (sa~make-sa 'hyp-intro =args)
   =goal>
      unconveyed =speech-act
   )
(spp produce-hyp-intro
     :a 0.001)

(P produce-par-intro
```

```
   "IF   Goal is a basic DS with the intention to inform about a parameter
         introduction proof node,
    THEN set unconveyed to a 'Par-Intro' SA."
   =goal>
      ISA basic-DS
      purpose    inform
      to-convey  =proof-node
      unconveyed =proof-node
      rule       =rule
      premises   =premises
   =rule>
      ISA rule
      name "DEC"
   =proof-node>
      ISA proof-node
      node =node
      fact =fact
   =fact>
      ISA fact
      succedent =succ
   =succ>
      ISA formula
      wff =wff
==>
   !bind! =args (list :parameter =wff :type (twa~judgment =node) :ds-node =goal)
   !bind! =speech-act (sa~make-sa 'par-intro =args)
   =goal>
      unconveyed =speech-act
   )
(spp produce-par-intro
     :a 0.001)

;## Operators (Bottom-Up Planning)

;;; The following production chooses the premises with the shortest subproof to be
;;; conveyed next, if there are unconveyed premises.
(P convey-premise-with-shortest-subproof
   "IF   Goal is a basic DS with unconveyed premises,
    THEN push as a subgoal to convey the unknown premises with the shortest subproof
         and remove it from the list of unconveyed premises."
   =goal>
      ISA basic-DS
      purpose     =purpose
      uncon-prems =premises
    - uncon-prems empty
==>
   !bind! =next (actr~get-next-node-to-explain =premises :choose 'shortest)
   !bind! =new-prems (actr~nil-2-empty (remove =next =premises))
   =proof-node>
      ISA proof-node
      node =next
   =subgoal>
      ISA basic-DS
      purpose    =purpose
      to-convey  =proof-node
      unconveyed =proof-node
      content-of =goal
      role       'contributes
   =goal>
      uncon-prems =new-prems
   !eval! (actr~add-to-chunk-slot =goal 'content =subgoal)
   !push! =subgoal
   )

;;; The following production pushes a basic DS with purpose inform.
(P verbalize-actual-node
   "IF   Goal is a basic DS with no unconveyed premises that has not been conveyed,
```

```
    THEN push the subgoal to verbalize the goal."
   =goal>
      ISA basic-DS
    - purpose     inform
      to-convey   =to-convey
    - unconveyed  empty
      uncon-prems empty
      rule        =rule
      premises    =premises
==>
   =verbalize>
      ISA basic-DS
      purpose    inform
      to-convey  =to-convey
      content-of =goal
      role       'verbalizes
      unconveyed =to-convey
      uncon-prems empty
      rule       =rule
      premises   =premises
      status     =status
   =goal>
      status     =status
      unconveyed empty
   !eval! (actr~add-to-chunk-slot =goal 'content =verbalize)
   !push! =verbalize
   )

;## Schemas (Top-Down Planning)

;;; The following production inserts a proof schema in a DST.
(P proof-schema
   "IF   Goal is a focus space DS to convey a proof,
    THEN insert a proof schema."
   =goal>
      ISA focus-space-DS
      purpose    =purpose
    - purpose repeat
      to-convey  =to-convey
    - unconveyed empty
   =to-convey>
      ISA proof
      proof =proof
==>
   !bind! =root (actr~conclusion =proof)
   !bind! =abstract (actr~abstract (twa~judgment =proof))
   !bind! =loc-ass (actr~local-assumptions =abstract)
   !bind! =mca-theorem (list :reasons
                             (mapcar #'(lambda (ass)
                                         (ref~fixed-reference-succedent ass 'implicit))
                                     =loc-ass)
                             :conclusion (actr~succedent =root))

   =begin-focus>
      ISA format
      type 'begin-focus
   =end-focus>
      ISA format
      type 'end-focus

   !bind! =sa-begin-th (sa~make-sa 'theorem)
   =begin-th>
      ISA basic-DS
      purpose    =purpose
      to-convey  =sa-begin-th
      unconveyed =sa-begin-th
      role       'opens
```

```
!eval! (sa~set-ds-node =sa-begin-th =begin-th)

=assumptions>
   ISA proof-node-list
   purpose =purpose
   nodes =loc-ass
   type 'assumptions
   content empty
!bind! =sa-th (sa~make-sa 'derive =mca-theorem)
=theorem>
   ISA basic-DS
   purpose    =purpose
   to-convey  =sa-th
   unconveyed =sa-th
   role       'verbalizes
!eval! (sa~set-ds-node =sa-th =theorem)

!bind! =sa-end-th (sa~make-sa 'end-theorem)
=end-th>
   ISA basic-DS
   purpose    =purpose
   to-convey  =sa-end-th
   unconveyed =sa-end-th
   role       'closes
!eval! (sa~set-ds-node =sa-end-th =end-th)

!bind! =th-focus-content (list =begin-th =assumptions =theorem =end-th)
=th-focus>
   ISA focus-space-DS
   purpose    =purpose
   to-convey  =sa-th
   unconveyed empty
   content    =th-focus-content
   content-of =goal
   role       'contributes
!eval! (actr~modify-chunks =th-focus-content 'content-of =th-focus)

!bind! =sa-begin-pf (sa~make-sa 'proof)
=begin-pf>
   ISA basic-DS
   purpose    =purpose
   to-convey  =sa-begin-pf
   unconveyed =sa-begin-pf
   role       'opens
!eval! (sa~set-ds-node =sa-begin-pf =begin-pf)

=root-node>
   ISA proof-node
   node =proof
=proof-root>
   ISA basic-DS
   purpose    =purpose
   to-convey  =root-node
   unconveyed =root-node
   role       'contributes

!bind! =sa-end-pf (sa~make-sa 'qed)
=end-pf>
   ISA basic-DS
   purpose    =purpose
   to-convey  =sa-end-pf
   unconveyed =sa-end-pf
   role       'closes
!eval! (sa~set-ds-node =sa-end-pf =end-pf)

!bind! =pf-focus-content (list =begin-pf =proof-root =end-pf)
!bind! =pf-focus-to-convey (list =root)
```

```
   =pf-focus>
      ISA focus-space-DS
      purpose    =purpose
      to-convey  =pf-focus-to-convey
      unconveyed empty
      content    =pf-focus-content
      content-of =goal
      role       'contributes
   !eval! (actr~modify-chunks =pf-focus-content 'content-of =pf-focus)

   !bind! =goal-content (list =th-focus =pf-focus)
   =goal>
      unconveyed empty
      content =goal-content
   !push! =pf-focus
   !push! =end-focus
   !push! =end-pf
   !push! =proof-root
   !push! =begin-pf
   !push! =begin-focus
   !push! =th-focus
   !push! =end-focus
   !push! =end-th
   !push! =theorem
   !push! =assumptions
   !push! =begin-th
   !push! =begin-focus
   )

(P judgment-schema
   "IF   Goal is a focus space DS with a proof node that has not been conveyed,
    THEN push the goal to convey a corresponding basic DS and add this to Goal's
         content."
   =goal>
      ISA focus-space-DS
      purpose    =purpose
      to-convey  =to-convey
    - unconveyed empty
   =to-convey>
      ISA proof-node
==>
   =subgoal>
      ISA basic-DS
      purpose =purpose
      to-convey =to-convey
   =goal>
      content =subgoal
      unconveyed empty
   !push! =subgoal
)
(spp judgment-schema
     :a 1)

;;; The following production conveys a hypothetical or parametric judgment by
;;; splitting it into two separate proof nodes.
(P convey-hyp-par-judgment
   "IF   Goal is a DS with a hypothetical or parametric judgment in its proof node,
    THEN open a focus to convey first the declaration, then the expression."
   =goal>
      ISA DS
      purpose   =purpose
      to-convey =proof-node
   =proof-node>
      ISA proof-node
      node  =node
      fact  =fact
      justs nil
```

```
   =fact>
      ISA fact
      succedent =succ
   =succ>
      ISA formula
      wff =pi-term
   !eval! (actr~hp-judgment-p =pi-term)
==>
   !bind! =split (actr~split-hp-judgment* =node)
   !bind! =first (butlast =split)
   !bind! =second (car (last =split))
   =ass>
      ISA proof-node-list
      purpose    =purpose
      nodes =first
      type 'assumptions
      content empty
   =conc-node>
      ISA proof-node
      node =second
   =conc>
      ISA basic-DS
      purpose    =purpose
      to-convey  =conc-node
      unconveyed =conc-node
      role       'contributes
   !bind! =contents (list =ass =conc)
   =focus>
      ISA focus-space-DS
      purpose    =purpose
      to-convey  =proof-node
      unconveyed empty
      content    =contents
      content-of =goal
      role       'contributes
   !eval! (actr~modify-chunks =contents 'content-of =focus)
   !bind! =goal-content (list =focus)
   =goal>
      content    =goal-content
      unconveyed empty
   !push! =focus ; necessary to carry on the entries in the derive slot
   !push! =conc
   !push! =ass
   )

;;; The following production processes the first node of a proof node list.
(P process-proof-node-list
   "IF   Goal is a proof node list
    THEN process the first node by pushing an appropriate DS node."
   =goal>
      ISA proof-node-list
      purpose =purpose
      nodes =nodes
    - nodes empty
      content =content
==>
   !bind! =actual (car =nodes)
   !bind! =rest (cdr =nodes)
   =node>
      ISA proof-node
      node =actual
   =DS>
      ISA basic-ds
      purpose    =purpose
      to-convey  =node
      unconveyed =node
      role       'contributes
```

```
      content-of =goal
   !bind! =new-content (append (actr~empty-2-nil =content) (list =ds))
   =goal>
      nodes =rest
      content =new-content
   !push! =ds
)

;;; The following production removes empty proof node list from the discourse
;;; structure tree.
(P pop-empty-proof-node-list
   "IF   Goal is an empty proof node list,
    THEN remove it from the discourse structure tree."
   =goal>
      ISA proof-node-list
      nodes nil
      content-of =parent-DS
      content =content
   =parent-DS>
      ISA DS
      content =parent-content
==>
   !bind! =new (actr~replace =content =goal =parent-content)
   =parent-DS>
      content =new
   !pop!
   )


;###  Reacting to User Interactions

;##  Productions for User Interaction

;;;; The following productions read and process interactions by the user.

;;; The following production reads a user interaction.
(P read-interaction
   "IF   Goal is to read a user interaction,
    THEN read the interaction with type and content and push the subgoal to process
         an interaction of that type and content with removing Goal form the goal
         stack."
   =goal>
      ISA interaction
      content nil
==>
   !bind! =type-and-content (ana~get-user-input)
   !bind! =type (car =type-and-content)
   !bind! =content (cdr =type-and-content)
   =subgoal>
      ISA interaction
      type =type
      content =content
   !focus-on! =subgoal
)

;;; The following production processes a user command.
(P process-command
   "IF   Goal is to process a user command,
    THEN push the subgoal to process a corresponding DS node with removing Goal
         from the goal stack"
   =goal>
      ISA interaction
      type "command"
      content =to-convey
==>
   !bind! =root (actr~find-root)
   =command>
```

```
      ISA basic-DS
      purpose request
      to-convey =to-convey
      content =to-convey
      content-of =root
      role 'commands
      status 'known
   !eval! (actr~add-to-chunk-slot =root 'content =command)
   !focus-on! =command
)


;;; The following production processes a user interruption.
(P process-interruption
   "IF   Goal is to process an interruption from the user,
    THEN push the subgoal to process a corresponding DS node with removing
         Goal from the goal stack."
   =goal>
      ISA interaction
      type "interruption"
      content =to-convey
==>
   =interrupt>
      ISA basic-DS
      purpose request
      to-convey =to-convey
      content =to-convey
      role 'interrupts
      status 'known
   !focus-on! =interrupt
   )

;# Commands

(P show-proof
   "IF   Goal is a basic DS with the request to show a proof
    THEN push the goal to show the proof."
   =goal>
      ISA basic-DS
      purpose request
      to-convey =sa
      content =sa
      content-of =parent
      role =role
   !eval! (find (cadr =role) '(commands interrupts))
   !eval! (sa~p =sa 'show!)
==>
   !bind! =purpose (sa~strategy =sa)
   !bind! =pf (actr~proof (sa~proof =sa))
   !bind! =name (actr~proof-name =pf)
   =proof>
      ISA proof
      proof =pf
      name =name
   =show>
      ISA focus-space-DS
      purpose =purpose
      to-convey =proof
      unconveyed =proof
      content-of =parent
   !eval! (actr~add-to-chunk-slot =parent 'content =show)
   !focus-on! =show
   )

(P repeat-proof
   "IF   Goal is a basic DS with the request to repeat the presentation of a proof
    THEN push the goal to repeat the presentation."
```

```
   =goal>
      ISA basic-DS
      purpose request
      to-convey =sa
      content =sa
      content-of =parent
      role =role
   !eval! (find (cadr =role) '(commands interrupts))
   !eval! (sa~p =sa 'repeat!)
   !bind! =name (sa~proof =sa)
   =proof>
      ISA proof
      proof =pf
      name =name
==>
   =repeat>
      ISA basic-DS
      purpose repeat
      to-convey =proof
      unconveyed =proof
      content-of =parent
      role 'contributes
   !eval! (actr~add-to-chunk-slot =parent 'content =repeat)
   !focus-on! =repeat
   )

(P exit-prex
   "IF   Goal is a basic DS with the request to exit P.rex
    THEN push the goals to say ``good bye'' and to exit."
   =goal>
      ISA basic-DS
      purpose request
      to-convey =sa
      content =sa
      content-of =parent
      role =role
   !eval! (find (cadr =role) '(commands interrupts))
   !eval! (sa~p =sa 'exit!)
==>
   !bind! =good-bye (sa~make-sa 'good-bye)
   =exit>
      ISA command
      type "exit"
   =bye>
      ISA basic-DS
      purpose inform
      to-convey =good-bye
      unconveyed =good-bye
      content-of =parent
   !eval! (actr~add-to-chunk-slot =parent 'content =bye)
   !push! =exit
   !push! =bye
   )

(P stop-prex
   "IF   Goal is a basic DS the request to stop P.rex
    THEN push the goal to stop it."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role =role
   !eval! (find (cadr =role) '(commands interrupts))
   !eval! (sa~p =to-convey 'stop!)
==>
   =stop>
      ISA command
```

```
      type "stop"
   !push! =stop
)

(P process-what-is
   "IF   Goal is a basic DS with the question what an object is
    THEN push the goal to explain that object."
   =goal>
      ISA basic-DS
      purpose request
      to-convey =sa
      content =sa
      role =role
   !eval! (find (cadr =role) '(commands interrupts))
   !eval! (sa~p =sa 'what-is?)
   !bind! =ds-node (sa~ds-node =sa)
   =ds-node>
      ISA basic-DS
      to-convey =node
   =node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      succedent =wff
   =wff>
      ISA formula
      wff =formula
==>
   !bind! =ex (sa~goal =sa)
   !bind! =expl (actr~make-proof-node =formula =ex)
   =explanandum>
      ISA identify
      object =expl
   =show>
      ISA basic-DS
      purpose explain
      to-convey =explanandum
      unconveyed =explanandum
      content-of =ds-node
   !eval! (actr~add-to-chunk-slot =ds-node 'content =show)
   !focus-on! =show
   )

;## Interruptions

(P too-implicit
   "IF   Goal is a DS with the notification that a step is too implicit,
    THEN push the goal to repair the presentation by a more explicit presentation."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'too-implicit)
   !bind! =node (sa~conclusion =to-convey)
==>
   !output! "~&;;; Node: ~S" =node
   =failed>
      ISA failed-presentation
      node =node
      problem "too implicit"
      remedy "more explicitly"
   =goal>
      content-of =node
      status 'known
   !eval! (actr~add-to-chunk-slot =node 'content =goal)
```

```
   !focus-on! =failed
   )

(P too-abstract
   "IF   Goal is a DS with the notification that a step is too abstract,
    THEN push the goal to repair the presentation at a lower level of abstraction."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'too-abstract)
   !bind! =node (sa~conclusion =to-convey)
==>
   !output! "~&;;; Node: ~S" =node
   =failed>
      ISA failed-presentation
      node =node
      problem "too abstract"
      remedy "check lower level"
   =goal>
      content-of =node
      status 'known
   !eval! (actr~add-to-chunk-slot =node 'content =goal)
   !focus-on! =failed
   )

(P too-difficult
   "IF   Goal is a DS with the notification that a step is too difficult,
    THEN push the goal to repair the presentation."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'too-difficult)
   !bind! =node (sa~conclusion =to-convey)
==>
   !output! "~&;;; Node: ~S" =node
   =failed>
      ISA failed-presentation
      node =node
      problem "too difficult"
      remedy "start repair"
   =goal>
      content-of =node
      status 'known
   !eval! (actr~add-to-chunk-slot =node 'content =goal)
   !focus-on! =failed
   )

(P too-detailed
   "IF   Goal is a DS with the notification that a step is too detailed,
    THEN push the goal to repair the presentation at a higher level of abstraction."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'too-detailed)
   !bind! =node (sa~conclusion =to-convey)
==>
   !output! "~&;;; Node: ~S" =node
   =failed>
      ISA failed-presentation
      node =node
      problem "too detailed"
```

```
      remedy "repair too detailed"
   =goal>
      content-of =node
      status 'known
   !eval! (actr~add-to-chunk-slot =node 'content =goal)
   !focus-on! =failed
   )

(P obvious
   "IF   Goal is a DS with the notification that a step is obvious,
    THEN push a dependency goal that allows for learning a production to omit that
         step."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'obvious-step)
   !bind! =node (sa~conclusion =to-convey)
   =node>
      ISA basic-DS
      rule =rule
   =rule>
      ISA rule
      name =rule-name
==>
   =before>
      ISA basic-DS
      purpose     inform
      rule        =rule
      uncon-prems empty
   =after>
      ISA basic-DS
      purpose    omit
      status     'known
      unconveyed empty
   =omit>
      ISA dependency
      goal =before
      modified =after
      constraints =rule
      specifics (inform omit empty 'known)
      dont-cares (nil)
   !focus-on! =omit
   )

(P trivial-derivation
   "IF   Goal is a DS with the notification that a derivation is trivial,
    THEN mark the derived fact as 'inferable."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'trivial-derivation)
   !bind! =ds-node (sa~conclusion =to-convey)
   =ds-node>
      ISA basic-DS
      to-convey =node
   =node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
==>
   !output! "~&;;; Node: ~S" =node
   =fact>
```

```
      status 'inferable
   !pop!
   )

(P continue-prex
   "IF   Goal is a DS with the request to continue,
    THEN pop Goal."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'continue!)
==>
   !pop!
)

(P stop-proof
   "IF   Goal is a DS with the request to stop the current proof,
    THEN stop the current proof."
   =goal>
      ISA DS
      purpose request
      to-convey =to-convey
      role 'interrupts
   !eval! (sa~p =to-convey 'stop!)
==>
   !eval! (actr~stop)
   !pop!
)

;## Reactions to Commands

(P process-exit
   "IF   Goal is to exit P.rex
    THEN throw a prex+exit error."
   =goal>
      ISA command
      type "exit"
==>
   !eval! (error (sys~make-condition 'prex+exit))
   )

(P process-stop
   "IF   Goal is to stop P.rex
    THEN throw a prex+stop error."
   =goal>
      ISA command
      type "stop"
==>
   !eval! (error (sys~make-condition 'prex+stop))
   )


;;; Repeating a proof
(P repeat
   "IF   Goal is a DS to repeat a proof,
    THEN reverbalize the last presentation of that proof."
   =goal>
      ISA DS
      purpose repeat
      to-convey =proof
      unconveyed =proof
      status 'unknown
==>
   !eval! (dst~verbalize actr*current-root)
   =goal>
```

```
      unconveyed nil
      status 'known
  )

;;; Identifying an Object
(P identification
   "IF   Goal is to explain an object,
    THEN push the goal to verbalize its identification."
   =goal>
      ISA basic-DS
      purpose    explain
      to-convey  =exp
      unconveyed =exp
   =exp>
      ISA identify
      object =expl
==>
   !bind! =sa (sa~make-sa 'identify
                          (list :object (twa~make-judgment (twa~context =expl)
                                                           (twa=proof-term =expl))
                                :class (twa~judgment =expl)))
   =verbalize>
      ISA basic-DS
      purpose    inform
      to-convey  =sa
      content-of =goal
      role       'verbalizes
      unconveyed =sa
      status     =status
   =goal>
      status     =status
      unconveyed empty
   !eval! (actr~add-to-chunk-slot =goal 'content =verbalize)
   !push! =verbalize
   )


;;; Reverbalization of a Proof Step
(P reverbalize
   "IF   Goal is a basic DS to reverbalize a proof step with all premises expressed
         explicitly,
    THEN push the goal to produce the appropriate 'derive' SA."
   =goal>
      ISA basic-DS
      purpose reverbalize
      to-convey =proof-node
   =inform>
      ISA basic-DS
      purpose inform
      to-convey =proof-node
      content-of =parent
      role 'verbalizes
      rule =rule
      premises =premises
   =rule>
      ISA rule
      name =rule-name
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      succedent =succ
   =succ>
      ISA formula
      wff =wff
==>
```

```
   !bind! =args (list :reasons
                      (mapcar #'(lambda (pre)
                                  (ref~fixed-reference-succedent pre 'explicit))
                              (actr~empty-2-nil =premises))
                      :method (prex::ref~reference-rule (sa~make-rule =rule-name))
                      :conclusion =wff)
   !bind! =new-sa (sa~make-sa 'derive =args)
   =reverbalize>
      ISA basic-DS
      purpose inform
      to-convey =proof-node
      content-of =goal
      role 'verbalizes
      unconveyed =new-sa
      rule =rule
      premises =premises
      status =status
   !eval! (sa~set-ds-node =new-sa =reverbalize)
   !bind! =content (list =reverbalize)
   =goal>
      status =status
      content =content
      content-of =parent
   !eval! (actr~add-to-chunk-slot =parent 'content =goal)
   !push! =reverbalize
   )
(spp reverbalize
     :a 0.03)


;## Repairing a Failed Presentation

;# Repair a Proof Step

(P start-repair-step
   "IF   Goal is to repair a proof step,
    THEN start by setting checking whether all premises were explicitly derived."
   =goal>
      ISA failed-presentation
      remedy "start repair"
==>
   =goal>
      remedy "check premises"
   )

(P stop-repair-step
   "IF   Goal is to repair a proof step and fact is known,
    THEN pop the goal."
   =goal>
      ISA failed-presentation
      fact-unknown nil
==>
   !pop!
   )
(spp stop-repair-step
     :a 0.001)

(P set-fact-unknown
   "IF   Goal is to repair a proof step whose fact is still marked as known,
    THEN mark the fact as unknown."
   =goal>
      ISA failed-presentation
      node =node
      fact-unknown t
   =node>
      ISA DS
      to-convey =proof-node
```

```
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
      status 'known
==>
   =fact>
      status 'unknown
   )
(spp set-fact-unknown
     :a 0.001)

;# Check Presence of Premises

(P repair-step-with-premises
   "IF   Goal is to repair a proof step that has some premises
    THEN check their explicitness."
   =goal>
      ISA failed-presentation
      node =node
      remedy "check premises"
   =node>
      ISA basic-DS
      premises =premises
    - premises empty
==>
   =goal>
      remedy "check explicitness"
   )

(P repair-step-with-empty-premises
   "IF   Goal is to repair a proof step that has no premises
    THEN check the existence of a lower level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      remedy "check premises"
   =node>
      ISA basic-DS
      premises empty
==>
   =goal>
      remedy "check lower level"
   )

(P repair-step-without-premises
   "IF   Goal is to repair a proof step that has no premises
    THEN check the existence of a lower level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      remedy "check premises"
   =node>
      ISA basic-DS
      premises nil
==>
   =goal>
      remedy "check lower level"
   )

;# Check Explicitness

(P repair-not-all-premises-explicit
   "IF   Goal is to repair a proof step whose premises were not all explicit,
    THEN try to repair it by reverbalizing it."
   =goal>
```

```
      ISA failed-presentation
      node =node
      remedy "check explicitness"
   =node>
      ISA DS
      to-convey =to-convey
   =inform>
      ISA basic-DS
      purpose inform
      to-convey =to-convey
      content =speech-act
      role 'verbalizes
   !eval! (not (sa~all-premises-explicit-p =speech-act))
==>
   =goal>
      remedy "reverbalize"
   )

(P repair-all-premises-explicit
   "IF   Goal is to repair a proof step whose premises were all explicit,
    THEN check whether all premises were understood."
   =goal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "check explicitness"
   =node>
      ISA DS
      to-convey =to-convey
   =inform>
      ISA basic-DS
      purpose inform
      to-convey =to-convey
      content =speech-act
      role 'verbalizes
   !eval! (sa~all-premises-explicit-p =speech-act)
==>
   =subgoal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "understood all premises?"
      fact-unknown =fact-unknown
   =goal>
      fact-unknown =fact-unknown
   !push! =subgoal
   )

;# Check Existence of Lower Level

(P repair-with-lower-level
   "IF   Goal is to check whether there is a lower level of abstraction and there is
         one,
    THEN use it."
   =goal>
      ISA failed-presentation
      node =node
      remedy "check lower level"
   =node>
      ISA basic-DS
      to-convey =to-convey
      rule =rule
   =to-convey>
      ISA proof-node
      justs =justs
   =rule>
      ISA rule
```

```
      name =name
   !eval! (actr~lower-level-p =name =justs)
==>
   =goal>
      remedy "lower level of abstraction"
   )

(P repair-without-lower-level
   "IF   Goal to check whether there is a lower level of abstraction, but there is
         none,
    THEN paraphrase the proof step."
   =goal>
      ISA failed-presentation
      node =node
      remedy "check lower level"
   =node>
      ISA basic-DS
      to-convey =to-convey
      rule =rule
   =to-convey>
      ISA proof-node
      justs =justs
   =rule>
      ISA rule
      name =name
   !eval! (not (actr~lower-level-p =name =justs))
==>
   =goal>
      remedy "paraphrase"
   )

;# Reverbalize

(P repair-by-reverbalizing
   "IF   Goal is to repair a proof step by reverbalizing it,
    THEN push the goal to first reverbalize it and then ask if the proof step is now
         understood."
   =goal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "reverbalize"
      fact-unknown t
   =node>
      ISA DS
      to-convey =to-convey
==>
   =reverbalize>
      ISA basic-DS
      purpose reverbalize
      to-convey =to-convey
      content-of =node
      role 'verbalizes
   =subgoal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "understood proof step?"
      fact-unknown =fact-unknown
   =goal>
      node =reverbalize
      fact-unknown =fact-unknown
   !push! =subgoal
   !push! =reverbalize
   )

;# Lower Level of Abstraction
```

```
(P lower-level-of-abstraction
   "IF   Goal is to repair a proof step by showing the next lower level of abstraction,
    THEN push the goals to first show the next lover level of abstraction and then ask
         whether the proof step is understood."
   =goal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "lower level of abstraction"
   =node>
      ISA basic-DS
      purpose =purpose
      to-convey =to-convey
      rule =rule
   =rule>
      ISA rule
==>
   =rule>
      status 'unknown
   =lower>
      ISA basic-DS
      purpose =purpose
      to-convey =to-convey
      unconveyed =to-convey
      content-of =node
      role 'contributes
   !eval! (actr~add-to-chunk-slot =node 'content =lower)
   =subgoal>
      ISA failed-presentation
      node =lower
      problem =problem
      remedy "understood proof step?"
      fact-unknown =fact-unknown
   =goal>
      lower =lower
      fact-unknown =fact-unknown
   !push! =subgoal
   !push! =lower
   )

;# Paraphrase a Proof Step

(P repair-by-paraphrasing-hyp
   "IF   Goal is to paraphrase a hypothesis,
    THEN push the goal to inform that this is a hypothesis."
   =goal>
      ISA failed-presentation
      node =node
      remedy "paraphrase"
   =node>
      ISA basic-DS
      rule =rule
      to-convey =proof-node
   =rule>
      ISA rule
      name "HYP"
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
==>
   !bind! =sa (sa~make-sa 'hypothesis)
   =inform>
      ISA basic-DS
      purpose inform
```

```
      to-convey =sa
      unconveyed =sa
      content-of =node
      role 'contributes
   !eval! (actr~add-to-chunk-slot =node 'content =inform)
   =fact>
      status 'known
   =goal>
      fact-unknown nil
   !push! =inform
   )
(spp repair-by-paraphrasing-hyp
     :a 0.01)

(P repair-no-lower-level
   "IF   Goal is to paraphrase that there is no lower level of abstraction,
    THEN push the goal to inform that ther is no lower level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "paraphrase"
   !eval! (find =problem '("too abstract"  "too difficult") :test #'string=)
   =node>
      ISA basic-DS
      rule =rule
      to-convey =proof-node
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
==>
   !bind! =sa (sa~make-sa 'least-abstract-available)
   =inform>
      ISA basic-DS
      purpose inform
      to-convey =sa
      unconveyed =sa
      content-of =node
      role 'contributes
   !eval! (actr~add-to-chunk-slot =node 'content =inform)
   =fact>
      status 'known
   =goal>
      fact-unknown nil
   !push! =inform
   )

(P repair-no-higher-level
   "IF   Goal is to paraphrase that there is no higher level of abstraction,
    THEN push the goal to inform that there is no higher level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      problem "too detailed"
      remedy "paraphrase"
   =node>
      ISA basic-DS
      rule =rule
      to-convey =proof-node
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
==>
```

```
   !bind! =sa (sa~make-sa 'most-abstract-available)
   =inform>
      ISA basic-DS
      purpose inform
      to-convey =sa
      unconveyed =sa
      content-of =node
      role 'contributes
   !eval! (actr~add-to-chunk-slot =node 'content =inform)
   =fact>
      status 'known
   =goal>
      fact-unknown nil
   !push! =inform
   )

;# Recursion on Premises

(P indices-to-premises
   "IF   Goal is a failed presentation with a 'premises' SA as answer,
    THEN replace the speech act by the corresponding premises nodes."
   =goal>
      ISA failed-presentation
      node =node
      remedy "understood all premises?"
      answer =answer
   !eval! (sa~p =answer 'premises)
   =node>
      ISA basic-DS
      premises =premises
==>
   !bind! =unknown-premises (actr~indices-2-premises (sa~items =answer) =premises)
   =goal>
      answer =unknown-premises
   )
(spp indices-to-premises
     :a 0.001)

(P recursion-on-premises
   "IF   Goal is to recurse on the premises,
    THEN push the goal to recurse on the first premises and remove that node from the
         premises to recurse on."
   =goal>
      ISA failed-presentation
      node =nodes
    - node empty
      problem =problem
      remedy "recursion on premises"
==>
   !bind! =first (car =nodes)
   !bind! =rest (actr~nil-2-empty (cdr =nodes))
   =failed-premise>
      ISA failed-presentation
      node =first
      problem =problem
      remedy "recursion on single premise"
   =goal>
      node =rest
   !push! =failed-premise
   )

(P recursion-on-single-premise
   "IF   Goal is to recurse on a single premise,
    THEN push the goal to repair the explanation of that premise."
   =goal>
      ISA failed-presentation
      node =node
```

```
      remedy "recursion on single premise"
   =proof-node>
      ISA proof-node
      node =node
   =premise>
      ISA basic-DS
    - purpose omit
      to-convey =proof-node
==>
   =goal>
      node =premise
      remedy "start repair"
   )

(P all-premises-done
   "IF   Goal is to recurse on the premises and the premises are empty,
    THEN pop Goal."
   =goal>
      ISA failed-presentation
      node empty
      remedy "recursion on premises"
==>
   =goal>
      fact-unknown nil
   !pop!
   )

;# Repair Explanation more Explicitly

(P repair-more-explicitly
   "IF   Goal is to repair a proof step by reverbalizing it more explicitly,
    THEN push the goal to reverbalize it."
   =goal>
      ISA failed-presentation
      node =node
      remedy "more explicitly"
      fact-unknown t
   =node>
      ISA DS
      to-convey =to-convey
==>
   =reverbalize>
      ISA basic-DS
      purpose reverbalize
      to-convey =to-convey
      content-of =node
      role 'verbalizes
   =goal>
      node =reverbalize
      fact-unknown nil
   !push! =reverbalize
   )

;# Repair too Detailed Explanation

(P repair-too-detailed
   "IF   Goal is to repair a proof step that was explained too detailed,
    THEN push the goal to show that step on the next higher level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      remedy "repair too detailed"
   =node>
      ISA basic-DS
      purpose =purpose
      to-convey =proof-node
      rule =rule
```

```
      premises =premises
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
   =find-just>
      ISA find-just
      previous =previous
      rule =rule
      premises =premises
==>
   !bind! =rule-name (actr~method =previous)
   =fact>
      status 'unknown
   =known>
      ISA rule
      name =rule-name
      status 'known
   =subgoal>
      ISA basic-DS
      purpose =purpose
      to-convey =proof-node
      role 'contributes
   =goal>
      fact-unknown nil
   !push! =subgoal
   )

(P least-detailed-available
   "IF   Goal is to repair a proof step and there is no higher level of abstraction,
    THEN paraphrase the proof step instead."
   =goal>
      ISA failed-presentation
      node =node
      remedy "repair too detailed"
   =node>
      ISA basic-DS
      purpose =purpose
      to-convey =proof-node
      rule =rule
      premises =premises
   =proof-node>
      ISA proof-node
      fact =fact
   =fact>
      ISA fact
   =find-just>
      ISA find-just
      previous nil
      rule =rule
      premises =premises
==>
   =goal>
      remedy "paraphrase"
   )

;## Clarification Dialogs

(P get-answer
   "IF   Goal is to get an answer of the user,
    THEN get the answer."
   =goal>
      ISA basic-DS
      purpose answer
      to-convey nil
==>
```

```
   !bind! =answer (cdr (ana~get-user-input))
   =goal>
      to-convey =answer
      content =answer
      unconveyed empty
      status 'known
   )

;# Ask if Proof Step Understood

(P understood-proof-step?
   "IF   Goal is to ask if a proof step is understood,
    THEN push the goals to enter a clarification dialog consisting of a question and
         an answer."
   =goal>
      ISA failed-presentation
      node =node
      remedy "understood proof step?"
==>
   !bind! =sa (sa~make-sa 'understood? (list :goal nil))
   =question>
      ISA basic-DS
      purpose ask
      to-convey =sa
      unconveyed =sa
      role 'contributes
   !eval! (sa~set-ds-node =sa =question)

   =answer>
      ISA basic-DS
      purpose answer
      to-convey =response
      role 'contributes
      status =status
   !bind! =contents (list =question =answer)
   =clarification>
      ISA focus-space-DS
      purpose clarify
      to-convey =sa
      content =contents
      content-of =node
      status =status
      role 'clarifies
   !eval! (actr~modify-chunks =contents 'content-of =clarification)
   ; add =clarification to content of the DS node =node
   !eval! (actr~add-to-chunk-slot =node 'content =clarification)
   =goal>
      answer =response
   !push! =clarification
   !push! =answer
   !push! =question
   )
(spp understood-proof-step?
     :a 0.7)

(P proof-step-understood
   "IF   Goal is to repair a proof step and the user answered to the question whether
         he understood the proof step with yes,
    THEN pop the goal."
   =goal>
      ISA failed-presentation
      remedy "understood proof step?"
      answer =yes
   !eval! (sa~p =yes 'yes)
==>
   =goal>
      fact-unknown nil
```

```
   !pop!
   )

(P proof-step-not-understood-1
   "IF   Goal is to repair a proof step and the user answered to the question whether
         he understood the proof step with no,
    THEN ask whether he understood all premises."
   =goal>
      ISA failed-presentation
      node =node
      remedy "understood proof step?"
      answer =no
   !eval! (sa~p =no 'no)
==>
   =goal>
      remedy "understood all premises?"
      answer nil
   )
(spp proof-step-not-understood-1
     :a 0.04)

(P proof-step-not-understood-2
   "IF   Goal is to repair a proof step and the user answered to the question whether
         he understood the proof step with no, and he has been asked whether he
         understood all premises,
    THEN check whether there is a lower level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      remedy "understood proof step?"
      answer =no
   !eval! (sa~p =no 'no)
   =premises>
      ISA failed-presentation
      node =node
      remedy "understood all premises?"
==>
   =goal>
      remedy "check lower level"
      answer nil
   )
(spp proof-step-not-understood-2
     :a 0.03)

(P proof-step-not-understood-3
   "IF   Goal is to repair a proof step and the user answered to the question whether
         he understood the proof step with no, and there is a lower level of
         abstraction,
    THEN push the goal to repair the step on the next lower level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "understood proof step?"
      answer =no
   !eval! (sa~p =no 'no)
   =higher>
      ISA failed-presentation
      remedy "lower level of abstraction"
      lower =node
==>
   =recursion>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "start repair"
      fact-unknown =fact-unknown
```

```
   =higher>
      remedy "done"
   =goal>
      answer nil
      fact-unknown =fact-unknown
   !push! =recursion
   )
(spp proof-step-not-understood-3
     :a 0.02)

;# Ask if Premises Understood

(P understood-all-premises?
   "IF   Goal is to ask the user whether he understood all premises of a proof step,
    THEN push the goals to enter a clarification dialog consisting of a question and
         an answer."
   =goal>
      ISA failed-presentation
      node =node
      remedy "understood all premises?"
==>
   !bind! =sa (sa~make-sa 'understood? (list :goal 'premises))
   =question>
      ISA basic-DS
      purpose ask
      to-convey =sa
      unconveyed =sa
      role 'contributes
   !eval! (sa~set-ds-node =sa =question)

   =answer>
      ISA basic-DS
      purpose answer
      to-convey =response
      role 'contributes
      status =status
   !bind! =contents (list =question =answer)
   =clarification>
      ISA focus-space-DS
      purpose clarify
      to-convey =sa
      content =contents
      content-of =node
      status =status
      role 'clarifies
   !eval! (actr~modify-chunks =contents 'content-of =clarification)
   ; add =clarification to content of the DS node that presented =node
   !eval! (actr~add-to-chunk-slot =node 'content =clarification)
   =goal>
      answer =response
   !push! =clarification
   !push! =answer
   !push! =question
   )
(spp understood-all-premises?
     :a 0.7)

(P all-premises-understood
   "IF   Goal is to repair a proof step and the user answered to the question whether
         he understood all premises of the proof step with yes,
    THEN re-explain the proof step on a lower level of abstraction."
   =goal>
      ISA failed-presentation
      node =node
      remedy "understood all premises?"
      answer =yes
   !eval! (sa~p =yes 'yes)
```

```
==>
   =goal>
      remedy "check lower level"
      answer nil
   )

(P not-all-premises-understood-explicit
   "IF   Goal is to repair a proof step and one of the premises has not been understood,
    THEN repair that premise."
   =goal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "understood all premises?"
      answer =answer
   !eval! (listp =answer) ; must be a list of DS nodes
   =node>
      ISA DS
      to-convey =to-convey
==>
   =failed-premises>
      ISA failed-presentation
      node =answer
      problem =problem
      remedy "recursion on premises"
   =reverbalize>
      ISA basic-DS
      purpose reverbalize
      to-convey =to-convey
      content-of =node
      role 'verbalizes
   =subgoal>
      ISA failed-presentation
      node =node
      problem =problem
      remedy "understood proof step?"
      fact-unknown =fact-unknown
   =goal>
      fact-unknown =fact-unknown
   !push! =subgoal
   !push! =reverbalize
   !push! =failed-premises
   )

(P all-premises-repaired
   "IF   Goal is to repair a proof step and all premises are repaired
    THEN ask whether the proof step is now understood."
   =goal>
      ISA failed-presentation
      remedy "understood all premises?"
      answer empty
==>
   =goal>
      remedy "understood proof step?"
      answer nil
   )

(P not-all-premises-understood-implicit
   "IF   Goal is to repair a proof step and some premises have not been understood,
    THEN ask which ones."
   =goal>
      ISA failed-presentation
      remedy "understood all premises?"
      answer =no
   !eval! (sa~p =no 'no)
==>
   =goal>
```

```
      remedy "which premises not understood?"
      answer nil
   )

;# Ask Which Premises Not Understood

(P which-premises?
   "IF   Goal is to ask which premises have not been understood,
    THEN push the goals to enter a clarification dialog consisting of a question and
         an answer."
   =goal>
      ISA failed-presentation
      node =ds-node
      remedy "which premises not understood?"
      answer nil
==>
   !bind! =sa (sa~make-sa 'which-not-known? (list :goal 'premises))
   =question>
      ISA basic-DS
      purpose ask
      to-convey =sa
      unconveyed =sa
      role 'contributes
   !eval! (sa~set-ds-node =sa =question)

   =answer>
      ISA basic-DS
      purpose answer
      to-convey =response
      role 'contributes
      status =status
   !bind! =contents (list =question =answer)
   =clarification>
      ISA focus-space-DS
      purpose clarify
      to-convey =sa
      content =contents
      content-of =ds-node
      status =status
      role 'clarifies
   !eval! (actr~modify-chunks =contents 'content-of =clarification)
   ; add =clarification to content of the DS node that presented =node
   !eval! (actr~add-to-chunk-slot =ds-node 'content =clarification)
   =goal>
      remedy "understood all premises?"
      answer =response
   !push! =clarification
   !push! =answer
   !push! =question
   )


;### Learning New Productions

(P learn-production
   "IF   GOAL is a dependency,
    THEN learn a new production by popping the goal."
   =goal>
      ISA dependency
==>
   !pop!
   )
```

Appendix D

The Interface

In this chapter, we shall give technical definitions we omitted in Section 7.2. First, we shall define arguments to instructions in Section D.1. Next, in Section D.2 we shall define the *P.rex* Markup Language (PML) formally.

D.1 Instructions

Recall from Section 7.2.1 that the generic interface takes as input a triple (A, a, s), where a is the action A taken by the user and s the arguments to a. In this section, we shall define the arguments to instructions for the generic interface, that is, we consider the case where A = `instruction`. Recall that instructions a are defined by the following grammar:

instruction ::= `NewProof` | `ReadSgnEntry` | `ReadInfoEntry` | `ReadJudgment` | `Reset` | `SetTheory` | `Load`

The following arguments s are allowed:

`NewProof`: s is the name of the new proof. A new current signature will be defined.

`ReadSgnEntry`: s is a signature entry (cf. Appendix A.1 for the syntax) to be added to the current signature.

`ReadInfoEntry`: s is an info entry.

`ReadJudgment`: s is a judgment to be explained.

`Reset`: s is empty or the name of a theory, which becomes the current theory.

`SetTheory`: s is the name of the new current theory.

`Load`: s is the pathname to a signature file, which will be loaded.

D.2 *P.rex* Markup Language

The *P.rex Markup Language (PML)* allows for the inclusion of layout information in the text, which is interpreted by the Emacs interface. As a small sublanguage of the Hypertext Markup Language HTML, it uses HTML syntax to encode the layout information. Note that hyperlinks are not supported in PML.

A PML string has one of the following forms:

`<directive>` or `<directive arguments*>` *text* `</directive>`

where `directive` is the PML directive, `arguments` is a string of the form `keyword=value` and *text* is arbitrary text.

The following directives are supported:

`<BR>` A new line is started.

`<I>`*text*`</I>` The text is set in italics.

`<B>`*text*`</B>` The text is set in boldface.

`<FONT COLOR=`*color*`>`*text*`</FONT>` The text is set in color *color*.

Note that the layout information can be nested, as the following example illustrates:

Example D.1
The string

```
"This text is normal, <I>italics</I>, <B>bold, <I>bold
 and italics</I>, <FONT COLOR=red>bold and red <I> and
 italics</I></FONT>, only bold</B>, normal again and
 <FONT COLOR=blue>blue</FONT>."
```

is displayed in the emacs interface as follows:

> This text is normal, *italics*, **bold**, ***bold and italics***, **bold and red *and italics***, **only bold**, normal again and blue.

□

Bibliography

Anderson, J. R. and Lebiere, C. (1998). *The Atomic Components of Thought.* Lawrence Erlbaum.

Anderson, J. R. (1990). *The Adaptive Character of Thought.* Studies in Cognition. Lawrence Erlbaum Associates, Hillsdale, NJ.

Andrews, P. B. (1980). Transforming matings into natural deduction proofs. In *Proceedings of the 5th International Conference on Automated Deduction*, pages 281–292. Springer Verlag.

Barendregt, H. P. (1992). Lambda calculi with types. In Abramsky, S., Gabbay, D. M., and Maibaum, T. S. E., editors, *Handbook of Logic in Computer Science*, Volume 2, pages 117–309. Oxford University Press.

Bateman, J. A., Kasper, R. T., Moore, J. D., and Whitney, R. A. (1990). A general organization of knowledge for natural language processing: the Penman upper model. Technical report, University of Southern California, Information Science Institute.

Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Melis, E., Meier, A., Schaarschmidt, W., Siekmann, J., and Sorge, V. (1997). ΩMEGA: Towards a mathematical assistant. In McCune [1997a], pages 252–255.

Buchberger, B. (1997). Natural language proofs in nested cells representation. In Siekmann, J., Pfenning, F., and Huang, X., editors, *Proceedings of the First International Workshop on Proof Transformation and Presentation*, pages 15–16, Schloss Dagstuhl, Germany.

Bundy, A., Harmelen, F. van, Horn, C., and Smaill, A. (1990). The Oyster-Clam System. In Stickel, M., editor, *Proceedings of the 10th Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 647–648, Kaiserslautern, Germany. Springer Verlag.

Cahill, L., Doran, C., Evans, R., Mellish, C., Paiva, D., Reape, M., Scott, D., and Tipper, N. (1999). In search of a reference architecture for NLG systems. In *Proceedings of the 7th European Workshop on Natural Language Generation*, pages 77–85, Toulouse, France.

Carpenter, P. A. and Just, M. A. (1995). 3CAPS—simulation systems for modeling a limited-capacity working memory. In *Proceedings of the 17th Annual Conference of the Cognitive Science Society.*

Cawsey, A. (1990). Generating explanatory discourse. In Dale, R., Mellish, C., and Zock, M., editors, *Current Research in Natural Language Generation*, number 4 in Cognitive Science Series, pages 75–101. Academic Press, San Diego, CA.

Cheikhrouhou, L. (in prep.). *A New Proof Planning Framework and Proofs by Diagonalization in* ΩMEGA. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany.

Chester, D. (1976). The translation of formal proofs into English. *AI*, 7:178–216.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68.

Coquand, T. and Huet, G. (1988). The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120.

Coquand, T. (1991). An algorithm for testing type conversion in type theory. In Huet, G. and Plotkin, G., editors, *Logical Frameworks*, pages 255–279. Cambridge University Press.

Coscoy, Y., Kahn, G., and Théry, L. (1995). Extracting text from proofs. In Dezani-Ciancaglini, M. and Plotkin, G., editors, *Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science, pages 109–123. Springer Verlag.

Dahn, B. I., Gehne, J., Honigmann, T., and Wolf, A. (1997). Integration of automated and interactive theorem proving in ILF. In McCune [1997a], pages 57–60.

Davis, M. (1957). A computer program for presburger's algorithm. In *Summary of talks presented at the Summer Institute for Symbolic Logic*, pages 215–233, Cornell University. Reprinted in [Siekmann and Wrightson, 1983].

de Bruijn, N. G. (1970). The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, number 125 in Lecture Notes in Mathematics, pages 29–61. Springer Verlag.

Dowek, G., Huet, G., and Werner, B. (1993). On the definition of the eta-long normal form in type systems of the cube. In Geuvers, H., editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, Nijmegen, The Netherlands.

Edgar, A. and Pelletier, F. J. (1993). Natural language explanation of natural deduction proofs. In *Proceedings of the 1st Conference of the Pacific Association for Computational Linguistics*, Vancouver, Canada. Centre for Systems Science, Simon Fraser University.

Elhadad, M. and Robin, J. (1992). Controlling content realization with functional unification grammars. In Dale, R., Hovy, E., Rösner, D., and Stock, O., editors, *Aspects of Automated Natural Language Generation*, number 587 in Lecture Notes in Artificial Intelligence, pages 89–104. Springer Verlag.

Feigenbaum, E. E. and Feldman, J., editors. (1995). *Computers and Thought*. AAAI Press / The MIT Press.

Felty, A. and Miller, D. (1988). Proof explanation and revision. Technical report MC-CIS-88-17, University of Pennsylvania, Philadelphia, PA.

Fiedler, A. (1996). Mikroplanungstechniken zur Präsentation mathematischer Beweise. Master Thesis, Computer Science Department, Universität des Saarlandes, Saarbrücken, Germany.

Franke, A. and Kohlhase, M. (2000). System description: MBASE, an open mathematical knowledge base. In McAllester [2000], pages 455–459.

Ganzinger, H., editor. (1999). *Automated Deduction — CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence. Springer Verlag.

Gelernter, H. (1959). Realization of a geometry-theorem proving machine. In *Proceedings of an International Conference on Information Processing*, pages 273–282, Paris. Reprinted in [Siekmann and Wrightson, 1983] and in [Feigenbaum and Feldman, 1995].

Gentzen, G. (1935). Untersuchungen über das logische Schließen I & II. *Mathematische Zeitschrift*, 39:176–210, 572–595.

Geuvers, J. H. (1993). *Logics and Type Systems*. Ph.D. thesis, Katholieke Universiteit Nijmegen.

Ghani, N. (1997). Eta-expansions in dependent type theory — the Calculus of Constructions. In de Groote, P. and Hindley, J., editors, *Proceedings of the 3rd International Conference on Typed Lambda Calculus and Applications (TLCA)*, number 1210 in Lecture Notes in Computer Science, pages 164–180, Nancy, France. Springer Verlag.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures de l'arithmetique d'ordre supérieur*. Ph.D. thesis, Université de Paris VII, France.

Grosz, B. J. and Sidner, C. L. (1986). Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12(3):175–204.

Halliday, M. A. K. (1994). *An Introduction to Functional Grammar*. Edward Arnold, 2. edition.

Harper, R. and Pfenning, F. (1999). On equivalence and canonical forms in the LF type theory. Technical report CMU-CS-99-159, School of Computer Science, Carnegie Mellon University.

Harper, R., Honsell, F., and Plotkin, G. (1993). A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184.

Holland-Minkley, A. M., Barzilay, R., and Constable, R. L. (1999). Verbalization of high-level formal proofs. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99) and Eleventh Innovative Application of Artificial Intelligence Conference (IAAI-99)*, pages 277–284. AAAI Press.

Horacek, H. (1997a). Generating referential descriptions in multimedia environments. In André, E., editor, *Referring Phenomena in a Multimedia Context and Their Computational Treatment*, pages 59–66, Madrid, Spain.

Horacek, H. (1997b). A model for adapting explanations to the user's likely inferences. *User Modeling and User-Adapted Interaction*, 7:1–55.

Horacek, H. (1999). Presenting proofs in a human-oriented way. In Ganzinger [1999], pages 142–156.

Hovy, E. H. (1988). *Generating Natural Language under Pragmatic Constraints*. Lawrence Erlbaum, Hillsdale, NJ.

Hovy, E. D. (1991). Approaches to the planning of coherent text. In Paris et al. [1991], pages 83–102.

Hovy, E. H. (1992). Sentence planning requirements for automated explanation generation. Diamod 23, GMD, St. Augustin, Germany.

Hovy, E. H. (1993). Automated discourse generation using discourse structure relations. *Artificial Intelligence*, 63:341–385.

Howard, W. A. (1980). The formulae-as-types notion of construction. In Seldin, J. P. and Hindley, J. R., editors, *To H. B. Curry: Essays on combinatory logic, lambda calculus and formalism*, pages 479–490. Academic Press. Hitherto unpublished note of 1969, rearranged, corrected and annotated by Howard, 1979.

Huang, X. and Fiedler, A. (1997). Proof verbalization as an application of NLG. In Pollack, M. E., editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 965–970, Nagoya, Japan. Morgan Kaufmann.

Huang, X. (1994a). *Human Oriented Proof Presentation: A Reconstructive Approach*. Ph.D. thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany.

Huang, X. (1994b). Planning argumentative texts. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 329–333, Kyoto, Japan.

Huang, X. (1994c). Reconstructing proofs at the assertion level. In Bundy, A., editor, *Proceedings of the 12th Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 738–752, Nancy, France. Springer Verlag.

INLG. (1994). *Proceedings of the 7th International Workshop on Natural Language Generation*, Kennebunkport, ME.

Joshi, A. K. (1985). An introduction to TAGs. Technical report MS-CIS-86-64, Department of Computer and Information Science, Moore School, University of Pennsylvania, Philadelphia, PA.

Kamp, H. (1981). A theory of truth and semantic representation. In Groenendijk, J., Janssen, T., and Stokhof, M., editors, *Formal Methods in the Study of Language*, pages 277–322. Mathematisch Centrum Tracts, Amsterdam, The Netherlands.

Kilger, A. and Finkler, W. (1995). Incremental generation for real–time applications. Research Report RR-95-11, DFKI, Saarbrücken, Germany, July.

Kursawe, P. (1982). Transformation eines Resolutionsbeweises: Der erste Schritt auf dem Weg zum natürlichen Schließen. Master Thesis, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany.

Leron, U. (1983). Structuring mathematical proofs. *The American Mathematical Monthly*, 90:174–185.

Leron, U. (1985). Heuristic presentations: the role of structuring. *For the Learning of Mathematics*, 5(3):7–13, November.

Lingenfelder, C. (1990). *Transformation and Structuring of Computer Generated Proofs*. Ph.D. thesis, Universität Kaiserslautern, Kaiserslautern, Germany.

Longo, G. and Moggi, E. (1988). Constructive natural deduction and its modest interpretation. Technical Report CMU-CS-88-131, Carnegie Mellon University, Pittsburgh, PA.

Luger, G. F., editor. (1995). *Computation and Intelligence.* AAAI Press / The MIT Press.

Mann, W. C. and Thompson, S. A. (1987). Rhetorical structure theory: A theory of text organization. ISI Reprint Series ISI/RS-87-190, Univerisity of Southern California, Information Science Institute, Marina del Rey, CA.

Martin-Löf, P. (1985). Truth of a proposition, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza, Firenze, Italy.

Martin-Löf, P. (1996). On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60.

Matthiesen, C. M. I. M. and Bateman, J. A. (1990). *Text Generation and Systemic-Functional Linguistics.* Pinter Publishers, London, United Kingdom.

McAllester, D., editor. (2000). *Automated Deduction – CADE-17*, number 1831 in Lecture Notes in Artificial Intelligence. Springer Verlag.

McCune, W. W. (1994). Otter 3.0 reference manual and guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA.

McCune, W., editor. (1997a). *Proceedings of the 14th Conference on Automated Deduction*, number 1249 in Lecture Notes in Artificial Intelligence, Townsville, Australia. Springer Verlag.

McCune, W. (1997b). Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276.

McDonald, D. D. (1984). Natural language generation as a computational problem. In Brady, M. and Berwick, R. C., editors, *Computational Models of Discourse.* The M. I. T. Press, Cambridge, Massachusetts/London.

McKeown, K. R. (1985). *Text Generation.* Cambridge University Press, Cambridge, United Kingdom.

Meier, A. (1997). Übersetzung automatisch erzeugter Beweise auf Faktenebene. Master Thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany.

Meier, A. (2000). System description: TRAMP: Transformation of machine-found proofs into ND-proofs at the assertion level. In McAllester [2000], pages 460–464.

Melis, E. and Leron, U. (1999). A proof presentation suitable for teaching proofs. In Lajoie, S. P. and Vivet, M., editors, *Artificial Intelligence in Education*, Volume 50 of *Frontiers in Artificial Intelligence and Applications*, pages 483–490. IOS Press.

Melis, E., Andres, E., Franke, A., Goguadse, G., Libbrecht, P., Pollet, M., and Ullrich, C. (2001). ACTIVEMATH system description. In Moore, J. D., Redfield, C. L., and Johnson, W. L., editors, *Artificial Intelligence in Education*, Volume 68 of *Frontiers in Artificial Intelligence and Applications*, pages 580–582. IOS Press.

Meteer, M. W. (1991). Bridging the generation gap between text planning and linguistic realization. *Computational Intelligence*, 7:296–304.

Meteer, M. W. (1992). *Expressibility and the Problem of Efficient Text Planning.* Pinter Publishes, London, United Kingdom.

Meyer, D. E. and Kieras, D. E. (1997a). EPIC: A computational theory of executive cognitive processes and multiple-task performance: Part 1. Basic mechanisms. *Psychological Review*, 104:3–65.

Meyer, D. E. and Kieras, D. E. (1997b). EPIC: A computational theory of executive cognitive processes and multiple-task performance: Part 2. Accounts of psychological refractory-period phenomena. *Psychological Review*, 104:749–791.

Miller, D. (1984). Expansion tree proofs and their conversion to natural deduction proofs. In Shostak, R. E., editor, *Proceedings of the 7th International Conference on Automated Deduction*, number 170 in Lecture Notes in Computer Science, pages 375–303. Springer Verlag.

Mooney, D. J., Carberry, S., and McCoy, K. (1991). Capturing high-level structure of naturally occurring, extended explanations using bottom-up strategies. *Computational Intelligence*, 7:334–356.

Moore, J. D. and Swartout, W. R. (1991). A reactive approach to explanation: Taking the user's feedback into account. In Paris et al. [1991], pages 3–48.

Moore, J. D. (1989). *A Reactive Approach to Explanation in Expert and Advice-Giving Systems.* Ph.D. thesis, University of California, Los Angeles, CA.

Newell, A., Shaw, C., and Simon, H. (1957). Empirical explorations of the logic theory machine. In *Proceedings of the Western Joint Computer Conference (WJCC)*, pages 218–239. Reprinted in [Siekmann and Wrightson, 1983], in [Feigenbaum and Feldman, 1995], and in [Luger, 1995].

Newell, A. (1990). *Unified Theories of Cognition.* Havard University Press, Cambridge, MA.

Ochs, E. (1979). Planned and unplanned discourse. *Syntax and Semantics*, 12:51–80.

Panaget, F. (1994). Using a textual representational level component in the context of discourse or dialogue generation. In INLG [1994], pages 127–136.

Paris, C. L., Swartout, W. R., and Mann, W. C., editors. (1991). *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, Boston, MA, USA. Kluwer.

Paris, C. (1988). Tailoring objects descriptions to a user's level of expertise. *Computational Linguistics*, 14:64–78.

Paris, C. (1991a). The role of the user's domain knowledge in generation. *Computational Intelligence*, 7:71–93.

Paris, C. L. (1991b). Generation and explanation: Building an explanation facility for the explainable expert systems framework. In Paris et al. [1991], pages 49–82.

Pfenning, F. and Schürmann, C. (1999). System description: Twelf — a meta-logical framework for deductive systems. In Ganzinger [1999], pages 202–206.

Pfenning, F. (1987). *Proof Transformations in Higher-Order Logic.* Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Pfenning, F. (in prep.). *Computation and Deduction.* Cambridge University Press. Draft from April 1997 available electronically.

Reichman, R. (1985). *Getting Computers to Talk Like You and Me.* MIT Press.

Reiter, E. (1994). Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In INLG [1994], pages 163–170.

Reithinger, N. (1991). *Eine parallele Architektur zur inkrementellen Generierung multimodaler Dialogbeiträge.* Ph.D. thesis, Universität des Saarlandes, Saarbrücken, Germany.

Rich, E. and Knight, K. (1991). *Artificial Intelligence.* McGraw-Hill.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence—A Modern Approach.* Prentice Hall, Upper Saddle River, NJ.

Sibun, P. (1990). The local organization of text. In McKeown, K. R., Moore, J. D., and Nirenburg, S., editors, *Proceedings of the 5th International Natural Language Generation Workshop*, pages 120–127, Dawson, PA.

Siekmann, J. and Wrightson, G., editors. (1983). *Automation of Reasoning 1: Classical Papers on Computational Logic 1957–1966*, Symbolic Computation. Springer Verlag.

Thompson, S. (1991). *Type Theory and Functional Programming.* International Computer Science Series. Addison Wesley.

van Benthem Jutting, L. S. (1993). Typing in pure type systems. *Information and Computation*, 105(1):30–41.

Wahlster, W., André, E., Finkler, W., Profitlich, H.-J., and Rist, T. (1993). Plan-based integration of natural language and graphics generation. *Artificial Intelligence*, 63:387–427.

Walker, M. A. and Rambow, O. (1994). The role of cognitive modeling in achieving communicative intentions. In INLG [1994], pages 171–180.

Weidenbach, C. (1997). SPASS: Version 0.49. *Journal of Automated Reasoning*, 18(2):247–252. Special Issue on the CADE-13 Automated Theorem Proving System Competition.

Wick, M. R. and Thompson, W. B. (1992). Reconstructive expert system explanation. *Artificial Intelligence*, 54:33–70.

Zukerman, I. and McConachy, R. (1993). Generating concise discourse that addresses a user's inferences. In Bajcsy, R., editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1202–1207, Chambery, France. Morgan Kaufmann, San Mateo, CA.

Zukerman, I. (1991). Using meta-comments to generate fluent text in a technical domain. *Computational Intelligence*, 7:276–295.

Index