LogiCalc: An Environment For Interactive Proof Development

Denys Duchier

YALEU/CSD/RR #862

July 1991

This work was supported in part by DARPA/BRL Contract DAAA15-87-K-0001, NSF grant IRI-8812790, and DARPA/ONR Contract N00014-91-J-1577.

Abstract

LogiCalc: An Environment For Interactive Proof Development

Denys Duchier

Yale University

1991

LogiCalc facilitates theorem-proving work with typical predicate-calculus formalizations of AI problems. It follows the LCF model: goals are analysed top-down by refining them into plans and subgoals. Proofs are synthesized bottom-up by executing the plans' validations. LogiCalc is based on a system of Natural Deduction for the sequent calculus, with a double twist: (1) formulae are skolemized by default and unification is built into the logic, (2) assumption sets are implemented with an ATMS. The underlying graph helps make the most of every answer: (1) goals are grouped in equivalence classes and thus share their answers, (2) related classes are linked together and communicate their answers to one another. Refinements often require premises. A transparent DWIM *detaching* mechanims allows the use of a subformula of an assertion as a premise. The system automatically determines the auxiliary goals required to infer it and adds them to the resulting plan; mismatches contribute equality goals. Proofs are generalized by a technique related to Explanation-Based Generalization, but operating on a reification of the logic. Finally, proof summaries can be automatically generated for publication in LaTeX. Heuristics determine how to compress/omit/in-line uninteresting parts. The thesis also contributes an axiomatization of Qualitative Physics.

LogiCalc: An Environment For Interactive Proof Development

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Denys Duchier

December 1991

© Copyright by Denys Duchier 1992
All Rights Reserved

To my brother, in Africa

Acknowledgements

Hervé Gallaire gave me the opportunity to work at CGE's AI lab and helped me get started in this field.

From my advisor, Drew McDermott, I learned so much that any attempt to acknowledge my intellectual debt to him can only be inadequate. I thank him for seeing me through all these years, for putting up with the slow pace of my writing, and for always being there to give me the benefit of his erudition and of his remarkable insight, or to share in a great hack.

I thank my readers P. Anandan and L. Henschen for their patience and for having sustained the extraordinary fortitude required by their task. I am also grateful to Larry Birnbaum for remaining supportive throughout the years.

Many people helped make my protracted stay at Yale an enjoyable experience. Steve Hanks and Jim Firby provided moral support and generated some fascinating discussions in what had to be AI's coolest office. Jim Firby and Brad Alpert introduced me to the joys of rock climbing. Andrew Gelsey was the perfect officemate, knowledgeable and intellectually challenging.

Elisabeth Rogers and Dominique Julien made my life interesting, if sometimes confusing. Randy Sepe helped me survive the final months of writing. He showed me that heavy metal and music theory are not mutually exclusive, and was always a source of challenging contradictions. I must also extend my thanks to my friends at the Moon, the lunatics who kept me sane: to Darren Sutphin for some great parties, and to Deborah Lovejoy for being the best dancer in town.

Finally, I am grateful to my sister Agnes for not finishing her doctoral work before I did mine. I would never have heard the end of that one!

Contents

1 Introduction **1**

1.1 Motivations and Requirements 2

1.2 Preview of the System 4

1.3 A Session with LOGICALC 7

1.4 Organization of the Thesis 39

2 Overview **41**

2.1 Introduction 42

2.2 Skolemization 46

2.2.1 Introduction 46

2.2.2 Predicate Calculus Databases 48

2.2.3 Skolemization of Axioms 50

2.2.4 Skolemization of Goals 52

2.3 Fundamental Concepts 53

2.3.1 Goals and Answers 53
2.3.2 Assumption Sets 53
2.3.3 Plans and Validations 54
2.3.4 Proofs 55
2.3.5 Goal Classes 56
2.4 Principles of Operation 58
2.4.1 Initiating a Session 58
2.4.2 Solving a Goal 59
2.4.3 Creating Plans 60
2.4.4 Initializing a Plan 65
2.4.5 Informing the User of the Results 67
2.4.6 Following an Answer 68
2.4.7 Concluding From a Plan 69
2.4.8 Creating a New Proof 70
2.5 Exceptions to Skolemization 71
2.5.1 Lambda Expressions 71
2.5.2 Skolemization of Lambda Expressions 72
2.6 Extensions to Unification 74
2.6.1 Quantified Formulae and Lambda Expressions 75

2.6.2 Segment Variables 75

2.7 Database Management 77

2.7.1 Indexation and Retrieval 77

2.7.2 Local Assumptions and Datapools 80

3 Logical System **85**

3.1 Introduction 85

3.2 Classical Formulation 87

3.2.1 The Language of $F^=$ 88

3.2.2 Substitutions 89

3.2.3 Axiomatic Structure of $F^=$ 90

3.2.4 Proofs 90

3.2.5 Major Results 91

3.3 LOGICALC's Axioms 92

3.4 LOGICALC's Inference Rules 93

3.5 Skolemizing Axioms 98

3.6 Inference Rule for Skolemization 99

3.7 Inference Rule for Quantification 100

3.8 Soundness and Completeness 101

3.9 Lambda Expressions 104

3.9.1 Comprehension Axioms 105

3.9.2 Rules for Reduction and Abstraction 106

3.9.3 Russell's Paradox 107

3.10 Unification and Substitutions 109

4 A Graph Editor for Theorem Proving 113

4.1 Introduction 113

4.2 Walk Mode 116

4.3 Display 118

4.4 Abbreviations 121

4.4.1 Definitions in NUPRL 121

4.4.2 Skolem Terms 122

4.4.3 Short and Long Forms 123

4.4.4 Input Forms 126

4.4.5 Generalized Abbreviations 128

4.4.6 Implementation 130

4.5 Commands 133

4.5.1 Command Line Structure 133

4.5.2 Macro Commands 135

4.5.3 Recalling An Earlier Command 138

4.5.4 Command Processing 139

4.6 Help 142

4.6.1 Implementation 145

5 Implementation **151**

5.1 Sequents and Assumption Sets 151

5.2 Goals and Goal Classes 154

5.2.1 Goals in Brief 154

5.2.2 Goal Classes 155

5.2.3 Goals Revisited 159

5.3 Plans 161

5.3.1 Validations 162

5.3.2 Successor Plans 164

5.3.3 Implementation 165

5.3.4 Initialization 166

5.3.5 Pushing a Plan 168

5.3.6 Producing a Plan's Conclusion 169

5.3.7 A Goal's Links 170

5.4 Proofs 171

5.4.1 Where Proofs Come From 171

5.4.2 Proof Generalization 176

5.4.3 Asserting Proofs in the Database 177

5.4.4 Implementation 178

5.5 Answers 181

6 Plan Generators **185**

6.1 Introduction 185

6.2 User Level Syntax and Notation 187

6.2.1 Premise Parameters 189

6.2.2 Subterm Parameters 194

6.3 Parsing The Command Line 198

6.4 Designing A Plan Description 200

6.4.1 Premise Description Language 200

6.4.2 Validation Hooks 204

6.4.3 Examples 207

6.5 Converting A Plan Description To A Plan Object 211

6.5.1 Automatic Expansion 212
6.5.2 Computing The Validation 213
6.5.3 Constructing The Plan Object 214
6.6 Available Plan Generators 215
6.6.1 Expand 215
6.6.2 Deduction Theorem 217
6.6.3 Modus Ponens 220
6.6.4 Modus Tollens 221
6.6.5 Resolution 222
6.6.6 Lemma 223
6.6.7 Substitution 224
6.6.8 Contradiction 224
6.6.9 Equality 225
6.6.10 Case 226
6.6.11 Reduction 230
6.6.12 Abstraction 231
6.6.13 Skolemize 234
6.6.14 Quantify 236

7 Validations and Detachments **239**

7.1 Introduction 239

7.2 Resolution and Natural Deduction 242

7.2.1 Non-Clausal Resolution 243

7.2.2 Translating Proofs to Natural Deduction 244

7.2.3 Conclusions for LOGICALC 245

7.3 Premises and Validations 245

7.3.1 The Point of Validations 246

7.3.2 Representation of Validations 248

7.3.3 Premises and Detachments 250

7.4 The Detaching Procedure 253

7.4.1 Detachment Records 254

7.4.2 The Detaching Algorithm 255

7.4.3 Converting a Detachment into a Premise 256

7.4.4 Converting a Detachment into a Proof 259

7.5 Detaching Modulo Equalities 261

7.5.1 E-Resolution 261

7.5.2 Congruence Closure 263

7.5.3 Extensions to Handle Mismatches 263

7.6 Conclusion 267

8 Automatic Generalization **271**

8.1 Introduction 271

8.2 Explanation-Based Generalization 273

8.3 Application to Proof Generalization 276

8.4 A Reified Sequent Logic For Horn Clauses 278

8.5 LOGICALC's Generalization Procedure 283

8.5.1 Reified Representation 283

8.6 The Regression Algorithm 287

8.6.1 Regressing A Proof 287

8.6.2 Punting 289

8.6.3 Regressing A Tautology 290

8.6.4 Where Automatic Generalization Fits 291

8.7 Regressors 292

8.8 Discussion Of Problems 295

8.8.1 Disappearing Terms 295

8.8.2 Built-In Theories 297

8.8.3 Regressors vs. Inference Rules 299

8.8.4 Regression vs. Inferencing 299

8.9 Generalization In Action 301

8.10 Comparison With ONTIC 303

9 Proof Editing And Formatting **305**

9.1 Introduction 305

9.2 The Proof Editor 308

9.2.1 The Notion of Users 310

9.2.2 The Notion of Visibility 310

9.2.3 Editing Operations 312

9.3 Construction of a Proof View 314

9.3.1 Data-Structures 314

9.3.2 Initialization 317

9.3.3 Computing a Line's Visibility 318

9.4 Generating a Summary 321

9.4.1 The Format of a Summary 322

9.4.2 Implementation 326

9.4.3 Example 330

9.5 Conclusion 333

10 An Axiomatization of Qualitative Physics **335**

10.1 Qualitative Simulation 336

10.2 Situations 341

10.3 Quantities 343

10.3.1 Landmarks 343

10.3.2 Qualitative States 345

10.3.3 Values of Quantities 345

10.4 Fluents and The Situation Calculus 347

10.5 Transitions 349

10.6 Qualitative State Changes 351

10.7 Constraints 356

10.8 Corresponding Values 357

10.9 Situated Arithmetic 359

10.10 Situated Inference Rules 360

10.10.1 Situated Negation 361

10.10.2 Timeless Truths 362

10.10.3 Situated Equality Substitution 364

10.10.4 Situated Symmetry 365

10.10.5 Generalization 366

10.11 The U-Tube Problem 367

10.11.1 Axiomatization 368

10.11.2 Proof Outline 369

10.11.3 Proof Summary 370

10.12 The Case of The Leaking Tank 376

10.12.1 Reachable Situations 376

10.12.2 Constants And Fluents 378

10.12.3 Derivatives 379

10.12.4 Arithmetic Revisited 380

10.12.5 The Problem Statement 381

10.12.6 Proof Outline 384

10.12.7 Proof Summary 384

10.12.8 Proof Commentary 391

10.13 Conclusion 394

11 Conclusion **395**

11.1 Summary Of The System 397

11.2 Related Work 405

11.3 Future Work 420

11.4 Conclusion 429

Bibliography **431**

A The Language **449**

A.1 Terms and Propositions 449

A.2 Read Macros 450

A.3 Syntax Macros 453

A.4 Types 453

A.4.1 Type Designators 454

A.5 Type Declarations 456

A.5.1 Global Type Declarations 456

A.5.2 Local Type Declarations 457

A.5.3 Controlling Type Constraints 457

A.5.4 Quantifying Over Types 459

A.5.5 Type Declarations And Lambda Expressions 459

A.6 Constraints In Binding Lists 460

A.7 Axioms And Rules 461

B Find Mode **465**

B.1 Generic Find Mode 465

B.2 Formula Find Mode 467

B.3 Subterm Find Mode 468

B.4 Target Find Mode 468

Chapter 1

Introduction

Mathematical proofs, as they are traditionally presented in textbooks, borrow a great deal of their elegance and apparent simplicity from the fact that they are neither completely formal nor do they spell out all the details, relying on the reader's intelligence to supply, reconstruct, or grasp the obviousness of what was left unspoken. In the past thirty years, computer-aided experimentation with formal systems has provided us with a better appreciation of just how much is usually left out.

Proving theorems formally involves a lot of bookkeeping, which can be a dauntingly tedious and error-prone task, if carried out by hand. Fortunately, it can be entirely taken over by the computer. However, the problem of discovering the proof remains and is not so easily dismissed; especially now that it is compounded with the necessity to spell out every detail. Consequently, considerable amounts of time and ingenuity have been invested in the development of techniques to automate and otherwise facilitate the search for and derivation of proofs.

A large spectrum of approaches have been attempted; from resolution to interactive proof editors. Since general techniques tend to perform poorly in the absence of problem- or domain-specific instructions to direct the search, there has been renewed

interest in interactive systems. Today, the availability of fast personal workstations makes such systems practical and attractive.

In this thesis, I propose a framework for interactive proof derivation based on graph editing and Natural Deduction. I also introduce several techniques which make such an approach practical rather than merely feasible. In particular, I wish to encourage an exploratory attitude in the user.

In contrast with the traditional stance which regards theorem proving as essentially an exercise in deduction—automated or otherwise—I believe that it is an activity with many facets, and that an interactive environment designed for it should reflect this plurality and provide adequate support in an integrated fashion. In particular, it should facilitate the process of initial formalization, of proof derivation, and also of proof editing and publication.

While conducting this research, I developed a rather large LISP program called LOGICALC, which implements the techniques presented in this thesis. Concurrently, I evaluated these ideas on a number of practical projects ranging from toy-sized problems, such as the derivation of a solution for a homework examined at the end of the present chapter, to far more ambitious undertakings in a formal theory of Kuipers-style qualitative physics described in Chapter 10.

1.1 Motivations and Requirements

The original motivation for the LOGICALC project was to develop a tool to assist our group in conducting research with, and proving theorems in, typical predicate calculus formalizations of AI problems; including, but not limited to, spatial reasoning, qualitative physics [Kui86], circumscription [e.g. Yale shooting problem], situation calculus and planning, etc...

Since the emphasis was on practical applications to formal AI research, LOGICALC was designed to facilitate precisely this type of work. In particular, we wanted to retain the simplicity afforded by the quantifier-free predicate calculus, and also the powerful convenience of unification. The reason for the former is that the quantifier-free predicate calculus is often exactly the right level of abstraction for many problems, and that it is a representation with which AI researchers are familiar. As for unification, it too is a familiar operation—perhaps the mechanism of choice in a significant fraction of AI programs—and is the expected mode of operation when dealing with the quantifier-free predicate calculus.

As a consequence, unlike NUPRL [C+86], this project has not attempted to be foundational. Rather, the intent was to arrive at a satisfactory compromise between adequate formality and manifest ease of use, helpfulness, and perspicuity.

A major requirement was that the system should lend itself not only to proof checking but to proof discovery as well. In fact, I have tried to encourage an exploratory style of proof whereby it is not necessary for the user to form a clear a-priori idea of how the proof should develop [although it is always an advantage to have such a guide], but will let him experiment, perform interactive searches, and allow the evolving context to suggest new or continued directions for the proof under construction.

The interface should make it easy to bring the user's knowledge to bear on the task at hand. For instance, when the user knows that a particular theorem can be proven using axiom $\mathcal{A}$, he should be able to say so simply and let the program figure out, and take care of, the inferential details. This thesis proposes a "detaching" mechanism which precisely fills this office.

Research in theorem proving has often taken the narrow view that proof derivation, in particular the automation thereof, is the only important problem in the field. Such a view is about as misguided as considering optimizing compilers to be the only important problem in the field of programming. The reality is that, when doing re-

search with logical formalisms, perhaps more so in AI, a lot of time must be spent on developing the axiomatization and getting it right. Attempting to prove theorems about and within the domain under consideration is an integral part of the process of formalization. It is not uncommon, while carrying out a proof, to discover that additional axioms are needed, or that an alternative representation would make the process easier. Thus the axiomatization as well as the ontology may be revised as a result of exploring the proof space. Therefore the system should allow rapid prototyping and, in so far as it is possible, should accommodate changes to the axiomatization on the fly.

The argument presented in the preceding paragraph is further evidence that interactive exploration of the proof space is an essential requirement, and also that proofs must be available for inspection. Which brings me to my last point: the researcher's task typically does not end when a proof has been derived; a representation must often be produced, e.g. for reference or inclusion in a paper. A unified environment for theorem proving ought to address this need too, and provide facilities to edit a proof and assist in its publication.

1.2 Preview of the System

LOGICALC offers a goal/plan based approach to incremental proof refinement. Each goal may have more than one plan attached to it. Thus the user can keep track of several alternative proof attempts at the same time.

A goal is a *sequent* of the form $A \Rightarrow p$, where p is the conclusion to be derived and A is the assumption set. A proof of this goal not only includes the corresponding instance of its sequent, but also captures the complete proof tree which validates the conclusion.

Assumptions and goals are skolemized, by default, thereby making quantifier-manipulation rules unnecessary and allowing unification to be used as the fundamental operation. Thus, the simplicity of quantifier-free predicate calculus may be retained, and the user will manipulate a familiar representation. However, it is possible to control the skolemizing behavior, and explicit quantification and skolemization rules are also available.

The primary mode of interaction with LOGICALC is through plan generators. A plan generator produces a set of plans on the basis of the current goal, as well as premises and parameters specified by the user. These plans will be filed under the current goal and represent admissible refinements, or alternative proof attempts. The idea of a plan generator is related to that of a tactic in LCF.

Whereas plan generators serve to refine the conclusion part of a goal sequent, inference generators operate on the assumption set and may augment it with new conclusions derived by forward inferencing rather than backward refinement.

The supporting framework is that of a graph of goals, plans, and proofs which is maintained and managed by the system. It is this structure which permits LOGICALC to address certain issues of subsumption in interesting ways: as we shall see, the system recognizes duplicate goals (modulo alphabetic renaming) and will establish an *equivalence class* to centralize bookkeeping for all of them so that they may share their answers. Also, links are maintained between similar equivalence classes; where 'similar' is taken to mean more, or less, specific, and specificity is determined by relative instantiation and difference in the number of assumptions. Whenever an answer is found, not only is it recorded with the particular class it pertains to, but it is also broadcast along its links, and thus propagates to similar classes which can then decide whether the new answer can be adapted to benefit them too.

Naturally, interaction with this structural framework is implemented through a graph editor. The editor is specified in an object-oriented fashion: each type of node in the

graph provides its own display procedure and its own command processor. From the point of view of the user, the interface is very similar to a line oriented shell such as is commonly found in the UNIX world, e.g. CSH. The major distinction is that the set of available commands varies from one type of node to another.

The user is expected to begin by loading in an axiomatization of the domain of interest, then state a theorem to be proven, at which point the actual session begins. The user is presented with a goal representing the target theorem, and should invoke a plan generator to refine this goal into a plan with subgoals, and proceed in the same manner with the plan's steps until all leaf goals have been matched to axioms or assumptions.

As I mentioned earlier, plan generators often expect premises and parameters to further specify the refinement to be performed, e.g. Modus Ponens expects a major premise in the form of an implication. Typically a premise is the name of an axiom or an assumption. However, LOGICALC generalizes this notion and provides a uniform and powerful extension through the "detaching" mechanism. Thus premises can be hypothetical, or they can be subformulae of assertions extracted by a procedure reminiscent of Unit Non-Clausal E-Resolution; in the latter case, additional subgoals may be required. Interactive browsing and selection of candidate detachments can be requested from the command line and will trigger activation of "Find Mode."

Since propositions are typically converted to their skolemized version, skolem terms occur frequently. They are well-known for possessing rather infelicitous representations. For the convenience of the user, an abbreviation scheme has been designed which considerably clarifies and simplifies both input and output. In this regime, a skolem term with function `foo` will ordinarily print as `!.foo`. This scheme was extended to allow arbitrary abbreviations of terms, as well as of interactive commands.

Since a proof contains a full trace of the deductive process that resulted in its derivation, it was possible to implement a mechanism for automatic generalization of con-

clusions in a manner that is related to Explanation-Based Generalization, but carries it further. This feature is particularly useful to counteract the over-specializing effects due to wanton skolemization.

The availability of proofs as self-contained objects also made it possible to produce textual, Natural Deduction-like representations for them. A specialized proof editor was implemented to facilitate the process of proof editing and publication. TEX output can automatically be prepared in this manner for inclusion in a document to be typeset.

LOGICALC was evaluated on various formal AI problems. Chapter 10 presents the result of my research on a first-order axiomatization of a Kuipers-style approach to qualitative physics. In the next section, I include an annoted transcript of a session with LOGICALC which illustrates, on a realistic example, how a user typically interacts with this program, and hopefully will give the reader a feel for the system. The chapter concludes with an outline of the thesis.

1.3 A Session with LOGICALC

This section contains an annotated session with LOGICALC, and serves as a gentle introduction to the system as well as an illustration of the more basic ways of interacting with it. The proof was derived as a solution for a class assignment.

LOGICALC has the ability to output text with visual enhancements for various kinds of devices, including TEX.[1] By merely setting TERM* to `tex`, I was able to use my transcript to produce slides, a hand-out, as well as the annotated version which follows.

The text of the assignment is included here for reference:

[1]In so far as TEX can be construed as a device.

---

homework

1. Express the following facts in predicate calculus, using a concise, clear vocabulary of predicates:
   - If x is a part of y, and y is a part of z, then x is a part of z.
   - If a creature has a thumb on its left hand, it has a thumb on its right hand, and vice versa.
   - Every hand is either left or right, but not both.
   - A thumb belongs to exactly one hand.
2. We wish to prove the following two statements:
   - A creature with just one hand has no thumbs.
   - If a creature has any thumbs, it has two distinct thumbs.

   What fact must be added to the list above for these two statements to be provable?
3. Pick one of the two, and give a formal proof, using either the tableau method, natural deduction, or resolution.

Below I include the axiomatization actually used in the solution. It is a straightforward restatement of the terms of the problem in a notational variant of first-order logic. Note that **creature-thumb** is an additional fact added as an answer to question 2: it states that if a creature has a thumb, then it must have a hand to which the thumb is attached (think of it as a weak statement to the effect that all parts of a creature must be connected).

---

axiomatization

```
(IN-PACKAGE 'NISP)
(DEPENDS-ON NISP)

(DEFDUCKTYPE creature obj)
(DEFDUCKTYPE hand     obj)
(DEFDUCKTYPE thumb    obj)
```

```
(DEFPRED (PART ?X ?Y - obj))
(DEFPRED (LEFT  ?H - hand))
(DEFPRED (RIGHT ?H - hand))

(AXIOM TRANSITIVITY
  (FORALL (X Y Z - part)
    (IF (AND (PART X Y) (PART Y Z)) (PART X Z))))

(AXIOM THUMBS
  (FORALL (C - creature ONE-HAND - hand ONE-THUMB - thumb)
    (IF (AND (PART ONE-THUMB ONE-HAND) (PART ONE-HAND C))
        (EXISTS (OTHER-HAND - hand OTHER-THUMB - thumb)
          (AND (PART OTHER-THUMB OTHER-HAND)
               (PART OTHER-HAND C)
               (IF (LEFT  ONE-HAND) (RIGHT OTHER-HAND))
               (IF (RIGHT ONE-HAND) (LEFT  OTHER-HAND)))))))

(AXIOM LEFT-OR-RIGHT
  (FORALL (H - hand)
    (AND (OR (LEFT H) (RIGHT H))
         (NOT (AND (LEFT H) (RIGHT H))))))

(AXIOM ONE-HAND-PER-THUMB
  (FORALL (T - thumb)
    (EXISTS (HAND-OF-THUMB - hand)
      (AND (PART T HAND-OF-THUMB)
           (FORALL (H - hand)
             (IF (PART T H) (= HAND-OF-THUMB H)))))))

(AXIOM CREATURE-THUMB
  (FORALL (C - creature T - thumb)
    (IF (PART T C)
        (EXISTS (ONE-HAND - hand)
          (AND (PART T ONE-HAND) (PART ONE-HAND C))))))
```

Question 3 of the problem set offered a choice of 2 propositions for which a formal proof should be attempted. I selected the second one:

If a creature has any thumbs, it has two distinct thumbs.

The corresponding proposition, expressed in first-order logic, is:

```
(FORALL (C - creature ONE-THUMB - thumb)
  (IF (PART ONE-THUMB C)
    (EXISTS (OTHER-THUMB - thumb)
      (AND (PART OTHER-THUMB C) (NOT (= ONE-THUMB OTHER-THUMB))))))
```

I will presently outline a sketch of the proof so that the reader may know what to expect and better follow the forthcoming account of this interaction as it proceeds.

1. First, we shall begin by assuming that creature C has ONE-THUMB; thus reflecting the presupposition *"if a creature has any thumbs..."* Our objective will be to find (or show the existence of) an OTHER-THUMB such that it is a part of creature C and is distinct from ONE-THUMB.

2. By axiom CREATURE-THUMB, we know that there is a hand connecting creature C to its thumb ONE-THUMB.

3. By axiom THUMBS, we know that, if there is a hand connecting a creature to a thumb on one side, then there there is a hand connecting the creature to a thumb on the other side.

4. Since it is not specified on which side the hand connecting creature C to ONE-THUMB is attached, we must perform a case analysis to conclude that there is a hand on the opposite side connecting creature C to another thumb.

5. Finally, we show that, if both thumbs were equal, then both hands would be equal too. However, since these hands are on opposite sides, they must be distinct. Therefore, by contradiction, the thumbs must be distinct.

Let's assume that the axiomatization presented earlier has been loaded into the LISP image. To initiate the session, we now invoke LOGICALC and state the theorem to be proven. The system responds by presenting us with a view of the top level goal.

```
> (LOGICALC)                                                      transcript
Theorem to be proven? (FORALL (C - creature ONE-THUMB - thumb)
                        (IF (PART ONE-THUMB C)
                           (EXISTS (OTHER-THUMB - thumb)
                              (AND (PART OTHER-THUMB C)
                                   (NOT (= ONE-THUMB OTHER-THUMB))))))
Return variables: ?OTHER-THUMB
Which ones are you interested in? -ALL
Goal View      <class CLASS.1>      (top goal)
   Find: (OTHER-THUMB) in:
      (IF (AND (IS CREATURE !.C[6])
               (IS THUMB !.ONE-THUMB[7])
               (PART !.ONE-THUMB[7] !.C[6]))
          (AND (IS THUMB ?OTHER-THUMB)
               (PART ?OTHER-THUMB !.C[6])
               (NOT (= !.ONE-THUMB[7] ?OTHER-THUMB))))
   (no assumptions)
   (no answers)
   (no local plans)
   (no class plans)
```

We enter the target theorem as a fully quantified formula. Note that each variable is followed by a type declaration, with the character '-' acting as a separator. Thus C was declared to be a creature and ONE-THUMB a thumb (see DEFDUCKTYPE specifications in axiomatization). After reading in this formula, the system applied skolemization to it and packaged the resulting expression as a goal. A view of this goal is displayed to the user.

The prefix '!.' is a notational convention to indicate a skolem term. !.C is a new skolem term which stands for an arbitrary and anonymous creature. The display actually shows !.C[6] where 6 is the skolem term's "id." Every time the system invents a new skolem term, it makes up a new id for it using a monotonically increasing global counter; this is why !.ONE-THUMB has id 7. You may read !.C[6] as C_6. The id is always printed the first time a skolem term is seen; however it is generally omitted on subsequent occasions, as long as the skolem term's name by itself (e.g. C) is unambiguous.

The prefix '?' is the traditional notational convention to indicate a free variable; e.g.

?OTHER-THUMB is a free variable to be subsequently instantiated in our search for a proof—we are looking for a term to serve as a value for ?OTHER-THUMB and for which the corresponding instance of the goal obtains.

Note that all type declarations have been transformed into propositions of the form (IS ⟨*type*⟩ ⟨*object*⟩) and appropriately inserted in the formula.

The fact that the goal is in the form of an implication suggests that we proceed by invoking the Deduction Theorem: let us assume the implication's antecedent and attempt to prove its consequent under the resulting additional assumptions.

```
[0] w> DEDUCTION                                                     transcript

There is 1 fate
1  (PLAN):
      PLAN --- (IF-INTRO)
      1  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          => (IS THUMB ?OTHER-THUMB))
      2  (|...| => (PART ?OTHER-THUMB !.C))
      3  (|...| => (NOT (= !.ONE-THUMB ?OTHER-THUMB)))
Make it local? YES
Move to it? YES

Plan View      <class CLASS.1>
   Documentation: "w> +PLAN DEDUCTION-THEOREM"
   Supergoal:
      (IF (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
          (AND (IS THUMB ?OTHER-THUMB)
               (PART ?OTHER-THUMB !.C)
               (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
   Will find:
      OTHER-THUMB = ?OTHER-THUMB
   Steps:
      1  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          => (IS THUMB ?OTHER-THUMB)) -- (no local plans, no class plans, 1 answer)
      2  (|...| => (PART ?OTHER-THUMB !.C)) -- (no local plans, no class plans, 1 answer)
      3  (|...| => (NOT (= !.ONE-THUMB ?OTHER-THUMB))) -- (no local plans, no class plans, no answe
   (no successors)
```

The `deduction` command is interpreted as a request to refine the current goal according to the Deduction Theorem. In response, a plan is constructed and proposed to the user. From proofs of the steps, we expect a proof of the original goal to be derivable according to the plan's validation (not shown here)—a validation is a description of a sequence of inferences expressed in terms of available inference rules.

In the resulting plan, the system automatically expanded the consequent into 3 steps, one for each conjunct. Each step is displayed in a sequent-like format:

$$\text{Assumptions} \Rightarrow \text{Goal}$$

The elision `|...|` means to repeat the preceding set of assumptions. Thus the system avoids cluttering the display with redundant repetitions.

Note that steps 1 and 2 each have one answer already associated with them. The reason is that, whenever a new goal is created, the system attempts to find easy answers to it; in particular, it checks whether it is a tautology—which neither of them are, here—and whether it unifies with some axiom or assumption—which they both do: step 1 unifies with the second assumption, and step 2 with the third one.

Unfortunately, neither of these answers is satisfactory since they both involve `!.ONE-THUMB` when we really meant to find *another* thumb. Therefore, we shall have to disregard them and proceed to look for other answers. At this point, we must select a step to work on. Since it is unlikely that we can proceed easily with either step 1 or step 3 without first instantiating `?OTHER-THUMB`, it is probably a better idea to try our luck with step 2. We simply type 2 to LOGICALC's shell, and the view is moved to the node representing step 2.

---

transcript

```
[1] w> 2
Goal View      <class CLASS.3>
   Find: (OTHER-THUMB) in:
      (PART ?OTHER-THUMB !.C)
   Assumptions:
```

```
    (IS CREATURE !.C) -- !:IS278
    (IS THUMB !.ONE-THUMB) -- !:IS279
    (PART !.ONE-THUMB !.C) -- !:PART280
 Answers:
    1  OTHER-THUMB = !.ONE-THUMB
 Supergoal:
    (IF (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
        (AND (IS THUMB ?OTHER-THUMB)
             (PART ?OTHER-THUMB !.C)
             (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
 (no local plans)
 (no class plans)
```

We want to find the creature's other thumb, i.e. the thumb on the side opposite `!.ONE-THUMB`. Clearly, we must use axiom `THUMBS`; however, `THUMBS` does not mention a proposition of the form *"thumb is part of creature"* but it does mention the two simpler propositions *"thumb is part of hand"* and *"hand is part of creature."* Therefore, before we can use axiom `THUMBS`, we must invoke the axiom of transitivity of parts in order to decompose the goal into these simpler components.

---

[2] w> USE TRANSITIVITY — transcript

```
There is 1 fate
1  (PLAN):
      PLAN --- (MODUS-PONENS)
      1  (PART ?OTHER-THUMB ?Y.1)
      2  (PART ?Y.1 !.C)
Make it local? YES
Move to it? YES

Plan View      <class CLASS.3>
   Documentation: "w> +PLAN RESOLUTION TRANSITIVITY"
   Supergoal:
      (PART ?OTHER-THUMB !.C)
   Will find:
      OTHER-THUMB = ?OTHER-THUMB
   Steps:
      1  (PART ?OTHER-THUMB ?Y.1) -- (no local plans, no class plans, 1 answer)
      2  (PART ?Y.1 !.C) -- (no local plans, 1 class plan, 1 answer)
   (no successors)
```

We have reduced the problem of finding a thumb that is a part of creature `!.C` to that of finding a hand `?Y.1` that is part of `!.C` and such that the thumb is part of it.

We can now move to either step and attempt to proceed using axiom THUMBS as we originally intended. Since step 2 is the more instantiated, it will presumably result in greater directionality and specificity.

---

transcript

```
[3] w> 2
Goal View      <class CLASS.3>
   Find: (Y.1) in:
      (PART ?Y.1 !.C)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   Answers:
      1  Y.1        = !.ONE-THUMB
   Supergoal:
      (PART ?OTHER-THUMB !.C)
   (no local plans)
   (1 class plan)

[4] w> USE THUMBS

There is 1 fate
1  (PLAN):
      PLAN --- (TAUT-TRANS)
      1  (IS HAND ?ONE-HAND.1)
      2  (IS THUMB ?ONE-THUMB.1)
      3  (PART ?ONE-THUMB.1 ?ONE-HAND.1)
      4  (PART ?ONE-HAND.1 !.C)
Make it local? YES
Move to it? YES

Plan View      <class CLASS.3>
   Documentation: "w> +PLAN RESOLUTION THUMBS"
   Supergoal:
      (PART ?Y.1 !.C)
   Will find:
      Y.1        = !.OTHER-HAND[2](?ONE-HAND.1 !.C)
   Steps:
   |-[1  (IS CREATURE !.C)]
      2  (IS HAND ?ONE-HAND.1) -- (no local plans, no class plans, no answers)
      3  (IS THUMB ?ONE-THUMB.1) -- (no local plans, no class plans, 1 answer)
      4  (PART ?ONE-THUMB.1 ?ONE-HAND.1) -- (no local plans, no class plans, 1 answer)
      5  (PART ?ONE-HAND.1 !.C) -- (no local plans, 2 class plans, 1 answer)
   (no successors)
```

In this view, ?Y.1 is the "other" hand, ?ONE-THUMB.1 stands for the given thumb

(i.e. !.ONE-THUMB), and ?ONE-HAND.1 is the "first" hand that connects ?ONE-THUMB.1 (which we will promptly instantiate with !.ONE-THUMB) to creature !.C. Step 3 already has an answer which presumably results from unification with the assumption !:IS279 that !.ONE-THUMB is a thumb. We ought to use that answer, thereby further instantiating this plan in the desired direction.

transcript

```
[5] w> 3
Goal View       <class CLASS.2>
   Find: (ONE-THUMB.1) in:
      (IS THUMB ?ONE-THUMB.1)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   Answers:
      1  ONE-THUMB.1 = !.ONE-THUMB
   Supergoal:
      (PART ?Y.1 !.C)
   (no local plans)
   (no class plans)

[6] w> FOLLOW
```

When an answer to a goal is "followed," a "successor" is created to the plan where the goal appears as a step. The successor plan looks just like the original plan, except that the goal whose answer was followed is marked as "solved" (indicated by brackets in the display), and the answer's substitution was applied to all the remaining steps.

transcript

```
Plan View       <class CLASS.3>
   Documentation: ">w> +PLAN RESOLUTION THUMBS"
   Supergoal:
      (PART ?Y.1 !.C)
   Will find:
      Y.1           = !.OTHER-HAND(?ONE-HAND.2)
   Steps:
   |-[1  (IS CREATURE !.C)]
      2  (IS HAND ?ONE-HAND.2) -- (no local plans, no class plans, no answers)
     [3  (IS THUMB ?ONE-THUMB)]
      4  (PART !.ONE-THUMB ?ONE-HAND.2) -- (no local plans, no class plans, 1 answer)
      5  (PART ?ONE-HAND.2 !.C) -- (no local plans, 3 class plans, 1 answer)
   (no successors)
```

Note that the variable denoting the "first" hand was renamed from ?ONE-HAND.1 to ?ONE-HAND.2. Our objective will be to derive an instantiation for this variable. Axiom CREATURE-THUMB states that, if a creature has a thumb, then there exists a hand that connects the creature to its thumb. This last observation suggest that we move to step 5 and tentatively apply the aforementioned axiom.

transcript

```
[7] w> 5
Goal View      <class CLASS.3>
   Find: (ONE-HAND.2) in:
      (PART ?ONE-HAND.2 !.C)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   Answers:
      1  ONE-HAND.2 = !.ONE-THUMB
   Supergoal:
      (PART ?Y.1 !.C)
   (no local plans)
   (3 class plans)

[8] w> USE CREATURE-THUMB

There are 2 fates
1  (PLAN):
      PLAN --- (TAUT-TRANS)
      1  (IS THUMB ?T.1)
      2  (PART ?T.1 !.C)
Make it local? NO
2  (CONCLUSION):
      ONE-HAND.2 = !.ONE-HAND[5](!.C !.ONE-THUMB)
Move to one of them? Type no.: 2
```

As expected, using axiom CREATURE-THUMB resulted in the creation of a two-step plan expressing the need to find a thumb ?T.1 that is part of creature !.C. Both steps have answers found by unification with an assumption, and both answers have compatible (in fact, identical) substitutions: They both assign !.ONE-THUMB to ?T.1. The system notices the answers can be merged, thereby producing a substitution which satisfies all the plan's steps, and proceeds to derive the corresponding conclusion which it then displays as fate 2.

In the proposed conclusion, the "first" hand is denoted by skolem term !.ONE-HAND[5] (!.C !.ONE-THUMB) which is a function of both the creature and the "first" thumb as reflected by the list of arguments following !.ONE-HAND. We select fate 2 by typing 2 to the prompt, thus following the conclusion, and are presented with a new successor plan.

```
Plan View      <class CLASS.3>                                    transcript
   Documentation: ">>w> +PLAN RESOLUTION THUMBS"
   Supergoal:
      (PART ?Y.1 !.C)
   Will find:
      Y.1          = !.OTHER-HAND[2](!.ONE-HAND !.C)
   Steps:
   |-[1  (IS CREATURE !.C)]
      2  (IS HAND !.ONE-HAND) -- (no local plans, no class plans, no answers)
     [3  (IS THUMB ?ONE-THUMB)]
      4  (PART !.ONE-THUMB !.ONE-HAND) -- (no local plans, no class plans, no answers)
     [5  (PART ?ONE-HAND !.C)]
   (no successors)
```

The brackets indicate those steps which have been solved. Also, the arguments to !.ONE-HAND, which were displayed earlier when !.ONE-HAND was first introduced, have been omitted here along with the "id" since the name by itself is unambiguous.

All that remains to be done in this plan is to show that !.ONE-HAND is a hand [step 2] and that !.ONE-THUMB is a part of it [step 4]. Remember that !.ONE-HAND was introduced by our use of axiom CREATURE-THUMB. Since, the latter "defines" the former, it is most likely that other properties of !.ONE-HAND will also be found there, e.g. its type, because they must have been expressed originally within the scope of the quantifier that introduced !.ONE-HAND. In fact, a cursory glance at the text of the axiom confirms that the properties expressed by both remaining steps are explicitly asserted. Therefore, we should be able to use the axiom again to quickly dispatch these two steps.

```
[9] w> 2                                                          transcript
```

```
Goal View     <class CLASS.9>
   Prove
      (IS HAND !.ONE-HAND)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   (no answers)
   Supergoal:
      (PART ?Y.1 !.C)
   (no local plans)
   (no class plans)

[10] w> USE -SKOLEM
```

The idiom "`use -skolem`" means: find all assertions mentioning any of the skolem terms that appear in the goal, and attempt to solve the goal using them. The great advantage of this idiom is that the user does not have to remember or find out where the skolem terms originated; the system will make this determination for him. I have found this feature to be extremely useful in practice, often solving the goal in a seemingly magical way.

---

transcript

```
There is 1 fate
1  (CONCLUSION): Yes
Move to it? YES

Goal View     <class CLASS.10>
   Prove
      (PART !.ONE-THUMB !.ONE-HAND)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   (no answers)
   Supergoal:
      (PART ?Y.1 !.C)
   (no local plans)
   (no class plans)
```

We just solved step 2, and only step 4 remains. Therefore, the system anticipates the user, and moves the focus directly to the sole remaining step instead of to the

successor plan where step 4 is the only unsolved step. Once again, we invoke the convenient idiom introduced above.

```
[11] w> USE -SKOLEM                                          transcript

There is 1 fate
1 (CONCLUSION): Yes
Move to it? YES

Goal View      <class CLASS.13>
   Find: (OTHER-THUMB) in:
      (PART ?OTHER-THUMB !.OTHER-HAND)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   (no answers)
   Supergoal:
      (PART ?OTHER-THUMB !.C)
   (no local plans)
   (no class plans)
```

As expected, an answer is immediately found, which we proceed to follow. Since step 4 was the last remaining step in the plan originally created on page 15, the plan's validation is executed to produce a conclusion. A validation is a description of how to derive a plan's conclusion from proofs of its steps, using inference rules.

This new conclusion is an answer to the plan's supergoal (see page 15). This supergoal appeared as step 2 in the plan displayed on page 14. Therefore, following the conclusion creates a successor plan with a single remaining step (where ?Y.1 is instantiated with !.OTHER-HAND), and the system moves us directly to that step. It is a view of that step which is displayed above. !.OTHER-HAND is the "other" hand and was introduced by our use of axiom THUMBS on page 15.

```
[12] w> USE THUMBS[2]                                        transcript

There are 3 fates
```

[2] I could have typed "USE -SKOLEM"

```
1  (PLAN):
      PLAN --- (TAUT-TRANS)
      1  (IS CREATURE !.OTHER-HAND)
      2  (IS HAND ?ONE-HAND.3)
      3  (IS THUMB ?ONE-THUMB.2)
      4  (PART ?ONE-THUMB.2 ?ONE-HAND.3)
      5  (PART ?ONE-HAND.3 !.OTHER-HAND)
Make it local? YES[3]
2  (PLAN):
      PLAN --- (TAUT-TRANS)
      1  (IS THUMB ?ONE-THUMB.3)
      2  (PART ?ONE-THUMB.3 !.ONE-HAND)
Make it local? YES
3  (CONCLUSION):
      OTHER-THUMB = !.OTHER-THUMB[3](!.ONE-HAND !.C)
Move to one of them? Type no.: 3
```

```
Plan View     <class CLASS.1>
   Documentation: ">w> +PLAN DEDUCTION-THEOREM"
   Supergoal:
      (IF (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
          (AND (IS THUMB ?OTHER-THUMB)
               (PART ?OTHER-THUMB !.C)
               (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
   Will find:
      OTHER-THUMB = !.OTHER-THUMB
   Steps:
      1  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          => (IS THUMB !.OTHER-THUMB)) -- (no local plans, no class plans, no answers)
     [2  (|...| => (PART ?OTHER-THUMB !.C))]
      3  (|...| => (NOT (= !.ONE-THUMB !.OTHER-THUMB))) -- (no local plans, no class plans,
   (no successors)
```

An answer was found and we decided to follow it by typing the corresponding fate number (i.e. 3) to the prompt. No unsolved step remains in the superplan and its validation is executed, thereby producing an answer to step 2 of the original plan (see page 12). Following this answer creates a successor plan and moves the focus to it. It is a view of this successor which we see above.

Now that ?OTHER-THUMB has been instantiated with !.OTHER-THUMB, we can expedite

[3] I answered YES because at this point it is still unclear that one of the 3 fates is in fact the desired conclusion.

step 1 with another "use -skolem."

transcript

```
[13] w> 1
Goal View      <class CLASS.18>
   Prove
      (IS THUMB !.OTHER-THUMB)
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   (no answers)
   Supergoal:
      (IF (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
          (AND (IS THUMB ?OTHER-THUMB)
               (PART ?OTHER-THUMB !.C)
               (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
   (no local plans)
   (no class plans)

[14] w> USE -SKOLEM

There is 1 fate
1  (CONCLUSION): Yes
Move to it? YES

Goal View      <class CLASS.17>
   Prove
      (NOT (= !.ONE-THUMB !.OTHER-THUMB))
   Assumptions:
      (IS CREATURE !.C) -- !:IS278
      (IS THUMB !.ONE-THUMB) -- !:IS279
      (PART !.ONE-THUMB !.C) -- !:PART280
   (no answers)
   Supergoal:
      (IF (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
          (AND (IS THUMB ?OTHER-THUMB)
               (PART ?OTHER-THUMB !.C)
               (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
   (no local plans)
   (no class plans)
```

This is a view of step 3 which is the only remaining subgoal in this proof. The fact that the goal has the form of an inequality suggests that we attempt a proof by contradiction: assuming that the two thumbs !.ONE-THUMB and !.OTHER-THUMB are identical, we will prove that the two hands must be identical as well (by axiom

ONE-HAND-PER-THUMB) and consequently must be on the same side; but we know that they are on oppposite sides by construction (cf. axiom THUMBS). Hence a contradiction.

---

[15] w> CONTRADICTION (LEFT !.ONE-HAND) *transcript*

```
There is 1 fate
1  (PLAN):
      PLAN --- (NOT-INTRO)
      1  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          (= !.ONE-THUMB !.OTHER-THUMB)
          => (LEFT !.ONE-HAND))
      2  (|...| => (NOT (LEFT !.ONE-HAND)))
Make it local? YES
Move to it? YES
```

Plan View <class CLASS.17>

```
   Documentation: "w> +PLAN CONTRADICTION (LEFT !.ONE-HAND)"
   Supergoal:
      (NOT (= !.ONE-THUMB !.OTHER-THUMB))
   Steps:
      1  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          (= !.ONE-THUMB !.OTHER-THUMB)
          => (LEFT !.ONE-HAND)) -- (no local plans, no class plans, no answers)
      2  (|...| => (NOT (LEFT !.ONE-HAND))) -- (no local plans, no class plans, no answers)
   (no successors)
```

The plan is to show that, assuming that !.ONE-THUMB and !.OTHER-THUMB (the two thumbs) are identical, then it follows that !.ONE-HAND (the first hand) both *is* and *is not* on the left side.

---

[16] w> 1 *transcript*

Goal View <class CLASS.19>

```
   Prove
      (LEFT !.ONE-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
```

```
    (NOT (= !.ONE-THUMB !.OTHER-THUMB))
  (no local plans)
  (no class plans)
```

First, we are going to prove as a lemma that the first hand is identical to the other hand (under the assumption that the first thumb is identical to the other thumb). Proving this lemma explicitly ahead of time is not strictly necessary since it would naturally occur as a subgoal when required, but it simplifies the exposition and makes the proof easier to follow.[4]

transcript

```
[17] w> LEMMA (= !.ONE-HAND !.OTHER-HAND)

There is 1 fate
1  (PLAN):
      PLAN --- (TAUT-TRANS)
      1  (= !.ONE-HAND !.OTHER-HAND)
      2  (LEFT !.ONE-HAND)
Make it local? YES
Move to it? YES

Plan View      <class CLASS.19>
  Documentation: "w> +PLAN LEMMA (= !.ONE-HAND !.OTHER-HAND)"
  Supergoal:
     (LEFT !.ONE-HAND)
  Steps:
     1  (= !.ONE-HAND !.OTHER-HAND) -- (no local plans, no class plans, no answers)
     2  (LEFT !.ONE-HAND) -- (no local plans, 1 class plan, no answers)
  (no successors)

[18] w> 1
Goal View      <class CLASS.21>
  Prove
     (= !.ONE-HAND !.OTHER-HAND)
  Assumptions:
     (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
  (no answers)
  Supergoal:
     (LEFT !.ONE-HAND)
  (no local plans)
  (no class plans)
```

[4] besides, it simply occurred to me at this point in the proof that it might be a good idea to have this lemma around.

Obviously, in order to prove this proposition, we shall need axiom ONE-HAND-PER-THUMB; not only is it the only one concerned with unicity, but it is also the only one that has an equality sign in it. I could choose to illustrate the use of the equality command and type something like:

```
equality !.one-hand - one-hand-per-thumb
```

This would "detach" an equality formula of the form (= !.ONE-HAND ⟨*term*⟩) from axiom ONE-HAND-PER-THUMB and would use it to substitute occurrences of term !.ONE-HAND with ⟨*term*⟩. Instead, I would rather illustrate another useful feature of LOGICALC's interface, which is the ability to perform approximate unification (near matches). For those terms that do not unify where they should, explicit subgoals are inserted to prove them equal.

Thanks to this feature, I will be able to continue invoking the simpler 'use' command and let the system figure out exactly what equality substitutions are required. I will also explicitly limit the number of allowed mismatches to no more than 1.

What I expect to happen is this: the system will try to "detach" the current goal's equality from axiom ONE-HAND-PER-THUMB. However, this cannot succeed with regular unification because the equality formula occurring in said axiom has one operand which is a free variable (which would be fine) and the other which is a skolem that unifies with neither the first nor the other hand. Therefore, we ought to permit 1 mismatch. The rest of the proof will involve proving that both hands are equal to this new term.

---

[19] w> USE -MISMATCH[5] 1 ONE-HAND-PER-THUMB transcript

```
There are 2 fates
1  (PLAN):
```

[5]-mismatch 1 can be abbreviated -m 1 or just +m.

```
      PLAN --- (EQUALITY)
      1  (= !.ONE-HAND !.HAND-OF-THUMB[4](?T.2))
      2  (IS THUMB ?T.2)
      3  (IS HAND !.OTHER-HAND)
      4  (PART ?T.2 !.OTHER-HAND)
Make it local? YES
2  (PLAN):
      PLAN --- (EQUALITY)
      1  (= !.OTHER-HAND !.HAND-OF-THUMB[4](?T.3))
      2  (IS THUMB ?T.3)
      3  (PART ?T.3 !.ONE-HAND)
Make it local? YES
Move to one of them? Type no.: 2

Plan View      <class CLASS.21>
   Documentation: "w> +PLAN RESOLUTION -MISMATCH 1 ONE-HAND-PER-THUMB"
   Supergoal:
      (= !.ONE-HAND !.OTHER-HAND)
   Steps:
      1  (= !.OTHER-HAND !.HAND-OF-THUMB(?T.3)) -- (no local plans, no class plans, no answers)
      2  (IS THUMB ?T.3) -- (no local plans, no class plans, 2 answers)
   |-[3  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          => (IS HAND !.ONE-HAND))]
      4  (PART ?T.3 !.ONE-HAND) -- (no local plans, no class plans, 1 answer)
   (no successors)
```

Obviously, step 4 already has the answer that the first thumb is a part of the first hand. It is probably a good idea to follow this answer, thereby further instantiating our plan.

```
[20] w> 4                                                            transcript

Goal View      <class CLASS.28>
   Find: (T.3) in:
      (PART ?T.3 !.ONE-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   Answers:
      1  T.3          = !.ONE-THUMB
   Supergoal:
      (= !.ONE-HAND !.OTHER-HAND)
   (no local plans)
   (no class plans)
```

```
[21] w> FOLLOW

Goal View     <class CLASS.30>
   Prove
      (= !.OTHER-HAND !.HAND-OF-THUMB[4](!.ONE-THUMB))
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (= !.ONE-HAND !.OTHER-HAND)
   (no local plans)
   (no class plans)
```

Following the answer instantiated ?T.3 with !.ONE-THUMB. Step 2 became the goal to prove that !.ONE-THUMB is a thumb, which was already known to be true. Therefore, the successor plan's only remaining unsolved step was step 1. The system moved us directly to it.

We shall simply repeat the "mismatch" trick to complete the transitive substitution.

```
[22] w> USE -MISMATCH 1 ONE-HAND-PER-THUMB                    transcript

There are 2 fates
1  (PLAN):
      PLAN --- (EQUALITY)
      1  (= !.OTHER-HAND !.HAND-OF-THUMB[4](?T.4))
      2  (IS THUMB ?T.4)
      3  (IS HAND !.HAND-OF-THUMB[4](!.ONE-THUMB))
      4  (PART ?T.4 !.HAND-OF-THUMB)
Make it local? YES
2  (PLAN):
      PLAN --- (SYMMETRY)
      1  (IS HAND !.OTHER-HAND)
      2  (PART !.ONE-THUMB !.OTHER-HAND)
Make it local? YES
Move to one of them? Type no.: 2

Plan View     <class CLASS.30>
   Documentation: "w> +PLAN RESOLUTION -MISMATCH 1 ONE-HAND-PER-THUMB"
   Supergoal:
      (= !.OTHER-HAND !.HAND-OF-THUMB)
   Steps:
   |-[1  ((IS CREATURE !.C)
```

```
        (IS THUMB !.ONE-THUMB)
        (PART !.ONE-THUMB !.C)
        => (IS THUMB !.ONE-THUMB))]
     2  (IS HAND !.OTHER-HAND) -- (no local plans, no class plans, no answers)
     3  (PART !.ONE-THUMB !.OTHER-HAND) -- (no local plans, no class plans, no answers)
   (no successors)
```

Step 2 should be easily handled with a "`use -skolem`" and, since we assumed that both thumbs were equal, a quick substitution of thumbs should take case of step 3.

```
[23] w> 2                                                              transcript

Goal View     <class CLASS.24>
   Prove
      (IS HAND !.OTHER-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (= !.OTHER-HAND !.HAND-OF-THUMB)
   (no local plans)
   (no class plans)

[24] w> USE -SKOLEM

There is 1 fate
1  (CONCLUSION): Yes
Move to it? YES

Goal View     <class CLASS.33>
   Prove
      (PART !.ONE-THUMB !.OTHER-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (= !.OTHER-HAND !.HAND-OF-THUMB)
   (no local plans)
   (no class plans)

[25] w> EQUALITY 1 --LOCAL

There is 1 fate
1  (CONCLUSION): Yes
Move to it? YES
```

```
Goal View      <class CLASS.19>
   Prove
      (LEFT !.ONE-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (LEFT !.ONE-HAND)
   (no local plans)
   (1 class plan)
```

Command 25 uses the "`-local`" designator which indicates that only local assumptions should be considered as premises for the requested refinement. Here, the only local assumption is that both thumbs are identical.

We are now done proving as a lemma that, under the assumption that both thumbs are equal, then both hands are equal too. We should proceed with the first half of the contradiction; and we shall do so by case analysis:

1. Axiom `LEFT-OR-RIGHT` expresses the fact that a hand is either on the left side or on the right side (but not both). This disjunction will constitute the starting point of our analysis. Hence the following two cases.
2. Either the first hand is on the left side, in which case the current goal (i.e. that `!.ONE-HAND` is a left hand) is trivially true.
3. Or it is on the right side; from which axiom `THUMBS` allows us to deduce that the other hand is on the left side. However, since both thumbs are equal by assumption, then both hands must be equal too, and it follows by equality substitution, from the conclusion that the other hand is on the left side, that the first hand is also on the left side.

---

[26] w> CASE (OR (LEFT !.ONE-HAND) (RIGHT !.ONE-HAND)) transcript

```
There is 1 fate
1  (PLAN):
```

```
      PLAN --- (CASE)
      1  (OR (LEFT !.ONE-HAND) (RIGHT !.ONE-HAND))
      2  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          (= !.ONE-THUMB !.OTHER-THUMB)
          (RIGHT !.ONE-HAND)
          (NOT (LEFT !.ONE-HAND))
          => (LEFT !.ONE-HAND))
Make it local? YES
Move to it? YES

Plan View      <class CLASS.19>
   Documentation: "w> +PLAN CASE (OR (LEFT !.ONE-HAND) (RIGHT !.ONE-HAND))"
   Supergoal:
      (LEFT !.ONE-HAND)
   Steps:
      1  (OR (LEFT !.ONE-HAND) (RIGHT !.ONE-HAND)) -- (no local plans, no class plans, no answers)
   |-[2  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          (= !.ONE-THUMB !.OTHER-THUMB)
          (LEFT !.ONE-HAND)
          => (LEFT !.ONE-HAND))]
      3  ((IS CREATURE !.C)
          (IS THUMB !.ONE-THUMB)
          (PART !.ONE-THUMB !.C)
          (= !.ONE-THUMB !.OTHER-THUMB)
          (RIGHT !.ONE-HAND)
          (NOT (LEFT !.ONE-HAND))
          => (LEFT !.ONE-HAND)) -- (no local plans, no class plans, no answers)
   (no successors)
```

I expect to be able to easily dispatch step 1 with axiom LEFT-OR-RIGHT, and step 2 is already solved because it is the trivial case as explained earlier.

```
[27] w> 1                                                         transcript
Goal View      <class CLASS.37>
   Prove
      (OR (LEFT !.ONE-HAND) (RIGHT !.ONE-HAND))
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (LEFT !.ONE-HAND)
   (no local plans)
```

```
   (no class plans)

[28] w> USE LEFT-OR-RIGHT

There is 1 fate
1  (CONCLUSION): Yes
Move to it? YES

Goal View      <class CLASS.39>
   Prove
      (LEFT !.ONE-HAND)
   Assumptions:
      (RIGHT !.ONE-HAND) -- !:RIGHT331
      (NOT (LEFT !.ONE-HAND)) -- !:NOT332
   (no answers)
   Supergoal:
      (LEFT !.ONE-HAND)
   (no local plans)
   (no class plans)
```

Only the non-trivial case was left, and the system moved us directly to it. As announced earlier, we shall first use axiom THUMBS to refine this goal into a new one to show that the other hand is on the right side. Then using our lemma that the two hands are equal, we should be able to perform an equality substitution and match the first local assumption !:RIGHT331. However, we can take advantage of the equality of the two hands right now, by allowing 1 mismatch in our projected refinement.

```
[29] w> USE -MISMATCH 1 THUMBS                              transcript

There is 1 fate
1  (CONCLUSION): Yes
Move to it? YES
```

This was one case where the system magically solved everything for us. Of course, it is not magical at all: there is simply enough information for the system to realize that all necessary subgoals are either trivial or have already been solved previously.

We must proceed with the alternative branch of the proof by contradiction which we initiated on page 23. We could follow the same procedure as before, but we shall see

that won't be necessary because one of the side effects of the case analysis we carried out earlier was to derive the conclusion that the other hand is on the right side.

All we need to do is to first convert the goal of proving that the first hand is *not* on the left side into the goal of showing that it is on the right side—this we can accomplish using axiom LEFT-OR-RIGHT—followed by an equality substitution replacing the first hand by the other hand, finally resulting in a goal (the other hand is on the right side) which, as anticipated, matches a conclusion derived during the earlier case analysis.

```
Goal View      <class CLASS.20>                                    transcript
   Prove
      (NOT (LEFT !.ONE-HAND))
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (NOT (= !.ONE-THUMB !.OTHER-THUMB))
   (no local plans)
   (no class plans)

[30] w> USE LEFT-OR-RIGHT

There is 1 fate
1  (PLAN):
      PLAN --- (MODUS-PONENS)
      1  (RIGHT !.ONE-HAND)
Make it local? YES
Move to it? YES

Goal View      <class CLASS.46>
   Prove
      (RIGHT !.ONE-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   (no answers)
   Supergoal:
      (NOT (LEFT !.ONE-HAND))
   (no local plans)
   (no class plans)

[31] w> USE -MISMATCH 1 THUMBS

There is 1 fate
1  (CONCLUSION): Yes
```

```
Move to it? YES

Answer to top Goal
   1  (goal): (IF (AND (IS CREATURE !.C)
                       (IS THUMB !.ONE-THUMB)
                       (PART !.ONE-THUMB !.C))
                  (AND (IS THUMB ?OTHER-THUMB)
                       (PART ?OTHER-THUMB !.C)
                       (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
     OTHER-THUMB = !.OTHER-THUMB
   (no fates)
```

The proof is now complete. We can request LOGICALC to show us a view of the top most inference.

```
[32] w> PROOF                                                    transcript

Proof
   Conclusion:
     (IF (AND (IS CREATURE ?Y) (IS THUMB ?X) (PART ?X ?Y))
         (AND (IS THUMB !.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y))
              (PART !.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y) ?Y)
              (NOT (= !.ONE-THUMB !.OTHER-THUMB))))
   (no local assumptions)
   Validation:
     (IF-INTRO (1 2) ())
   follows from:
     1  (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
     2  (AND (IS THUMB !.OTHER-THUMB[3](!.ONE-HAND[5](!.C !.ONE-THUMB) !.C))
             (PART !.OTHER-THUMB[3](!.ONE-HAND[5](!.C !.ONE-THUMB) !.C) !.C)
             (NOT (= !.ONE-THUMB
                     !.OTHER-THUMB[3](!.ONE-HAND[5](!.C !.ONE-THUMB) !.C))))
```

The surprising feature of this proof's conclusion is that, where we were expecting to see the creature !.C and its thumb !.ONE-THUMB, we find instead, and respectively, the variables ?Y and ?X. This is the result of LOGICALC's automatic generalization facility. Since this last conclusion discharges all assumptions about the creature and its thumb, LOGICALC was able to generalize over them.

Line no. 1 is the assumption being discharged, while line no. 2 is the conclusion which was derived under this assumption. We could manually explore the proof tree, but

this would not be very enlightening. Instead, I shall illustrate another feature of LOGICALC: the proof editor.

```
[33] w> EDIT                                                          transcript

Proof View
  12 (LEFT !.OTHER-HAND)                                              [THUMBS 1 2 !:RIGHT331]
  13 (LEFT !.ONE-HAND)                                                [11 12]
  14 (IF (AND (RIGHT !.ONE-HAND) (NOT (LEFT ?X))) (LEFT !.ONE-HAND))  [!:RIGHT331 !:NOT332 13]
  15 (LEFT !.ONE-HAND)                                                [6 7 14]
  16 (RIGHT !.OTHER-HAND)                                             [THUMBS 1 2 15]
  17 (RIGHT !.ONE-HAND)                                               [11 16]
  18 (NOT (LEFT !.ONE-HAND))                                          [LEFT-OR-RIGHT 17]
  19 (NOT (= !.ONE-THUMB !.OTHER-THUMB))                              [15 18]
=>20 (IF (AND (IS CREATURE ?Y) (IS THUMB ?X) (PART ?X ?Y)) (AND (IS THUM... [!:PART280 5 19]
```

The view is now that of a very abbreviated Natural Deduction-like presentation of the proof. Each line is numbered and represents a conclusion. Only a small window of lines (12 through 20) is displayed by default. Various editing operations could be performed such as hiding lines, in-lining certain conclusions, inserting comments, etc... but the default presentation is usually satisfactory. This presentation maybe requested with a "DUMP" command. I produced the hand-out by asking for a LaTeX dump to file "proof.tex." With the aid of the FLOAT command, I also slightly improved the legibility of the proof summary by moving a couple of lines closer to the point where they are used.

```
[34] w> / FLOAT 12 7 6 / UPDATE                                       transcript
!
[35] w> DUMP -DOCUMENT -TEX "proof.tex"
```

The resulting code was included in *this* document and produces the following formatted proof. First comes a reference list of all abbreviations mentioned in the proof. Second is an alphabetically ordered list of all axioms used in the proof. Finally we have the proof proper. Each line is numbered as is traditional for Natural Deduction proofs. Assumptions are *not* lines in the proof and therefore are not numbered; like axioms they are referred to by name. Each line's indentation reflects the level of

assumptions it depends on. Whenever new assumptions must be introduced a line of the form:

$$\textbf{Assume } \mathrm{H}_n \equiv a_1\ a_2 \ldots a_k + \mathrm{H}_m$$

defines a new assumption set named H_n which supplements the assumptions made by H_m with the additional a_1 through a_k. Each a_i is then defined by producing its name followed by its formula. When the same assumption set must be mentioned again at a subsequent point in the proof, the word "**Assume**" is replaced with "**Assuming,**" and the definitions of the a_i's are omitted.

The justification for each line is not always expressed in terms of the specific inference rule responsible for the corresponding conclusion; instead, it often says "from ⟨these premises⟩ conclude," where ⟨these premises⟩ are names of axioms or assumptions or line numbers denoting the corresponding conclusions. The reason for this deliberate vagueness is twofold: firstly, omitting such details makes the proof easier to read; specifying too much would detract from the frequent obviousness of the inference steps, and there is already typically so much to read that brevity must be a major concern. Secondly, not all conclusions are displayed as lines. Many inferences are too obvious to warrant a numbered line in the proof; instead they are often in-lined in ⟨these premises⟩, i.e. where they would otherwise be mentioned by line number, the names or line numbers of their respective justifying premises are inserted instead. This may sound somewhat confusing but it really tends to produce justifications which are intuitively easy to understand.

ABBREVIATIONS

```
!.ONE-THUMB  = (SK ONE-THUMB 7)
!.C          = (SK C 6)
!.OTHER-HAND[2](?ONE-HAND ?C)
             = (SK OTHER-HAND 2 ?ONE-HAND ?C)          -- THUMBS
!.OTHER-THUMB[3](?ONE-HAND ?C)
```

```
            = (SK OTHER-THUMB 3 ?ONE-HAND ?C)          -- THUMBS
!.ONE-HAND[5](?C ?T)
            = (SK ONE-HAND 5 ?C ?T)                    -- CREATURE-THUMB
!.ONE-HAND[5](!.C !.ONE-THUMB)
            = (SK ONE-HAND 5 !.C !.ONE-THUMB)          -- CREATURE-THUMB
!.OTHER-THUMB[3](!.ONE-HAND !.C)
            = (SK OTHER-THUMB 3 !.ONE-HAND !.C)        -- THUMBS
!.OTHER-HAND[2](!.ONE-HAND !.C)
            = (SK OTHER-HAND 2 !.ONE-HAND !.C)         -- THUMBS
!.HAND-OF-THUMB[4](?T)
            = (SK HAND-OF-THUMB 4 ?T)                  -- ONE-HAND-PER-THUMB
!.HAND-OF-THUMB[4](!.ONE-THUMB)
            = (SK HAND-OF-THUMB 4 !.ONE-THUMB)         -- ONE-HAND-PER-THUMB
!.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y)
            = (SK OTHER-THUMB 3 !.ONE-HAND(?X ?Y) ?Y) -- THUMBS
```

AXIOMS

```
CREATURE-THUMB
          (IF (AND (IS CREATURE ?C) (IS THUMB ?T) (PART ?T ?C))
              (AND (IS HAND !.ONE-HAND[5](?C ?T))
                   (PART ?T !.ONE-HAND[5](?C ?T))
                   (PART !.ONE-HAND[5](?C ?T) ?C)))

LEFT-OR-RIGHT
          (IF (IS HAND ?H)
              (AND (OR (LEFT ?H) (RIGHT ?H)) (NOT (AND (LEFT ?H) (RIGHT ?H)))))

ONE-HAND-PER-THUMB
          (IF (IS THUMB ?T)
              (AND (IS HAND !.HAND-OF-THUMB[4](?T))
                   (PART ?T !.HAND-OF-THUMB[4](?T))
                   (IF (AND (IS HAND ?H) (PART ?T ?H))
                       (= !.HAND-OF-THUMB[4](?T) ?H))))

THUMBS    (IF (AND (IS CREATURE ?C)
                   (IS HAND ?ONE-HAND)
                   (IS THUMB ?ONE-THUMB)
                   (PART ?ONE-THUMB ?ONE-HAND)
                   (PART ?ONE-HAND ?C))
              (AND (IS HAND !.OTHER-HAND[2](?ONE-HAND ?C))
                   (IS THUMB !.OTHER-THUMB[3](?ONE-HAND ?C))
                   (PART !.OTHER-THUMB[3](?ONE-HAND ?C)
                         !.OTHER-HAND[2](?ONE-HAND ?C))
                   (PART !.OTHER-HAND[2](?ONE-HAND ?C) ?C)
                   (IF (LEFT ?ONE-HAND) (RIGHT !.OTHER-HAND[2](?ONE-HAND ?C)))
                   (IF (RIGHT ?ONE-HAND) (LEFT !.OTHER-HAND[2](?ONE-HAND ?C)))))
```

```
TRANSITIVITY
        (IF (AND (PART ?X ?Y) (PART ?Y ?Z)) (PART ?X ?Z))
```

PROOF

Assume H1 ≡ !:PART280

```
!:IS278
      (IS CREATURE !.C)
!:IS279
      (IS THUMB !.ONE-THUMB)
!:PART280
      (PART !.ONE-THUMB !.C)
```

1 ——— From CREATURE-THUMB !:PART280 Obviously:
(PART !.ONE-THUMB !.ONE-HAND[5](!.C !.ONE-THUMB))

2 ——— From CREATURE-THUMB !:PART280 Obviously:
(PART !.ONE-HAND !.C)

3 ——— From THUMBS **1 2** Obviously:
(PART !.OTHER-THUMB[3](!.ONE-HAND !.C) !.OTHER-HAND[2](!.ONE-HAND !.C))

4 ——— From THUMBS **1 2** Obviously:
(PART !.OTHER-HAND !.C)

5 ——— From TRANSITIVITY and **3 4** Conclude:
(PART !.OTHER-THUMB !.C)

Assume H2 ≡ !:=305 + H1
!:=305 (= !.ONE-THUMB !.OTHER-THUMB)

6 ——— By equality substitution of !:=305 in **3** Conclude:
(PART !.ONE-THUMB !.OTHER-HAND)

7 ——— From ONE-HAND-PER-THUMB and **6** Conclude:
(= !.HAND-OF-THUMB[4](!.ONE-THUMB) !.OTHER-HAND)

8 ——— From ONE-HAND-PER-THUMB and **1** Conclude:
H1 ≡ !:PART280 ⊢
(= !.HAND-OF-THUMB !.ONE-HAND)

Assuming H2 ≡ !:=305 + H1

9 ——— By equality substitution of **7** in **8** Conclude:
(= !.ONE-HAND !.OTHER-HAND)

10 ——— From LEFT-OR-RIGHT Obviously:
H1 ≡ !:PART280 ⊢

```
(OR (LEFT !.ONE-HAND) (RIGHT !.ONE-HAND))

  Assuming H2 ≡  !:=305 + H1

  11 ——— Discharging Obviously:
     (IF (LEFT ?X) (LEFT ?X))

       Assume H4 ≡  !:RIGHT331 !:NOT332 + H2
          !:RIGHT331
                (RIGHT !.ONE-HAND)
          !:NOT332
                (NOT (LEFT !.ONE-HAND))

       12 ——— From THUMBS 1 2 and !:RIGHT331 Conclude:
          (LEFT !.OTHER-HAND)

       13 ——— By equality substitution of 9 in 12 Conclude:
          (LEFT !.ONE-HAND)

  14 ——— From 13 Discharging !:RIGHT331 !:NOT332 Conclude:
     H2 ≡ !:=305 H1 ⊢
     (IF (AND (RIGHT !.ONE-HAND) (NOT (LEFT ?X))) (LEFT !.ONE-HAND))

  15 ——— By case analysis 10 11 14 Conclude:
     (LEFT !.ONE-HAND)

  16 ——— From THUMBS 1 2 and 15 Conclude:
     (RIGHT !.OTHER-HAND)

  17 ——— By equality substitution of 9 in 16 Conclude:
     (RIGHT !.ONE-HAND)

  18 ——— From LEFT-OR-RIGHT and 17 Conclude:
     (NOT (LEFT !.ONE-HAND))

19 ——— By contradiction 15 18 Conclude:
   H1 ≡ !:PART280 ⊢
   (NOT (= !.ONE-THUMB !.OTHER-THUMB))

20 ——— From 5 19 Discharging !:PART280 Conclude:
  H0 ⊢
  (IF (AND (IS CREATURE ?Y) (IS THUMB ?X) (PART ?X ?Y))
      (AND (IS THUMB !.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y))[6]
           (PART !.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y) ?Y)
           (NOT (= !.ONE-THUMB !.OTHER-THUMB))))
```

[6]Proofs of uninteresting facts are suppressed from the summary. A formula predicating the type of an object is considered uninteresting. For more information, see Chapter 9.

1.4 Organization of the Thesis

Chapter 2 provides a synthetic overview of the system, covering all major aspects, features, and mechanisms, but mostly in a superficial manner. These subjects will be fully developed in subsequent chapters. Chapter 3 is a formal exposition of the logic. Chapter 4 describes the graph editor and general principles of the command shell. Chapter 5 is a fairly detailed presentation of the framework's implementation, in terms of data-structures and the bookkeeping operations managing the graph. Chapter 6 discusses plan generators and plan generation. Chapter 7 explains the "detaching" mechanism and provides high-level descriptions of the algorithms in a functional programming style with pattern matching. Chapter 8 describes the mechanism for automatic generalization of conclusions, and investigates its relation to Explanation-Based Generalization. Chapter 9 presents the proof editor as well as its formatting conventions and default heuristics, and explains how to prepare a proof for publication, in particular for inclusion in a TEX or LATEX document. In practice, little editing is necessary as the default heuristics give surprisingly good results. Chapter 10 presents an axiomatization of Kuipers-style qualitative physics which served as the major source of realistic problems for the testing and evaluation of LOGICALC. Finally, Chapter 11 Concludes this dissertation with a recapitulation of the system and its constributions, a discussion of related work, an implementation of tactics and tacticals in our framework, and some suggestions for further research. Appendix A briefly discusses the language for writing axiomatizations, while Appendix B describes Find Mode.

Chapter 2

Overview

LOGICALC follows the LCF paradigm [Mil79a, Pau87]: goals are analysed top-down, by refining them into plans and subgoals, until all leaves have been identified with axioms or assumptions. Proofs are synthesized bottom-up by executing the plans' validations.

The process of analysis is carried out interactively through a graph editor. The user navigates and extends a graph of goals, plans, and proofs, until a satisfactory answer has been synthesized for the top-goal (i.e. the theorem to be proven).

In this chapter, I will first discuss skolemization and axiomatic definitions. Secondly, I will introduce the conceptual entities of the framework, such as goals and plans. Thirdly, I will outline the principles of operation and give the reader a sense of how the various mechanisms, described in the remainder of this thesis, fit together. Finally, I will conclude with further details on skolemization and unification, as well as with a brief presentation of the ATMS database, and the mechanisms for indexation and retrieval.

Implementation Language and Tools. LOGICALC is written in NISP [McD83,

McD88] and benefits from tools provided by the DUCK system [McD85]. NISP essentially supplements LISP with type-checking, a powerful and extensible type-system with polymorphism,[1] a comprehensive library, and is portable between Common Lisp and Scheme (T). The DUCK system is writen in NISP and provides tools for manipulating formulae and managing databases of predicate calculus assertions with an ATMS; LOGICALC takes advantage of these tools and often extends them.

2.1 Introduction

DUCK provides a set of tools to manage and interact with multiple predicate calculus databases driven and arbitrated by an active reason maintenance system. In this section, I briefly recapitulate these notions and mechanisms inherited from DUCK which are relevant to LOGICALC; further details can be found in Appendix A and [McD85]. My secondary purpose is to anticipate and motivate subsequent sections.

Stating An Axiom. New axioms are introduced by means of the `AXIOM` special form whose syntax is `(AXIOM` *name formula*`)`.

Formulae are written in a LISPish notation. For instance, $p_1 \vee p_2 \vee \ldots \vee p_n$ and $p \supset q$ are respectively written `(or` p_1 p_2 `...` p_n`)` and `(if` p q`)`.

Quantified propositions are expressed naturally in a similar fashion; e.g. $\forall x\, y\ P$ is written `(forall (`x y`)` P`)`—similarly for $\exists$ (`exists`).

When a new axiom is introduced, DUCK verifies that its formula is well-formed and well-typed, then stores it in the database in a form appropriate for efficient subsequent retrieval. I will now elaborate on the details of this procedure.

[1] Parametric, ad-hoc, and inclusion polymorphism are all present in one form or another, although, not necessarily in their full generality.

Syntax and Type Checking. When DUCK is handed a formula, it makes sure that it is well-formed and well-typed.[2] For example, the `AXIOM` special form enforces this condition on its *formula* argument. The procedure implementing this capability is driven by type declarations. There are two categories of type declarations: (1) declarations of constants, and (2) declarations of (bound) variables.

The types of constant functions and predicates are declared globally with `duclare`, `deffun`, and `defpred` [Appendix A]. Thus `if` has type `(prd [prop prop])`, i.e. it is a predicate of two propositional arguments. Therefore, DUCK will enforce the restriction that an implication can only appear where a proposition is expected, and that it must have exactly two arguments, both propositions.

A variable—introduced by a quantifier—may also be associated with a type declarations, e.g:

```
(forall (n - integer) (= n (+ n 0)))
```

In the quantifier's binding list, variables are interleaved with types. In order to distinguish variables from types, types are preceded by a dash—all variables between this type and the previous one (or the beginning of the binding list) are declared to be of this type. In the formula above, `n` is declared to be an integer.

Type Definitions. DUCK knows about a number of predefined types. However, it is often convenient, when axiomatizing a domain, to introduce new types. The form `(defducktype` *new old*`)` defines *new* to be a subtype of *old*. Most commonly, this is used with *old* ≡ `obj`, where `obj` is at the top of the type lattice (everything is an `obj`), e.g:

[2]Since we are manipulating actual LISP expressions, low-level issues of syntax have already been taken care of by the LISP *reader*, and, at this point, well-typing subsumes well-formedness.

```
(defducktype quantity obj)
```

defines `quantity` to be a new type with no interesting properties.

Querying The Database. The point of having a database is so that we can ask questions and get answers. Traditionally, facts are stored in the database, e.g. `(p a)`, and a query often contains free variables, e.g. `(p ?x)`, which stands for the question *"does* p *hold for certain values of* ?x, *and, if yes, which ones?"*; answers take the form of substitutions, e.g. {?x ← a}.

DUCK also supports backward-chaining rules of the form (<- p (and q_1 ... q_n)) much like PROLOG's p :- q_1, ... ,q_n. LOGICALC takes advantage of this facility to implement built-in theories.

In order to allow querying the database in this manner, it is necessary to *skolemize* queries and formulae in the database, and to use *unification* during the retrieval process. Thus, `(p ?x)` is the skolemized form of the query `(exists (x) (p x))`.

When a new axiom is introduced with the `AXIOM` special form, after it has been syntax-checked, it is automatically skolemized prior to being recorded in the database.

Goals in a Theorem-Prover. In a theorem-proving system, queries become goals and the technique for deriving answers using backward-chaining rules is generalized into the construction of a proof tree by means of inference rules.

Here too, skolemization and unification can be enlisted to considerable advantage; (1) to search the database for relevant axioms (i.e. containing occurrences of the goal); (2) to manipulate goals, and effect *refinements*.

Sequents. The traditional technique for proving a proposition of the form $p \supset q$ is to assume p and prove q. In order to formalize this technique, it is convenient to use a notation that makes explicit the set of assumptions under which we are currently operating.

For example, to indicate that r follows from the set of assumptions $\{a_1, \ldots, a_n\}$, we write:

$$a_1, \ldots, a_n \Rightarrow r$$

and we call such an object a *sequent*. From $A, p \Rightarrow q$—where $A, p \equiv A \cup \{p\}$—the DEDUCTION THEOREM [And86, p59] permits us to conclude $A \Rightarrow p \supset q$.

In LOGICALC, not only are sequents used in the representations of conclusions, but in the representations of goals as well. In the context of a goal, the sequent $A \Rightarrow p$ means that we are trying to derive a proof of an instance of p (which may have free variables) from the set of assumptions A.

Assumption Sets and Datapools. DUCK provides a particularly suitable concept to capture the notion of a set of assumptions: *datapools* [McD79]. The database is managed by an Assumption-Based Truth Maintenance System (ATMS)[3] whose task is to maintain a view of the database according to the *current* datapool: all formulae which are assumptions of the current datapool are labelled IN, which means that they are visible and known to be true in this context. All formulae derived from premises which are IN are similarly labelled IN as well; and so on by transitive closure. All other formulae are labelled OUT and are not visible (either they are false, or merely not known to be true). Thus, by changing the current datapool (such as when we move from one goal to another) we get a different view of the database.

The more primitive notion is that of a *bead*. We can mark a formula with one or more beads. When a bead is selected, all those formulae which have been marked

[3] See e.g. [Doy79,McA78,McA80,dK84,dK87,dK88,McD79,McD85,McD89].

with it are labelled IN. A datapool is a set of beads; selecting the current datapool simultaneously selects every one of its beads. Datapools are more convenient because they make it easy to capture the notion of hierarchical sets of assumptions: let $\mathcal{DP}$ be a datapool whose set of beads is $\{b_1, \ldots, b_n\}$, and let b_{n+1} be another bead; then $\mathcal{DP}'$ defined by $\{b_1, \ldots, b_n, b_{n+1}\}$ inherits all the formulae in $\mathcal{DP}$ plus those specified by b_{n+1}.

In a system of Natural Deduction where proofs are constructed by incremental refinements, hierarchical sets of assumptions arise naturally. Therefore datapools turn out to be particularly well-suited to our purpose.

2.2 Skolemization

Skolemization and unification are fundamental operations in LOGICALC. Much of the benefits afforded by the system are derived directly or indirectly from their use. In this section, I provide a brief, non-technical, introduction to skolemization, and discuss some extensions to the classical unification algorithm.

2.2.1 Introduction

In a practical sense, skolemization can be regarded as an operation which transforms a formula by removing the quantifiers occurring within and replacing quantified variables with free variables or skolem terms in a manner that preserves the meaning of the original formula; e.g. the assertion that `(forall (x) (p x))` becomes `(p ?x)`, where `?x` represents the free variable `x`.

In what sense is the meaning preserved? From the original assertion and for any term `t` we were licensed to infer `(p t)` since the assertion stated that `p` held for all `x`'s.

Similarly, for any assertion which contains free variables, we are allowed to infer an instance of it by plugging in other terms for these variables.

For existentially quantified variables the transformation is somewhat more involved. First, consider the statement `(exists (x) (q x))`. The interpretation is that there exists some object which satisfies **q**. The object itself is unspecified, but it is typically convenient to name it arbitrarily; e.g. we would say: "let **a** be such an **x**," where **a** is a new name we just made up—the name has to be new so that it doesn't conflict with another object named **a**. Skolemization will also make up a new name, usually notated `!.x`, and produces the formula `(q !.x)`.

`!.x` is called a skolem term, and is represented internally by a LISP expression of the form (**sk x** n), where n is an integer which is different for every new skolem term. You can think of (**sk x** n) as a composite name for the new constant $\mathbf{x}_n$. Since n is increased for every new skolem term, these names can never conflict with existing constants.

When a formula has both universally and existentially quantified variables, skolemization becomes slightly more complex. Consider for example the assertion:

```
(forall (x) (exists (y) (< x y)))
```

Removing the first quantifier, we obtain `(exists (y) (< ?x y))`. However, we cannot simply remove the second quantifier and replace y with the new constant `!.y` because the assertion `(< ?x !.y)` would imply that any `?x` is less than this `!.y`. Clearly, this is not what was originally intended. Instead, we should construe y—that is, the y whose existence is asserted by `(exists (y) (< ?x y))`—as a function of `?x`: given an `?x`, it is possible to find a y whose value depends on the particular value of `?x` and such that `(< ?x y)` is true for those values.

Thus, from the above assertion, skolemization produces the formula `(< ?x !.y(?x))`, where `!.y(?x)` is the printed representation for a skolem term whose function is y

and whose argument is ?x. Internally, this term is represented by the expression (sk y *m* ?x).

2.2.2 Predicate Calculus Databases

Skolemization is a technique particularly well suited to the tasks of maintaining and querying databases of propositional assertions. Every formula is skolemized prior to being recorded in the database; e.g. (forall (x) (p x)) will be recorded as (p ?x). Subsequently, we can inquire whether (p a) is true by attempting to match it against each pattern in the database.

Tentative matching of this sort is carried out by a procedure called "unification" [Rob65]. In the case of (p ?x) against (p a), not only will unifiation succeed, but it will also return the information that the term a must be plugged into the free variable ?x; the latter piece of information is usually notated $\{ ?x \leftarrow a \}$ and is called a "substitution."

Querying a database does not necessarily require successively attempting unification with every element stored therein; such an approach would be prohibitively expensive in practice. Instead, some sort of indexing scheme is usually maintained to facilitate and speed up access to relevant assertions. A widely used technique is to file each assertion under the heading of its main symbol. Thus, (p ?x) would be indexed under p (see section 2.7.1 for a discussion of indexing). Answering the query (p a) simply requires unifying with the patterns indexed under p.

Note that, if we did not skolemize, the technique presented above would place all formulae of the form (forall (*vars...*) *formula*) under the heading forall which would be useless. Furthermore, we would not be able to use unification to retrieve information in as free and uniform a fashion. Consider the assertion:

```
(forall (x) (exists (y) (if (p x) (q y))))
```

It is equivalent to `(forall (x) (if (p x) (exists (y) (q y))))`, yet such a fact is difficult to recognize automatically simply because of the syntactical difference; unification, being essentially a syntactical matching operation, utterly fails to identify the two expressions.

Also, given the query `(if (p a) (q ?z))`, no useful answer can be derived even though the above assertion guarantees the existence of a `?z` which satisfies the query. However, here again, the syntactical differences get in the way of successful unification. Furthermore, even if unification were to be extended to correctly handle the syntax problem, it remains that the unskolemized assertion does not give a name to the object whose existence it guarantees. Consequently, we cannot plug that name into the free variable `?z` to obtain the desired answer. In contrast, the skolemized version of the assertion is:

```
(if (p ?x) (q !.y(?x)))
```

and an answer can be constructed by unification, thereby producing the substitution { `?z` ← `!.y(a)` }.

As we can see, the great advantages of skolemization are that it factors out syntactical variations in quantification, allows better indexing techiques, makes it possible to query a database using unification, and gives names to objects whose existence is asserted.

2.2.3 Skolemization of Axioms

I will now illustrate skolemization on an example taken from an earlier version of my axiomatization of Qualitative Physics.[4]

```
(AXIOM NEXT-OF-AT
  (FORALL ((BEFORE ??T1 FOREVER) - timepoint)
    (IF (REACHABLE (AT T1))
        (EXISTS ((BEFORE T1 ??T2) - timepoint)
          (= (NEXT (AT T1)) (BETWEEN T1 T2))))))
```

In my formalization of Qualitative Physics, there is the notion of a situation. A situation is either a *point situation*, and corresponds to a specific time-point, or it is an *interval situation*, and corresponds to the interval between two time-points. The two kinds of situations alternate. In particular, the axiom above expresses the condition that a point situation must necessarily be followed by an interval situation. The only way to stop is when the far end of an interval situation is at infinity.

This axiom illustrates several features of the language. Firstly, it gives the flavor of its LISPish syntax. Secondly, it shows that type declarations are allowed in the binding lists of quantifiers—a binding list is the expression that immediately follows a quantifier, and is usually a list of the variables which are being quantified over; a type declaration is preceded by a dash and applies to the objects before it (e.g. `(forall (x - integer) (p x))`). Thirdly, a binding list may also contain *constraints*, such as `(before ??t1 forever)`; the variables being quantified over are identified by the `??` prefix.

When this axiom is presented to the system, it will be skolemized into the following expression prior to being stored in the database:

[4]The version presented in Chapter 10 is more recent, and no longer includes axiom `next-of-at` which is used as an illustration here.

```
(IF (AND (IS timepoint ?T1)
         (BEFORE ?T1 FOREVER)
         (REACHABLE (AT ?T1)))
    (AND (IS timepoint !.T2(?T1))
         (BEFORE ?T1 !.T2(?T1))
         (= (NEXT (AT ?T1)) (BETWEEN ?T1 !.T2(?T1)))))
```

The universally quantified variable `t1` was replaced with the free variable `?t1`, and the existentially quantified variable `t2` with a skolem term of the form e.g. `(sk t2 17 ?t1)`[5] that prints as `!.t2(?t1)`.

Type declarations resulted in the addition of type constraints to the formula. Each type constraint is represented by an `IS`-expression.

Similarly, constraints occurring in binding-lists are added to the formula: for universal quantification, a constraint contributes an additional condition, whereas, for existential quantification, it contributes an additional conclusion. This is why the first constraint was added to the implication's antecedent, while the second one was conjoined with the consequent.

Negation switches the roles of $\forall$ and $\exists$. So does implicit negation, such as in the antecedent of an implication (because $p \supset q \equiv \neg p \vee q$). Thus, the proposition:

```
(IF (FORALL (X) (P X)) (FORALL (Y) (P Y)))          (Υ)
```

is skolemized as follows:

```
(IF (P !.X) (P ?Y))
```

[5]The number 17 has no special significance; the number which is selected is the value of the global counter **sknum***, which is then incremented. Thus a different number is chosen for each new skolem term.

2.2.4 Skolemization of Goals

Queries to the database and goals in a theorem-prover are skolemized in a dual manner, i.e. as if within a negation. Thus, as a goal, the proposition (Υ) is skolemized as follows:

```
(IF (P ?X) (P !.Y))
```

It is clear that, if we want to show that "there exists" some x such that the formula is satisfied, all we need to do is exhibit such an x; i.e. find a substitution for the free variable x such that the resulting instance holds.

Conversely, if we want to show that "for all" ys the formula is valid, then all we need to do is make up a new name and prove that the formula holds of it—the intuition being that, if the proof obtains for this arbitrary constant, about which we made absolutely no assumptions, then it would work just as well for any term substituted throughout in its place. In other words, the conclusion would hold for all terms.

This technique captures the following mathematical practice: to prove that $\forall x{:}\tau\ P(x)$, let a stand for an object of type τ, and show that $P(a)$.

Skolemization automatically provides such new names in the form of skolem terms.

The logical intuition behind the duality of skolemization for axioms and goals can also be captured by either of the following remarks:

- Proving that p follows from the set of assumptions A can be done by adjoining $\neg p$ to A and showing that the resulting set $A \cup \{\neg p\}$ is inconsistent.
- The object is to show that the implication $A \supset p$ is valid, where A is interpreted as the conjunction of its elements. By virtue of being on opposite sides of an implication, A and p have opposite polarities and must be skolemized with reversed conventions.

2.3 Fundamental Concepts

In this section, I provide brief introductions to those conceptual entities which form the basis of LOGICALC's framework.

2.3.1 Goals and Answers

A goal consists primarily of a formula p and a set of assumptions A. Conventionally, this is represented by the notation $A \Rightarrow p$. The point is to find a substitution θ of the free variables of p such that θp follows logically from A.

An answer to the goal $A \Rightarrow p$ is such a θ, together with a proof tree demonstrating how θp can be inferred from A. A goal may have several answers if different instances of p can be derived from A. Answers are normally arrived at either by unifying the goal's formula with an axiom or an assumption, or by proposing, then solving a plan. Several alternative plans may be attached to the same goal.

2.3.2 Assumption Sets

At the top-level, the assumption set is the global database, which contains axioms for various domain theories. In the course of developing a proof, it is possible to obtain subgoals whose assumption sets contain additional hypotheses. Consider a goal of the form $A \Rightarrow p \supset q$. The natural technique for proving such a goal is to assume p and try to prove q. Plan generator DEDUCTION-THEOREM will do this for you: for the goal $A \Rightarrow p \supset q$, it will propose a plan with the 1 step $A; p \Rightarrow q$, where $A; p$ denotes the set $A \cup \{p\}$. The plan's validation will discharge the assumption when the step has been solved: from a proof of $A; p \Rightarrow q$, it concludes $A \Rightarrow p \supset q$ by IF-INTRO.

For a Natural Deduction style of proof, the ability to make assumptions and subsequently discharge them in this manner is essential.

Since the DEDUCTION-THEOREM can be applied to goals that already make assumptions, a natural hierarchy (partial order) exists among assumption sets. For example, A is said to be the *parent* of $A; p$. The global database is the ancestor of all assumption sets.

2.3.3 Plans and Validations

Plans are constructed as a result of invoking *plan generators*, which will be further discussed later. A plan is attached to a *supergoal*, and consists of a set of steps together with a *validation*. Each step is a subgoal. A validation is the description of a procedure to derive a new logical conclusion from premises supplied as inputs. When all steps have been proven, their proofs are fed to the validation which combines them, using inference rules, into a new proof. This proof becomes an answer to the supergoal.

Steps are really of two kinds: *solved* and *pending*. Initially, all steps are pending. When a step has acquired one or more answers, the user has the option to *follow* one of them. Following an answer to a step means this: a copy is made of the plan in which the step occurs. This copy is called a *successor plan*. It differs from the original plan in that the step in question is removed from the *pending* list, and its answer is inserted in the *solved* list instead. The pending list is instantiated using the answer's substitution θ.

Following an answer to the last pending step in a plan causes the validation to be executed and an answer to be derived for the supergoal.

2.3.4 Proofs

A proof is a sequent $A \Rightarrow p$ indicating that p logically follows from A, together with a *proof tree* recapitulating the various inferences required to effect its derivation. Each node of this proof tree is itself a proof. The leaves of this tree are axioms and assumptions.

Proofs are created by inference rules. An inference rule derives a conclusion from premises; both consist of proofs. An inference rule may also require parameters to further specify the details of the derivation.

A conclusion proof recapitulates its own genesis: it is an aggregate consisting of a sequent representing the conclusion, the inference rule responsible for its derivation, the premises it was derived from, and the parameters characterizing the details of this inference.

Since a proof packages its own premises, it can also be regarded as a tree: a proof tree. Thus a proof (node) has a dualistic interpretation: (1) it contains a sequent representing a conclusion; (2) it *is* a proof tree recapitulating the derivation of this conclusion.

As I mentioned earlier, when the user follows an answer to the last pending step of a plan, the plan's validation is executed. The validation combines the proofs of the plan's steps using inference rules. A proof is thereby constructed which solves the plan's supergoal. This mechanism is the primary source of new proofs.

Another way of obtaining new proofs is by forward inferencing through the mediation of *inference generators*. They allow the user to augment the database with new conclusions derived from existing assertions. Since the preferred mode of interaction with LOGICALC is backward chaining using *plan generators*, inference generators will not be further discussed in this thesis.

LOGICALC incorporates a mechanism to automatically generalize proofs. In this context, "to generalize" means precisely this: to replace certain terms with free variables while preserving validity. For example $A \Rightarrow p(a) \supset p(a)$ is generalized into $A \Rightarrow p(x) \supset p(x)$.

2.3.5 Goal Classes

The picture painted so far is unfortunately inaccurate in that it blissfully ignores a major contribution of LOGICALC's graph structure: namely, *goal classes*. I will now introduce this notion.

It is not uncommon for the same goal, or variants of the same goal (modulo renaming free variables), to occur in different plans in various places of the growing graph structure. For example, if we often mention a particular term a, which is of type τ, *proving* that a is of type τ may well be a frequently recurring subgoal. It would be nice if we could just prove it once in a way that would automatically benefit all occurrences of this goal, rather than to be forced to go through the same tedious motions for each occurrence. The major source of recurring subgoals, however, is the mechanism creating successor plans [see above], because it must make new copies of the plan's remaining pending steps.

in LOGICALC, goals are grouped in equivalence classes. For example, the two goals $A \Rightarrow p(x)$ and $A \Rightarrow p(y)$ are said to be (alphabetical) variants of one another: they are identical modulo renaming free variables (e.g. renaming y to x). Both goals belong to the same equivalence class.

An equivalence class is represented by an aggregate called a *goal class* (or *class* for short). It serves as a centralized bookkeeping device for answers: whenever a goal—member of a class $\mathcal{C}$—acquires an answer, it registers this answer in $\mathcal{C}$, thus sharing it with its fellow members.

Now that I have eased the reader into this notion of goal classes, I will remove the final layer of deception. Goals do not in fact play the central rôle in which I have cast them so far; classes do. Contrary to what I claimed earlier, plans are not attached to a supergoal but to a superclass. Correspondingly, when an answer is constructed by executing a plan's validation, it is recorded with the superclass, not with a supergoal.

So what is a goal? In a sense, it is nothing more than a funny-shaped window through which we view its class. Each member of a class $\mathcal{C}$ gives us a different view of $\mathcal{C}$, but it is a different view of the same thing. A goal is simply the representation of an occurrence of a particular class, with the variables renamed in some way. For example, when a plan contains the step $A \Rightarrow p(x)$, this step will indicate that it is an occurrence of a class $A \Rightarrow p(u)$ with the renaming $\{u \rightarrow z\}$. If another plan contains the step $A \Rightarrow p(y)$, this step will similarly be represented as an occurrence of class $A \Rightarrow p(u)$ with the renaming $\{u \rightarrow y\}$.

For practical purposes and convenience, goals are a little more complex than just a pointer to a class plus a renaming. I will say more about this later.

Classes were designed to address the issue of recurring subgoals; but what of *related* goals? Consider a class $\mathcal{C}_1$: $A \Rightarrow p(a)$, and a class $\mathcal{C}_2$: $A \Rightarrow p(u)$. Clearly, an answer to $\mathcal{C}_1$ is also an answer to $\mathcal{C}_2$ that instantiates u with the constant a. Therefore, it would be desirable to have a mechanism to automatically communicate this answer to $\mathcal{C}_2$ when it is recorded in $\mathcal{C}_1$. Conversely, should $\mathcal{C}_2$ acquire an answer that assigns a to u, this answer would also work for $\mathcal{C}_1$, and the latter should be told about it. For precisely this purpose, each class has links to *similar* classes. When a class acquires an answer, it *broadcasts* it along these links. The recipient classes must then each determine whether the answer is admissible—in which case it is recorded—or not—in which case it is ignored.

The measure of similarity used to establish these links is based on the notion of generality. For example, if p' is an instance of p (i.e. there exists a substitution θ such

that $\theta p \cong p'$[6]), then $A \Rightarrow p$ is more general than $A \Rightarrow p'$. Also, if $B \subset A$, $A \Rightarrow p$ is more general than $B \Rightarrow p$. Each class has links to all those classes which are more or less general than itself.

Thus, the underlying graph helps make the most of each answer by handling directly certain practical issues of subsumption: (1) goals are grouped in equivalence classes and share their answers; (2) related classes are connected by links, and communicate their answers to one another by broadcasting them along these links.

In practice, the user can remain completely oblivious to the existence of classes, and interact exclusively with goals and plans [actually, *views* of same].

2.4 Principles of Operation

In this section, I am going to introduce a number of underlying mechanisms and explain how they all fit in the overall framework. I will do this by reviewing typical interactive operations and outlining what is really happening behind the scenes.

2.4.1 Initiating a Session

The most important preliminary step is to *axiomatize* the domain of interest (e.g. qualitative physics or some such). For example, in Chapter 10, I present an axiomatization of Kuipers-style qualitative physics. Normally, the user will write these axioms in various files and *load* these files into the LISP system prior to initiating the session proper.

The session is started by invoking the `logicalc` procedure. In response to a prompt, the user states the theorem to be proven. This formula is syntax/type-checked. Next,

[6] I am using $\cong$ to express syntactic identity because I will usually write $\equiv$ to denote the equivalence connective.

it is skolemized[7] and a goal class is constructed to represent the resulting pattern. Then, the system installs the top-level goal which is a member (occurrence) of this class.

The top-level goal is the only goal which is not a step in some plan. The object of the session with LOGICALC is to interactively—and monotonically—grow a graph of goals, classes, plans, proofs and answers, until a satisfactory answer to the top-goal has been synthesized.

2.4.2 Solving a Goal

A goal is a member of a class. As I explained earlier, a class is essentially a centralized bookkeeping device for answers. When a goal is created which is a member of a preexisting class, it is possible that the class already possesses answers. In this case, the new goal will also have these answers simply by virtue of being a member of the class. Thus a goal may be solved by *inheriting* solutions derived earlier.

If a goal is created which cannot be assigned to a preexisting class, then a new class must be constructed to house it. This new class is *initialized* by a two step process: firstly, links are established to and from more/less general classes, and secondly, the system attempts to derive *obvious* answers for the new class.

Obvious answers are acquired in the following ways:

- Related classes are examined and admissible answers are borrowed from them. This step makes up for the fact that the new class was never broadcast to.
- A decision procedure is applied to determine whether the class is a tautology.

[7] Unless skolemization has been turned off by setting `lc-skolemize*` to NIL.

- The assumption set is consulted to find axioms or assumptions that unify with the class's conclusion.
- Built-in theories or decision procedures are invoked. In LOGICALC, these are currently implemented by backward-chaining logic programs.

Thus a goal may simply be solved as a consequence of the initialization mechanism. If a goal does not acquire a satisfactory answer either by inheritance or by initialization (as the case may be), then one must resort to proposing, then solving, plans for it.

2.4.3 Creating Plans

When viewing a goal (or a class) through LOGICALC's graph editor, the user may create one or more (additional) plans for it by invoking a *plan generator*. This is done by issuing a command of the form *plangen args...* where *plangen* names the desired plan generator, and *args...* are arguments denoting the premises and parameters required to perform and/or specify the expected refinement.

Suppose that we are now viewing (we say *at*) a goal of the form $A \Rightarrow q(f(x))$ and that **foo** is an assertion in A of the form $p_1(u) \wedge p_2(u) \supset g(u) = f(f(u))$, then the command '**equality f - foo**' will result in the creation of a plan with the following 3 steps:

$$1.\ p_1(u),\ \ 2.\ p_2(u),\ \ 3.\ q(g(u))$$

Furthermore, the plan's conclusion imposes the following assignment: $x \leftarrow f(u)$.

I am now going to explain how the system goes from the command issued earlier, to the plan shown above. This is accomplished in 4 successive phases:

1. Processing the arguments.

2. Constructing plan descriptions.

3. Expanding plan descriptions.

4. Creating plans according to these descriptions.

Processing the Arguments

The plan generator must first make sense of the arguments typed by the user. In the case of the EQUALITY plan generator, the user must specify both the term(s) to be substituted by equality, and the equality premise to use for this purpose. To make parsing the arguments easier, the command line must be an instance of the following template:

$$\texttt{equality}\ \langle \textit{term designators} \rangle\ \texttt{-}\ \langle \textit{equality designator} \rangle$$

In our example,

$$\langle \textit{term designators} \rangle \equiv \texttt{f}$$
$$\langle \textit{equality designator} \rangle \equiv \texttt{foo}$$

Note that in both cases, further processing is required to turn the user's specifications into a useful parameter or premise:

- From the symbol `f`, the system must determine that the user means to denote the occurrence of the term $f(x)$ in goal $A \Rightarrow q(f(x))$. In general, there is a procedure that turns ⟨*term designators*⟩ into a set of pointers[8] to specific occurrences of terms in the goal.

[8] A so-called pointer is actually a list of integers representing a selection path into the formula and leading to the denoted occurrence, and is ordinarily simply called a *path*.

- From the symbol `foo`, the system must determine that the user is interested in the equality formula $g(u) = f(f(u))$ occurring in assertion `foo`. Also, since this equality is a subformula of `foo`, in order to become available for use, it must be inferred from `foo`. Therefore, the system must come up with a plan to derive the desired equality from `foo`. It is the description of this plan which will be used in place of the equality premise (its projected conclusion *is* the equality premise).

  The procedure, which here turns the ⟨*equality designator*⟩ into a plan description, is called *detaching*. It determines that $g(u) = f(f(u))$ can be inferred from `foo` by MODUS-PONENS, but requires $p_1(u) \wedge p_2(u)$ as a minor premise. The resulting plan description is:

  (plan $g(u) = f(f(u))$ **modus-ponens (foo (goal** $p_1(u) \wedge p_2(u)$**)))**

The results of this phase—i.e. the set of specific term occurrences, and the plan description for inferring the equality—are packaged in a standardized format and handed over to the next phase.

Constructing Plan Descriptions

The second phase is given a description of the current goal, and the set of premises and parameters assembled by the previous (parsing) phase. The object is to construct a description of the plan corresponding to the requested refinement.

In the case of our example, there is one premise whose conclusion is $g(u) = f(f(u))$, and one parameter which denotes the occurrence of term $f(x)$ in goal $A \Rightarrow q(f(x))$. The system must perform the equality substitution backwards on the current goal to obtain a new subgoal—I say 'backwards' because the substitution here is $g(u) \leftarrow f(f(u))$, whereas, when the plan has been solved and its validation is executed to

infer a conclusion from a proof of the subgoal, the substitution is performed in the other direction, i.e. $g(u) \rightarrow f(f(u))$.

$f(f(u))$ is unified with subterm $f(x)$, thus establishing the assignment $x \leftarrow f(u)$, and the following description is constructed for the subgoal:

```
(goal q(g(u)))
```

Note that the subgoal's assumption set is implicitly the same as that of the current goal. If additional assumptions had to be introduced, they could be specified as extra arguments to the `(goal ...)` construct.

Now, the system must assemble the various parts into a plan description which captures the idea that the resulting proof is to be inferred by EQUALITY from a proof of the premise and a proof of the subgoal:

```
(plan q(f(f(u))) equality
   ((plan g(u) = f(f(u)) modus-ponens
        (foo (goal p1(u) ∧ p2(u))))
    (goal q(g(u))))
   ⟨parameter denoting subterm g(u)⟩)
```

Notice that the respective descriptions of both premise and subgoal were simply plugged in.

Expanding Plan Descriptions

For the user's convenience, the system does a little extra work on the plan description. For example, goals which are conjunctions are *expanded* into subplans with one step for each conjunct. Similarly, goals which are double-negations are expanded into

subplans whose step has the double-negation stripped off. This behavior can be disabled by the user.

During this phase, there is the potential to perform arbitrarily complex plan transformations. However, LOGICALC does not at present attempt to handle anything but conjunctions and double-negations.

The example description assembled earlier is thus transformed into:

```
(plan q(f(f(u))) equality
   ((plan g(u) = f(f(u)) modus-ponens
       (foo (plan p1(u) ∧ p2(u) and-intro
               ((goal p1(u))
                (goal p2(u))))))
    (goal q(g(u))))
   ⟨parameter denoting subterm g(u)⟩)
```

Creating a Plan According to a Description

Finally, from the plan description, a plan structure must be constructed, and the resulting plan and goals aggregates must be appropriately inserted in the existing graph.

In order to construct a plan, we must be able to determine (1) the steps, and (2) the validation. Both can be established from the description. The steps are simply the set of goal descriptions, and names of assertions used as premises. Inspection of our example description yields the following set of steps:

0. foo, 1. (goal $p_1(u)$), 2. (goal $p_2(u)$), 3. (goal $q(g(u))$)

`foo` is assigned step number 0: steps which correspond to assertions do not have to be proven since they are elements of the assumption set. They are assigned numbers starting from 0 downwards, which makes it easy to not display them when a plan node is being examined by setting variable **`show-plan-negative-steps*`** to NIL.

Similarly, the plan description encodes the inferential structure in a fairly straightforward way. The following validation is extracted:

```
(equality ((modus-ponens (0 (and-intro (1 2)))) 3) ⟨param...⟩)
```

The integers denote proofs of the corresponding steps.

For each goal description, a goal structure must be constructed; I went over this procedure in section 2.4.2. As I explained then, goals so constructed may already possess answers. A useful notion to introduce here is that of a *yes* answer. This is an answer which does not assign any of the goal's free variables; i.e. its substitution is empty. A *yes* answer is as general as an answer can get.

If the newly created plan has steps which have *yes* answers, these steps do not have to be further considered since they have been solved in the most general manner possible. It is useful to make a distinction between those steps which still require work from those that do not.

As discussed in section 2.3.3, a plan partitions its steps in two sets: those which are *solved*, and those which are *pending*. Steps which correspond to assertions, as well as steps which have *yes* answers are immediately assigned to the first set; all others to the second one.

2.4.4 Initializing a Plan

When a new plan has been constructed, it must be initialized. Part of this process was just described in the above, but other questions have to be answered too; such

as: *is the plan redundant? Do all of its steps already have satisfactory answers? If yes, can these answers be combined, and a conclusion be drawn right away?*

Checking for Redundancy

The idea here is to avoid inserting a new plan in the graph if it is not going to help. A plan is considered redundant if its projected conclusion is an instance of an answer already recorded in its superclass. For example, if the plan's projected conclusion is $q(f(f(u)))$ and its superclass already has an answer that concludes $q(f(z))$, there is no point in pursuing this plan since it would only derive an instance of a known answer.[9]

Another possibility is that there exists another plan for the same class, or a related class, and which subsumes the new plan in some sense which I won't detail here. The idea is that, if there is some other plan which will do the same job, or better, there is no point in adding a new one. The issue, however, is a little more subtle than it looks at first because, for practical reasons (such as meeting with the user's expectations about progress—concerning successor plans, for instance), it may be desirable to allow some amount of redundancy.

If a plan is found to be redundant, it is discarded and a *fate* is constructed summarizing this redundancy. Fates are what the user gets back after invoking a plan generator. They state what happened, hence the name "fates." For instance, some say *"a plan was derived, but was found redundant because of this answer"* or *"a plan was derived, but was found to be redundant because of this other plan"* or, usually, *"a plan was derived, and here it is."*

[9] Now that the generalization technique has been added to LOGICALC, it is possible for the answer effectively obtained to be more general than the projected conclusion. Thus, the decision to not pursue a plan because its projected conclusion is an instance of a known answer may in fact rule out a plan whose conclusion would have been generalizable into a more general solution than was available before.

Checking for Overall Solutions

The first case is when all steps have *yes* answers; i.e. there are no remaining *pending* steps. In this case, a conclusion can be immediately derived by executing the plan's validation; and the plan is discarded. A fate is returned indicating that this conclusion was derived.

The second case is when all steps have answers, but they do not all have *yes* answers. The object here is to determine if it is possible to use these existing answers to derive alternative conclusions for the plan. The difficulty is that, if we pick an answer for each step, these answers may make incompatible assignments to free variables; for instance, $x \leftarrow a$ for one and $x \leftarrow b$ for another. The system determines which combinations of answers are compatible, if any, and executes the validation for each one of them. This process is called *pushing* a plan. Fates are constructed for the resulting conclusions. There is also a fate for the plan itself, since it cannot be discarded.

Finally, if some steps do not have answers, a fate is returned that simply accounts for the new plan.

In all cases, unless the plan is discarded, it is added to the set of plans for its class.

2.4.5 Informing the User of the Results

Initializing a plan results in one or more fates. Furthermore, there may have been more than one plan to initialize: for example, the user might have specified two premises, instead of just one (namely `foo`), in the earlier example of equality substitution; the same procedure would have been followed for each one, presumably resulting in the construction of two plans, each requiring initialization, and each yielding its own set of fates. All these fates are collected and displayed to the user who must then chose one to proceed with.

In addition, for each fate that represents a plan, the user is asked whether he wants to make the plan *local*. Here is what this means: a class not only serves to store answers, but also plans. All members of a class have equal access to either. However, it would be very confusing if the display of a goal listed all the plans recorded in its class; it is better to list only those plans which the user has specifically designated as being relevant to the goal. These plans have to be imported from the class;[10] they have to be made *local* to the goal.

When the user has selected a fate, the graph editor moves the current view to the corresponding node, which is either an answer or a plan. If it is a plan and the plan has only one pending step, then, the current view is moved directly to that subgoal.

2.4.6 Following an Answer

In this section, and the next, I discuss the question of how plans become solved.

When the current view is that of a plan with one or more pending steps, the user should select one of them and proceed to solve it. When that step has acquired a satisfactory answer, the user should *follow* this answer. Following an answer moves the current view to the node representing the answer, and displays the fates associated with it. I am now going to explain where these fates come from, and what they are.

The point of following an answer is to construct a *successor* plan for the plan in which the answer's goal[11] is a step. The successor is a copy of the plan, with the following difference: the step in question is removed from the list of *pending* steps and its answer is inserted in the list of *solved* steps; furthermore, the remaining pending steps are instantiated according to the answer's substitution.

[10] The command `^^plan` can be used to import a plan from a class, while the command `-plan` can be used to drop a local plan (it still remains available in the class, and can be imported again later).

[11] The goal which the answer is an answer to.

Since a step may have more than one answer, and the user is at liberty to follow any number of them, a plan may have more than one successor.

The new successor plan must be initialized in the manner described in section 2.4.4, thus resulting in the collection of a set of fates. These fates are then recorded with the answer; following the same answer again will not cause them to be recomputed, instead the system will use the cached values.

2.4.7 Concluding From a Plan

Following an answer to a plan's last pending step causes the plan's validation to be executed and an answer to the plan's class to be constructed. Here is how it works: we have a set of answers—one for each step—and a validation which is an expression describing how to infer the plan's conclusion from these answers. A validation has the form:

(*inference-rule* (*premises...*) (*parameters...*))

where a *premise* is either a step number, and denotes the proof of the corresponding step (available through the answer), or is a validation, and must be recursively interpreted. Interpreting such an expression yields a proof: first, each premise is evaluated and yields a proof. Then, the *inference-rule* is called with two arguments: the list of these proofs, and the *parameters*. It returns a proof.

Now, the system must decide whether this proof provides an answer which is worth recording in the class. If the new conclusion is an instance of an existing answer, then it is redundant and is discarded; a fate is returned that mentions the subsuming answer.

Otherwise, the proof is packaged as an answer and recorded in the class. Furthermore, it is broadcast to all related classes through the system of links introduced in section 2.3.5.

2.4.8 Creating a New Proof

New proofs can only be created by inference rules. More precisely, an inference rule is a function which takes premises and parameters as input and returns a *description* of a conclusion as output. This description is then *generalized* by the system.

The point of this generalization is to replace as many terms as possible with new free variables while preserving validity. For example, the tautology $p(f(a), b) \supset p(f(a), b)$ will be generalized into $p(x, y) \supset p(x, y)$ (the greater gains come from generalizing over discharged assumptions). The technique used for this purpose is an extension of Explanation-Based Generalization: a variabilized skeleton of the conclusion (i.e. a formula which has the same propositional shape, but in which all terms have been replaced by new free variables) is regressed through the recorded inference tree.

The system then checks if there exists a proof whose conclusion is an (alphabetic) variant of this new conclusion. If such is the case, the old proof is returned—this technique is known as *uniquifying* proofs—otherwise a new proof is constructed to package the conclusion together with the inference tree of its derivation.

If a new proof is constructed, it is also inserted in the database. More precisely, if the proof corresponds to the sequent $A \Rightarrow p$, then the formula p is given a new arbitrary name and is added to the datapool of assumption set A. Thus new conclusions become assertions and can be used as premises in subsequent refinements.

2.5 Exceptions to Skolemization

The decision to apply skolemization to assertions and goals is only the default policy. This policy may be explicitly revoked and reinstated at any time to affect subsequent behavior—past decisions are *not* reconsidered. Plan generators are available for *quantification* and *skolemization*.

I will now discuss a situation where skolemization is unable to remove certain quantifiers from a formula. The situation arises with the use of λ-expressions.

2.5.1 Lambda Expressions

The language includes the notion of λ-expressions. A λ-expression is a term of the form:

$$\texttt{(LAMBDA (}x_1 \ \ldots \ x_n\texttt{)} \ t[x_1 \ldots x_n]\texttt{)}$$

and denotes either an anonymous function if t is a term, or an anonymous predicate if t is a proposition. The language reserves the operator % to represent application of a λ-expression to arguments. The operator % takes an indefinite number of arguments (> 1). The first argument is supposed to denote a function (or a predicate), and the following arguments are the parameters the function is to be applied to. For example:

$$\texttt{(\% (LAMBDA (}x_1 \ \ldots \ x_n\texttt{)} \ t[x_1 \ldots x_n]\texttt{)} \ a_1 \ \ldots \ a_n\texttt{)}$$

If the 1st argument is n-ary and there are exactly n parameters, then it is possible to use the rule of β-reduction and replace the % term with the term (or proposition) that results from applying the function (or predicate) to the parameters. For instance, β-reducing the above example yields $t[a_1 \ldots a_n]$.

The unifier does not attempt higher-order unification [Hue72, Hue75, Pie73]; rather, it limits itself to the straightforward extension to the classical unification algorithm [see section 2.6]. This is a restriction in the implementation, but not one of principle.

The introduction of λ-expressions in the language was motivated by practical considerations. For example, for reasoning by induction it is often convenient to have a second-order axiom expressing Mathematical Induction which we can instantiate at will to derive specific first-order induction formulae.

This dissertation does not address the issues raised by the formal treatment of λ-expressions. In particular, all the classical problems remain. See Chapter 3.

2.5.2 Skolemization of Lambda Expressions

Since an anonymous predicate can be expressed as a λ-expression, the body of said expression may contain quantified formulae. Skolemization, however, is unable to remove the corresponding quantifiers: to remove a quantifier, one must determine whether it has universal or existential force in order to decide whether the quantified variables should be replaced by free variables or skolem terms. This determination cannot be made for the body of a λ-expression because it cannot be ascribed a polarity. Consider the formula:

```
(Q (LAMBDA (X) (EXISTS (Y) (< X Y))))
```

The λ-expression appears as an argument to the (second-order) predicate `Q`. It is not possible to decide on a polarity for the body of this expression. If you feel tempted to say *'positive'*, then consider that `Q` might be equivalent to $\lambda p\, \lambda y\, \neg p(y)$.

A λ-expression that appears as an argument to a (higher-order) predicate cannot be skolemized. Skolemization must be postponed until after β-reduction.

I shall now illustrate this discussion with an example of mathematical induction. In standard notation, the second-order axiom expressing mathematical induction is:

$$(\forall P : \mathbb{N} \mapsto \text{prop})\,[P(0) \wedge (\forall n \in \mathbb{N})\, P(n) \supset P(n+1)] \supset (\forall m \in \mathbb{N})\, P(m)$$

In LOGICALC, it is written thus:

```
(FORALL (P - (PRD [integer]))
  (IF (AND (% P 0)
           (FORALL (N - integer)
             (IF (% P N) (% P (+ N 1)))))
      (FORALL (M - integer) (% P M))))
```

Note that $P(m)$ has positive polarity, while $P(0)$ and $P(n+1)$ have negative polarity because they appear in the *antecedent* of the implication. $P(n)$ is nested in the antecedent of two implications: the effect of the two implicit negations cancel each other, and $P(n)$ has positive polarity. Clearly, the λ-expression to be substituted in for P cannot be skolemized in advance.

After skolemization, the formula expressing Mathematical Induction becomes:

```
(IF (AND (IS (PRD [integer]) ?P)
         (% ?P 0)
         (IF (AND (IS integer !.N(?P))
                  (% ?P !.N(?P)))
             (% ?P (+ !.N(?P) 1))))
    (IF (IS integer ?M) (% ?P ?M)))
```

Suppose now that we wish to produce an instance of this formula by plugging the following λ-expression for the free variable ?P:

```
(LAMBDA (X - integer) (EXISTS ((< ?X ??Y) - integer)))
```

After carrying out the substitution, β-reducing the redexes, and skolemizing, we obtain:

```
(IF (AND (IS (PRD [integer])
             (LAMBDA (X)
               (AND (IS integer X)
                    (EXISTS (Y) (AND (IS integer Y) (< X Y))))))
         (AND (IS integer 0) (AND (IS integer ?Y.1) (< 0 ?Y.1)))
         (IF (AND (IS integer !.N)
                  (AND (IS integer !.N)
                       (AND (IS integer !.Y[1]) (< !.N !.Y[1]))))
             (AND (IS integer (+ !.N 1))
                  (AND (IS integer ?Y.2) (< (+ !.N 1) ?Y.2)))))
    (IF (IS integer ?M)
        (AND (IS integer ?M)
             (AND (IS integer !.Y[2](?M)) (< ?M !.Y[2](?M))))))
```

2.6 Extensions to Unification

The motivation for skolemizing formulae by default was to allow unification as a fundamental operation. Unification is a very powerful, and, I claim, natural operation [but see below]. It frees the user from having to decide on, and apply, substitutions, and facilitates search through the database.

Some would argue that the details of unification are often too complex to be carried out by people, and therefore would object to unification being called a "natural" operation; after all, where is the naturalness in it if people can't do it? This objection, however, misses the point: it is not the "carrying out" which is predicated to be natural, but rather the principle of the operation. People can easily identify "similar" patterns; based on this presumption of similarity, they may then request that the machine carry out unification; the result, assuming the operation succeeds, makes sense to people in exactly the right ways: (1) it confirms their expectations that those things indeed match, (2) it fills in the blanks, and (3) they don't have to figure out the details.

There are two topics which I would like to dicuss briefly here: first, the extensions to unification necessary to handle quantified formulae and λ-expressions, and second, the notion of segment variables.

2.6.1 Quantified Formulae and Lambda Expressions

We have seen that, eventhough the default policy is to skolemize all formulae, there may be circumstances when it is either necessary or desirable to manipulate explicitly quantified formulae. It would be convenient if, when confronted with quantified formulae, we did not suddenly lose all the advantages of unification. For example, $\forall x\, p(x)$ and $\forall y\, p(y)$ should unify—in the sense that the operation should succeed, eventhough the two expressions being unified do not become identical in the process.

Similarly for λ-expressions: $\lambda x\, f(x)$ and $\lambda y\, f(y)$ should unify. In fact unifying λ-expressions is no different from unifying quantified formulae; both are instances of the more general idea of unifying expressions of the form:

(binder (variables...) expression)

The algorithm I adopted is a straightforward extension of Robinson's unification procedure rather than a higher-order approach [Pie73, Hue72, Hue75], and can be interpreted as the unification of the corresponding de Bruijn formulae [dB72], with the restriction (to preserve soundness) that free variables can only be assigned well-formed de Bruijn terms (i.e. closed).

2.6.2 Segment Variables

In PROLOG, it is possible to denote the *tail* of a list using the vertical bar notation. For example, the term `[X|L]` denotes a list with `X` as its first element and `L` as its tail

(or remainder). In LOGICALC, the corresponding idiom is `[?X !?L]`. The term `!?L` is called a *segment variable*, which is a generalization of a *tail* variable: it stands for an unspecified number of adjacent terms. A segment variable is not restricted to appear at the end of a list. For example, here is how the `member` predicate might be defined:

```
(DEFPRED (MEMBER ?X - obj ?L - (LST obj))
  (MEMBER ?X [!?L1 ?X !?L2]))
```

The above formula expresses the idea that `?X` is an element of a list if said list is of the form `?X`, preceded by a certain number of elements (`!?L1`), and followed by a certain number of elements (`!?L2`).

Of course, the astute reader will have noticed by now that a single unification may produce more than one solution. Consider unifying `[A B C D]` with `[!?L1 ?X !?L2]`; 4 solutions result from this attempt:

```
{?L1 ← []     , ?X ← A, ?L2 ← [B C D]}
{?L1 ← [A]    , ?X ← B, ?L2 ← [C D]  }
{?L1 ← [A B]  , ?X ← C, ?L2 ← [D]    }
{?L1 ← [A B C], ?X ← D, ?L2 ← []     }
```

From this remark it follows that, in LOGICALC, we can no longer retain the notion of a most general unifier; rather, unification returns a list of unifiers (i.e. substitutions). Unfortunately, it is also unreasonable to expect unification to return a *complete* set of unifiers. Consider unifying `[!?X A]` and `[A !?X]`. The problem is to find a substitution which makes identical the two sides of the following equation:

$$\texttt{[!?X A]} = \texttt{[A !?X]} \qquad (\Phi)$$

Clearly there are 2 cases:

- Either `?X` = `[]`, in which case (Φ) simplifies to `[A]` = `[A]`.

- Or `?X` must begin with `A`, i.e. be of the form `[A !?Y]`. By substitution, (Φ) becomes `[A !?Y A]` = `[A A !?Y]`, which simplifies to `[!?Y A]` = `[A !?Y]`. In other words, `?Y` is defined by the same equation (Φ) as `?X`, using the name `Y` instead of `X`.

From these remarks we may conclude that the set of unifiers in this example can be recursively enumerated: {`?X` = `[]`}, {`?X` = `[A]`}, {`?X` = `[A A]`} ... Since this set is infinite, it is probably not a good idea to try and compute it all; instead, LOGICALC simply returns the first two elements (i.e. `?X` = `[]` and `?Y` = `[]`).

2.7 Database Management

2.7.1 Indexation and Retrieval

Simply adding a new node to the ATMS is not sufficient; we also need means to query, or, more generally, search the database. To support this capability efficiently, it is necessary to cleverly index the contents of the database.

Indexing was mentioned earlier. I stated then that, in order to speed up access to the database, the simplest technique is to file each formula under its principal symbol, e.g. `(p a)` under `p`. When presented with a query `(p ...)`, the system doesn't have to search the whole database, but can lookup the relevant formulae indexed under `p`.

Indexing can be and usually is done somewhat more cleverly than the simple-minded one-level technique suggested above. For example, PROLOG systems often use a two-level indexing scheme: the first level of indexing is done using ⟨*predicate name/arity*⟩ as a key, while the second level might be based on the nature of the first argument—whether it is a constant, a list, a functional term, or a variable, say.

In LOGICALC, indexing is done using a discrimination tree. The simple-minded technique presented earlier corresponds to a discrimination only 1 level deep; the PROLOG technique to a discrimination 2 levels deep. In LOGICALC, we can discriminate to levels as deep as required by considerations of specificity against space/time efficiency. The current implementation allows multiple indexing, i.e. it is possible to index the same piece of data in more than one way. It is self-reorganizing: if discrimination according to a particular key turns out to not work very well at a certain point in the tree, the system will automatically select another key that works better; thus, whereas PROLOG may be stuck using a predicate's 1st argument as a secondary key, LOGICALC might pick another argument when it becomes obvious that the 1st one results in poor discrimination. Also, our implementation is data-driven so that the objects themselves (e.g. syntactic constructs) may dictate how they wish to be indexed.

Simply adding a new node to the graph managed by the reason maintenance system isn't enough. Convenient and efficient means of retrieval have to be provided as well. To this end, several indexation schemes are maintained concurrently.

The principal means of access to the database is through DUCK's multiply-indexed discrimination tree. I will not go into the details here (but see e.g. [McD85]). The general idea is that one can use `FETCH` to retrieve all solutions in the current datapool that unify with the argument pattern. `FETCH` will also return the solutions that can be derived by backward-chaining rules [Ibid.].

It is often useful in theorem proving to be able to search the database for assertions containing occurrences of a specific formula that is of interest. In particular, I will later introduce the idea of detaching formulae from assertions (this idea is related to non-clausal resolution). For an efficient implementation of this idea it is crucial that we be able to find out quickly all the assertions a particular formula occurs in.

The naïve way to achieve this sort of cross-indexation (and indeed this was how the first implementation worked) is to produce additional assertions of the form:

(LOGICALC_INDEX *formula assertion sign*)

where *formula* is a subformula occuring in *assertion* (which is the name of a node) with the corresponding *sign*. Thus, finding candidate assertions from which to detach a given formula is simply a matter of calling FETCH with the appropriate LOGICALC_INDEX pattern. Of course this wastes a lot of memory because each subformula has to be copied into its own LOGICALC_INDEX assertion.

The current scheme is a simplification of the above. Instead of the first argument to LOGICALC_INDEX being a formula, it is now the principal symbol in the formula (i.e. connective or predicate). This scheme is a lot more economical in terms of memory.

The interface is implemented by function CONSULT-INDEX. It is given a pattern as argument and returns a list of assertions that may possibly detach the given pattern (i.e. contain a subformula that unifies with the pattern modulo its sign). Initial selection is done according to the pattern's principal symbol. Then filtering is performed based on the connectives and predicates naturally occurring in the pattern; those candidates that do not also mention them are discarded (modulo some care).

The original scheme is more specific because the pattern must unify with the recorded formula, however, not only does it waste storage, it also wastes time and memory performing unification, the result of which will be discarded anyway since "detaching" has to start again from scratch. The current scheme performs no other unification than those implicit in the retrieval of LOGICALC_INDEX assertions (which are flat). Yet, experience shows that the filtering does about just as well as unification would, at a much lesser cost, and is much faster. Since "detaching" is subsequently applied, the occasional remaining false positives are then identified when unification is attempted.

There is another indexing scheme which is concerned with skolem terms. Experience showed that it is very useful to be able to determine what assertion a given skolem term came from (presumably through unification). The reason this is useful is that

if you want to prove something about a particular skolem term, the most likely place to find that information is in the assertion where the skolem term was defined. For this purpose, I maintain assertions of the form:

`(INDEX->SKOLEM` *id assertion*`)`

where *id* is this integer that uniquely identifies the skolem term (or rather the corresponding bound variable), and *assertion* is the name of the assertion in which it is mentioned. It is by this method that the `-SKOLEM` option [Chapters 1 and 6] is supported.

2.7.2 Local Assumptions and Datapools

In a system based on natural deduction, in order to prove a goal of the form $A \Rightarrow p \supset q$ one would ordinarily invoke the Deduction Theorem and attempt a proof of $A; p \Rightarrow q$. In the later sequent, we interpret p as a local assumption. If q is itself an implication, the process may be repeated resulting in an additional assumption.

In the course of a proof, hierarchical sets of assumptions will thus naturally arise. Therefore a method is needed to efficiently represent multiple assumption sets organized hierarchically. This is precisely what datapools have to offer [McD79,McD85].

Beads and Datapools

Datapools allow multiple sets of assumptions, and the conclusions derived from them, to coexist coherently and consistently in a global database managed by an ATMS. To interact with the database, a current view must be selected which specifies which of these sets will be presently assumed; the ATMS then labels as TRUE (or rather IN

according to traditional RMS parlance) all formulae which belong to these sets or are known to be supported by (e.g. because they were derived from) formulae in these sets.

The trick is accomplished with the aid of *beads*. To simplify in the extreme, a bead can be thought of as a distinguished mark. A formula can be marked with a bead. The bead can be selected or not. When the bead is selected, all formulae marked with this bead are labelled as TRUE. In other words, a bead can also be thought of as a set of assumptions: when the bead is selected, the corresponding set is assumed.

A datapool is essentially a set of beads. When the datapool is selected, all the beads in its set are selected as well. Such an approach makes it very easy to implement hierarchical sets of assumptions: a new datapool is created by making up a new bead and adjoining it to the set of beads inherited from the parent datapool. The new bead created on this occasion is called the datapool's characteristic bead. All assumptions specific to the new datapool will be marked with this characteristic bead and will be labelled TRUE only when said bead is selected, i.e. when this datapool or one of its descendent is selected.

The complete theory of beads and datapools is much more general since it allows deletions as well as additions, and supports non-monotonic dependencies. It is fully documented in [McD79].

Assumption Sets and Datapools

Goals and proofs are represented by sequent-like objects of the form $A \Rightarrow p$ where A is an assumption set uniquely associated with its own datapool. This approach was very attractive because it permitted a natural and painless retrofitting of sequents into a framework we were familar with, using tools readily available in DUCK.

As I mentioned earlier, during the course of a proof, a natural hierarchy of assumption sets will arise, e.g. when the goal $A \Rightarrow p \supset q$ is refined into $A; p \Rightarrow q$. The datapool for $A; p$ has the datapool for A as its parent, i.e. it inherits all the assumptions (and conclusions) in A. However, p is marked with the characteristic bead of $A; p$.

Assumptions and Free Variables

The scheme outlined above works like a charm when the assumption p is ground. Unfortunately, it becomes unsound when p contains free variables. I will now show how the problem manifests itself in practice, and I will then present two possible techniques to preserve soundness. LOGICALC does not allow unsound refinements, and automatically resorts to a policy that combines the two aforementioned techniques.

When a goal sequent is of the form $A \Rightarrow p(x) \supset q$ where x is a free variable, it is not legal to "move" $p(x)$ to the assumption set and attempt a proof of $A; p(x) \Rightarrow q$. The reason is that moving $p(x)$ to the assumption set really means to assert $p(x)$ in a local datapool and we lose the notion that the variable x was somehow connected to the goal; this happens because "fetching" from the assumption set conceptually involves making copies (in particular renaming free variables) of those assertions which are extracted.[12]

Let me illustrate this with a example. Suppose you wanted to prove the following proposition:

```
(EXISTS (X) (IF (P X) (P A)))
```

You would then attempt to prove the skolemized version `(IF (P ?X) (P A))`. If you were allowed to simply move `(P ?X)` to the assumption set, you would now have

[12]Another way to look at the problem is to realize that, in a goal, a free variable represents an existentially quantified variable, whereas, in an assumption set, it represents a universally quantified variable.

a local assumption which can be unified with the remaining goal `(P A)`, and you would correctly conclude that you have a proof that `(P A)` follows from this set of assumptions. However, after discharging the local assumption, you would erroneously conclude that you derived a proof of `(IF (P ?X) (P A))`. Clearly this conclusion is wrong since substituting `?X` with `B` yields `(IF (P B) (P A))`, which is not a valid formula.

Making a local assumption that contains free variables is therefore disallowed in LOGICALC. When it is desirable to move to the assumption set a proposition that contains free variables, 2 options are available: (1) either instantiate the variables in question with ground terms, or (2) capture those variables by quantification using a preliminary refinement step reintroduing a quantifier—e.g. replace `(if (p ?x) (p a))` with `(if (forall (x) (p x)) (p a))`. The user may either perform such operations manually, or let the system resort to its default policy which is to instantiate those variables shared by the antecedent (p) and the consequent (q) with arbitrary new constants (in the form of skolem terms), and capture by quantification the remaining free variables of the antecedent. When this preliminary procedure has been carried out (either manually by the user, or automatically by the system), the resulting antecedent no longer contains free variables and can be safely moved to the assumption set.

Chapter 3

Logical System

In this chapter I describe LOGICALC's axiomatic structure. First I recapitulate the classical formulation of $\mathcal{F}^{=}$. Then I discuss each inference rule, showing how it is either a primitive or a derived inference rule in the classical formulation (thus establishing soundness). I introduce the notion of "skolemizing axioms" to support skolemization and quantification as inference rules, and show how my system captures the axiomatic structure of the classical formulation (thus establishing completeness). Finally, I discuss the treatment of λ-expressions, and also unification and substitution.

3.1 Introduction

LOGICALC is typically used as a natural-deduction proof system, which makes the idea of a sequent-based logic [Gen35] very attractive because the notation explicitly captures the notion of a conclusion following from a set of assumptions. A sequent is written $A \Rightarrow p$, where A is a set of assumptions and p is a formula; the sequent is "true" iff p logically follows from A.

There is perhaps greater elegance in the symmetric formulation of sequents [Smu68] where both sides are taken to be sets of formulae: in $U \Rightarrow V$, U plays the role of a conjunction and V of a disjunction. However, LOGICALC can also be used to derive conclusions in a forward manner; and, whereas is easy to interpret p as the conclusion in $A \Rightarrow p$, it is less clear what meaning should be ascribed to V in the symmetric formulation $U \Rightarrow V$.

I will generally write $A, p \Rightarrow q$ to mean that q can be derived from an assumption set containing the formula p. I will occasionally write $A; p \Rightarrow q$ to mean more specifically that q can be derived from an assumption set whose "local assumption" is p. The notion of a local assumption arises from the following theorem-proving technique: when attempting to prove $p \supset q$, assume p and try to prove q under this additional assumption; i.e. given the goal of proving $A \Rightarrow p \supset q$, try to show $A; p \Rightarrow q$ instead. This technique is justified by the Deduction Theorem.

The reader may wonder why I make this distinction between an assumption and a "local" assumption. The reason is that, because of the way assumption sets are implemented [see chapter 5], only "local" assumptions may be discharged.

An inference rule prescribes what conclusion may be derived from given premises. For instance, Modus Ponens states that, from the premises $A \Rightarrow p \supset q$ and $A \Rightarrow p$, one may validly conclude $A \Rightarrow q$. Therefore, an inference rule may be viewed as a procedure taking sequent premises as input and returning a conclusion sequent. LOGICALC generalizes this notion and implements inference rules as functions taking a list of premises *and* a list of parameters, and returning a list of conclusions [more about this later]. Parameters serve to further specify the particular inference to be performed. For instance, Equality Substitution requires a "path" to identify the subterm to be replaced.

In the classical formulation of Modus Ponens, i.e. from $p \supset q$ and p infer q, the implicit condition is that p in the major premise and p in the minor premise are syntactically

identical. LOGICALC is more powerful because it relies on unification rather than syntactic identification. Therefore, when I state that one may infer $A \Rightarrow q$ from $A \Rightarrow p \supset q$ and $A \Rightarrow p$, I really mean that the p in the major premise and the minor premise are unifiable by θ, and that the q in the conclusion is really θq. Stating this explicitly each time would make the exposition difficult to read. Furthermore, I am going to start out by assuming that syntactic identification is used, so that the relationships with the classical formulation will be simpler to establish. Later, in section 3.10, I will argue that substituting unification for syntactic identification only makes the system more powerful as a tool, but does not change the logic.

LOGICALC also relaxes the process of identification on the side of the assumption set. In the example above, both major and minor premises were shown as having the same assumption set A. The actual requirement is that the assumption sets of the premises may be ordered by hierarchical inclusion. The assumption set selected for the conclusion is the one that makes the most assumptions. For example, from $A \Rightarrow p \supset q$ and $A; r \Rightarrow p$, Modus Ponens will infer $A; r \Rightarrow q$. Once again, this will not be explicitly stated in the following exposition because (1) doing so would obscure the discussion, and (2) this trick is by no means necessary; just like unification, it simply enhances the power of the system considerably from the user's point of view. The only exception to this rule is when a "local" assumption is explicitly mentioned, e.g. $A; p \Rightarrow q$, in which case the assumption set for this particular premise is not allowed to be stronger; it must have p as its local assumption.

3.2 Classical Formulation

I am going to start out by giving a brief account of a classical formulation of first-order logic with equality. For this purpose I chose the $\mathcal{F}^{=}$ system of [And86]. All the notions and results mentioned in this section will subsequently prove useful when I formally capture and justify LOGICALC's logic.

3.2.1 The Language of $\mathcal{F}^=$

First, we need to define the primitive symbols of $\mathcal{F}$ and specify the formation rules for terms and well-formed formulae (henceforth *wffs*). The primitive symbols of $\mathcal{F}$ are:

1. Improper symbols:[1] ( ) [ ] $\neg$ $\vee$ $\forall$.
2. Individual variables: $u\ v\ w\ x\ y\ z\ u_1\ v_1\ w_1\ \ldots\ u_2\ \ldots$
3. n-ary function variables: $f^n\ g^n\ h^n\ f_1^n\ g_1^n\ h_1^n \ldots$ for each natural number $n \geq 1$.
4. Propositional variables: $p\ q\ r\ s\ p_1\ q_1\ r_1\ s_1\ \ldots$
5. n-ary predicate variables: $P^n\ Q^n\ R^n\ S^n\ P_1^n\ Q_1^n\ R_1^n\ S_1^n \ldots$ for each natural number $n \geq 1$.

There are denumerably many variables of each type. There may also be finitely or infinitely many individual, function, propositional, or predicate constants. Terms and *wffs* are defined inductively as follows:

1. Each individual variable or constant is a term.
2. If $t_1 \ldots t_n$ are terms and f^n is a n-ary function variable or constant, then $f^n(t_1, \ldots, t_n)$ is a term.
3. If $t_1 \ldots t_n$ are terms and P^n is a n-ary predicate variable or constant, then $P^n(t_1, \ldots, t_n)$ is a *wff*.
4. If **A** is a *wff*, so is $\neg$**A**.

[1] Brackets are only required to disambiguate the "textual" representation of certain formulae. If we agree that the actual representation of a formula is not a linear text string but a tree (e.g. a LISP s-expression), then brackets are only useful to disambiguate the representation of these trees in our textual metalanguage.

5. If **A** and **B** are *wffs*, so is $[\mathbf{A} \vee \mathbf{B}]$.

6. If **B** is a *wff* and x is an individual variable, then $\forall x\, \mathbf{B}$ is a *wff*.

In LOGICALC, free variables are denoted by expressions of the form $?X$; also, if P^n is a predicate variable, then $P^n(t_1, \ldots, t_n)$ will be written $(\%\ ?P^n\ t_1\ \ldots\ t_n)$ (similarly for functional variables), while predicate constants will follow the normal LISP convention. I will ignore these distinctions for the time being.

It is customary to let $[\mathbf{A} \wedge \mathbf{B}]$ stand for $\neg[\neg\mathbf{A} \vee \neg\mathbf{B}]$, $[\mathbf{A} \supset \mathbf{B}]$ for $[\neg\mathbf{A} \vee \mathbf{B}]$, $[\mathbf{A} \equiv \mathbf{B}]$ for $[[\mathbf{A} \supset \mathbf{B}] \wedge [\mathbf{B} \supset \mathbf{A}]]$, and $\exists x\, \mathbf{B}$ for $\neg\forall x\, \neg\mathbf{B}$. In LOGICALC, only '$\equiv$' is actually implemented as an abbreviation.

3.2.2 Substitutions

If $x_1\ \ldots\ x_n$ are individual variables and $t_1\ \ldots\ t_n$ are terms, then $\mathsf{S}^{x_1 \ldots x_n}_{t_1 \ldots t_n}\mathbf{A}$ (resp. $\underset{\cdot}{\mathsf{S}}^{x_1 \ldots x_n}_{t_1 \ldots t_n}\mathbf{A}$) denotes the result of simultaneously substituting t_i for all (resp. all free) occurrences of x_i in **A** for $1 \leq i \leq n$.

I will often write θ to denote a substitution, and $\theta\mathbf{A}$ to represent the application of this substitution to **A**. Also $\mathbf{A}[x]$ will stand for a *wff* with 0 or more free occurrences of the individual variable x; subsequently, $\mathbf{A}[t]$ will represent the *wff* obtained by substituting t for the free occurrences of x in $\mathbf{A}[x]$.

It is occasionally useful to generalize the notation and write $\underset{\cdot}{\mathsf{S}}^{\mathbf{A}}_{\mathbf{B}}\mathbf{C}$ to denote the *wff* obtained from **C** by substituting free occurrences of **A** with **B** such that the free variables of **B** are not captured. Also, sometimes we want to substitute some occurrences (not necessarily all of them), in which case I shall write e.g. $\overline{\underset{\cdot}{\mathsf{S}}}^{\mathbf{A}}_{\mathbf{B}}\mathbf{C}$; in LOGICALC, the particular occurrences to be substituted will have to be specified by parameters.

3.2.3 Axiomatic Structure of $\mathcal{F}^=$

Modus Ponens: From $\mathbf{A}$ and $\mathbf{A} \supset \mathbf{B}$ infer $\mathbf{B}$.

Generalization: From $\mathbf{A}$ infer $\forall x\, \mathbf{A}$, where x is any individual variable.

Axiom of Reflexivity: $x = x$

Axiom Schemata:

1. $\mathbf{A} \vee \mathbf{A} \supset \mathbf{A}$

2. $\mathbf{A} \supset {\scriptstyle\blacksquare}\mathbf{B} \vee \mathbf{A}$, where ${\scriptstyle\blacksquare}$ indicates an implicit left bracket whose matching right bracket is placed as far to the right as is consistent with explicit brackets; often, this means: to the end of the formula.

3. $\mathbf{A} \supset \mathbf{B} \supset {\scriptstyle\blacksquare}\mathbf{C} \vee \mathbf{A} \supset {\scriptstyle\blacksquare}\mathbf{B} \vee \mathbf{C}$

4. $\forall x\, \mathbf{A} \supset \mathsf{S}^x_t \mathbf{A}$ where t is a term free for the individual variable x in $\mathbf{A}$.[2]

5. $\forall x[\mathbf{A} \vee \mathbf{B}] \supset {\scriptstyle\blacksquare}\mathbf{A} \vee \forall x\, \mathbf{B}$ provided that x is not free in $\mathbf{A}$.

6. $x = y \supset {\scriptstyle\blacksquare}\mathsf{S}^z_x \mathbf{A} \supset \mathsf{S}^z_y \mathbf{A}$ where $\mathbf{A}$ is an atomic formula.

3.2.4 Proofs

Here again, I shall follow Andrews:

A proof of a *wff* $\mathbf{B}$ from the hypotheses $\mathbf{A}_1 \ldots \mathbf{A}_n$ is a finite sequence $\mathbf{B}_1 \ldots \mathbf{B}_m$ of *wffs* such that $\mathbf{B}_m$ is $\mathbf{B}$ and for each $1 \leq i \leq m$ one of the following conditions is satisfied:

[2] t is free for x in $\mathbf{A}$ means that no free variable of t is captured by some quantifier in $\mathbf{A}$ when the substitution is performed: if y is a free variable of t, there is no free occurrence of x in $\mathbf{A}$ which is in the scope of a $\forall y$. LOGICALC automatically handles this problem by effecting an Alphabetic Change of Bound Variables whenever required [see later].

1. $\mathbf{B}_i$ is an axiom.

2. $\mathbf{B}_i$ is an hypothesis $\mathbf{A}_j$.

3. $\mathbf{B}_i$ follows by Modus Ponens from preceding members of the sequence.

4. $\mathbf{B}_i$ follows by Generalization from a preceding member of the sequence. The individual variable generalized upon must not be free in any hypothesis.

5. $\mathbf{B}_i$ follows from some preceding member of the sequence by the Rule of Alphabetic Change of Bound Variable [see later].

I will write $\mathbf{A}_1 \ldots \mathbf{A}_n \vdash \mathbf{B}$ to indicate that $\mathbf{B}$ has a proof (a derivation) from hypotheses $\mathbf{A}_1 \ldots \mathbf{A}_n$.

3.2.5 Major Results

The most important result is that $\mathcal{F}^=$ is sound and complete, i.e. every theorem of $\mathcal{F}^=$ is valid, and every valid formula of $\mathcal{L}(\mathcal{F}^=)$ is a theorem of $\mathcal{F}^=$: the set of theorems and valid formulae coincide. A formula is a theorem if it has a proof. A formula is valid if it is true in every model of $\mathcal{F}^=$. Since I will examine LOGICALC's system by comparing it with Andrews' $\mathcal{F}^=$ system, I will not need the notion of validity for the following discussion and will say no more about it. The interested reader should consult any textbook on Mathematical Logic such as [And86].

By showing that LOGICALC's rules are derived rules of inference in $\mathcal{F}^=$ and therefore preserve validity, I will show that LOGICALC is sound. Also, by showing how LOGICALC captures the axiomatic structure of $\mathcal{F}^=$ and therefore can produce all of $\mathcal{F}^=$'s theorems, I will establish completeness.

But first, here is a list of derived rules of inference for $\mathcal{F}^=$:

Extended Rule P: If $H \vdash \mathbf{A}_1 \ldots H \vdash \mathbf{A}_n$ and if $[\mathbf{A}_1 \wedge \ldots \wedge \mathbf{A}_n \supset \mathbf{B}]$ is tautologous, then $H \vdash \mathbf{B}$.

Substitutivity of Implication: If $\vdash \mathbf{A} \supset \mathbf{B}$ then $\vdash \mathbf{C} \supset \S^{\mathbf{A}}_{\mathbf{B}}\mathbf{C}$ for positive occurrences of $\mathbf{A}$ in $\mathbf{C}$, and $\vdash \S^{\mathbf{A}}_{\mathbf{B}}\mathbf{C} \supset \mathbf{C}$ for negative occurrences.

Substitutivity of Equivalence: Similarly, if $\vdash \mathbf{A} \equiv \mathbf{B}$ and $\vdash \mathbf{C}$, then $\vdash \S^{\mathbf{A}}_{\mathbf{B}}\mathbf{C}$.

Rule of Alphabetic Change of Bound Variables: If $\vdash \mathbf{C}$ then $\vdash \S^{\forall x \mathbf{A}}_{\forall y \S^{x}_{y} \mathbf{A}}\mathbf{C}$ provided y is not free in $\mathbf{A}$ and is free for x in $\mathbf{A}$ (i.e. doesn't get captured).

Universal Instantiation: If $H \vdash \forall x\, \mathbf{A}$ then $H \vdash \S^{x}_{t}\mathbf{A}$ provided t is a term free for x in $\mathbf{A}$.

Deduction Theorem: if $H, \mathbf{A} \vdash \mathbf{B}$ then $H \vdash \mathbf{A} \supset \mathbf{B}$.

Rule of Substitution: If $H \vdash \mathbf{A}$, and x is an individual (resp. propositional) variable, and t is a term (resp. a *wff*), then $H \vdash \S^{x}_{t}\mathbf{A}$ provided x doesn't occur free in H and t is free for x in $\mathbf{A}$.

Rule of Existential Generalization: If $H \vdash \mathbf{A}$, then $H \vdash \exists x\, \S^{t}_{x}\mathbf{A}$ provided x isn't free in $\mathbf{A}$ and t is free for x in $\mathbf{A}$.

Rule of Cases: If $H \vdash \mathbf{A} \vee \mathbf{B}$ and $H, \mathbf{A} \vdash \mathbf{C}$ and $H, \mathbf{B} \vdash \mathbf{C}$ then $H \vdash \mathbf{C}$.

Indirect Proof: If $H, \neg\mathbf{A} \vdash \mathbf{B}$ and $H, \neg\mathbf{A} \vdash \neg\mathbf{B}$, then $H \vdash \mathbf{A}$.

Weakening: If $H_1 \vdash \mathbf{A}$ and $H_1 \subseteq H_2$, then $H_2 \vdash \mathbf{A}$.

3.3 LOGICALC's Axioms

Like every sequent system, LOGICALC's principal axiom schema is $H, p \Rightarrow p$. It is also traditional to let every tautology be an axiom: that is, if p is a tautology, $A \Rightarrow p$

is an axiom. In LOGICALC, this tradition is acknowledged by the `obvious` inference rule, but tautologies can also be derived by means of `taut-trans` [see later].

LOGICALC also possesses the Axiom of Reflexivity $\Rightarrow x = x$. From this axiom, it is possible to infer all instances of it (i.e. $\Rightarrow t = t$) by the Rule of Substitution; LOGICALC of course saves you the trouble by relying on unification in the first place.

3.4 LOGICALC's Inference Rules

I will now list all the basic inference rules available in LOGICALC. A discussion of those rules related to quantifier-handling will be postponed until later sections. For each rule, I will give the name by which it is known to the system as well as a diagrammatic representation of the corresponding inference where premises are presented above a fraction line and the conclusion lies below. I will also provide a description of purpose, a justification in terms of the classical system (thus establishing soundness). Finally, I will include a representation of the validation for the corresponding inference; this is for reference purposes only and may be safely ignored for the moment.

Identity $$\frac{A \Rightarrow p}{A \Rightarrow p}$$

The *identity* inference rule doesn't do anything interesting. Its sole purpose is to serve as a no-op place holder in certain validations.

(`identity` (*premise*) ())

Substitution $$\frac{A \Rightarrow p}{A \Rightarrow \theta p}$$

The substitution inference rule requires a substitution θ to be specified as a parameter. This rule serves little purpose since LOGICALC uses unification anyway and was intended primarily as a documenting device in a proof. However, with the implementation of proof generalization, the status of this inference

rule has become even more dubious. *Proof:* Rule of Substitution. The classical system requires that for each individual assignment t/x in θ, t be free for x in p; in LOGICALC, applying a substitution that violates this requirement will result in a preliminary alphabetic change of bound variables being effected on p [see section 3.10]. (substitution (*premise*) (θ))

Assumption-Intro

$$\frac{A \Rightarrow p}{A; q \Rightarrow p}$$

This inference rule captures the idea of weakening a proof by introducing an extraneous assumption (q in the above schema). It is used internally by the system: whenever a class receives an answer [cf. broadcasting], it attempts to "adapt" it to itself; doing so may involve weakening the proof until its assumption set matches the class's. *Proof:* by Weakening.

(assumption-intro (*premise*) (q))

And-Intro

$$\frac{\begin{array}{l} A \Rightarrow p_1 \\ \vdots \\ A \Rightarrow p_n \end{array}}{A \Rightarrow p_1 \wedge \ldots \wedge p_n}$$

From n premises, *and-intro* derives their conjunction. *Proof:* by Rule P, using $p_1 \wedge \ldots \wedge p_n \supset p_1 \wedge \ldots \wedge p_n$ as the tautology.

(and-intro (*premise*$_1$ ... *premise*$_n$) ())

If-Intro

$$\frac{A; p \Rightarrow q}{A \Rightarrow p \supset q}$$

If-intro captures the essence of the Deduction Theorem; i.e. given the premise $A; p \Rightarrow q$, it will discharge the local assumption p and conclude $A \Rightarrow p \supset q$.[3] However, it imposes the restriction that p and q may not share free variables. The reason for this is discussed later: briefly, it is because free variables in the assumption set are independent from free variables in the goal, just like

[3]In the distant past, a technical difficulty forced me to make the assumption to be discharged an explicit premise; in other words, the rule takes two premises as input: $A; p \Rightarrow p$ and $A; p \Rightarrow q$. This problem has since been resolved but the rule was not changed to remove the now-redundant 1st premise.

in PROLOG variables in the query are independent from those in the database. Thus, even when $p[x]$ and $q[x]$ both mention a free variable x, they are really talking about two distinct variables. If we were to infer $A \Rightarrow p[x] \supset q[x]$ from $A; p[x] \Rightarrow q[x]$ the two distinct free variables, both called x, would become confused. I could arbitrarily restrict the applicability of *if-intro* to only those cases where p and q do not have free variables with the same names, however it is more useful to define the rule so that it renames those free variables in p that would otherwise collide with free variables in q. Such a renaming is justified simply by the Rule of Substitution: new free variables are substituted for old ones. `(if-intro (`*assumption conclusion*`) ())`

If-Elim

$$\frac{A \Rightarrow p \supset q}{A; p \Rightarrow q}$$

If-elim implements the converse of *if-intro*. This is not terribly useful in practice, but is provided for the convenience of proof manipulation procedures: an earlier version of LOGICALC used it to perform some non trivial footwork in connection with the *case* inference rule; a better way, not involving *if-elim*, has since been designed. *Proof:* if $A \vdash p \supset q$, then also $A, p \vdash p \supset q$, and since $A, p \vdash p$, by Modus Ponens conclude $A, p \vdash q$. `(if-elim (`*implication*`) ())`

Taut-Trans

$$\frac{\begin{array}{c} p \supset q \text{ is a tautology} \\ A \Rightarrow \theta p \end{array}}{A \Rightarrow \theta q}$$

This is essentially Andrews' Rule P. The basic idea is that, if $p \supset q$ is a tautology of the propositional calculus and θp and θq are *wffs* of first-order logic, if θp is a theorem then θq is also a theorem. *Proof:* Rule P.

`(taut-trans (`*premise*`) (`*p q*`))`

Modus-Ponens

$$\frac{\begin{array}{c} A \Rightarrow p \supset q \\ A \Rightarrow p \end{array}}{A \Rightarrow q}$$

From an implication as the major premise and a proof of the implication's antecedent as a minor premise, infer the implication's consequent. *Proof:* Modus

Ponens.

`(modus-ponens` (*implication antecedent*) `())`

Modus-Tollens

$$\frac{A \Rightarrow p \supset q \qquad A \Rightarrow \neg q}{A \Rightarrow \neg p}$$

From an implication as the major premise and a proof of the negation of the implication's consequent as a minor premise, infer the negation of the implication's antecedent. *Proof:* From $\vdash p \supset q$ and the tautology $p \supset q \supset . \neg q \supset \neg p$, infer $\vdash \neg q \supset \neg p$ by Rule P, then from $\vdash \neg q$ by Modus Ponens conclude $\vdash \neg p$.

`(modus-tollens` (*implication negated_consequent*) `())`

Symmetry

$$\frac{A \Rightarrow p(x,y)}{A \Rightarrow p(y,x)}$$
provided p is symmetric

Traditionally, in order to state that a predicate is symmetric, one would have to write an axiom of the form $\forall x \forall y \, p(x,y) \equiv p(y,x)$. However, such an approach is not only verbose, it also makes manipulations that require using the property of symmetry quite tedious. In LOGICALC, the user has the option to simply give the LISP symbol naming p the property **symmetric**; this is what the *symmetry* rule checks for. Furthermore, the "detaching" procedure will automatically take advantage of such declarations [see Chapter 7 p256]. *Proof:* by Substitutivity of Equivalence.

`(symmetry` (*premise*) `())`

Not-Intro

$$\frac{A; p \Rightarrow q \qquad A; p \Rightarrow \neg q}{A \Rightarrow \neg p}$$

If assuming p leads to a contradiction, then $\neg p$ must be the case. The rule discharges assumption p and concludes its negation. *Proof:* Indirect Proof.

`(not-intro` (*premise negated_premise*) `())`

Not-Elim

$$\frac{A \Rightarrow p \qquad A \Rightarrow \neg p}{A \Rightarrow q}$$

Not-elim captures the idea that anything can be inferred from an inconsistent set of assumptions. The inconsistency is expressed by the ability to derive

both p and its negation. The rule's actual conclusion must be provided as a parameter. *Proof:* by Rule P, from tautology $p \wedge \neg p \supset q$.

`(not-elim` *(premise negated_premise)* *(q)*`)`

Equality

$$\frac{\begin{array}{l} A \Rightarrow a = b \\ A \Rightarrow p[a] \end{array}}{A \Rightarrow p[b]}$$

If $a = b$, and $p[a]$ is a theorem containing occurrences of a, then $p[b]$ (i.e. $\bar{\S}^a_b p[a]$) is also a theorem and is obtained by replacing some occurrences of a in p by b. Note that some care must be taken with bound variables and potential captures [see later]. The occurrences to be replaced must be specified with a list of "path" parameters. *Proof:* let $\mathbf{A} \cong \bar{\S}^a_z p[a]$ where z is an individual variable that does not occur in $\mathbf{A}$, by Axiom Schema 6 we have $x = y \supset . \mathrm{S}^z_x \mathbf{A} \supset \mathrm{S}^z_y \mathbf{A}$; by Substitution $\S^{xy}_{ab}$ we derive $a = b \supset . \mathrm{S}^z_a \mathbf{A} \supset \mathrm{S}^z_b \mathbf{A}$; by Modus Ponens from $a = b$ we infer $\mathrm{S}^z_a \mathbf{A} \supset \mathrm{S}^z_b \mathbf{A}$, i.e. $p[a] \supset p[b]$; finally, again by Modus Ponens, from $p[a]$ we conclude $p[b]$.

`(equality` *(equality target)* *(path*$_1$ ... *path*$_n$*)*`)`

Case

$$\frac{\begin{array}{ll} A \Rightarrow & p_1 \vee \ldots \vee p_n \\ A \Rightarrow & p_1 \supset q \\ A \Rightarrow & p_2 \wedge \neg p_1 \supset q \\ \vdots & \\ A \Rightarrow & p_n \wedge \neg p_{n-1} \wedge \ldots \wedge \neg p_1 \supset q \end{array}}{A \Rightarrow \quad q}$$

When a proposition can be derived in each one of an exhaustive list of cases, then it is a theorem. The exhaustive list of cases is represented by a disjunction $p_1 \vee \ldots \vee p_n$. Each case has the form $A \Rightarrow p_k \wedge \neg p_{k-1} \wedge \ldots \wedge \neg p_1 \supset q$ so as to keep it disjoint from the others; this is not strictly required by the Rule of Cases but (1) making each assumption more specific also results in the corresponding case being easier to prove (which is an important consideration in a proof system), and (2) in order to derive anything at all from $p(a) \vee p(b)$, $p(a) \Rightarrow p(a)$, and $p(b) \Rightarrow p(b)$, one must introduce a new term c defined to be a in the first case and

b in the second case, and for this definition to be consistent with the assumption set, it is necessary that the cases be disjoint [more about this in Chapter 6 p204 and p226]. *Proof:* If we have $H \vdash \mathbf{A} \vee \mathbf{B}$, $H \vdash \mathbf{A} \supset \mathbf{C}$ and $H \vdash \neg\mathbf{A} \wedge \mathbf{B} \supset \mathbf{C}$, then by Rule P and the tautology $[\mathbf{A} \supset \mathbf{C}] \wedge [\neg\mathbf{A} \wedge \mathbf{B} \supset \mathbf{C}] \supset_{\bullet} [\mathbf{A} \vee \mathbf{B}] \supset \mathbf{C}$, we infer $[\mathbf{A} \vee \mathbf{B}] \supset \mathbf{C}$ and finally conclude $\mathbf{C}$ by Modus Ponens.

```
(case (disjunction case1 ... casen) ())
```

3.5 Skolemizing Axioms

Since skolemization plays such a central rôle in LOGICALC, it would be convenient if it could be viewed as an inference rule rather than as an operation on theories. This is what gave me the idea of "skolemizing axioms." For every quantified formula $\exists x\, \mathbf{A}[x]$, we can imagine there is a corresponding "skolemizing axiom" $\exists x\, \mathbf{A}[x] \equiv \mathbf{A}[\alpha]$, where α is a new skolem constant denoting the x which satisfies $\mathbf{A}$. Skolemizing positive occurrences of the quantified formula $\exists x\, \mathbf{A}[x]$ is now simply a matter of using the corresponding skolemizing axiom and invoking the Substitutivity of Equivalence. What is needed to formalize this notion and fold it back into the logic is an axiom schema that will correctly generate all skolemizing axioms. The following explains how this can be done.

I restrict myself to a countable formulation of $\mathcal{F}^=$. Each *wff* in $\mathcal{L}(\mathcal{F}^=)$ can be uniquely associated with a different integer by fixing a particular enumeration. Let ι be an indexing function that maps a *wff* to an integer in precisely this manner.

A new (infinite and countable) set of functional constants α_i^n is introduced, where α_i^n has arity n, and $n = 0$ indicates an ordinary constant, and the following axiom schema is added:

$$\mathcal{Q}x\, \mathbf{A}[x] \equiv \mathbf{A}[\alpha^n_{\iota(\mathcal{Q}x\mathbf{A})}(y_1, \ldots, y_n)]$$

where $\mathcal{Q}$ is a quantifier, **A** is a *wff* of $\mathcal{L}(\mathcal{F}^{=})$, x is some individual variable, and y_1 through y_n are all the free variables of $\mathcal{Q}x\,\mathbf{A}$ in an order specified by some arbitrary scheme such as order of occurrence or order of enumeration. A skolemizing axiom is an instance of the above schema.

The result is a conservative extension of $\mathcal{F}^{=}$ (see 3301 p124 in [And86]). It simply serves to make the notion of skolemization into an explicit theory.

During a session with LOGICALC, skolemizing axioms are added to the database when they are needed. Only plan generators and inference generators perform such additions; inference rules only operate on available premises. α_i^n is represented by a skolem function of n arguments. Also, since one direction in the equivalence is always obvious (by Existential Generalization or Universal Instantiation), only the other direction is effectively asserted.

Skolemizing axioms are defined only for formulae $\mathcal{Q}x\,\mathbf{A}[x]$ of $\mathcal{L}(\mathcal{F}^{=})$, i.e. that do not themselves contain skolem terms. I do not need skolemizing axioms for formulae containing skolem terms because I am trying to capture only and precisely those inferences allowed in $\mathcal{F}^{=}$ (wherein neither premises nor conclusions mention skolem terms). On the other hand, it is possible to iterate any number of times the same process of introducing skolemizing axioms, or to otherwise generalize the principle of the skolemizing axiom schema which I presented earlier; however, such a more complex formal apparatus is not needed here.

3.6 Inference Rule for Skolemization

In LOGICALC, skolemization is implemented as an inference rule. In the following I will use $\mathcal{Q}$ to denote a quantifier (either $\forall$ or $\exists$), and each time I will specify whether it has universal or existential force—e.g. $\forall$ has universal force in $p \supset \forall x\, q$, but existential force in $[\forall x\, p] \supset q$.

skolemize *(universal)*
$$\frac{A \Rightarrow F[\mathcal{Q}x\, p[x]]}{A \Rightarrow \quad F[p[y]]}$$

When the quantifier $\mathcal{Q}$ has universal force and the subformula $\mathcal{Q}x\, p[x]$ does not occur within the scope of another quantifier in F, and y is free for x in F, then the quantifier can be removed and the variable x replaced with the free variable y. LOGICALC doesn't have a rule of universal instantiation as such. It can be simulated by this version of skolemization followed by a substitution step (typically unification). *Proof:* by Axiom Schema 4, $\forall x\, p[x] \supset p[y]$ is an axiom (resp. by Existential Generalization and the Deduction Theorem $\vdash p[y] \supset \exists x\, p[x]$). From $F[\forall x\, p[x]]$ (resp. $F[\exists x\, p[x]]$), by Substitutivity of Implication for positive (resp. negative) occurrences of $\forall x\, p[x]$ (resp. $\exists x\, p[x]$) infer $F[p[y]]$.

skolemize *(existential)*
$$\frac{\begin{array}{l} A \Rightarrow \quad F[\mathcal{Q}x\, p[x]] \\ A \Rightarrow \mathcal{Q}x\, p[x] \equiv p[\alpha] \end{array}}{A \Rightarrow \quad F[p[\alpha]]}$$

When the quantifier $\mathcal{Q}$ has existential force and the subformula $\mathcal{Q}x\, p[x]$ does not occur within the scope of another quantifier in F, and there is a skolemizing axiom for $\mathcal{Q}x\, p[x]$, then the quantifier can be removed, and x replaced throughout p by α. *Proof:* by Substitutivity of Equivalence.[4]

3.7 Inference Rule for Quantification

Quantification is an inference rule that performs a function inverse of skolemization, i.e. it reinserts quantifiers in subformulae. I will use the same conventions as above.

quantify *(existential)*
$$\frac{A \Rightarrow \quad F[p[a]]}{A \Rightarrow F[\mathcal{Q}x\, p[x]]}$$

When $p[a]$ is a subformula of F not in the scope of a quantifier, and a is some

[4] Actually, since only the non trivial direction of the equivalence is recorded, the justification should really be by Substitutivity of Implication, which works out just the same since $\mathcal{Q}$ must have existential force.

term, then some occurrences of a in p may be existentially abstracted by a variable and the subformula $p[a]$ replaced with the quantified formula $\mathcal{Q}x\, p[x]$, where $\mathcal{Q}$ has existential force, i.e. is $\exists$ if the subformula has positive sign, $\forall$ otherwise. *Proof:* by Existential Generalization (resp. Universal Instantiation) and the Deduction Theorem infer $p \supset \exists x\, \overline{\varsigma}^a_x p$ (resp. $\forall x\, \overline{\varsigma}^a_x p \supset p$); then, by Substitutivity of Implication for positive (resp. negative) occurrences of p in F conclude $F[\exists x\, \overline{\varsigma}^a_x p]$ (resp. $F[\forall x\, \overline{\varsigma}^a_x p]$).

quantify *(universal)*

$$\frac{\begin{array}{ll} A \Rightarrow & F[p[\alpha]] \\ A \Rightarrow & \mathcal{Q}x\, p[x] \equiv p[\alpha] \end{array}}{\begin{array}{ll} A \Rightarrow & F[\mathcal{Q}x\, p[x]] \end{array}}$$

Here again, a skolemizing axiom and the principle of Substitutivity of Equivalence are together at work. Note that the premise $A \Rightarrow F[p[\alpha]]$ is often really of the form $A \Rightarrow F[p[x]]$, where x is a free variable which gets unified with α for this particular inference step.

3.8 Soundness and Completeness

A logical system is sound iff all its theorems are valid: that is, if its axioms are valid and its inference rules preserve validity. I have shown that all of LOGICALC's rules of inference are derived rules of inference in a conservative extension of $\mathcal{F}^=$, therefore they are sound by virtue of the system $\mathcal{F}^=$ being sound.

The system $\mathcal{F}^=$ is complete in the sense that every *wff* of $\mathcal{L}(\mathcal{F}^=)$ which is valid is also a theorem. I will show that LOGICALC shares this property. However, I will only be interested in completeness with respect to formulae which are strictly in $\mathcal{L}(\mathcal{F}^=)$. Remember that LOGICALC's system is an extension of $\mathcal{F}^=$. In particular, its language has additional symbols (such as skolem terms) which are not in $\mathcal{L}(\mathcal{F}^=)$. I will *not* consider completeness relative to this extended language, but rather to its proper subset $\mathcal{L}(\mathcal{F}^=)$ which is all that is required to express sentences in First-Order Logic.

The result will be established by showing that LOGICALC faithfully captures the axiomatic structure of $\mathcal{F}^=$ and therefore that every theorem of $\mathcal{F}^=$ is also a theorem in LOGICALC.

Modus Ponens: is also a rule of inference in LOGICALC.

Generalization: is subsumed by quantification. *Proof:* if $\mathbf{A}[x]$ is a formula, there must be a skolemizing axiom of the form $\forall x\, \mathbf{A}[x] \equiv \mathbf{A}[\alpha]$. From $\mathbf{A}[x]$ infer $\mathbf{A}[\alpha]$ by Substitution, then $\forall x\, \mathbf{A}[x]$ by Substitutivity of Equivalence from the skolemizing axiom; this is exactly what Quantify (universal) accomplishes.

Axiom of Reflexivity: is also an axiom in LOGICALC.

Axiom Schemata 1, 2 and 3: if $\mathbf{A}$ is a tautology, so is $x = x \supset \mathbf{A}$, therefore infer $\mathbf{A}$ by Taut-Trans.

Axiom Schema 4: $\forall x\, \mathbf{A}[x] \supset \forall x\, \mathbf{A}[x]$ is a tautology, therefore a theorem. By Skolemization into the consequent infer $\forall x\, \mathbf{A}[x] \supset \mathbf{A}[y]$, then by Substitution conclude $\forall x\, \mathbf{A}[x] \supset \mathbf{A}[t]$.

Axiom Schema 5: $\forall x\, [\mathbf{A} \vee \mathbf{B}[x]] \supset \forall x\, [\mathbf{A} \vee \mathbf{B}[x]]$ is a tautology, therefore a theorem. By Skolemization into the consequent infer $\forall x\, [\mathbf{A} \vee \mathbf{B}[x]] \supset [\mathbf{A} \vee \mathbf{B}[y]]$. There is a skolemizing axiom of the form $\mathbf{B}[\alpha] \equiv \forall z\, \mathbf{B}[z]$. By substitution infer $\forall x\, [\mathbf{A} \vee \mathbf{B}[x]] \supset [\mathbf{A} \vee \mathbf{B}[\alpha]]$, then by Quantification of the second occurrence of $\mathbf{B}$ conclude $\forall x\, [\mathbf{A} \vee \mathbf{B}[x]] \supset [\mathbf{A} \vee \forall z\, \mathbf{B}[z]]$ and by Alphabetic Change of Bound Variable $\forall x\, [\mathbf{A} \vee \mathbf{B}[x]] \supset [\mathbf{A} \vee \forall x\, \mathbf{B}[x]]$.

For this derivation to be acceptable, we must show that the rule of Alphabetic Change of Bound Variable is a derived inference rule of LOGICALC. *Proof:* From $\forall x\, \mathbf{A}[x]$ infer $\mathbf{A}[z]$ by Skolemization, where z is a variable free for x in $\mathbf{A}[x]$. For any variable y, there is a skolemizing axiom of the form $\forall y\, \mathbf{A}[y] \equiv \mathbf{A}[\alpha]$. By Substitution, from $\mathbf{A}[z]$ infer $\mathbf{A}[\alpha]$; and finally, by Substitutivity of Equivalence (or Modus Ponens), conclude $\forall y\, \mathbf{A}[y]$.

Axiom Schema 6: it would seem to be a simple matter to infer $x = y \supset \centerdot \mathbf{A}[x] \supset \mathbf{A}[y]$ by first assuming $x = y$, then applying the Equality rule to $\mathbf{A}[x] \supset \mathbf{A}[x]$, and finally discharging the assumption. There is a limitation in LOGICALC which prevents you from making an assumption in this way (to be subsequently discharged by the Deduction Theorem) if it has free variables (and more importantly if it shares free variables with the consequent).

When you must find a proof for $A \Rightarrow p \supset q$, you typically want to set up a subgoal of the form $A; p \Rightarrow q$ (the refinement to be validated by If-Intro). If $p[x]$ and $q[x]$ share the free variable x, then the correct semantics for the subgoal $A; p[x] \Rightarrow q[x]$ is that, as you further refine and develop the proof and x becomes more and more instantiated (remember we rely on unification), the proper and selfsame substitution is reflected in both the assumption $p[x]$ and the subgoal's conclusion $q[x]$.

Unfortunately, this cannot be done because when $p[x]$ is asserted in the assumption set (datapool), the free variable x loses its former identity and in some sense becomes renamed. A similar (mis)feature is present in PROLOG: when you `assert(p(X))` and `X` becomes subsequently instantiated, this instantiation will not be reflected in the assertion you made earlier.

Therefore, LOGICALC will not allow you to "move" p to the assumption set if it contains free variables; these variables will have to be "captured" either by quantification or by instantiating them with ground terms (the system will offer to do this automatically whenever necessary).

The *if-intro* inference rule makes sure to enforce this particular restriction and will arbitrarily rename all free variables (should there be any for some reason) in the assumption being discharged. For instance $H, p[x] \Rightarrow q$ would result in $H \Rightarrow p[z] \supset q$, where z is not free in q and is free for x in $p[x]$.

Despite this limitation, it is not very difficult to derive Axiom Schema 6. Let $\mathbf{A}[u, x]$ be a *wff* such that x is a variable free in $\mathbf{A}$, and u represents all the

other variables free in **A**. There are skolemizing axioms of the form:

$$\forall u\,[\mathbf{A}[u,z] \supset \mathbf{A}[u,z]] \;\equiv\!\centerdot\; \mathbf{A}[\gamma(z),z] \supset \mathbf{A}[\gamma(z),z] \quad \text{(i)}$$

$$\forall xy\,[x = y \supset \forall u\centerdot\mathbf{A}[u,x] \supset \mathbf{A}[u,y]] \;\equiv\!\centerdot\; \alpha = \beta \supset \forall u\centerdot\mathbf{A}[u,\alpha] \supset \mathbf{A}[u,\beta] \quad \text{(ii)}$$

1.	$\Rightarrow \mathbf{A}[u,z] \supset \mathbf{A}[u,z]$	is a tautology
2.	$\Rightarrow \forall u\centerdot\mathbf{A}[u,z] \supset \mathbf{A}[u,z]$	by Quantification using (i)
3.	$\alpha = \beta \Rightarrow \forall u\centerdot\mathbf{A}[u,z] \supset \mathbf{A}[u,z]$	by Assumption-Intro
4.	$\alpha = \beta \Rightarrow \forall u\centerdot\mathbf{A}[u,\alpha] \supset \mathbf{A}[u,\alpha]$	by Substitution
5.	$\alpha = \beta \Rightarrow \forall u\centerdot\mathbf{A}[u,\alpha] \supset \mathbf{A}[u,\beta]$	by Equality
6.	$\Rightarrow \alpha = \beta \supset \forall u\centerdot\mathbf{A}[u,\alpha] \supset \mathbf{A}[u,\beta]$	by If-Intro
7.	$\Rightarrow \forall xy\centerdot x = y \supset \forall u\centerdot\mathbf{A}[u,x] \supset \mathbf{A}[u,y]$	by Quantification using (ii)
8.	$\Rightarrow x = y \supset \forall u\centerdot\mathbf{A}[u,x] \supset \mathbf{A}[u,y]$	by Skolemization
9.	$\Rightarrow x = y \supset \centerdot\mathbf{A}[u,x] \supset \mathbf{A}[u,y]$	by Skolemization

The simpler case where there are no other free variables u is dealt with simply by omitting axiom (i) as well as lines 2 and 9, and by removing $\forall u$ and any other mention of u throughout.

I have demonstrated that LOGICALC completely captures the axiomatic structure of $\mathcal{F}^=$, therefore it is complete in the sense that all valid formulae of $\mathcal{L}(\mathcal{F}^=)$ can be derived.[5]

3.9 Lambda Expressions

LOGICALC also allows terms of the form $\lambda x_1 \ldots x_n\,\mathbf{A}$ which denote anonymous predicates or functions. The motivations for introducing this notation were:

- It is occasionally convenient to state a general principle as an axiom in terms of a free predicate (or function) variable and be able to derive instances tailored to

[5] The language of LOGICALC covers more formulae since it includes the skolem constants α_i^n.

the current problem by plugging in the appropriate λ-expression. Mathematical Induction is such a principle:

$$(\forall P : \mathbb{N} \mapsto \text{prop}) \, [P(0) \wedge (\forall n \in \mathbb{N}) \, P(n) \supset P(n+1)] \supset (\forall m \in \mathbb{N}) \, P(m)$$

By substituting $\lambda x \left[\sum_{i=0}^{x} i = \frac{x(x+1)}{2}\right]$ for P, we are able to derive the following induction theorem:

$$\left[\sum_{i=0}^{0} i = \frac{0(0+1)}{2} \wedge (\forall n \in \mathbb{N}) \, \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \supset \sum_{i=0}^{n+1} i = \frac{(n+1)(n+1+1)}{2}\right] \supset (\forall m \in \mathbb{N}) \, \sum_{i=0}^{m} i = \frac{m(m+1)}{2}$$

- λ-expressions are a convenient notation for expressing set theoretic notions. For instance, $\lambda x \, \mathbf{A}[x]$ can be interpreted as the set of those x which satisfy $\mathbf{A}$. Set intersection is defined by:

$$\forall x \centerdot x \in S_1 \cap S_2 \equiv x \in S_1 \wedge x \in S_2$$

by viewing $x \in S$ as an abbreviation for $S(x)$, once again it is possible to plug in λ-expressions for S_1 and S_2 and derive instances of the above definition.

3.9.1 Comprehension Axioms

Naturally, the logical system needs to provide ways of manipulating λ-expressions. For instance, plugging such expressions in the above schemas involved a little more than mere substitution: it was also necessary to reduce[6] the resulting redexes—a redex is an expression of the form $(\% \ (\lambda x_1 \ldots x_n.B) \ t_1 \ldots t_n)$ denoting the application of a λ-expression to arguments. Reducing the preceding redex means replacing it with $B[t_1/x_1 \ldots t_n/x_n]$—Thus $P(m)$ after substitution became $[\lambda x \sum_{i=0}^{x} i = \frac{x(x+1)}{2}](m)$, which was then converted to $\sum_{i=0}^{m} i = \frac{m(m+1)}{2}$.

[6] What LOGICALC calls "reduction" is also known as λ-conversion.

The intuitive way to justify this reduction operation is to assume the existence of Comprehension Axioms of the form:

$$[\lambda x_1 \ldots x_n\, \mathbf{A}](t_1, \ldots, t_n) \equiv \S^{x_1 \ldots x_n}_{t_1 \ldots t_n} \mathbf{A}$$
$$[\lambda x_1 \ldots x_n\; s](t_1, \ldots, t_n) = \S^{x_1 \ldots x_n}_{t_1 \ldots t_n} s$$

and then invoke Substitutivity of Equivalence (or Equality); similarly for Abstraction. Unfortunately, it is well known that the resulting system is inconsistent.

3.9.2 Rules for Reduction and Abstraction

Before agonizing further on the fundamental inconsistency introduced by this treatment of λ-expressions, I will state formally the rules of Reduction and Abstraction, however dubious they may be.

Reduction $\dfrac{H \Rightarrow \mathbf{A}[(\%\ (\lambda x_1 \ldots x_n.p)\ t_1 \ldots t_n)]}{H \Rightarrow \mathbf{A}[\S^{x_1 \ldots x_n}_{t_1 \ldots t_n} p]}$

Replace a redex with the body of the λ-expression with each occurrence of a formal parameter x_i replaced by the corresponding actual parameter t_i. The t's are supposed to be free for the x's in $\mathbf{A}$, and p may be either a term or a *wff*.

Abstraction $\dfrac{H \Rightarrow \mathbf{A}[p[t_1, \ldots, t_n]]}{H \Rightarrow \mathbf{A}[(\%\ (\lambda x_1 \ldots x_n.p[x_1, \ldots, x_n])\ t_1 \ldots t_n)]}$

Replace a term or *wff* p in $\mathbf{A}$ with a redex (i.e. the application of a λ-expression to arguments) that reduces to p. $t_1 \ldots t_n$ are term or *wffs* that may or may not occur in p. $x_1 \ldots x_n$ are variables which do not occur free in p. Each x_i abstracts out of p zero or more occurrences of t_i. t_i must not contain any variable bound (by quantifier or lambda) within a subterm or subformula of p.

3.9.3 Russell's Paradox

When applied without discrimination, these rules send soundness down the tube. In particular, here is the derivation of a contradiction inspired by Russell's famous paradox.

The set of all sets which are not member of themselves might be represented by the expression $\lambda y \neg y(y)$.

$$[\lambda y \neg y(y)]\lambda y \neg y(y) \Rightarrow [\lambda y \neg y(y)]\lambda y \neg y(y) \tag{i}$$

must be an axiom in LOGICALC. By Reduction we conclude

$$[\lambda y \neg y(y)]\lambda y \neg y(y) \Rightarrow \neg[\lambda y \neg y(y)]\lambda y \neg y(y) \tag{ii}$$

which is in contradiction with (i). Therefore, by NOT-INTRO, we infer:

$$\Rightarrow [\lambda y \neg y(y)]\lambda y \neg y(y)$$

We can proceed similarly with the negation and infer:

$$\Rightarrow \neg[\lambda y \neg y(y)]\lambda y \neg y(y)$$

Hence the paradox.

LOGICALC will give you as much rope as you need to hang yourself, and then some. Type theory is a rather attractive alternative which does not have these problems [And86, AINP88c, C+86]; however, it would require all variables and constants to be explicitly typed (annotated with a type). Implementing such a scheme would not be terribly difficult but it would necessitate a number of additions and extensions to the code to make the approach practical:

1. The representation would have to be extended: every variable and every constant must be associated with a type expression.

2. Unification would have to be revised to properly deal with types, e.g. it should be replaced with some variant of higher-order unification such as Huet's or Elliott's.

3. Input syntax would have to be extended to allow the specification of type expressions. Also, if the user had to type in all type annotations, such a task would quickly become unbearably tedious. Therefore, the system would have to be able to supply the proper type annotations when they are left out by the user, e.g. by means of a type inference/reconstruction algorithm.

4. Syntax checking would have to be revised accordingly. In fact it would have to be part of the type reconstruction procedure.

5. Printing of terms and *wffs* should indicate type annotations but not overwhelm the display with them.

For the purpose of mathematical investigations, this would be a worthwhile project, but, since I was mostly interested in facilitating experimentations with AI formalisms/ problems, it seemed that the benefits to be derived from such an enterprise would hardly be worth the effort. Instead, I suggest we follow traditional practice in mathematics which is to omit type annotations, but be mindful to only perform those manipulations that would be permissible in a typed system.

LOGICALC allows the user to specify type declarations for variables and constants. On the strength of such declarations, syntax checking [see Appendix A] will detect and prohibit the more blatant infractions to the principle of type well-formedness.

Since neither variables nor constants are typed symbols (i.e. they do not have a type annotation), what good are type declarations after the process of statically checking the syntax of a formula or expression has been carried out? LOGICALC will typically translate a type declaration into an IS-expression of the form (IS ⟨*type*⟩ ⟨*term*⟩) and

insert it in the resulting formula (using either $\wedge$ or $\supset$ as appropriate) as an additional constraint. The type theory proper is specified as a theory extension. Note that this practice serves to limit the scope of certain axioms, but will not prohibit the derivation of paradoxical results, e.g. when a substitution step violates an implicit type constraint.

If we represent a type constraint by a proposition of the form $x{:}T$, the axiom of Mathematical Induction will be encoded by a formula such as:

$$P : \mathbb{N} \mapsto \text{prop} \supset \centerdot P(0) \wedge \forall n\, [n{:}\mathbb{N} \supset \centerdot P(n) \supset P(n+1)] \supset \forall m\, [m{:}\mathbb{N} \supset P(m)]$$

3.10 Unification and Substitutions

So far, my exposition of the logic did not presuppose more than the ability to determine syntactical identity in order to apply rules of inference. However, LOGICALC affords the user considerably more flexibility and power by relying on unification instead. Thus, Modus Ponens should be understood as stating that from $p_1 \supset q$ and p_2 and the existence of a substitution θ such that $\theta p_1 \cong \theta p_2$ (where $\cong$ denotes syntactic identity), one may infer θq.

Naturally, the problem of unifying expressions is made slightly more complicated by the fact that quantifiers and lambdas have to be properly dealt with. In particular, we should require that $\forall x\, \mathbf{A}[x]$ and $\forall y\, \mathbf{A}[y]$ must unify, but no substitution will make the two expressions syntactically identical; such an identification could only be effected through an alphabetic change of bound variables (such as $x \rightarrow y$). Should we then extend the notion of unification to include not only a substitution but also a renaming? I suggest that such an extension is unnecessary. Whereas the substitution truly carries additional information, the renaming does not and is merely a change of notation. Instead we should broaden the notion of syntactic identity to cover notational variants as well.

The classical method for factoring out the choice of names for bound variables is to use De Bruijn's notation in which binding lists are omitted and each occurrence of a bound variable is replaced with the appropriate De Bruijn's index [dB72]. A De Bruijn's index is an integer denoting the reference depth of the occurrence of the variable it stands for: i means *"the variable introduced by the ith quantifier or lambda up from here."* Thus $\lambda f\, \lambda g\, [f(\lambda h\, h(g))](g)$ would become $\lambda\, \lambda\, [2(\lambda\, 1(2))](1)$.

Two formulae are equivalent modulo an alphabetic change of bound variables iff they are syntactically identical when written using De Bruijn's notation [dB72]. This observation leads me to formulate the following improved notion of syntactic identity:

> *Two expressions will be declared syntactically equivalent if they are identical when written using De Bruijn's notation.*
> *More practically: the only places where they may differ are occurrences of bound variables, and these occurrences would be assigned the same index by the process of translation to De Bruijn's notation.*

The justification for using unification rather than mere syntactic identification is that it simulates substitution steps. Unification does not permit inferences which do not also have a derivation using syntactic identification and a few additional substitution steps, but it makes their proofs shorter since these steps do not have to appear explicitly.

If we decided to make the aforementioned steps explicit, obviously the corresponding substitutions would have to satisfy the requirements of the classical Substitution Rule: i.e. for a given unifier $\theta = \{\, t_1/x_1, \ldots, t_n/x_n \,\}$, that t_i be free for x_i in the corresponding formula. As we shall see, this imposes a natural restriction on the unification procedure.

t is free for x in $\mathbf{A}$, iff for each free variable y in t there is no free occurrence of x in $\mathbf{A}$ that is in the scope of a $\forall y$, $\exists y$ or λy. If we consider this requirement in light of de Bruijn's notation, we find that what it means is that t is allowed to contain an

occurrence of a de Bruijn's index n iff this occurrence appears in the scope of at least n quantifiers or lambdas in t. In other words, a free variable can only be assigned a well-formed De Bruijn's term, i.e. one in which every index n occurs within the scope of at least n quantifiers or lambdas. I call this the requirement of well-formedness for substitutions.

Using de Bruijn's notation, LOGICALC's extended notion of unification is simply ordinary unification with the addition of the restriction just stated. It is exactly ordinary unification when the formulae contain no quantifier or lambda. It is also exactly ordinary unification when they do, except that it will report failure more often. It will report failure in exactly those cases where the resulting substitution would contain assignments involving non-well-formed De Bruijn's terms, thereby violating the requirement of the Substitution Rule.

LOGICALC's implementation of unification is of the incremental depth-first recursive algorithm variety. Bound variables are compared by computing their de Bruijn indices (for this reason I must keep track of binding lists as they are being encountered). When about to assign a term to a free variable, the algorithm not only performs the "occur" check, but also makes sure that said term does not contain free occurrences of bound variables (this again relies on keeping track of binding lists).

Usually, unification of two expressions is done assuming that their respective free variables are independent (i.e. that x in one expression is distinct from x in the other). LOGICALC's unification procedure cleverly keeps track of which side each term or variable pertains to; for instance unifying $f(x)$ and $f(g(x))$ would produce a pseudo-substitution of the form $\{ \langle x, \mathbf{left} \rangle \leftarrow \langle g(x), \mathbf{right} \rangle \}$ from which it is possible to extract (compute) a substitution applicable to the left side (which would be something like: $\{ x \leftarrow g(y) \}$) and another one applicable to the right side (empty in this case).

Unfortunately, it is possible that extracting a substitution in this manner might violate the requirement of well-formedness. Consider the problem of unifying $\lambda y\, f(x)$

with $\lambda z\, f(g(x))$. Clearly, the same pseudo-substitution will be computed by the unification procedure. However, if extracting a substitution for the left side results in $\{\, x \leftarrow g(y) \,\}$ where y was thought to be a "new" variable, we can see that the intended meaning of y in the substitution conflicts with the bound variable in $\lambda y\, f(x)$ and unwittingly violates the requirement of well-formedness.

The solution is to keep in mind that "pseudo-substitutions" are constructed according to the principle of well-formedness; therefore, any violations of this principle encountered while applying a "true" substitution must be the result an unfortunate choice of name when extracting said substitution from the corresponding pseudo-substitution. One possible fix would be to revise the substitution by chosing other names until it was admissible again. I opted for a different approach: I rename bound variables until they no longer conflict with free variables.[7]

[7] In fact, since constants and bound variables are both represented by LISP symbols, it is also necessary to rename bound variables so that they do not conflict with constants by the same names.

Chapter 4

A Graph Editor for Theorem Proving

In this chapter I introduce the idea of theorem proving as graph editing. Each type of node, such as goals and plans, must define its own interface by means of a display function and a command processor. I will look at the issues of Display (introducing the notion of a view), Abbreviations (input and output), Commands, and Help.

4.1 Introduction

A session with LOGICALC begins with the user stating the theorem to be proven, which the system formats and presents to him as a "goal." By invoking a "plan generator," he should then propose a refinement of the goal into a "plan" whose steps are themselves goals, and proceed to select one of these subgoals to work on next in a similar fashion. The process bottoms out when a subgoal matches an axiom or an assumption.[1] When one branch has thus been closed, the user will move on to the

[1] or is otherwise obvious, e.g. a tautology.

next remaining open branch, and so on until the proof tree has been completed.

This view of interactive proof development is clearly an instance of the more general notion of graph editing. The user works towards constructing a proof tree by progressively extending a tree of goals and plans until each branch is closed.

Others too have looked at interactive proof construction as a process of graph editing. NUPRL's user interface [C+86] is implemented on *ted*, the Text EDitor, and *red*, the Refinement EDitor; both are based on the concept of structure editor. The Environment for Formal Systems (EFS) [Gri87a] is also based on a structure editor, but generated by the Cornell Synthesizer Generator [RT87]. The Interactive Proof Editor (IPE) [RT88] more specifically applies the notion of an attribute grammar to structure the proof tree and investigates the idea of "proof by clicking."

IPE lends itself well to an exploratory style of proof. LOGICALC was also designed precisely with this type of interaction in mind. Unlike systems such as LCF [Mil79a, Pau87] and the Boyer-Moore theorem prover [BM79] where it is necessary to determine in advance the key lemmas that will be required in the course of the proof, IPE and LOGICALC both allow the user to start out and proceed without a-priori knowledge of what will be required or even what direction the course of the proof will take. The ability to switch between alternative proof attempts encourages the user to "try things out." Also, it is not unusual to realize after a few such attempts that there are axioms "missing" in the theory; it should be possible (and it is) to extend the theory "on the fly" without forcing the user to start all over again.

The "detaching" mechanism, which will be discussed in a separate chapter, allows LOGICALC to go one step further and provides the user with another degree of freedom. The basic idea is that, when confronted with a goal, the user often has the intuition that a particular axiom should be used to solve it, but the details and the particulars of the inferential manipulations actually required are difficult to work out. With LOGICALC, the "detaching" mechanism will take care of it automatically by a

combination of unification and a process that superficially resembles non-clausal resolution [Mur82] but is actually justified in terms of regular inference rules [Chapters 3 and 7].

As I mentioned earlier, IPE investigates the possibility of "proof by clicking." While this is an attractive model for the non-sophisticate (or casual) user, it only allows 1-dimensional exchange of information and does not support compositionality well. Furthermore, because of its low bandwidth, the preferred mode of interaction must be supplemented with other forms of input (e.g. textual). I find that graphical mouse-based interfaces, while they make for much sexier demos, just don't have the generality, simplicity, flexibility, or power of the more traditional command-line-oriented shells. Also, it is often a lot faster to just type in a command than to pull down a menu and select an item (and then be prompted for arguments).

LOGICALC offers a model of interaction that is reminiscent of a UNIX-type shell; there are motion commands to follow parent/child or other kinds of links, just like `cd` allows the user to move about the file system by following super/subdirectory or symbolic links. The system keeps track of a stack of visited nodes which can be "popped" or "jumped to" in the manner of `pushd`/`popd`. There is a command stack which plays the same role as the UNIX shell "history;" commands can be recalled from this stack and substitutions applied. Abbreviations can be used to define macro commands, much in the manner of aliases. Finally there are commands to extend the graph.

In an article on graphic user interfaces for computer algebra systems, Kajler [Kaj] identifies a number of essential issues; in particular, the following:

- The display of formulae. Subtopic: the display of "long" formulae.
- Abbreviating parts of formulae. Subtopic: provision for a "magnifying glass" (i.e. display of abbreviation contents).
- History mechanism.

- Attaching comments to objects.
- Interactive help.

All of these are addressed in LOGICALC.

LOGICALC's graph editing functionality is implemented in an object-oriented way: each type of node provides its own specific display procedure and command processor. Thus the implementation is natural, modular and extensible. For obvious ergonomic reasons (principle of least astonishment), it is desirable to impose some kind of uniformity and consistency of interfaces across node types. In the following sections, I will discuss the topics of Display, Abbreviations, and Commands, which are the principal components of this overall "look and feel," as the catch phrase goes these days. Other issues, such as "detaching" and "find mode," subterm and premise descriptors, etc... which also contribute to uniformity of command specification will be treated in separate chapters.

4.2 Walk Mode

LOGICALC's interface is based on DUCK's walk mode [McD85]. Many entities in DUCK can be thought of as graph structures. These include goal hierarchies constructed during deductions, data type hierarchies, etc... The "walker" or "browser" is a sort of graph editor which allows you to move about these graphs, inspecting, and, in some cases, altering them. When the walker has control, the system is said to be in "walk mode," and prompts with "`w>`". You can type commands to move around in the graph or act on it, or LISP expressions to be evaluated.

In "walk mode," the system displays the current "focus of attention" which is the graph's node currently under scrutiny. We also say that we are *at* the node.

When the focus is moved to a new node, or when the PP command is entered, the system outputs a detailed display of the node, which was designed to be as informative as possible and consists of helpfully formatted material, often including labeled and/or numbered items and pointers to other nodes. Simply numbered items are typically subparts, and typing n to the shell is interpreted as a request to move to the node representing subpart n. In other words, positive integers act as moving commands [more about this later].

The system keeps a stack of those nodes which have been inspected. Whenever you move to a new node, it is pushed on the stack, except when it is already in the stack, i.e. has already been visited, in which case everything above it is "popped" off. The **stack** command displays all elements currently on the stack in an abbreviated form, prefixing each one with a negative (or null) integer: the current node is prefixed with 0 (zero), the previous node with -1, etc... Typing $-n$ moves you back to the node prefixed with $-n$ (in essence, n nodes are popped off the stack). Typing 0 (zero) acts just like pp: the current node is redisplayed in the verbose form.

Each node is really represented by a pair $\langle node, node_descriptor \rangle$, where *node_descriptor* is a 4-uple of the form:

$$\langle type, abbreviate, display, command_processor \rangle$$

Commands such as **stack** or $-n$, which are stack commands are intercepted by the browser's interpreter and directly executed: **stack** invokes the *abbreviate* procedure of each node on the stack in order to produce the condensed display. pp and 0 (zero) are similarly intercepted and invoke the *display* procedure of, and on, the node on top of the stack. All other commands[2] are handed to the current node's *command_processor*. However, should the *command_processor* report failure to handle the command, then the global command processor (named `logicalc`) gets a turn; this global processor centralizes those commands common to all modes, such as `abbreviation`, `execute`,

[2]Except a handful which are also intercepted by the browser.

xabbrev, **commands**, **history**, **information**. If this too should fail, the command is then evaluated as a LISP expression.

4.3 Display

Every type of node has its own display function, but they are all constructed along the same principles. Enough information must be provided to identify and permit useful work on the node. Also, since the paradigm is that of graph editing, sufficient information must be provided to navigate to other nodes. A typical display includes the following:

- A title which indicates type and identity of this node.
- A representation of the node itself. For instance, a goal would display its formula.
- Numbered subparts. For example, a plan will output an enumerated display of its steps. The user can move to subpart n simply by typing n to the shell.
- Other links. They are typically grouped by category. A subtitle indicates the name of the category, and it is followed by an enumeration as above. For example, in the display of a goal, we can find the subcategory "answers;" each answer is numbered and an abbreviated description is provided. The user will be able to move to answer n by typing **answer** n.
- Other information, such as the number of class plans for a goal, or the number of successors for a plan.

Many examples of displays have already been encountered in the introductory example [pp. 7–38]. Here is another illustration:

```
Goal View      <class CLASS.3>
  Find: (X)  in:
    (< (F ?X) !.C)
  Assumptions:
    (< (F !.A) !.C) -- !:<198
  Answers:
    1  X           = !.MID_A_XX2
    2  X           = !.A
  Supergoal:
    (IF (< (F !.A) !.C) (AND (< !.A ?X) (< (F ?X) !.C)))
  Local Plans:
    PLAN --- (TAUT-TRANS)
    1  (< ?Y1.3 (F ?X.4))
    2  (< (F ?X.4) !.C)
    3  (< !.X1(?X.4 ?Y1.3) ?X)
    4  (< ?X !.X2(?X.4 ?Y1.3))
  (2 class plans)
```

As I hope you begin to see, all this is organized to facilitate movement about the graph; recall that $-n$ moved back n nodes; 0 or **pp** redisplays the current node; n moves to subpart number n; and **answer** n moves to n^{th} answer—more generally `foo` n moves to n^{th} element in the enumeration of subcategory `foo`, or n^{th} link of type `foo`. Proof construction may occasionally look like an adventure game, but navigation should not be the difficulty!

Node display is terminal-oriented rather than graphical,[3] but the most widely used terminals are supported (i.e. **xterm**, **vt100**, **hpterm**), as well the TI Explorer, and TEX. The global variable **term*** controls which kind of device output is prepared for. In particular, I produced all examples included in this thesis by setting **term*** to **tex**. What **term*** controls is how certain kinds of textual enhancements are effected, such as selecting boldface or reverse-video. For regular terminals, this is achieved by escape sequences, on the Ti Explorer a **:set-current-font** message must be sent to the stream, while for TEX, appropriate control commands must be inserted (e.g. **\bf**).

[3]The project started out before X was available.

The code is easily extensible, but nowadays it would make more sense to invest in a graphical interface (maybe based on CLX). Note that setting `term*` to `nil` disables all enhancements.[4]

All formulae are displayed using a more powerful version of the traditional LISP pretty-printing procedure. This version is data-driven (therefore easily extensible) and can handle abbreviations (these will be described in the next section). Today, the XP proposal [Wat89] might be a viable alternative although I believe that in certain respects it is not quite as powerful.

In every display and for each category of formulae, there is a corresponding global variable that controls whether and how much of the formula to display. These variables have uniform semantics (and namings) across all node types and are documented in The LOGICALC Manual [Duc88]. Thus the user is able to customize all displays according to his preferences.

The truth abouth LOGICALC is that what you see is *not* what you get. For most type of nodes, LOGICALC provides an extra level of packaging, namely "*views*." You may have noticed that the title of a goal display stated **Goal View** rather than just **Goal**. The reason is that I lied to you: you weren't really looking at a goal, but rather at a view of that goal. The fact is that you *never* get to see goals, but only views of them.

The principal distinction between a goal and a view of same, is that the view has a table which specifies a renaming of the goal's variables. The reason for wanting to do such a thing is rather subtle: in a complex graph, the same node may be accessible from different places. However, these various places may have wildly differing variable namings. Therefore, in order to produce the least astonishment in the user, it is often desirable to carry over the naming conventions of the current node when moving to another. Thence views!

[4] Preferred setting when running under EMACS for instance.

When the pretty-printing procedure must output the representation of a free variable, it will compute a renaming for it by regressing its translation through a list of tables.[5] For instance, printing a goal involves posting the goal's table (i.e. correspondance between the class variables and the names by which they are known in the goal) to this global stack of tables and then pretty-printing the goal's class's formula according to the renamings that can be computed through this stack. For a view, it is the same same thing with the additional posting of the view's own table.

4.4 Abbreviations

A major concern in an interactive system must be to make displays as legible as possible. Similarly, input should allow short-cuts to spare the user tedious (and error prone) typing. These requirements suggest the idea of abbreviations to improve the readability of terms and formulae, and minimize the number of keystrokes required to type them in.

4.4.1 Definitions in NUPRL

NUPRL has a *definition* facility whereby one can say, e.g:

```
DEF GE
 <x:int1>>=<y:int2>==((<x><<y>) -> void)
```

in order to define `>=` in terms of `<`; the definition itself is named `GE`. Subsequently, occurrences of expressions matching the right-hand side will be displayed as the corresponding instance of the definition's left-hand side. For input however, you cannot simply type in an instance of the left-hand side; instead you are required to *instantiate*

[5]Contained in the global variable **namings***.

a definition. In NUPRL's text editor, this means typing `^I` and being prompted for the name of the definition to be instantiated (e.g. `GE`). An instance of the definition is then inserted at the current position in the tree (remember, this is a structure editor) with place holders for the as yet unspecified parameters. The user must then edit each place holder and fill them in with the desired values.

EFS has a similar mechanism: the user would type `@GE` instead of `^IGE`. It goes somewhat further and allows the user to type `@(GE t1 t2)` to produce an instantiation of definition `GE` with its first two slots filled with `t1` and `t2`. In other words: `x` gets replaced by `t1` and `y` by `t2`. As we shall see, LOGICALC's abbreviations are more flexible and easier to use than NUPRL's definitions or EFS improved version of same, and, in some respects, more general.

4.4.2 Skolem Terms

In LOGICALC, most formulae are represented by their fully skolemized form. Among other advantages, this choice of representation permits the consistent use of unification to drive the construction of the proof tree and retrieve premises from the database. However, it means that skolem terms will play a major and pervasive role. Unfortunately, they tend to possess large and infelicitous representations, which is why skolemization is often portrayed as a human-hostile transformation. I contend that the judicious use of abbreviations, by successfully palliating the obfuscatory nature of skolem terms, removes this last objection.

A skolem term is represented by a LISP s-expression of the form `(SK` *name id args...*`)`, where *name* is typically the name of the bound variable (presumably existentially quantified) that gave rise to this term, *id* is a unique identifier (actually an integer), and *args...* are the terms this skolem term depends on (originally these are the universally quantified variables (or free variables) the corresponding bound variable *name* was in the scope of).

Ordinarily, a skolem term is created as a result of skolemizing a quantifed formula. For example:

```
(AXIOM NOFLOOR (FORALL (X) (EXISTS (Y) (< Y X))))
```

is skolemized into, and represented internally by:

```
(< (SK Y 103 ?X) ?X)
```

where 103 is just a value I chose arbitrarily to stand for Y's *id*, but would in reality be whatever value the monotonically increasing global counter `sknum*` happened to have at that time. The example above is rather innocuous, but when the skolem term depends on more variables, the internal form becomes harder to read; and when, in the course of a proof, these variables become instantiated with arbitrarily large terms, it quickly turns into an undecipherable mess. Consider for example:

```
(SK MID 6 (SK A 10) (SK X2 9 (SK A 10) (SK BELOW 7 (F (SK A 10))) (SK
                                C 11)))
```

Similarly on input: skolem terms internal forms are hard and tedious to type. Their size can become arbitrarily large and unnecessarily try the user's typing skills as well as his patience.

4.4.3 Short and Long Forms

The solution to both these problems (input and output) is to come up with a more condensed, more friendly notation for skolem terms, or, at least, a less openly hostile one. Here is what happens in LOGICALC: a skolem term of the form:

```
(SK BAZ 57 A (F B))
```

which depends only on the two ground terms `A` and `(F B)` will be printed in the abbreviated short form:

```
!.BAZ
```

whereas a skolem term depending on non ground terms, such as the following:

```
(SK BAR 109 A (F ?X) ?Y)
```

will be printed together with a summary of the variables it depends on:

```
!.BAR(?X ?Y)
```

That is the general idea. A couple of considerations slightly complicate the picture. Firstly, it is good manners to "define" an abbreviation before using it (consider this a precept of well behaved user interface design). For this reason, the first time a skolem term is printed, it comes out in a slightly more verbose form: the *long* form. Thus, the first time the two skolem terms used for illustration above are introduced, they would respectively come out as:

```
!.BAZ[57](A (F B))
!.BAR[109](A (F ?X) ?Y)
```

thereby establishing identity (the *id* is mentioned between brackets), and listing the actual arguments. In subsequent occurrences, however, they would be displayed in the "short" forms that I exhibited earlier.

Secondly, two skolem terms may be competing for the same abbreviation. Consider for instance:

```
           (SK BAZ 57 A (FOO B))
    and    (SK BAZ 238 (FOO A) A)
```

They both compete for `!.BAZ` as their short form. If the system were to indiscriminately print `!.BAZ` for both, total confusion would ensue. To prevent such a deplorable state of affairs, the system abides by the rule that where the short form would be ambiguous, the long form should be used instead. Of course, the user could always abbreviate one of them differently to resolve the conflict.

I will now recapitulate the notions introduced above and formally state how skolem terms are printed. When a skolem term is printed, it may appear in one of two forms, the *short* form or the *long* form:

Short form: if the skolem term is ground (i.e. mentions no free variable in its arguments, cf. *args...* above), then only its name is printed after the '`!.`' prefix; otherwise, the name is followed by a list of the free variables occuring in *args....* For example:

```
(SK FOO 3 (F A) B)     ⟹  !.FOO
(SK FOO 3 (F ?X) ?Y)   ⟹  !.FOO(?X ?Y)
```

Long form: the '`!.`' prefix is followed by the skolem term's *name* then by its *id* surrounded by square brackets, and finally by a list of its arguments *args....* For example:

```
(SK FOO 3 (F A) B)     ⟹  !.FOO[3]((F A) B)
(SK FOO 3 (F ?X) ?Y)   ⟹  !.FOO[3]((F ?X) ?Y)
```

The decision of whether to chose the short or the long form is based on the following rule:

> *If the skolem term has already been printed on some earlier occasion and no other skolem term by the same name has been printed since, then select the short form, otherwise the long form.*

A skolem term will be printed in the long form the first time it is encountered; on subsequent occasions it will be printed in the short form unless it is competing for the same abbreviation with another skolem term with the same name.

4.4.4 Input Forms

Clearly, a similar mechanism is needed for input, whereby skolem terms can be abbreviated and no longer require to be typed in their entirety. In LOGICALC, the simplest way to input `(SK BAZ 57 A (FOO B))`—assuming it last printed as `!.BAZ`—is to type `!.BAZ`.

The general rule is that "`!.`*n name*" denotes the nth most recently printed skolem term with function *name*. "`!.1` *name*" can be abbreviated "`!.`*name*." Neither the *id* or the *args* have to or should be specified.

The system keeps track of printing chronology by means of histories. When you type "`!.`*n name*," you are designating the nth entry in *name*'s history.

Histories

A *history* is simply a list of skolem terms with identical skolem functions, ordered by printing recency; the most recently printed first. Each skolem function has its own history. According to the examples I have used so far, there must be a `BAR` history, a `BAZ` history, and so forth.

Histories can be inspected by means of the `history` command.

`HISTORY` *names...*

`HISTORY -ALL`

Prints out the histories corresponding to the given names (or prints out all

histories if the -ALL switch is provided instead). For each skolem term in a history, the following correspondance is displayed:

$$\langle\text{non abbreviated long form}\rangle \longrightarrow \langle\text{abbreviated short form}\rangle$$

An example history might be:

```
w> HISTORY MID

History for: MID
  1  !.MID(?X ?Y)
          --> !.MID(?X ?Y)
  2  !.MID[6](?A !.X2[9](?A !.BELOW(?A) ?C))
          --> !.MID_A_X2(?A ?C)
  3  !.MID[6](!.A !.X2[9](!.A !.BELOW[7]((F !.A)) !.C))
          --> !.MID_A_XX2
```

Whenever a skolem term is printed for the first time, it is recorded at the top of its function's history. On subsequent printing occasions, it is removed from the history and reentered at the top. Thus, pretty-printing a formula will generally involve updating several histories.

Conclusion

It is always possible to input skolem terms by typing their complete internal form, but doing so is a tedious and error prone process. As an alternative LOGICALC allows the use of abbreviations:

!.*name*

Denotes the most recently printed skolem term with function *name*.

`!.`*n name*

Denotes the nth entry in *name*'s history—i.e. the nth most recently printed skolem term with function *name*. `!.1` *name* and `!.`*name* are equivalent.

`!.(`*name args...*`)`

This is almost like typing the full internal form, except that the `SK` function and the "id" are omitted. Naturally, this notation may be ambiguous—consider 2 skolem terms with same function, same args, but different ids. Fortunately, when the read-function encounters an ambiguous designator like this, it displays the possible candidates and prompts the user to select one of them.

Note that, while these candidates are being displayed, the normal updating of histories is suspended. The rationale for this exception is that if the disambiguation process—interrupting the reading process—was side effecting histories as a result of printing out candidates, some abbreviations (of the form `!.`*n name*) in the rest of the input might be thrown off.

4.4.5 Generalized Abbreviations

While originally designed to improve the legibility of skolem terms, LOGICALC's notion of abbreviations was extended and generalized to non-skolem terms. The principle of this extension is that the user should be allowed to specify a name for a particular pattern, say **`green`** for **`(color [blue yellow])`**, and whenever said pattern is to be printed it will be shown as if it was a skolem term with the given name (e.g. `!.`**`green`**).

An abbreviation can be defined through the `abbreviation` command:

```
w> abbreviation green (color [blue yellow])
```

The system responds by printing something like `!.`**`green`**`[258]`. What happened is that a new skolem term was created with the internal form `(sk green 258)` and an

assertion was made that it stood for the expression (color [blue yellow]). Note that an expression such as (color [red]) will still print the same as before. More surprisingly (color ?x) won't be affected either even though it matches (color [blue yellow]) because of the free variable ?x; LOGICALC is careful to use abbreviations only when applicable, and, when several are applicable, to select the most specific one. The precise details of this process will be explained later.

It is possible to define an abbreviation for a term containing free variables, in which case the abbreviating skolem terms would have these free variables as its arguments (i.e. *args*). In particular, If the user were now to define an abbreviation for (color ?x):

```
abbreviation hue (color ?x)
```

(color ?x) would, from now on, print as !.hue(?x), and (color [red]) as !.hue([red]), but (color [blue yellow]) would remain unaffected and would continue to print as !.green.

Input works just as you would expect: !.green denotes (color [blue yellow]) and !.(hue [red]) stands for (color [red]). Of course, on input, it is now no longer sufficient to determine what skolem term is denoted by "!....," for that skolem term itself may be an abbreviation. Each skolem term must be recursively disabbreviated to ultimately yield the actual referent.

Here is a summary of the functionality provided through the abbreviation command:

ABBREVIATION *new old*

Defines *new*—which normally must be a symbol—to be an abbreviation for *old*—which is an arbitrary non atomic term. If *old* is a ground term (i.e. it contains no variables), then (SK *new id*), where *id* is brand new, becomes its abbreviation. If *old* contains variables $v_1, \ldots, v_n$ and no other, then (SK *new id* $v_1 \ldots v_n$) becomes its abbreviation.

It is also permitted for *new* to be a functional term instead of just a symbol, with the proviso that *new* and *old* must have identical sets of variables. If *new* = $(f\ t_1\ \ldots\ t_n)$, then (SK f id t_1 ... t_n) becomes *old*'s abbreviation.

`ABBREVIATION` *term*

`ABBREVIATION -ALL`

Displays all possible abbreviations of *term*. The system has heuristics to select the most specific abbreviations. Normally, one would expect most candidates to be filtered out by these heuristics, and at most one to pass. However, under certain circumstances there may be several remaining candidates that are undistinguishable by the heuristics as far as the specificity is concerned. When that happens on printing, the system arbitrarily selects the first one.

When the **-ALL** option is given as an argument, all abbreviation templates are displayed. Whenever an abbreviation is defined, the system asserts an abbreviation template in a special datapool. The **-ALL** option makes it possible to list them.

`ABBREVIATION -REMOVE` *term*

`ABBREVIATION -REMOVE -ALL`

For each abbreviation of *term* the system asks the user whether the corresponding template should be removed. If **-ALL** is specified instead, the same thing happens for all abbreviation templates.

4.4.6 Implementation

When the user defines an abbreviation such as:

```
abbreviation smaller (< ?x ?y)
```

the system first determines the set of free variables occurring in the term to be abbreviated (i.e. { **?x**, **?y** }), then it makes up a new skolem term with the name as

specified by the user, and with the free variables as its arguments; e.g. the system might come up with `(sk smaller 31 ?x ?y)`. Finally an assertion is made to the effect that `(sk smaller 31 ?x ?y)` is an abbreviation for the term `(< ?x ?y)`.[6] When I say assertion, I mean precisely that. LOGICALC sets aside a special datapool in which such bookkeeping assertions can safely be made. In the present case the assertion has the form:

```
(skolabbreviate (< ?x ?y) (sk smaller 31 ?x ?y))
```

In general the form is `(skolabbreviate` ⟨*expression*⟩ ⟨*abbreviation*⟩`)`, where ⟨*abbreviation*⟩ is always a skolem term, and has exactly the same free variables as ⟨*expression*⟩. The system also makes 2 additional notes: (1) the first to the effect that `<` has an abbreviation (this is used for speed in the pretty-printer), and (2) the second to the effect that 31 is an *id* for a skolem term that is an abbreviation (this is used for speed in the reader).

Abbreviation Algorithm

The pretty-printing procedure operates in the natural recursive way, and maintains a cache for the duration of its execution. This cache associates expressions with their abbreviations; thus, if an abbreviation has already been computed, it can be speedily retrieved from the cache. For each subexpression,[7] the system first checks the cache. If that fails, the system checks whether an abbreviation has ever been defined for the expression's main functor (this justifies making note (1) earlier). If yes, the proper abbreviation must be computed (and recorded in the cache) to replace

[6] In this example the abbreviation is actually larger than the term it abbreviates, but we will ignore this minor detail!

[7] Here, perforce I must simplify the exposition to an extreme, lest we get bogged down in mostly irrelevant details.

this expression. In any case, the pretty-printing procedure then proceeds with the sub-expressions (e.g. the arguments to the abbreviation).

I will now describe how an abbreviation is effectively computed. Let us assume that the expression to be abbreviated is $\langle term \rangle$.

1. First, we will query the bookkeeping datapool for assertions which unify with `(skolabbreviate` $\langle term \rangle$ `?any)`. This unification will succeed for these assertions:

   `(skolabbreviate` $\langle ugly \rangle$ $\langle pretty \rangle$`)`

   for which there exists a substitution θ, such that $\theta\langle term \rangle = \theta\langle ugly \rangle$.

2. We shall then reject all candidates which assign values to free variables of $\langle term \rangle$: abbreviating a term should not make it appear to be more instantiated than it really is.

3. Next we shall rank the candidates according to how close $\langle ugly \rangle$ is to $\langle term \rangle$. The idea is to pick those closest on account that they are more specific. In a first approximation, if $\langle ugly \rangle_1$ has fewer free variables than $\langle ugly \rangle_2$, it is probably more specific. On further examination, it appears that the size of the terms required to instantiate the free variables in each case also bears on the issue of specificity. The heuristic I use computes the "size" of $\langle ugly \rangle$ by counting all atoms but not free variables; it works quite well in practice. Only those candidates with maximum size are retained.

4. Similarly, we then rank the remaining candidates according to the size of $\theta\langle pretty \rangle$. Only those candidates with minimum size are kept. The rationale is that we want the abbreviation to be as small and simple as possible.

5. Finally, we just pick randomly from the remaining candidates.

4.5 Commands

The user interacts with LOGICALC through a "shell" which shares a number of similarities with e.g. a UNIX shell. In particular, it is line oriented: a entire line of input is read and the first expression on this line is considered to be the "command" while the rest of the line is taken to constitute the command's arguments.

Unlike a traditional shell however, this one offers a different set of commands depending on the context (current node or current mode), and is implemented in an object oriented manner. Such a design is very natural and intuitive in the context of graph editing, both for the user and the implementor. A further advantage is that context-specific disambiguation of commands can easily be provided. As a consequence, all "named" commands can be abbreviated by a prefix; e.g. **abbreviation** may be invoked by **abbrev** or **abb**, **xabbrev** by **x**, **+plan** by **+**. Should the prefix turn out to be ambiguous, the user would be informed of the candidates, and then be prompted for further completion until the ambiguity is finally resolved.

Also, a command is not required to be denoted by a name, but can be any arbitrary expression, as long as the current "command processor" knows what to make of it. For example, integers often serve as "moving" commands (typing n moves to subpart number n). I have not found a use for stranger sorts of commands, and any input that cannot be interpreted as a command is evaluated as a LISP expression.

4.5.1 Command Line Structure

In a regular shell, the user may specify several commands on the same line by inserting appropriate separators (e.g. a semi-colon ';'). In LOGICALC, a similar segmentation can be effected by insertion of '//'. For instance:

$$cmd_1\ args_1 \ldots\ //\ cmd_2\ args_2 \ldots$$

would first execute "cmd_1 $args_1 \ldots$," then "cmd_2 $args_2 \ldots$." In fact, both '`/`' and '`//`' are command *prefixes* which also act as separators. '`/`' as the additional effect of "silencing" the next command: e.g. if it is a moving command, the destination node will *not* be displayed. Unfortunately, the exact semantics of silencing have not been completely worked out and some commands may violate your expectations, [e.g. what to do when interaction with the user would otherwise be initiated?].

Each command is responsible for making sense of its own arguments, but for the sake of uniformity (and familiarity) I have borrowed yet another convention from the UNIX tradition. Commands accept "options," which, by tacit agreement,[8] are distinguished by a leading dash '`-`'. For example:

```
abbreviation -all
```

lists "all" abbreviations. Note that options too may be abbreviated: **`-all`** may be typed **`-a`**; however, unlike in the case of commands, no provision for interactive disambiguation is made: the first match is selected. This should never be a problem because options accepted by the same command are unlikely to begin with identical letters. Making sure this doesn't happen is the responsibility of the designer [let me help your here: this means me].

In Chapter 6, I will show how this idea of options was also adopted and extended for use in subterm and premise descriptors.

As long as each parameter to a command can be represented by one expression on the command line, parsing arguments is a breeze. Unfortunately, some commands may expect or admit more complex parameters which may require several expressions to notate, e.g. **`-all foo`**. Also, the command may allow the specification of several arguments of the same type: several premises to successively fill the same role, several

[8] Not enforced, but respected throughout.

subterms to act on, etc... Therefore, an additional structuring convention had to be designed. I chose single dashes '`-`' as explicit separators to be inserted at strategic points within the argument list itself to enable the command to correctly partition this list into groups of arguments. For example:

```
+plan equality a b - foo1 foo2 foo3
```

invokes plan generator `equality` with 2 groups of arguments: the first group contains `a` and `b` and denotes subterms to be replaced by equality, and the second group contains `foo1`, `foo2`, and `foo3` which are premises where the necessary equalities are to be found.

Occasionally, for very complex commands, yet another level of structuring may be required. In such cases, I use '`--`' to denote major breaks and '`-`' to denote subdivisions. For instance, specifying the arguments to `abstraction` is rather tricky: you must specify which subexpressions will become redexes, for each one what variables to introduce, and for each variable which subexpressions it will abstract away. The syntax is:

```
+plan abstraction    subfmla - var subterms - var subterms ...
                  -- subfmla - var subterms - var subterms ...
                  -- subfmla - var subterms - var subterms ...
```

The above appears on several lines for clarity, but it must be typed on one line to the shell.

4.5.2 Macro Commands

Often, it would be convenient if the user were allowed to abbreviate certain command lines or certain recurring command patterns. Unfortunately, abbreviations were designed for "expressions," not whole lines of input. This gave me the idea to extend LOGICALC's shell to also accept input of the form:

```
(execute +plan equality foo - bar)
```

and treat it as if the words following **execute** had directly been typed to the shell as a line of input. The parentheses make a world of difference, because **(execute ...)** is an expression, and therefore can be abbreviated.

Defining Macro Commands

execute is a command implemented by the global command processor `logicalc`. An input line of the form **execute** *cmd args...* will be interpreted as if *cmd args...* had been typed directly to the shell. By special dispensation, **(execute** *cmd args...*) is also permitted. Because of the surrounding parentheses, the preceding is an expression and can be abbreviated by the mechanism and notation for generalized abbreviations [see earlier]. For instance:

```
abbreviation foo1 (execute +plan equality foo - bar)
```

defines `!.foo1` as an abbreviation for **(execute +plan equality foo - bar)**. It is possible to define parameterized macros by including free variables. For example:

```
abbreviation foo2 (execute +plan equality foo - ?x)
```

leaves one parameter unspecified by replacing it with a free variable, and defines `!.foo2(?x)` which can be interpreted as a macro with one argument. Subsequently, typing `!.(foo2 bar)` will have the same effect as typing `!.foo1`; other parameters can be supplied depending on what the situation calls for.

Invoking Macro Commands

Macro commands are regular abbreviations and consequently are invoked simply by using the abbreviation notation characterized by the "`!.`" prefix, e.g. `!.(foo2 bar)`. On the other hand, it would be nice, especially for macros with arguments, if they could be invoked more like regular commands; that is by typing their name followed by the required arguments. LOGICALC lets you do something that is almost as nice:

```
xabbrev foo2 bar
```

is interpreted as if `!.(foo2 bar)` had been typed. In general,

xabbrev *cmd* arg_1 ... arg_n

is interpreted as `!.`(*cmd* arg_1 ... arg_n). Note also that, since commands can be abbreviated by a prefix [see next section], the user could simply type `x foo2 bar` which is almost as short as if the `x` had been omitted altogether.

xabbrev is a command implemented by the global command processor `logicalc`. The difference between **xabbrev** and the `!.` prefix is that the latter performs expansion at read-time, whereas the command delays the expansion process until execution time. Like the **execute** command, an **xabbrev** command may be enclosed in parentheses. Therefore, you could define the following macro:

```
abbreviation foo3 (xabbrev foo2 ?y)
```

which is not very interesting because the `foo3` macro does the same thing as the `foo2` macro. However, suppose you removed the definition of the `foo2` macro and redefined it to do something different, the `foo3` macro would automatically follow the new definition. On the other hand, if you had defined `foo3` by:

```
abbreviation foo3 !.(foo2 ?y)
```

`foo3` would keep following the old definition because expansion of `!.(foo2 ?y)` happened at read-time, instead of being delayed until execution in the case of `(xabbrev foo2 ?y)`.

4.5.3 Recalling An Earlier Command

A UNIX shell such as CSH offers a mechanism known as "history substitutions," whereby the user may use words from previous commands as portions of new commands. A typical application of this facility is to redo an earlier command exactly, or with one argument changed or corrected. Support for this feature is provided by the shell in the form of a "history" list which saves a fixed number of the most recent commands together with their command number. The notation `!`n is interpreted as a request to insert at this point the text of command n; the text may be further processed before insertion through the use of "modifiers," e.g. `!`n`:s/foo/baz/` will insert the text of command n with the first occurrence of `foo` replaced by `baz`; several modifiers can be specified.

A similar facility is available in LOGICALC. Its shell also maintains a list of the more recent commands. This list can be displayed by typing `commands`. For example, you might see:

```
[5] w> COMMANDS
   [1]    +PLAN DEDUCTION
   [2]    USE -FIND
   [3]    3
   [4]    PROOF
   [5]    COMMANDS
```

Just as in CSH, you may type `!`n to recall command n, or `!`$-n$ for the nth most recent command. What really happens is that `!`n denotes the expression `(execute` ⟨*text*

of command n⟩); thus, according to the list shown above, `!2` (or, equivalently `!-4`) denotes `(execute use -find)`.

Modifiers can also be applied, but they are not as general as in CSH. You can type `!`n`:(foo baz)` to denote the text of command n with every occurrence of `foo` replaced by `baz`. This is not "textual" substitution, but "symbolic" substitution; in particular, if the text of command n contains a symbol named `foo1`, it will *not* be replaced by `baz1`, only occurrences of the symbol `foo` will be affected.

In a modifier such as `!`n:(⟨*old text*⟩ ⟨*new text...*⟩), ⟨*old text*⟩ doesn't have to be a symbol; it can be any expression. Also, ⟨*new text...*⟩ can be composed of several expressions; the effect is to "splice" them in. For example, assuming that `!12` denotes:

```
(execute +plan equality (foo a) - bar)
```

then, `!12:((foo a) (baz a) (baz b) (baz c))` would denote:

```
(execute +plan equality (baz a) (baz b) (baz c) - bar)
```

Furthermore, it is possible to control which occurrences of ⟨*old text*⟩ are to be affected, by including as a prefix their respective indices as determined in the left to right, depth first order; e.g:

```
        !12:((foo a) (baz a) (baz b) (baz c)):1 3:(baz moe)
denotes (execute +plan equality (moe a) (baz b) (moe c))
```

4.5.4 Command Processing

As I mentioned earlier, every type of node provides its own "command processor." Thus, the shell's functionality will depend on context in a natural and intuitive way.

Furthermore, its implementation is greatly facilitated by this modularity, and the object oriented approach makes it both flexible and extensible.

In this section, I will recapitulate and/or elucidate the major ideas (structural and algorithmic) that underlie command processing.

Structuring and Segmentation of Input Lines

There are two levels of concern here:

1. **sequences of commands:** Both / and // can serve to prefix, and therefore delimit, several commands given on the same line. / has the additional effect of silencing the following command—in general this means that it won't print what information would otherwise be displayed; this is particularly convenient for moving commands, because by default they have the effect of verbosely displaying the node which has become the new focus of attention.

2. **the structure of argument lists:** It is sometimes necessary to insert explicit separators to indicate natural groupings of parameter specifications. In general '-' serves this purpose, e.g:

   cmd `foo1 foo2 - bar - baz1 baz2 baz3`

   invokes *cmd* with 3 groups of parameters. The first group `foo1 foo2` serves as argument 1, the second group `bar` as argument 2, and the third group `baz1 baz2 baz3` as argument 3.

   Certain very complex commands require yet another level of argument segmentation. In such a case, I use '--' to indicate major groupings within which '-' specifies subdivisions.

The Shell's Interpretive Loop

First, a complete line of input is read.[9] It is then segmented according to top-level occurrences of / or //, thus resulting in a list of commands which are successively and individually interpreted according to the following algorithm.

1. If the command begins with a / (resp. //), turn the silencing switch on (resp. off) temporarily. strip the prefix off and continue interpreting the rest of the command.

2. If the command is `quit`, terminate the shell and return control to the LISP interpreter.

3. Intercept stack oriented commands, e.g. **`stack`** (list stack contents in a condensed format), **pp** or 0 (display current node), and $-n$ (pop n nodes off the stack).

4. Try current node's command processor. If the processor knows how to handle the command it will either return a new node (which represents the destination if a motion was involved) or T; otherwise, it will return NIL, thereby indicating failure. Note that named commands may be specified by just a prefix; therefore, each command processor will automatically compute the completion of that prefix according to the set of commands it knows about; if the prefix is ambiguous (i.e. more than one completion is possible), the user will be prompted for additional characters until the ambiguity is resolved.

5. If the current command processor fails to handle the command, try the global command processor named LOGICALC (all command processors are named—usually after the type of nodes they cater to). It implements those commands that make sense in all contexts, e.g. **`abbreviation`**, **`execute`**, **`xabbrev`**, **`commands`**, **`history`**, **`information`**.

[9]It is also recorded in the command history.

6. Otherwise, if the command is a symbol and there is a global variable by that name, print its value; else, if it is a list whose first element is a symbol with a function definition, evaluate the expression and print the result.

7. If all else fails, display an error message.

4.6 Help

In an interactive system, a major design concern must be to achieve an effective and uniform integration of help facilities with all other aspects of the user interface. Ideally, the principle should be that wherever you may type something, you should be able to ask for help too. For a practical implementation, we must first identify the various kinds of help a user might need.

1. First and foremost, since all interactions take place through commands, the user should be able to ask what commands are available in the current context. This is all the more essential in LOGICALC since command sets vary from one type of node to another.

2. Secondly, it should be possible to request information about one particular command, à la UNIX `man`.

3. As a further refinement of the above, it is occasionally useful to list the documentation for one particular option to a given command.

4. The calling syntax for a command can be thought of as a template; the user may require help filling in that template. Here again, the needs may be broken down into several categories:

   explanations: It should be possible to ask for an explanation of what type of parameter or specification is expected.

defaults: It should be possible to omit the specification of a parameter or parameter group and have the system supply an appropriate default.

prompts: It should be possible to omit or underspecify a parameter and be prompted by the system, with a intuitively meaningful message, for more input [also, see next item]. At this point it should also be possible to ask for help.

interactive determination and construction of parameter specs: It should be possible to ask the system for help in interactively constructing the specification for a particular parameter. For example, many parameters are premises (axioms or assumptions). Clearly, if the database is large, the user will require help in searching for pertinent premises [see Chapter 7 and Appendix B]. Also, other parameters are specifications of subterms (see e.g. `equality`); here too, interactive selection can be helpful.

As we can see, the various kinds of help fall in two broad categories: (1) information, and (2) parameter determination and construction. LOGICALC implements support for both through the respective switches `-help` and `-find`; either may be typed wherever input is expected. Find Mode, invoked through `-find`, will be documented separately in Appendix B. For the moment, I will consider the information help facility built into LOGICALC.

Available Commands: `-help`

When `-help` is typed directly to the shell, the system responds with a listing of all commands available in the current context, together with their respective documentation. It is possible to control whether information about global commands (i.e. `abbreviation` through `information`) is printed by setting the global switch HELP-LOCAL-P*.

Documentation of Command: *cmd* `-help`

Lists the documentation about *cmd*. For example:

```
[2] w> ABBREVIATION -HELP
ABBREVIATION new old
                   -- define 'new' to be an abbreviation for 'old'.
                      'new' may be a symbol or an expression of the
                      form (symbol args...).
ABBREVIATION expr
                   -- show all abbreviations of 'expr'
ABBREVIATION -ALL
                   -- show all abbreviations
ABBREVIATION -REMOVE expr
                   -- for each abbreviation of 'expr', prompt the user
                      to determine whether it should be removed
ABBREVIATION -REMOVE -ALL
                   -- for each abbreviation template, prompt the user
                      to determine whether it should be removed
```

Documentation of Option: *cmd option* `-help`

Lists the documentation about the specified option. For example:

```
[3] w> ABBREVIATION -REMOVE -HELP
ABBREVIATION -REMOVE expr
                   -- for each abbreviation of 'expr', prompt the user
                      to determine whether it should be removed
ABBREVIATION -REMOVE -ALL
                   -- for each abbreviation template, prompt the user
                      to determine whether it should be removed
```

Documentation on a Plan Generator: `+plan` *plan_generator* `-help`

List the documentation about the specified plan generator. For example:

```
[4] w> +PLAN MODUS-PONENS -HELP
MODUS-PONENS (Plan Generator)

   A=>q  --->  A=>p->q, A=>p

   Using an implication whose consequent unifies with the current
   goal, attempts to prove the corresponding instance of the
   implication's antecedent. etc...
```

Getting Help at a Prompt:

When a parameter is omitted, the system will typically prompt the user for it. At that point, it is possible to type in `-help` to see the documentation for the parameter specification syntax. For example:

```
[5] +PLAN MODUS-PONENS
Major premise? -HELP

You are being prompted for a premise. You are expected to
type in a descriptor to denote premises. A descriptor may be:
...[see chapter on plan generators]
```

Another way to access information is through the `information` facility, which provides a front-end with a categorized index to available documentation.

The problem with this approach is that all commands must be designed to recognize and handle the `-help` option, not only as their first argument (in which case handling of `-help` could easily be factored out), but also after admissible options; and since each command is responsible for parsing its own argument list, it must also account for possible occurrences of `-help` at meaningful places in the argument list. This state of affairs is regrettable since providing help is an issue orthogonal to that of implementing the basic functionality. It would be nice if the two could be specified independently, or, at least, in a more decoupled fashion. However, I do not, at this point, have a more satisfactory design, and the two issues have remained conflated.

Similarly for all prompting procedures...

4.6.1 Implementation

In this subsection, I wish to afford the reader a glance at the implementation. Those not keen on such details may safely skip to the next chapter.

Each node in the browser is represented by a pair $\langle object, descriptor\rangle$, where the *descriptor* provides all the interactive functionality associated with the *object*. A *descriptor* is a 4-tuple:

$$\langle type, abbreviate, display, interpreter\rangle$$

type is a NISP type; *abbreviate* is a function that maps an instance of *type* to a small s-expression by which it can be represented when brevity is required; *display* is a procedure that produces a verbose description of an instance of *type*. Finally, *interpreter* is a procedure that takes an instance of *type* as its first argument, and a list of expressions (command line) as its second argument, and returns either a pair $\langle object_2, descriptor_2\rangle$ if this was interpreted as a motion to $object_2$, T if this was only a side-effecting command, NIL if not handled.

What I have outlined above is the structure of the browser as originally implemented in DUCK. In LOGICALC, all *interpreters* invoke a command processor that provides the additional processing (e.g. command completion) and design enhancements (e.g. help facility) discussed earlier. The *interpreter* for a goal is defined thus:

```
(defun goal-interpreter (the-goal the-cmd-line)
  (process-command goal the-cmd-line the-goal))
```

where the role of `process-command` is simply to invoke the processor named by its 1st argument, with, as parameters, the command proper (i.e. `(car the-cmd-line)`), the command's parameters (i.e. `(cdr the-cmd-line)`), and the node itself (`the-goal`)— in fact any additional arguments you wish to pass on.

A command processor can be defined by the the `define-command-processor` macro whose syntax is:

(`define-command-processor` ⟨*name*⟩ (`cmd args node`) ⟨*clauses...*⟩)

where each clause is one of the following:

(⟨*command name*⟩ ⟨*command code ...*⟩)

This defines a named command. However, the preferred way to define named commands is incrementally using the `datafun` construct as illustrated later in this section.

(⟨*test*⟩ ⟨*code ...*⟩)

A command can be recognized by an arbitrary ⟨*test*⟩; in which case, the corresponding ⟨*code ...*⟩ is executed.

(`-else` ⟨*code ...*⟩)

A default action may be specified for the case when the command expression is not recognized by any other clauses.

The semantics is this: if the command is `-help` then provide information about all commands available in the current context. Otherwise, all ⟨*test*⟩ clauses are attempted first in the order in which they were specified. If they all fail to recognize the command, then an attempt is made to identify it with one of the named commands known to this processor—modulo the prefix expansion business. If that too fails and there is an `-else` clause, then that clause is finally tried.

The command processor for a goal node is defined thus:

```
(define-command-processor goal (cmd args node)
  ((integerp cmd)
   (_goal-goal-command cmd (list cmd) node))
  (-else
   (let ((pg (disabbreviate-plan-generator cmd)))
     (and pg (+plan-goal-command '+plan (cons pg args) node)))))
```

The first clause shows how an integer can be interpreted as a motion command: a ⟨*test*⟩ clause intercepts all integers and directly calls the function implementing

motion to a subgoal. The **-else** clause is rather interesting: since invoking plan generators is probably the major activity at goal nodes, it is convenient to be able to omit the **+plan** command altogether and simply name the plan generator directly. This is what the **-else** clause does for you: if the command was not otherwise recognized, maybe it is a plan generator (or prefix thereof), in which case it should be invoked as if **+plan** had been typed.

Other named commands can be incrementally defined through the following syntax:

```
(datafun ⟨processor⟩-command ⟨command name⟩
  (defun (cmd args node) ⟨code...⟩))
```

For example, the command to move from a goal node to its class is defined thus:[10]

```
(datafun goal-command class
  (defun (cmd args node)
    (match-cond args
      (\? (?"-H|ELP" . ??ignoring)
          (out 0 cmd (t 15) " -- move to goal's class" t) t)
      (t (ignoring args)
         (make gwalkstate class-descrip*¹¹ (goal-class goal))))))
```

This example illustrates 2 things: first, every single command must anticipate possible occurrences of the option `-help` and be prepared to respond appropriately. Secondly, the vertical bar in `"-H|ELP"` indicates the minimal prefix to the matching code.

I am dissatisfied with the obligation imposed on each command to parse its own arguments. Since there is a lot of commonality, I have abstracted quite a bit of functionality into various macros and auxiliary functions, but it still feels like there ought to be a better way, a general parsing interface. It is very possible that an

[10]There is a little white lie in the definition, because one doesn't deal with goal nodes directly but with goal views.

[11]Descriptor for class nodes.

analysis of the needs and requirements of the current implementation could lead to the design of compositional parsing tools that would simplify and unify command definition. Unfortunately, I have not had the time to look into this.

Chapter 5

Implementation

In this chapter I describe the data-structures for the internal representation of a proof graph. They capture the notions of sets of assumptions, goals and goal classes, plans, proofs and answers.

5.1 Sequents and Assumption Sets

LOGICALC is based on a system of natural deduction operating on a sequent logic, with unification built in as a fundamental operation. Proofs and goals (i.e. theorems to be proven) are represented by sequent-like objects which, for convenience's sake, we both denote by expressions of the form $A \Rightarrow p$, where A is a set of assumptions and p is either the conclusion or the goal formula.

Natural-deduction proofs occasionally require making assumptions local to a sub-proof. Consider for instance a goal of the form $A \Rightarrow p \supset q$. The natural approach is to invoke the deduction theorem, assume the implication's antecedent p and try to prove the implication's consequent q under this additional assumption. In other

words, the new goal has the form $A, p \Rightarrow q$, where A, p is the traditional abbreviation for $A \cup \{p\}$, and p is now an assumption local to the subgoal.

In the course of a proof, hierarchical sets of assumptions will thus naturally arise. Therefore, we require an implementation of assumption sets that will reflect this hierarchical structure and efficiently support multiple assumption sets. There should be minimal overhead involved with the creation of new assumption sets; in particular, copying of contents should not be required. It should be easy to switch between different assumption sets, and querying should have uniformly good performance. I have argued earlier that datapools satisfy those requirements quite nicely.

Datapools are described at length in [McD79]. They can be created in a hierarchical manner by means of the function `DP-PUSH`. Given a datapool P, `(DP-PUSH P)` creates a new datapool whose parent is P. The new datapool initially inherits the assertions of P (no copying is involved), but is independently side-effectable and modifications made to it do not affect P. In particular, it is possible to add assertions to the new datapool that were not (and will not be visible in) its parent. This is how local assumptions are implemented; a new datapool is created by `DP-PUSH` and the local assumption is asserted in it.

Assumption sets are implemented as data structures with the following 4 slots:

Assumption: the formula which is the local assumption

Assertions: making the above assumption results in the creation of one or more assertions [see below]; these are their names.

Datapool: where the assumption is being made.

Parent: the assumption set without the local assumption.

When an assumption such as a conjunction is being made, it is often convenient to pretend that each conjunct is being asserted separately. The procedure `ASSUME`, which

is the one that actually inserts the local assumption in the new datapool, recognizes all common cases such as conjunctions, double negations, negated disjunctions, etc... and asserts the components separately; it returns the list of the resulting assertions, which is then stored in the corresponding slot of the assumption set.

In the introduction, there was a goal of the form:

```
(IF (AND (IS CREATURE !.C)
         (IS THUMB !.ONE-THUMB)
         (PART !.ONE-THUMB !.C))
    (AND (IS THUMB ?OTHER-THUMB)
         (PART ?OTHER-THUMB !.C)
         (NOT (= !.ONE-THUMB ?OTHER-THUMB))))
```

After invoking the deduction theorem the new datapool contains the following local assertions (this is taken from the display of the goal on page 13):

Assumptions:

```
   (IS CREATURE !.C) -- !:IS278
   (IS THUMB !.ONE-THUMB) -- !:IS279
   (PART !.ONE-THUMB !.C) -- !:PART280
```

As you can see, each conjunct resulted in a different assertion; the funny name to the right of each formula (e.g. `!:IS278`) is the name of the assertion. It is automatically made up by the system on the basis of the principal symbol in the formula and the value of a monotonically increasing global counter; this is why the 3 consecutive assertions are named `!:IS278`, `!:IS279`, and `!:PART280`. The reason for the strange '`!:`' prefix is that these names are not really symbols but symbol-like structures which become "interned" in some sense only when they are printed; they can essentially be used like symbols but they are cheaper because, as long as they are not printed, they can be garbage-collected.

Assumption sets are weakly "uniquified" in the following sense: the assumption set constructor checks whether there already exists an assumption set with the same

parent and same assumption slots; if one is found it is returned; otherwise a new one is constructed and added to the list of existing assumption sets. I use the qualifier "weakly" because assumptions are compared with a simple syntactic equality test rather than with some more general subsumption check.

As I mentioned earlier, the natural-deduction way of solving a goal of the form $A \Rightarrow p \supset q$ is to attempt a proof of the subgoal $A, p \Rightarrow q$. However, once such a proof has been derived, the Deduction-Theorem inference rule must still be invoked via a validation in order to produce a proof of the original goal.

The Deduction-Theorem rule "discharges" a local assumption. The way it works is really quite simple: it is given a proof of $A, p \Rightarrow q$, where A, p is an assumption set whose local assumption is p and whose parent is A, and it returns a proof object for $A \Rightarrow p \supset q$.

5.2 Goals and Goal Classes

5.2.1 Goals in Brief

Goals are sequent-like objects which I usually depict using the notation $A \Rightarrow p$, where A represents the current set of assumptions implemented as a datapool, and p is the formula which is to be shown follows logically from A (actually: that there exists some instance of p which logically follows from A). Although both proofs and goals are sequent-like objects, both frequently depicted with the same notational device, I believe that they are distinct notions and should not be conflated. Others have sometimes introduced the idea of partial proofs; I do not pursue this avenue; instead I chose a goal/plan based approach. Here is the display of a goal taken from the extended example of the introduction chapter:

```
Goal View      <class CLASS.28>
   Find: (T.3) in:
      (PART ?T.3 !.ONE-HAND)
   Assumptions:
      (= !.ONE-THUMB !.OTHER-THUMB) -- !:=305
   Answers:
      1  T.3          = !.ONE-THUMB
   Supergoal:
      (= !.ONE-HAND !.OTHER-HAND)
   (no local plans)
   (no class plans)
```

When the user initiates a session with LOGICALC, he is asked to state the theorem to be proven. This theorem is then presented as a goal. In order to solve a goal, the user will typically invoke a plan generator which will result in one or more plans being created and indexed under the goal as alternative ways of attempting the proof. Sometimes other plan generators may be invoked on the same goal to generate further plans; for instance if those available so far have all proved unsuccessful, or a better way occurred to the user.

Thus, goals are refined into plans with subgoals and so on recursively in some form of and-or tree where plans are and-nodes (conjunctions of goals) and goals are or-nodes (disjunctions of plans). The process essentially bottoms out when a goal can unify with one (or more) assumptions (or axioms), or otherwise aquire an answer.

5.2.2 Goal Classes

It is pretty common for similar goals to occur in different parts of the proof tree. For instance, there might be some plan with a step of the form $A \Rightarrow p(x)$ and some other plan with a step of the form $A \Rightarrow p(y)$; these steps are equivalent modulo a renaming of variables. When one has been solved, it should not be necessary to duplicate the work in order to obtain a solution for the other.

The notion of a goal class was introduced as a bookkeeping device that would support the sharing of answers between goals. A goal class is an equivalence class for goals; goals are simply instances of an equivalence class. Plans and answers are actually recorded with the class, not with the goal. When the user is given the impression of working on a plan to solve a goal, he is really working on a plan to solve the corresponding class, but, by working *through* the goal, he is granted the benefit of certain cosmetic adjustments which sustain the illusion that he is working on solving that very goal itself.

So why can't we do away with goals altogether and use classes directly as steps in plans? The reason is that variable names must be consistent across all steps of a plan. Suppose there already exists classes for goal-sequents $A \Rightarrow p(x)$ and $A \Rightarrow q(y)$, and further suppose that we want to create a plan consisting of the steps $A \Rightarrow p(z)$, $A \Rightarrow q(z)$. It is clear that transforming the plan into $A \Rightarrow p(x)$, $A \Rightarrow q(y)$ would not preserve its (computational) meaning. A goal essentially acts as a view of its class through the renaming required by the plan of which it is a step.

Here is how a class is implemented:

name: a symbol such as `CLASS.28` that uniquely identifies the class.

$A \Rightarrow p$: the sequent representative of what is to be proven.

answers: all answers found so far for this class. Each answer is little more than a packaging of a proof for the above sequent.

plans: all plans proposed so far for this class. Not only may different plans have been tried in successive attempts at deriving an answer for this class, but different plans may be appropriate for different goals (different instantiations may be desired).

members: a list of all the member goals in this class.

stronger: a list of all the classes which are more general than this one.

weaker: a list of all the classes which are less general than this one.

Related Classes

The last two fields require a little explanation. They serve to maintain links to "related" classes. The rationale is that a class may occasionally be able to adapt an answer from a related class and transform it into an answer for itself. Therefore, it is convenient to keep these links around so that answers may be "broadcast" along them to other classes that may also benefit from them.

Consider a class $\mathcal{C}$ defined by $A \Rightarrow p$ and a class $\mathcal{C}'$ defined by $A' \Rightarrow p'$. $\mathcal{C}$ is said to be *stronger* than $\mathcal{C}'$ if it attempts to prove a more general statement. In principle, this more or less means that p subsumes p' and $A \subseteq A'$, i.e. $\mathcal{C}$ makes fewer assumptions than $\mathcal{C}'$. In practice, true subsumption is too expensive and hard to check, and LOGICALC restricts the notion of a stronger class to the case where p' is an instance of p and A is an ancestor of A' in the assumption set hierarchy; both these conditions are easy to check.

So, why is it a good idea to keep links between related classes? Consider the two classes $A \Rightarrow p$ and $A \Rightarrow p'$ where p' is an instance of p, i.e. $p' = \sigma_1 p$. Clearly, any answer $A \Rightarrow \sigma_2\sigma_1 p$ to the second class is also a valid answer to the first one.

Conversely, consider the two classes $A \Rightarrow p$ and $A \cup B \Rightarrow \sigma_1 p$, where the first one is more general than the second one, and suppose an answer $A \Rightarrow \sigma p$ is found for the first one. In the case where there exist σ_2 such that $\sigma = \sigma_2\sigma_1$ (composition of substitutions), then the answer $A \Rightarrow \sigma_2\sigma_1 p$ can be adapted (by assumption introduction) into an answer for the second class: $A \cup B \Rightarrow \sigma_2\sigma_1 p$.

Since every class has to figure out for itself whether it can adapt an answer received through the broadcasting apparatus, it might be simpler to merge the lists of stronger/weaker classes into a single one of related classes. I originally thought that maintaining this distinction might be a useful thing to do for the benefit of the user (who might want to prove a more general goal for instance); however, in practice this distinction is never taken advantage of by the user. It survives because there was no good reason to remove it either, and, who knows, it might yet turn out to be profitable in some circumstances.

Creation

When a goal is created (at the top level or as a plan step), a corresponding class must be found in which to insert it. This is accomplished by the function `FIND-CLASS` which is given a sequent $A \Rightarrow p$ as argument. First, `FIND-CLASS` determines whether there is an already existing class for this sequent as follows: it queries the global database for assertions of the form `(GOAL_CLASS` $\langle name \rangle$ p`)`; for each class there is an assertion of this form that relates its $\langle name \rangle$ to its pattern. We want to find those which at the very least unify with p. For each candidate, we can recover the original class through the `GOAL_CLASS` property of its $\langle name \rangle$. We then compare $A \Rightarrow p$ to the recovered class $A' \Rightarrow p'$. If $A = A'$ and $p = p'$ (modulo renaming variables), then `FIND-CLASS` will return this class.

If no appropriate equivalence class is already in existence, we must construct one. Its "stronger" and "weaker" slots are initialized with related classes identified by a process similar to the one described above; each one of these related classes is also updated to record the correct back-pointer to the new class. The new class is recorded on the `GOAL_CLASS` property of its name, and a global `GOAL_CLASS` assertion is made that relates its name to its pattern p. Note that these side-effecting bookkeeping tricks force LOGICALC to do some amount of cleaning whenever a new session is initiated.

Initialization

After a class has been created, it goes through a process of initialization whose purpose is to find answers to this new class, presumably in the hope that the user won't have to be bothered. There are 3 ways answers can be acquired:

1. Consult all related classes and attempt to borrow and adapt their answers.
2. Find all trivial answers. First, check whether the class's pattern can be regarded as a tautology of propositional calculus. If such is the case, then the class is considered to be solved (with a *yes* answer [see later]). Otherwise, find all assertions that unify with the pattern.
3. Find all answers that can be derived by back-chaining. For instance, LOGICALC has a built-in theory of types implemented by DUCK back-chaining rules which does a really good job of automatically solving `IS`-goals.

Method 1 is always attempted. Whether method 2 (resp. method 3) is attempted is controlled by switch **`class-init-fetch-nobc*`** (resp. **`class-init-fetch*`**). Thus classes, and a fortiori goals, may acquire answers without user intervention.

5.2.3 Goals Revisited

As I mentioned earlier, a goal is an element of a goal class. Its principal function is to provide a renaming between the variables of its class and those required by the plan of which it is a step.

Remember however that a class may have many plans associated with it, not all of which are relevant to the current goal. Therefore, the duties of a goal are slightly extended to include performing a little contextual packaging. In other words, a goal

also serves to record information directly relevant to it (as a step in its plan, or a subgoal of its supergoal) and suppress the extraneous stuff associated with its class.

For instance, when the user views a goal, he sees only those plans that have been explicitly associated with this goal; all the other plans of the class remain hidden, eventhough they are all available and can selectively be brought to the foreground with a command.

Also, whenever an answer if followed from a goal, it is useful to record with that goal what the result of following said answer was; typically the result would be the construction of a successor plan or the derivation of a proof of the supergoal.

Here is how a goal is implemented:

class: the goal's class.

table: the correspondence between variable names in the class and variable names in the goal's plan.

plans: a subset of the class's plans. A plan is included in this set either as a result of invoking a plan generator and answering "yes" to the question "make it local?" or by explicitly requesting it from the class (`^^plan` command). Plans can also be removed from this list (`-plan` command).

superplan: the plan this goal is a step of. This goal's supergoal is the goal that the superplan is attempting to solve. The top-level goal has no superplan.

rank: an integer numbering this goal as a step within its superplan. This number is used by the superplan's validation to refer to a proof of the corresponding step.

links: each link is of the form (⟨*answer*⟩ ⟨*fate*⟩) where ⟨*answer*⟩ is an answer that was followed from this goal, and ⟨*fate*⟩ is what it led to and is of the form (⟨*status*⟩ ⟨*successor*⟩); ⟨*successor*⟩ is either a successor plan to the goal's superplan, or an answer to the goal's supergoal. ⟨*status*⟩ indicates whether the actual fate was found to be subsumed by an already existing fate, in which case ⟨*successor*⟩ is really the subsuming fate.

5.3 Plans

The preferred method to solve a goal is to reason backwards and progressively refine goals into subgoals until the process bottoms out when answers are found by unifying goals with assumptions or axioms. The process then unravels upwards producing proofs validating each successive refinement.

The notion of refining a goal into subgoals is captured by the concept of a plan. A plan is in general constructed as a result of the user invoking a plan generator. For instance, invoking the deduction theorem plan generator on the initial goal of the introductory example resulted in the creation of the following plan:

```
Plan View      <class CLASS.1>
   Documentation: "w> +PLAN DEDUCTION-THEOREM"
   Supergoal:
      (IF (AND (IS CREATURE !.C) (IS THUMB !.THUMB1) (PART !.THUMB1 !.C))
          (AND (IS THUMB ?THUMB2)
               (PART ?THUMB2 !.C)
               (NOT (= !.THUMB1 ?THUMB2))))
   Will find:
      THUMB2      = ?THUMB2
   Steps:
      1  ((IS CREATURE !.C)
          (IS THUMB !.THUMB1)
          (PART !.THUMB1 !.C)
          => (IS THUMB ?THUMB2)) -- (no local plans, no class plans, 1 answer)
```

```
    2  (|...| => (PART ?THUMB2 !.C)) -- (no local plans, no class plans, 1 answer)
    3  (|...| => (NOT (= !.THUMB1 ?THUMB2))) -- (no local plans, no class plans, no answers)
  (no successors)
```

The user then proceeds to solve the plan's steps. Whenever a step has acquired one or more answers, the user may elect to *follow* one of them, which typically results in a *successor* being constructed. The user then proceeds to deal with the successor's remaining steps until all steps have thus been eliminated; at which time, the plan's *validation* is executed to produce a proof of the plan's supergoal from the corresponding proofs of its steps.

5.3.1 Validations

No validation was shown in the display of the plan above because I find that such information unnecessarily clutters my screen, and I set up my environment so that validations remain hidden. This preference is entirely customizable by the user. Here is the short story about validations:

Definition

A validation is an expression of the form:

(*inference_rule* (*premises*...) (*parameters*...))

where *inference_rule* is the name of the inference rule validating this refinement, each *premise* is either an integer n denoting a proof of the plan's n^{th} step (i.e. of rank n), or another validation denoting the proof resulting from its recursive evaluation, and the *parameters* are arbitrary expressions further specifying the inference to be derived. For example:

```
(MODUS-PONENS (0 (AND-INTRO (1 2) ())) ())
```

is a validation which means to conjoin a proof of step 1 with a proof of step 2 by *and-introduction*, then to use the resulting conclusion as minor premise and a proof of step 0 as major premise to derive a conclusion by *modus-ponens.* Assuming the proofs of these steps are respectively $A \Rightarrow p_1 \wedge p_2 \supset q$, $A \Rightarrow p_1$, $A \Rightarrow p_2$, executing the validation results in a proof of $A \Rightarrow q$.

Recursive Validations

Why allow a $\langle$*premise*$\rangle$ to be a validation? The idea of recursive validations is not just for generality. When a simple plan generator such as "modus-ponens" is invoked, it can be expected to have a trivial validation expressed in term of the inference rule "modus-ponens." However, plan generators can be arbitrarily complex. Consider for instance an hypothetical plan generator that would take a conjunctive goal with nested conjunctions and *flatten* it out into individual steps. The goal $(p_1 \wedge p_2) \wedge p_3$ would be refined into a plan with the three steps p_1, p_2, p_3 and a validation of the form:

```
(AND-INTRO ((AND-INTRO (1 2) ()) 3) ())
```

In fact, something very much like this happens automatically whenever a new plan is proposed by a plan generator; the system performs "automatic expansion" on the plan's description, e.g. conjunctive goals are refined into separate steps, double negations are stripped off, etc... These automatic refinements (not explicitly requested by the user or implied by the plan generator) introduce recursive validations.

Another source of recursive validations is the "detaching" mechanism by which premises for plan generators are often obtained. I will not go into any details here, but the

idea is that when a plan generator expects a premise as input, the user may instead point to a subformula in an assertion and let the system figure out what ought to be done in order to infer this subformula from the indicated assertion. This will result in some kind of subplan, with a validation determined by the system.

5.3.2 Successor Plans

As I mentioned earlier, a successor plan is created as a result of following an answer to a plan's step. Within a plan the complete set of steps is partitioned in two subsets: the solved steps and the remaining steps. The solved steps are those steps of the original plan for which answers have been found and followed.

A solved step is represented by a pair $\langle answer, goal \rangle$ associating the goal that was solved with the answer that was selected to solve it. Occasionally a solved step may also be of the form $\langle answer, n \rangle$, where n is the rank of the corresponding step; this only happens when step n could be determined to have a "yes"[1] answer at the time the original plan was initialized; in such a case, there is no reason to actually construct a representation for this step; instead, it is abbreviated by its rank.

When an answer $\mathcal{A}$ to a step $\mathcal{S}$ is followed, a new plan instance is created. The remaining steps of the new plan are computed as follows: for each remaining step of the previous plan except $\mathcal{S}$, create a new goal by instantiating the step's pattern according to the substitution as specified in $\mathcal{A}$. Doing so may involve creating new classes. The solved steps in the new plan are simply those of the old plan to which we add $\langle \mathcal{A}, \mathcal{S} \rangle$. The new plan $\mathcal{P}$ is then initialized according to a procedure that I will describe later.

$\mathcal{P}$ is described as a *fate* of $\mathcal{A}$. A note is made in $\mathcal{S}$ that following $\mathcal{A}$ from $\mathcal{S}$ resulted in the creation of $\mathcal{P}$. Such a note is called a link and is added to $\mathcal{S}$'s set of links. I

[1] a yes answer is an answer that binds no variable; this is as general as an answer can get.

will postpone discussing links further until plan initialization has been presented.

One plan may have many successors acquired either by following different answers to the same step or answers to different steps. Typically, however, there will be one straightforward sequence of successor plans ultimately leading to a fate that is an answer to the supergoal.

Although I will not discuss aspects of the user interface in this chapter, I would like to mention that each plan has a documentation string associated with it; it is by default a representation of what the user typed in to create the original plan. Each successor plan inherits the documentation string from its predecessor prepended with the string ">" for visual feedback.

5.3.3 Implementation

Here is a description of the data-structure implementing a plan:

class: the class for which this plan attempts to derive a proof.

alist: typically, certain variables in the class have already been instantiated by terms as a result of creating the plan in the first place. For instance, if the class is $A \Rightarrow q(x)$ and we back-chain by modus ponens using an axiom of the form $p \supset q(f(a))$, the resulting plan is known to bind x to $f(a)$.

All free variables in the class have an entry in this alist. Some entries simply indicate by what name a free variable of the class is known in the plan.

This alist must also be instantiated according to the appropriate substitution when an answer to a step is followed and a successor plan is constructed.

remaining steps: the plan's remaining steps.

solved steps: the plan's solved steps.

validation: an expression describing how to infer a proof of the supergoal from proofs of the plan's steps.

5.3.4 Initialization

After a plan has been constructed, it must be initialized. If the supergoal already has a "yes" answer $\mathcal{A}$, there is no need to search for another answer; therefore, the plan is deemed superfluous and is deleted, and a fate of the form ⟨`plan-subsumed-by-answer`, $\mathcal{A}$⟩ is returned.

Otherwise, the plan's steps are initialized, which may involve initializing their classes (i.e. finding trivial answers). I know you may have been led to believe that class initialization happened immediately after class creation, but it really happens now. When this is done, some steps may have acquired "yes" answers. Since these steps have already been solved in the most general way possible, they ought to be considered solved once and for all; therefore it is clear that they should be put in the "solved" list. Also, since their representations will never be used, instead of putting them in the form $\langle answer, goal\rangle$ in the solved list, we can do better and insert $\langle answer, n\rangle$, where n is the corresponding rank. Thus, their representations, which are now no longer required, can be garbage collected.

Let us first suppose that not all steps have thus been solved: there are a few remaining steps. It must now be determined whether the plan is redundant. For instance, it is likely that by following answer $\mathcal{A}_1$ to step 1 and then answer $\mathcal{A}_2$ to step 2 we will get a plan similar to the one we would obtain by following $\mathcal{A}_2$ first, then $\mathcal{A}_1$. The redundancy check will take care of this case. If a plan $\mathcal{P}$ is found that is deemed

to render the new plan redundant, the new plan is deleted and a fate of the form $\langle$**plan-subsumed-by-plan**, $\mathcal{P}\rangle$ is returned.

Another case of redundancy is when the superclass has an answer $\mathcal{A}$ whose substitution is more general than the one determined by the plan's alist. In such a case, if the plan leads to a conclusion, it will be necessarily less general than the one available in $\mathcal{A}$. The plan is considered pointless and a fate of the form $\langle$**plan-subsumed-by-answer**, $\mathcal{A}\rangle$ is returned.

This expectation of redundancy was quite reasonable in the earlier design of LOGICALC. The more recent design includes a proof generalizer which might generalize the conclusion expected from the plan into one that is more general than $\mathcal{A}$'s. The redundancy check does not take this possibility into consideration and still functions according to the old regime. I do not believe this to be a serious problem.

The redundancy check not only examines the superclass, but also its related weaker classes: a plan may be redundant because of an existing plan for a weaker class.

If the plan $\mathcal{P}$ is not found to be redundant, then it is "pushed" according to a procedure which I will describe next; this results in a list of fates; to this list is added a fate of the form $\langle$**plan**, $\mathcal{P}\rangle$ and the result is returned.

Now, let us consider the case where, after initialization, all steps are discovered to have been solved. Then, a conclusion is produced by evaluating the plan's validation with the list of solved steps to pick the argument proofs from. Thus an answer $\mathcal{A}$ to the superclass is derived and broadcast, and we return a fate of the form $\langle$`conclusion`, $\mathcal{A}\rangle$. If however the answer $\mathcal{A}$ is found to be subsumed by a preexisting answer $\mathcal{A}'$, then $\mathcal{A}$ is discarded and a fate of the form $\langle$**conclusion-subsumed-by-answer**, $\mathcal{A}'\rangle$ is returned.

5.3.5 Pushing a Plan

As a result of earlier work and/or through initialization all steps may have acquired answers, although not necessarily "yes" answers. Sometimes, completing the plan as it stands then is simply a matter of the user successively selecting and following a particular answer for each step. LOGICALC alleviates this tedium to some extent by "pushing" plans.

When all steps of a plan have answers, it is possible that by picking one answer for each step we may be able to derive a conclusion for the whole plan. All that is required for this to happen is that the selected answers be compatible. Here is what compatible means: the selected answers $\mathcal{A}_1 \ldots \mathcal{A}_n$ define corresponding substitutions $\sigma_1 \ldots \sigma_n$. These are compatible if there exists a substitution σ such that:

$$\sigma\sigma_1 = \ldots = \sigma\sigma_n$$

If $p_1 \ldots p_n$ are the plan's steps, then $\epsilon \stackrel{\text{def}}{=}$ `(plan` $p_1 \ldots p_n$`)` is an expression representing the plan's sequence of steps, and σ is the most general unifier of $\{\sigma_1\epsilon, \ldots, \sigma_n\epsilon\}$. Pushing a plan involves trying all possible combinations of answers, and for each one for which there exists a unifier σ a conclusion is produced.

My implementation is the straightforward one that picks answers left to right checking compatibility incrementally after each selection. Thus not all combinations have to be explored: when an answer for step k is picked that is incompatible with those selected so far for step 1 through $k-1$, there is no need to make any selection to the right of k; instead another answer must be picked for step k.

Conclusions are derived according to the method described next. For each conclusion a fate is produced. The collected list of all these fates is returned.

5.3.6 Producing a Plan's Conclusion

As a result of pushing a plan or following answers to its steps, there comes a time when an answer has been selected for each step and a conclusion must be derived for the plan's superclass. This conclusion is constructed according to the plan's validation.

As I explained earlier, a validation is an expression of the form:

(*inference_rule* (*premises...*) (*parameters...*))

which is interpreted as follows: the named *inference_rule* is called with a list of premises and a list of parameters as arguments. Each *premise* is either an integer i, in which case it denotes the corresponding proof of step i to be looked up in the list of selected answers, or it is a (recursive) validation, in which case it denotes the conclusion to be obtained by recursively interpreting this validation in the same manner.

The parameters are arbitrary expressions which further specify the inference to be made. For example, equality substitution requires that a path be provided to denote the subterm to be substituted by equality.

When an answer has finally been constructed, the algorithm must determine whether it is redundant or not. An answer $\mathcal{A}_1$, whose corresponding substitution is σ_1, is redundant if there already exists an answer $\mathcal{A}_2$ to the same class whose substitution is σ_2 and such that there exists σ satisfying $\sigma_1 = \sigma\sigma_2$. In other words: the class already has a more general answer. $\mathcal{A}_1$ is said to be subsumed by $\mathcal{A}_2$.

If $\mathcal{A}_1$ is not redundant, a fate of the form $\langle$`conclusion`$, \mathcal{A}_1\rangle$ is returned. If $\mathcal{A}_1$ is subsumed by $\mathcal{A}_2$, $\mathcal{A}_1$ is discarded and a fate of the form $\langle$`conclusion-subsumed-by-answer`$, \mathcal{A}_2\rangle$ is returned.

5.3.7 A Goal's Links

In conclusion to this section, I want to come back to the notion of a link. Whenever an answer $\mathcal{A}$ to a goal is followed, a list of fates is produced. For each fate $\mathcal{F}$, a new link $\langle \mathcal{A}, \mathcal{F} \rangle$ is added to the goal's links. A fate must be one of the following:

$\langle$`plan`$, \mathcal{P}\rangle$

Following $\mathcal{A}$ produced a successor plan $\mathcal{P}$ which was not redundant.

$\langle$`plan-subsumed-by-plan`$, \mathcal{P}\rangle$

The successor plan was found to be subsumed by the preexisting plan $\mathcal{P}$, and was discarded.

$\langle$`plan-subsumed-by-answer`$, \mathcal{A}'\rangle$

It was noticed that the successor plan could not produce a conclusion more general than the one already available in answer $\mathcal{A}'$. The successor plan was discarded.

$\langle$`conclusion`$, \mathcal{A}'\rangle$

After following $\mathcal{A}$ either there were no remaining steps or they all had acquired answers either by initialization or through "pushing," and a conclusion was produced for the superclass. The corresponding answer $\mathcal{A}'$ was found to be non redundant.

$\langle$`conclusion-subsumed-by-answer`$, \mathcal{A}'\rangle$

Similarly a new answer was produced which was found to be subsumed by the preexisting answer $\mathcal{A}'$. The new answer was discarded.

5.4 Proofs

A proof is an aggregate comprising a conclusion sequent and a complete trace of the sequent's derivation in the form of a proof tree—thus, it is an instance of a recursive datatype. A proof is usually depicted using the notation $A \Rightarrow p$, where A represents a set of assumptions and p is the conclusion which has been shown to logically follow from this set. For the introductory example [pp. 7–38], the proof of the ultimate conclusion was displayed as follows:

```
Proof
   Conclusion:
      (IF (AND (IS CREATURE ?Y) (IS THUMB ?X) (PART ?X ?Y))
          (AND (IS THUMB !.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y))
               (PART !.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y) ?Y)
               (NOT (= !.ONE-THUMB !.OTHER-THUMB))))
   (no local assumptions)
   Validation:
      (IF-INTRO (1 2) ())
   follows from:
      1  (AND (IS CREATURE !.C) (IS THUMB !.ONE-THUMB) (PART !.ONE-THUMB !.C))
      2  (AND (IS THUMB !.OTHER-THUMB[3](!.ONE-HAND[5](!.C !.ONE-THUMB) !.C))
              (PART !.OTHER-THUMB[3](!.ONE-HAND[5](!.C !.ONE-THUMB) !.C) !.C)
              (NOT (= !.ONE-THUMB
                      !.OTHER-THUMB[3](!.ONE-HAND[5](!.C !.ONE-THUMB) !.C))))
```

Although I often choose to denote a proof by a sequent $A \Rightarrow p$, a proof captures more information than just such a conclusion; it is really a node of an inference tree (actually a *dag*). Thus, the graph makes available a trace of the whole computation.

5.4.1 Where Proofs Come From

The most elementary kind of proof is an axiom: an axiom is considered to be a proof of itself. As a matter of convenience, it is truly the name of the assertion representing an axiom which is taken to constitute a proof of same. Thus, in the

introductory example, the symbol `transitivity` is a proof of the axiom of transitivity of parts. Local assumptions are also elementary, but I will postpone discussing their representation until later.

Most other proofs come from invoking an inference rule. The only other case consists of conclusions derived through backward-chaining rules of logic programs implementing built-in theories (such as LOGICALC's simple theory of types, or of tuples and lists); these will be discussed in the closing paragraph of this subsection. In LOGICALC, inference rules are functions which capture computationally the meaning of their logical counterparts. As a function, an inference rule takes two arguments: a list of sequents and a list of parameters that further specify the inference to be made (such as a path to a subterm to be substituted by equality); it returns a list of conclusion sequents. The system then takes care of packaging these conclusions as proofs which record all the relevant information.

The common case is that a proof will be constructed as a result of executing a plan's validation, thus producing a conclusion for the plan's superclass. Proofs can also be derived by performing explicit forward inferences by means of inference generators.

The Justification for Plan Validations

Why are plans validated? For instance, in a PROLOG system, goals are also refined into subgoals in a backward chaining manner, but, once all subgoals have been satisfactorily resolved, there is no need for an additional process of validation to be performed bottom-up. There are 4 reasons for having such a process in LOGICALC: (1) to double check the validity of the plan's refinement, (2) to perform some post-processing, after the plan is completed, that is necessary but could not be done ahead of time, (3) to make the conclusion available to subsequent deductive operations, and (4) to package the conclusion as an object available for inspection.

Double-Checking the Validity of the Inference Step

Plan generators are arbitrary procedures which, given the current goal, some premises and some parameters, create a plan object that we then interpret as a refinement of the goal. PROLOG search happens as if directed by a built-in "modus-ponens" plan generator; each refinement is guaranteed to correspond to a valid inference; therefore an explicit validation mechanism would be completely superfluous. There is no such guarantee in LOGICALC. A plan generator is no more than a function which maps a goal to a pair consisting of a list of subgoals and a validation [Mil79a,Pau87]. Once all subgoals have been solved, the validation is executed in order to produce a proof. In principle, executing the validation might fail to produce a proof and no conclusion could be drawn from the completed plan, or it might produce a proof that would not unify with the formula of the plan's superclass and would thus fail to solve it.

Consider for instance a plan generator that would take a goal of the form $A \Rightarrow p(x) \wedge q(x)$ and would propose a plan with the two steps $A \Rightarrow p(y)$, $A \Rightarrow q(z)$, where x, y and z are distinct free variables, and the validation `(AND-INTRO (1 2) ())`. Suppose further that step 1 acquires a proof $A \Rightarrow p(a)$ and step 2 $A \Rightarrow q(b)$, where a and b are distinct constants. Executing the validation will result in the proof $A \Rightarrow p(a) \wedge q(b)$ which won't unify with the original goal. The plan failed to fulfill its contractual agreement to solve the goal when completed.

By design, the existing plan generators produce plans which are expected to solve their supergoal on completion. Executing the validation *will* return a proof which *will* unify with the supergoal. LOGICALC will signal an error when this expectation fails.[2]

[2]which hopefully should never happen.

Post-Completion Processing

Given the preceding design decision and assuming that the implementation is bug-free, is there any other reason to retain the notion of a validation process? The answer is "yes." The reason is that some really tricky plan generators cannot do all the work in advance, they require that some further processing happen when the plan has been completed.

Consider solving $p(a) \vee p(b) \Rightarrow p(x)$, where x is a free variable and a and b are distinct constants, by case analysis. The resulting plan has the two steps:

$$\begin{array}{lcl} p(a) \vee p(b), p(a) & \Rightarrow & p(x_1) \\ p(a) \vee p(b), \neg p(a), p(b) & \Rightarrow & p(x_2) \end{array}$$

where x_1 and x_2 are distinct free variables. Clearly we obtain the two proofs $p(a) \vee p(b), p(a) \Rightarrow p(a)$, and $p(a) \vee p(b), \neg p(a), p(b) \Rightarrow p(b)$, and the question is *"what should x be bound to?"* The answer, which I won't detail here, involves making up a new constant c which is defined to be equal to a in the first case and to b in the second case. Clearly, this cannot be done until bindings have been found for x_1 and x_2.

Making the Conclusion Available

After a conclusion has been derived, it is made publicly available through two channels. First, the corresponding answer is broadcast from its class to all related classes. Classes that will be constructed in the future may also benefit from this answer by borrowing it from some related class at initialization time. Second, the proof is made available to all deductive operations (proof or plan generation) by inserting it in the database according to a process to be described in Section 5.4.3. Thus, when subsequently invoking a plan generator, the user will be able to use this new conclusion as a premise.

Packaging Inferences for Inspection

A proof is a self-contained object. It includes the entire inference tree (graph) that resulted in the conclusion and makes it available for inspection by the user. I believe access to this information is essential in an interactive theorem proving system. Some reasons are: (1) curiosity, to find out how an inference was derived, (2) for debugging purposes (to debug both the system and the axiomatization), (3) to print, or otherwise prepare for publication, a representation of the proof, (4) to permit generalization.

LOGICALC affords the user 3 ways to access this information: (1) either by directly browsing through the proof graph, or (2) by using the old-style proof-dumper to output a natural deduction like account of the proof, or (3) by invoking the special purpose proof editor which also produces a natural deduction style account, but is much more powerful. The listing of the proof at the end of the introductory example was produced by the editor.

Proofs Derived Through Backward-Chaining

Not all conclusions are arrived at by executing inference rules: LOGICALC also supports backward-chaining rules[3] which represent built-in theories and are usually activated during the initialization of a class. How does LOGICALC reconcile the two mechanisms? When a conclusion has been derived through the office of DUCK's backward-chaining facility, LOGICALC converts it to a proof whose steps consists in all the assertions involved in the derivation, and whose *pretend* inference rule is `obvious`. Needless to say this integration is not exactly seamless; but this turns out not to be very important (except for generalization), and could be fixed without too much pain [p181].

[3]principal symbol is <-; constrast with modus-ponens operating on implications (i.e. principal symbol **IF**)

5.4.2 Proof Generalization

After a proof has been constructed, it is optionally generalized before it is released by the system; whether this generalization is actually performed is controlled by variable **proof-generalize*** which should be either NIL, in which case no generalization is attempted, or a function that will map a proof into a list of formulae more general than its conclusion (the simplest way to achieve this end is to return a list that contains the proof's formula as its sole element).

The default generalizer is implemented by **#'LC-GENERALIZE** which will be described at length in a separate chapter. The generalization technique is related to Explanation-Based Generelization (henceforth EBG). Since the whole deduction tree is available, the idea is to determine the most general conclusion that can be drawn through the same sequence of inference steps. The generalized conclusion will have the same shape as the original conclusion, but will be less instantiated, i.e. some of the original terms will have been replaced by free variables. A proof of $A \Rightarrow p[t]$ which does not depend on the identity of the term t will typically be generalized into $A \Rightarrow p[x]$, where x is a free variable.

As a result of generalization, proofs will unify with a wider range of goal sequents. Reasonably similar goals which would require analogous deduction trees can all be solved simply by deriving an answer to one of them; generalization will take care of making this answer applicable to all of them. This feature can often considerably reduce the amount of work required of the user. For instance, the generalized proof $A \Rightarrow p[x]$ corresponding to goal $A \Rightarrow p[t]$ will also apply to a goal of the form $A \Rightarrow p[t']$, thus sparing the user the tedium involved with the recreation of an analogue of the original deduction.

Also, proving a universally quantified goal $A \Rightarrow (\forall z)q(z)$ will typically involve a subgoal of the form $A \Rightarrow q(c)$, where c is a new constant introduced by skolemization.

Since a proof of the latter goal cannot depend on the identity of c (since it is new), it will be generalized into $A \Rightarrow q(x)$, where x is a free variable not already occurring in $q(c)$.

5.4.3 Asserting Proofs in the Database

After a proof has been derived, it is a good idea to make it available to all subsequent deductive operations by asserting it in the database. If the proof is $A \Rightarrow p$, the assumption set A is augmented with the additional formula p. p will be visible in all assumption sets that make more assumptions than A, but not in those that make fewer (or incommensurable) assumptions.

I require that it be possible for any one assertion to determine whether it is an axiom, a local assumption, or the result of an inference. In the later case, I also want to be able to recover the proof which produced it. Therefore, the assertion must be somehow annotated with the proof.

When a formula is asserted in a datapool, DUCK requires that you specify (1) the expression representing the formula, and (2) a TMS style justification. Such a justification is conceptually a pair $\langle in, out \rangle$ of two lists naming other assertions (or characteristic beads of datapools).

When a proof is to be inserted in the database, it is necessary to determine by inspection of the proof structure all the non-discharged assertions mentioned in the proof tree. These are all the assertions that the proof depends on; they will serve as the *in* component of the justification. I said "non-discharged" because the proof tree may contain proof nodes that perform "if-introduction" inference steps such as:

$$A, p \Rightarrow q \longrightarrow A \Rightarrow p \supset q$$

After such an inference step, the local assumption p is said to have been discharged.

Since no non-monotonic reasoning is allowed to occur in LOGICALC, the justification's *out* list would always be empty. I take advantage of this fact to annotate the assertion with the proof. Here is how:

1. First, I create a dummy node whose formula is (PROOF!! *proof*), where *proof* is the proof object. This node masquerades as an assertion except that it is never asserted (or indexed) in any way. In reason maintenance terms, the node will always by "out" by design.

2. The justification's *out* list is set to contain this dummy node as its sole element. Since, the dummy node is always "out," this trick is essentially a noop as far as reason maintenance is concerned; and since we know that no non-monotonic justification is allowed to be specified by the user, when we find one it must necessarily be an annotation that gives us access to a proof through a dummy node.

Whenever a deductive operation uses an assertion as a premise, what really happens is that a proof is first recovered for said assertion and then passed on to the operation proper. The trick outlined above makes it possible to recover proofs for assertions obtained through inference.

5.4.4 Implementation

Here is how a proof is implemented:

$A \Rightarrow p$: a sequent expressing the fact that p logically follows from the set of assumptions A.

validation: an expression describing this inference. It is of the form described earlier, except that no recursive validations can

occur (they are explicitly represented by the sub-proof trees). The important information here is the name of the inference rule and the parameters.

steps: typically a list of subproofs from which the inference is being drawn by the inference rule [but see below].

node: the name of the corresponding assertion in A.

How can LOGICALC distinguish between an assertion that is an axiom, one that is an assumption, and one that is the result of an inference? The answer is that each assertion is supported by a set of justifications (this is a TMS technique).

- An axiom is distinguished by having one justification of the form $\langle$`nil`, `nil`$\rangle$.

- An assumption is very much like an axiom except that its "in" list contains the characteristic bead of the corresponding assumption set, i.e. it is of the form $\langle(b)$, `nil`$\rangle$ where b is a bead.

  An assumption set is associated with a datapool, and a datapool is essentially a list of beads. Each bead represents a "context." Justifications can mention beads as well as assertion nodes. Thus, only those assertions which belong to one of the currently active contexts are visible. When a datapool is selected to become current, its beads are marked "in," and all other beads become "out."

  Whenever a new assumption set is created, a new datapool is also constructed by calling `(dp-push` P`)` where P is the parent's datapool. As a result, a new bead P' is created, and the union of P' to the parent's beads makes up the new datapool's set of beads. P' is called the new datapool's characteristic bead.

- An inference has a justification of the form $\langle(a_1, \ldots, a_n), (d)\rangle$ where a_1 through a_n are the various assertions the inference depends on, and d is the dummy node through which the proof object can be recovered.

- An conclusion derived by DUCK through forward-chaining only has "in" supporters, and an empty "out" list: $\langle(a_1,\ldots,a_n),()\rangle$.

Given an assertion, LOGICALC will always translate it into a proof before it will operate on it. The distinctions made above are reflected in the resulting proofs. For expository reasons, I will represent a proof object by $\langle A \Rightarrow p, (s_1,\ldots,s_n), validation\rangle$ for the duration of the following items (s_1 through s_n are the proof's steps).

- For an axiom, no proof is constructed; its name is used instead. Occasionally, for obscure reasons the system may have to construct a proof object anyway; in this case, assuming that the axiom's name is `foo`, it will be of the form:

  $$\langle \texttt{nil} \Rightarrow p, (\texttt{foo}), (\texttt{obvious (1) ()})\rangle$$

- For an assumption, a similar trick is used. Again, let's pretend the assertion representing p is called `foo`:

  $$\langle A, p \Rightarrow p, (\texttt{foo}), (\texttt{obvious (1) ()})\rangle$$

- For an inference, LOGICALC will recover the original proof through the dummy node in the justification's "out" list. The proof has the expected form:

  $$\langle A \Rightarrow p, (s_1,\ldots,s_n), validation\rangle$$

- For both forward and backward DUCK deductions, the trick with the bogus inference rule `obvious` once again suffices. The proof has the form:

  $$\langle A \Rightarrow p, (a_1,\ldots,a_n), (\texttt{obvious (1 ... n) ()})$$

  Actually, when attempting backward-chaining, DUCK has this built-in rule that a disjunction may be solved by selecting a disjunct and solving it instead. For

technical reasons, it is necessary to make a note in the proof when this happens on a single assertion, and the validation becomes:

$$\langle A \Rightarrow p, (a_1, \ldots, a_n), (\texttt{obvious (1) (or } k\texttt{)})\rangle$$

where k is the index of the selected disjunct.

No inference rule named `obvious` actually exists. For axioms and assumptions, its is simply a convenient notational device. It is also used to represent DUCK deduction chains as if they were the result of executing this hypothetical `obvious` rule. A more principled way to represent those chains would be to define explicit inference rules to operate on backward-chaining rules and translate a DUCK deduction into a proof tree expressed in terms of those rules. All the information required to do this is already available (well, most of it anyway). I never bothered to implement it because there never was any reason to do so. All conclusions derived in this manner are in some sense trivial and uninteresting.

5.5 Answers

As we have seen earlier, answers are typically constructed either during the initialization of a class (by borrowing answers from related classes or by backward-chaining on DUCK rules), or when a plan reaches completion and broadcasts its conclusion. Yet another way to cause new answers to be found is by means of the `fetch` command; it essentially executes again the procedure that is performed at class initialization time. There are two major circumstances in which you might want to invoke `fetch`: (1) you added new axioms to the database since the class was initialized, (2) more conclusions have been added as a result of proving other goals [e.g. lemmas].

Answers are merely a convenient package that associate classes with the proofs that solve them. The data-structure implementing an answer has the following fields:

class: the class which this is an answer to.

substitution: how the free variables in the class's formula are instantiated by this answer.

proof: the proof object itself; its conclusion unifies with the class's formula modulo the substitution given above.

Checking for Redundancy

Before an answer is recorded with a class, the system checks whether it is redundant. This check is performed on the basis of the answer's substitution. If the class already has an answer whose substitution is comparable with this one and more general (i.e. instantiates the formula less), then the new answer is said to be "subsumed" by the preexisting one. In that case, it is discarded. A substitution σ_1 is more general than σ_2 if there exists σ such that $\sigma_2 = \sigma\sigma_1$.

The most general substitution possible is the empty substitution, i.e. the one that binds no variable. An answer whose substitution is empty is called a "yes" answer. A class which acquires a "yes" answer has been solved in the most general way possible. No other answer or plan will subsequently ever be indexed under it, as they will all be found redundant.

Answer Broadcasting

Whenever a class acquires a new answer, it will broadcast it to related classes. Thus, an answer will propagate to all the classes in the graph that might possibly be interested in it. Each one of them will attempt to adapt the answer to suit its own needs, and will then recursively broadcast it to its own related classes. Here is an outline of the broadcasting algorithm:

1. Record the answer in the class.

2. Axioms and assumptions are *not* broadcast.

3. Inferences *are* broadcast to all related classes by a straightforward recursive technique that keeps track of all the classes already visited so as to avoid loops and repeats.

4. Each class attempts to "adapt" the answer to itself. First the proof is extracted from the answer and it is *strengthened*, which means that if the proof was obtained by weakening through assumption introduction, then as many of these steps are removed from the top as possible. Then the strengthened proof is tentatively unified with the class's formula; if unification fails, the answer cannot be adapted; if it succeeds, we must make sure that the proof makes as many or fewer assumptions than the class (i.e. the proof's assumption set is an "ancestor" of the class's). If such is the case, an answer is constructed whose substitution is determined by unifying the proof's conclusion with the class's formula. Finally, it is checked that the answer is not made redundant by one already recorded with the class. If all this succeds, we recursively go to step 1.

Chapter 6

Plan Generators

In this chapter I document the mechanism whereby the user can generate alternative refinements for a goal by invoking plan generators. I begin with a description of user-level syntax for premises and subterm designators, and then explain the various steps involved in constructing the resulting plans from the user's input. The chapter ends with a reference listing of all plan generators available in LOGICALC.

6.1 Introduction

The preferred method of solving a goal is to refine it into a plan whose steps are subgoals, and to thus proceed recursively until all fringe goals in a proof tree either are tautologous or have been identified with axioms or assumptions.

At a goal node, the user may issue a command to invoke a plan generator. This results in the creation of 0 or more plans to solve the goal. A plan consists of 1 or more steps together with a *validation* which explains how to infer a proof of the goal from proofs of the steps; when all its steps have been solved, the plan's validation is executed to compute this proof.

Consider a goal of the form $A \Rightarrow q$, such that the assertion `foo` of the form $p \supset q$ is in A. Invoking plan generator MODUS-PONENS with `foo` as an argument premise will result in the creation of the plan:

$$\begin{array}{ll} [0. & \texttt{foo} \quad] \\ 1. & A \Rightarrow p \end{array}$$

where step 0 is already solved because it corresponds to the existing premise named by `foo`, and step 1 is the expected subgoal resulting from backward-chaining. The plan is also equipped with a validation. In the present case, it consists of the expression:

```
(modus-ponens (0 1) ())
```

The intended meaning is that, in order to derive a proof of the original goal, one should perform an inference using MODUS-PONENS as the rule, a proof of step 0 as the major premise (here, this would simply be `foo`), and a proof of step 1 as the minor premise. Validations are discussed in detail in Chapter 7.

I will now outline the algorithm which the system follows to go from user input to a set of new plans. Subsequent sections will provide the details.

Each plan generator really consists of two procedures: one for parsing user input and one for constructing descriptions of plans. Let's call them the *parser* and the *designer*, respectively. The designer takes 2 arguments: a list of premises (such as assertion names; but see Chapter 7 for the true, generalized story), and a list of parameters (e.g. designators for subterms to be affected by equality substitution). The parser's job is to take user input and convert it to a list of *argument records*, each one of which comprises a list of premises and a list of parameters. The designer is repeatedly applied until all argument records have been exhausted. Each invocation of the designer produces a list (possibly empty) of plan descriptions.

For each plan description, the system performs some additional [optional] processing, such as to automatically expand conjunctions into as many steps as there are conjuncts. Then, from this final description, a real plan is constructed and its validation automatically assembled from the various pieces. Finally, the plan is initialized, checked for redundancy, subsumption, etc... (see Chapter 5).

If truth be told, a plan generator consists of three procedures: *parser*, *designer*, and *helper*. The latter is executed whenever a plan generator is invoked with the option **-help**, and, in response, prints documentation about usage and purpose.

6.2 User Level Syntax and Notation

A plan generator *plangen* may be invoked with arguments to be determined from the specifications *specs* by invoking the **+plan** command thus:

+plan *plangen specs*

plangen is a name such as MODUS-PONENS, or DEDUCTION-THEOREM, or LEMMA—for a complete list see section 6.6. Like any name meaningful to LOGICALC's shell, it may be abbreviated. For instance, **+plan deduction**, or even **+ d**, will have the same effect as **+plan deduction-theorem**.

For the user's convenience, the command processors [Chapter 4] for class and goal nodes have been been written in such a way that, when a command is not otherwise recognized, the system will check to see if it could be interpreted as the name of a plan generator, in which case the **+plan** command will be invoked automatically. Thus, the name of a plan generator acts as a command to invoke this plan generator in the regular fashion.

The *specs* specify the arguments to the plan generator. For example, MODUS-PONENS requires a premise which has the form of an implication whose consequent unifies with the current goal. Thus, **+plan modus-ponens foo baz** will attempt backward chaining using assertion **foo** and also assertion **baz**. In general this will result in the creation of two plans. When an argument is omitted, sometimes a sensible default is provided, but usually the user is prompted for further specifications.

Plan generator EQUALITY requires an equality as a premise, but also needs to be told which subterms in the current goal should be subjected to equality substitution. Therefore, the user must provide two kinds of specifications: one kind to determine the premise(s), and another kind for the targeted subterms. In order to let the "parsing" function know where in the command line the first kind ends and the second kind begins, they must be separated by the character '-'. For example:

```
+plan equality a - foo baz
```

will use assertions **foo** and **baz** as premises, and will operate on subterms of the current goal denoted by **a**—typically this would be the constant **a** itself, but see Section 6.2.2.

For very complex plan generators such as ABSTRACTION and QUANTIFY, the specifications require an additional level of structuring. At the top level, we use **--** to segment them into related chunks; each chunk is further subdivided by occurrences of **-**.

Arguments to plan generators are of 4 kinds: formulae, terms, premises, and subterm designators. Formulae and terms have the usual syntax [Appendix A]. The notation for premise and subterm designators will be the subject of the next two sections.

6.2.1 Premise Parameters

Some plan generators not only operate on the current goal, but also require one or more premises to completely specify the desired refinement. For example, MODUS-PONENS expects an implication. The conclusion denoted by a premise is meant to serve in the sequence of inferences performed when the plan's validation is finally executed. The premise's conclusion may or may not have to be proven: if the premise is an assumption, then it does not; if it is a conjecture, it must be. A premise is a description of the subplan necessary to infer its conclusion. This subplan is inserted in the refinement.

When the premise's conclusion is readily available as an assertion in the current assumption set, it may be denoted simply by naming the assertion. The premise, as well as the corresponding subplan description, will simply be this name. When inserted in the refinement, it will turn into a *solved* step whose recorded proof is the aforementioned assertion.

Occasionally, the user may want to use a conjecture as a premise. The classical approach would be to prove a lemma. However it has the limitation that, in order to use the lemma's conclusion, the latter must be proven first. LOGICALC's notion of a premise removes this limitation: a hypothetical premise is such that its corresponding subplan description contains the conjectured formulae as its sole subgoal. The user may use the notation '`=` *formula*' for this purpose.

Most of the time, however, the desired premise is a subformula in an assertion. For example, supposing that `foo` is an assertion of the form $p \supset a = b$, the user might be interested in performing a refinement using EQUALITY to substitute a for b in the current goal. Classically, the user would be required to first prove p, then infer $a = b$ from `foo` and p. In LOGICALC, the premise may simply be constructed such that its corresponding subplan includes `foo` as its first (solved) subgoal, and p as its second

(presumably unsolved) subgoal, and such that its validation performs an inference using MODUS-PONENS as the inference rule, `foo` as its major premise and the proof of p as its minor premise. The premise is automatically constructed by the *detaching* mechanism [Chapter 7]; the user need only name the assertion which it should be applied to.

Actually, the detaching mechanism is always active (but see the `-raw` option). Thus, the more complex case of the preceding paragraph (detaching a subformula) is not distinct from the simpler case of the opening paragraph (using the whole assertion). Detaching also occurs with conjectures. Such uniformity is achieved thus: the detaching procedure is given a description of a plan to infer a certain proposition; this is always either the name of an assertion or a plan with one subgoal representing the conjecture—an arbitrarily complex plan is allowed, but LOGICALC does not take advantage of this generality. The detaching procedure returns a more elaborate plan which infers a subformula of the conclusion of the plan it was given as an argument.

Since *detaching* is often complex and may produce several candidates, it is a good idea for an interactive system to provide a tool for search, browsing, and selection of candidate detachments. This is the rôle of LOGICALC's *find mode* which is discussed and documented in Appendix B; it may be invoked through the `-find` option.

Similar remarks apply to searching the database for assertions to serve as input to the detaching mechanism. There is no need for the user to memorize the contents of the assumption set. Instead, he may specify *views* to be searched for relevant assertions. The `-all` view denotes the whole database. This system of views naturally works in combination with *find mode*: `-find -all` lets the user interactively browse through all possible detachments of the target formula from the database.

Looking up relevant assertions in the database is realized through an indexing scheme described in Section 2.7.1. Typically the target formula has at least a principal symbol, e.g. $\supset$ for MODUS-PONENS, which serves as the primary index. Occasionally

the target is unspecified; e.g. this happens when making forward inferences using *inference generators*. In this case (1) every subformula of any proposition yields a candidate detachment, therefore detaching should only be performed if the user agrees to it, not by default, (2) searching the whole database would be pointless since every assertion is relevant, therefore **-all** should complain and require that the user explicitly provide a view.

A premise parameter, also called a premise *descriptor*, must be one of the following:

-HELP

Prints a summary of the expected syntax and prompts the user again.

-ALL

Automatically consults the whole database for relevant assertions. In order for this to work, the target formula (i.e. the one we are attempting to detach from assertions in the database) must not be underspecified. Essentially this means it should have a principal symbol to serve as the primary index. The exact definition of an underspecified formula is: either it is NIL, conventionally representing an unspecified target, or it is a variable, which would match anything, or it is a variable negated an even number of times, in which case the negations cancel out and this is just like the previous case.

When the target is underspecified, rather than return all formulae detachable from (i.e. occurring in) the database (a staggering number), the system prompts the user for a more specific descriptor.

-ALL *specs...*

Automatically consults the view specified by the *specs*. Only these propositions occurring in the view will be considered. Each *spec* must be one of:

-HELP

Prints a summary of the syntax of view *specs*.

assertion

This is the name of an assertion. The system checks that the name corresponds to an existing assertion and also that this assertion is in the current assumption set. If a problem is noticed (e.g. the name was mistyped and fails to denote an assertion), the user will be given a chance to effect corrections interactively. Detaching will be attempted on the proposition represented by this assertion.

= *formula*

The pattern is fully quantified, and contains no free variables. As explained earlier, it will be set up as an auxiliary goal in the resulting plan. Detaching is also attempted on this formula. Naturally, such a spec is not available to inference generators since they cannot accommodate auxiliary goals.

`-LOCAL`

Every assumption set makes one more assumption than its parent. This assumption results in one or more additional assertions, which are local to the assumption set and not visible to its parent. It is often convenient to refer to them as *the local assumptions.* `-local` denotes precisely the local assumptions in the assumption set of the current goal.

`-ASET`

Denotes the local assumptions for the current assumption set and all its ancestors. These are all the assumptions in effect at this point in the proof.

`-SKOLEM`

Denotes the set of assertions that mention at least one of the skolem terms occurring in the target. This is decided solely on the basis of skolem terms' ids [see p80].

`(-SKOLEM` *names...*`)`

Same as above, but only for those skolem terms whose function is in the given set of *names*.

`-NIL`

The empty set. Because of the recursive nature of various prompting procedures, it is sometimes convenient to have a denotation for the empty set just so you can type something that denotes nothing and lets you exit a particular recursive prompting loop.

`-RAW` *assertions...*

The system does not attempt to detach *subformulae* of the named assertions. Instead, it simply filters out those that do not unify with the target. This is particularly convenient when the target is underspecified (is a variable).

`-FIND`

Invokes *find mode*, searching the whole database. The user may interactively browse through candidate detachments and select those of interest.

`-FIND` *specs...*

Invokes *find mode*, searching the view specified by the given *specs*, where each *spec* is as described in the entry for `-all` above.

`-NIL`

Denotes the empty set.

descs...

A descriptor may be a sequence of appropriately parenthesized descriptors.

(*descs...*)

Parentheses may be used for grouping and for limiting the scope of certain options, e.g. `(-find foo) -all bar baz` limits the scope of `-find` to just `foo`.

$i_1 \ldots i_n$ *descs...*

Integers may be used to further select detachments out of the list produced by the remainder of the descriptor, i.e. *descs*. Each integer i_k may be indifferently positive or negative—only its absolute value matters—and it selects the i_kth detachment, should there be one, in the list denoted by *descs*.

specs...

A descriptor may be a view denoted by *specs* as described in the entry for `-all`. Typically these are names of assertions.

`+MISMATCHES`

Allows mismatches during detachment [Section 7.5]. Each mismatch adds an equality goal to the premise's subplan.

`-MISMATCHES` *spec*

Allows mismatches according to *spec*. Typically *spec* is an integer indicating the maximum number of mismatches permitted for any one detachment. *spec* may also be NIL, in which case any number of mismatches are permitted. It may be a symbol, in which case only those subexpressions whose principal symbol is *spec* are allowed to mismatch. Finally, it may be a list of symbols, optionally beginning with NIL or an integer; e.g. `-mismatches (3 f g)` allows a maximum of 3 mismatches, and only for those subterms that have either `f` or `g` as their principal symbol.

As customary in LOGICALC, every one of these options may be abbreviated; e.g. `-f` instead of `-find`, or `+mis`, `+m`, or just `+` for `+mismatches`.

6.2.2 Subterm Parameters

Certain plan generators, such as EQUALITY, require parameters that specify those subterms concerned by the refinement. Consequently, it was necessary to devise a

notation for these parameters.

The *path* to a subexpression of a given formula is a sequence of integers. An empty sequence denotes the whole formula. Otherwise, let n be the first element of the sequence; the intended subexpression is the one denoted by the remainder of the path into the nth element of the formula considered as a list. `(1 2)` denotes `(g b)` in `(p (f a (g b)) c)`. Therefore, supposing the current goal has the form just shown, the user might type '**`+plan equality`** `1 2 - foo`' in order to replace `(g b)` using an equality detached from `foo`.

The sequence '`1 2`' typed by the user is called a *path descriptor*. LOGICALC generalizes this natural notation, and allows a path descriptor to contain not only positive integers, but also expressions such as `!&`n denoting the nth tail, and `!&(`n m`)` denoting the segment composed of elements n through m. `(g b)` could also be denoted by the path descriptor '`!&(1 2) 0 !&1 1`'. The more complex notation is occasionally useful for lists and tuples.

It is possible to refer to the same subexpression using various exotic path descriptors. However, plan generators and inference rules need to be able to compare these paths, for instance to determine whether one expression is a subexpression of another. To facilitate these computations, the system will automatically convert all paths to their normal forms.

Path descriptors are completely general but they require too much typing, and it is often difficult to get these numbers right. For these reasons, there is an alternative way to denote subterms, designed for brevity, and better suited to an interactive setting. Basically, you may type a constant to denote all of its occurrences (not in functional position); you may also type the name of a function to denote all the subterms where it appears as the principal symbol. Thus, the earlier example becomes: **`+plan equality g - foo`**.

A generalization of this last idea is to allow a pattern. It denotes all the subterms which unify with it. However, in the special case where the pattern is a variable, it would not be very useful to have it denote the set of all subterms; instead it denotes all occurrences of this particular variable.

A subterm descriptor must be one of the following:

`-HELP`

Prints a summary of the syntax of subterm descriptor and prompts the user again.

`-FIND`

Enters another incarnation of *find mode*, which lets the user browse through subterms and select those of interest.

`-FIND` *descs...*

Enters find mode, browsing through the subterms denoted by the remainder of the descriptor, e.g. *descs.*

path

A subterm may be denoted precisely with a path as explained earlier.

name

A much more concise notation. Denotes all subexpressions whose principal symbol is *name*, i.e. all subexpressions which have *name* in functional positional, as well as all occurrences of *name* in non-functional position.

$i_1 \ldots i_n$ *descs...*

To remedy the possible ambiguity of the above notation (among other things), a descriptor may be prefixed with *negative* integers. i_k selects subterm $-i_k$ in the list of subterms denoted by the remainder of the descriptor, i.e. *descs.* The reason that these integers must be negative is so as to avoid confusion with path descriptors which are sequences of positive integers.

pattern

Denotes all unifying subterms. A variable, however, denotes all occurrences of identically this variable.

NIL

NIL (that is ()) means to take the expression as a whole, and corresponds to the empty path.

`-NIL`

Denotes nothing [see discussion of `-NIL` for premises].

`-SKOLEM`

Denotes all the skolem terms occurring as subterms.

`-SKOLEM` *names...*

Same as above, but only for those skolem terms whose function is in the given set of *names*.

descs...

A descriptor may be a sequence of appropriately parenthesized descriptors.

(*descs...*)

Parenthesized descriptors are also allowed; however, this conflicts with the interpretation of a list as a *pattern*. The only way to resolve this conflict is to begin the parenthesized list with an expression that is allowed as a descriptor and cannot be mistaken for a legitimate function or predicate.

Therefore, *descs* may begin with an integer or an expression of the form `!&`*n* or `!&(`*n m*`)`, or with one of the 4 symbols NIL, `-nil`, `-skolem`, or `-find`. Anything else will be considered a pattern.

6.3 Parsing The Command Line

When a plan generator is invoked from LOGICALC's shell, the rest of the command line determines its arguments. Each plan generator has its own *parser* whose purpose is to convert these specifications into a list of *argument records*. Each argument record has two components: a list of premises and a list of parameters. The plan generator's *designer* [p186] will be invoked successively for each argument record.

Why return a *list* of records rather than merely a single one? For two reasons: firstly, the specifications may be ill-formed, or otherwise fail to correspond to an admissible record; in this case, an empty list of records is returned. Secondly, the specifications may meaningfully correspond to more than one record, e.g. in **`modus-ponens foo baz`**, if both **`foo`** and **`baz`** are implications whose consequent unifies with the current goal, then it is clear that the user's intention is that one argument record should have **`foo`** as a premise, and another **`baz`**. More generally, since "detaching" may produce several answers, each detachment should go into a separate argument record.

First, the specifications are segmented according to occurrences of `--` and `-`. Each item is then interpreted according to the kind of argument it is expected to be. For a formula (e.g. the parameter to LEMMA), it will be syntax and type checked. For a premise, the item must be interpreted as explained in section 6.2.1. Similarly for a subterm parameter, it must be interpreted as explained in section 6.2.2. For each kind of argument, there exists a prompting procedure which will convert the item into a list of arguments of this kind.

PROMPT-FOR-FORMULA

Syntax and type checks the item. If the item was not provided, or failed the check, the user is prompted for a formula and the process iterates.

PROMPT-FOR-PREMISE

Given a target proposition (which may be unspecified) and a descriptor, returns

a list of detachments. This procedure is essentially an interpreter for the language described in section 6.2.1. In general, further processing must be applied to the resulting detachment records: plan generators want them converted to plan descriptions, while inference generators want them turned into proofs. Consequently, the preferred interfaces are, respectively, PROMPT-DETACHING and PROMPT-ASSERTING; the latter also makes sure that the detaching procedure is invoked with the DEEP switch (see p255) set to FALSE, so that detachments which require auxiliary goals are not permitted. The algorithms for turning detachment records into plan descriptions and proofs are described on pages 257 and 259.

PROMPT-FOR-SUBTERMS

Given an expression and a descriptor, returns a list of paths denoting subexpressions. This procedure is essentially an interpreter for the language described in section 6.2.2.

Below, I include the code of the parser functions for plan generators MODUS-PONENS and EQUALITY.

---

code

```
(DEFFUNC PONENS-PARSE-PLANGEN - (LST (LRCD (LST <premise_t>)
                                           (LST sexp)))
   (SEQUENT - sequent_t PARMS - (LST sexp))
  (<# (\\ (P) (LRECORD (LIST P) NIL))
     (PROMPT-DETACHING (MAKE sequent_t
                         !>SEQUENT.ASET
                         '(IF ?* ,!>SEQUENT.CONCLUSION))
                       PARMS
                       (\\ () (OUT "Major premise? ")))))

(DEFFUNC EQUALITY-PARSE-PLANGEN - (LST (LRCD (LST <premise_t>)
                                              (LST sexp)))
   (SEQUENT - sequent_t PARMS - (LST sexp))
  (MATCH-COND PARMS
    ;;--no parameters: must prompt. do this by defaulting to 2nd clause.
    ((NULL PARMS)
     (EQUALITY-PARSE-PLANGEN SEQUENT '(-)))
    ;;--both descriptors are given
```

```
(\? (!?DESC - . ?VIEW)
    ;;--DESC == [-FIND] desc...
    (LET ((PATHS (PROMPT-FOR-SUBTERMS
                  !>SEQUENT.CONCLUSION DESC
                  (\\ () (OUT 0 "Enter descriptor of subterm to be"
                               T "subtituted by equality: "))))
          - (LST (LST fixnum)))
      (<# (\\ (PREM - <premise_t>)
             (LRECORD (LIST PREM) PATHS))
          (PROMPT-DETACHING
           (MAKE sequent_t !>SEQUENT.ASET
             '(= ,(SUBTERM !>SEQUENT.CONCLUSION (CAR PATHS)) ?*))
           VIEW
           (\\ () (OUT "Equality premise? "))))))
;;--only the subterm descriptor is given
(T (EQUALITY-PARSE-PLANGEN SEQUENT '(,@PARMS -)))))
```

6.4 Designing A Plan Description

A premise record has the form ⟨CONC, HOW⟩, where CONC is a conclusion, and HOW is the description of a plan to prove it. A plan designer takes a list of such premise records, as well as a list of parameters, and returns a list of plan descriptions. For a plan generator such as MODUS-PONENS, or EQUALITY, determination of the necessary subgoals is effected by a procedure which amounts to running the corresponding inference rule backwards. For more complex generators, such as CASE, QUANTIFY, or SKOLEMIZE, the designing procedure is much more elaborate.

In this section, I will first present the plan description language, then I will illustrate plan designing with examples of code and applications.

6.4.1 Premise Description Language

The HOW slot of a premise record is called a *premise descriptor*. It must be one of:

assertion name Type: `namddnode`

The corresponding conclusion is the formula associated with the named assertion. Since it is in the assumption set, it requires no proof and will result in a "solved step" when the plan description is converted to an actual plan.

goal descriptor Type: `<goal_t>`

Typically, a `<goal_t>` does not appear by itself, but only as part of a `<plan_t>` (see below). A goal descriptor represents a subgoal, and instances are constructed with the form:

```
(make <goal_t> PATTERN [ASSUMPTIONS RENAMED WEAKEN])
```

where the last three arguments are optional.

PATTERN is the formula to be set up as a subgoal.

ASSUMPTIONS is a list of formulae (NIL by default) indicating additional assumptions to be made for this goal.

RENAMED is a list (usually empty) of *renamings*. Each renaming is a pair associating an old variable name with a new variable name. First, let me explain why some variables may have to be renamed.

Suppose I wished to prove an instance of $p(x)$ from $p(a) \vee p(b)$. Proceeding by case analysis, if I set up a plan of the form:

1. $p(a) \Rightarrow p(x)$
2. $p(b) \Rightarrow p(x)$

I will not get very far because, as soon as I solve e.g. step 1, x becomes instantiated with a and the second step, now of the form $p(b) \Rightarrow p(a)$, has no proof.

What is needed here, is a way to allow x in step 1 and x in step 2 to be instantiated independently. This may be achieved simply by renaming x differently in each step:

1. $p(a) \Rightarrow p(x_1)$
2. $p(b) \Rightarrow p(x_2)$

However, now another problem creeps in: whenever an answer is found for a goal, LOGICALC trims it by removing the bindings for non-return variables. A *return variable* is a free variable occurring in the goal and such that its value is relevant: either it also occurs in the supergoal, or it is mentioned by at least one other sibling goal (i.e. goal in the same plan).

In the above, x_1 and x_2 are new, therefore cannot be return variables according to the definition I gave above. Consequently, LOGICALC would discard their bindings. On the other hand, they both stand for x which is a return variable: if the value of x is considered relevant, then surely so should be those of x_1 and x_2.

The purpose of the RENAMED list is precisely to overcome this problem. New variables inherit the status of return/non-return variable from the corresponding old variable. This feature is almost never used. Only a plan generator that does very strange things for the convenience of the user may have to worry about it.

WEAKEN is a boolean that is true iff it is essential that a proof of this goal have the exact same assumption set as the goal. Normally, a proof's assumption set is the strongest that suffices for its derivation—typically, it is the weakest (the one that makes the most assumptions) of the assumption sets of its immediate steps.

For example, a goal of the form $p \Rightarrow q \supset q$ will have $\Rightarrow q \supset q$ as its proof because $q \supset q$ is a tautology and does not depend on the assumption p.

However, suppose $p \Rightarrow q \supset q$ is a refinement of the goal $\Rightarrow p \supset \centerdot q \supset q$. From the stronger proof $\Rightarrow q \supset q$ we cannot derive $\Rightarrow p \supset \centerdot q \supset q$ using IF-INTRO because there is no way to discharge a non-existent assumption; and the refinement's validation fails.

To avoid this problem, we must require that the proof be weakened to have the same assumption set as the goal. [see * hook, Section 6.4.2].

plan descriptor Type: `<plan_t>`

Instance of plan descriptors are constructed with the form:

(make <plan_t> CONCLUSION RULE STEPS PARAMETERS [ASSUMPTIONS WEAKEN])

CONCLUSION is a formula representing (as best as can be determined at this time) what the plan will conclude. The actual conclusion, in general, will be an instance of this formula (although automatic generalization may occasionally intervene to produce something more general than the projected conclusion).

RULE names the inference rule, to be invoked during validation, to infer the correct conclusion from proofs of the steps. Note that, for this to work properly, the steps must appear in the order expected by the RULE.

STEPS is a list of premise descriptors which must appear in the order expected by the RULE.

PARAMETERS is a list of expressions further specifying the inference.

ASSUMPTIONS is optional and has the same interpretation as before.

WEAKEN is optional and has the same interpretation as before.

6.4.2 Validation Hooks

Case Analysis

Earlier, I discussed the problem of deriving an instance of $p(x)$ by case analysis from the assumption that $p(a) \vee p(b)$, and I suggested that a possible approach was to create a plan of the form:

$$1.\ p(a) \Rightarrow p(x_1)$$
$$2.\ p(b) \Rightarrow p(x_2)$$

thus allowing the two avatars x_1 and x_2 of x to be independently instantiated. However, the question remains: once we have obtained the two proofs $p(a) \Rightarrow p(a)$ and $p(b) \Rightarrow p(b)$, what can we conclude from them, and, more specifically, what should x be instantiated with?

The solution adopted here is to introduce a new constant, say α, which we define to be equal to a in the first case, and to b in the second case. x becomes instantiated with α. In order for such a definition to be safe, the cases must be mutually exclusive (if they overlapped, you could derive e.g. $a = b$ in the intersection). Therefore, the plan must really be:

$$1.\ p(a) \Rightarrow p(x_1)$$
$$2.\ \neg p(a), p(b) \Rightarrow p(x_2)$$

and the proofs become $p(a) \Rightarrow p(a)$ and $\neg p(a), p(b) \Rightarrow p(b)$. α is defined by the two formulae $p(a) \supset \alpha = a$ and $\neg p(a) \wedge p(b) \supset \alpha = b$.

The idea is that the validation should infer the appropriate equality in each case and use it to substitute, by EQUALITY, α for the corresponding term, thereby producing

the two proofs $p(a) \Rightarrow p(\alpha)$ and $\neg p(a), p(b) \Rightarrow p(\alpha)$. Now, from these (and a proof of $p(a) \vee p(b)$) the CASE rule can easily infer $p(\alpha)$.

Obviously, the necessary definitions of α cannot be introduced when the plan generator is invoked because, at that time, it is not known that x_1 will become bound to a or x_2 to b. However they are required for the equality substitutions which must be performed before the validation by CASE rule has any chance of succeeding. Therefore, we need some sort of *trapping* mechanism that will interrupt the normal execution of the validation to examine the proofs, construct and assert the necessary definitions.

Furthermore, we don't always want to introduce a constant α: for instance, when the various cases all have identical conclusions, it would be silly and unhelpful to introduce α to stand for the very same term in all of them. In fact, I made the decision to introduce new constants only for disagreeing subexpressions: if we replace a and b with $f(a)$ and $f(b)$ throughout the earlier example, I would introduce α to stand for a and b, not for $f(a)$ and $f(b)$.

Another approach to solving this example would be to capture x by existential quantification, and then reskolemize in each case. However, the first approach is more useful because it gives you bindings for the goal's free variables, and such that each binding retains as much that is the same in all cases as possible.

The point of this discussion is to convince the reader that the validation language and the language of premise descriptors both must provide a way to specify an arbitrary computation to be performed on a set of proofs at validation time.

Weakening a Proof

I mentioned earlier that, for the validations produced by certain plan generators (e.g. DEDUCTION) to perform successfully, it is necessary to make sure that certain proofs

are weakened down to a particular assumption set. Such weakening is effected by repeated application of the ASSUMPTION-INTRO inference rule.

In a plan description, WEAKEN slots may be set to TRUE to convey this requirement. However, ultimately it is necessary to have it reflected in the validation, but we can't know in advance the number of ASSUMPTION-INTRO inferences that will be needed; therefore, this extra computation cannot be specified directly in terms of inference rules, but instead, as for Case Analysis above, it must be an arbitrary computation to infer more proofs in a manner to be determined at validation time.

The Validation Hook

Both in plan descriptors and in validations, '*' is allowed to mascarade as a pseudo inference rule. For example:

```
(make <plan_t> CONC '* STEPS '(FCN . PARAMETERS))
```

indicates that, at validation time, FCN should be called with a list of the proofs of STEPS as its first argument and PARAMETERS as its second argument. It will return a list of proofs to be used instead of the proofs of the STEPS.

This scheme is not as general as it ought to be: FCN should really return a list of lists of proofs, for those cases when more than one transformation is applicable.

Lest the reader get the wrong idea, let me state that the *hook* mechanism is not an extra-logical hack. New proofs can only be created by application of inference rules. The hook mechanism merely extends the basic validation language to allow arbitrary invocations of inference rules to be determined and effected at validation time, instead of having to be fully specified in advance.

6.4.3 Examples

In this section, I will present code from LOGICALC's implementation of plan generators MODUS-PONENS and EQUALITY, and I will illustrate their behavior and corresponding output on simple examples.

Modus Ponens

Here is the code for MODUS-PONENS's *designer:*

code

```
(DEFFUNC PONENS-PLANGEN - (LST <plan_t>)
    (SEQUENT - sequent_t
     PREMS   - (LST <premise_t>)
     PARMS   - (LST sexp))
  (MATCH-COND !>(CAR PREMS).CONCLUSION
    (\? (IF ?P ?Q)
        (LIST (MAKE <plan_t> !>SEQUENT.CONCLUSION 'Modus-Ponens
                (LIST !>(CAR PREMS).HOW (MAKE <goal_t> P)) NIL)))
    (T NIL)))
```

The `match-cond` form does pattern-matching on the premise's conclusion: if it is not an implication, no plans are constructed; otherwise, `P` is set to the antecedent and `Q` to the consequent, and a list of one plan is constructed. The plan will conclude the desired formula (as expressed in the target sequent), using MODUS-PONENS, from a proof of the implication `(IF P Q)` and a proof of `P`. The plan description of how to prove the implication is gotten from the `HOW` slot of the premise record, and is inserted as the first step. The second step is a goal descriptor to prove `P`.

The reason I do not bother to unify `Q` with the sequent's conclusion to determine the proper instantiation of step `P` is that this has already been done by the *parser*. The premise records' conclusions are always already properly instantiated [but see equality later].

I will now illustrate how this code works on an example. Suppose the current goal is $p(a)$ and `foo` is an assertion of the form $[q(x) \supset p(x)] \vee r(x)$. The user invokes MODUS-PONENS thus:

```
+plan modus-ponens foo
```

The plan generator's parser will invoke the detaching procedure to obtain a premise record for a formula of the form $?? \supset p(a)$, where ?? stands for an unknown antecedent. This will result in a list containing the following premise record:

```
⟨q(a) ⊃ p(a),
   (<plan_t> q(a) ⊃ p(a) modus-ponens
       ((<plan_t> ¬r(a) ⊃.q(a) ⊃ p(a) taut-trans
            (foo) ([x1 ⊃ x2] ∨ x3   ¬x3 ⊃.x1 ⊃ x2))
        (<goal_t> ¬r(a)))
       ())⟩
```

The first element of the $\langle \ldots, \ldots \rangle$ record indicates the conclusion; while the second one provides a description of a plan to prove it.

Now the plan generator's designer can use the above premise record to construct the following plan description:

```
(<plan_t> p(a) modus-ponens
    ((<plan_t> q(a) ⊃ p(a) modus-ponens
        ((<plan_t> ¬r(a) ⊃.q(a) ⊃ p(a) taut-trans
             (foo) ([x1 ⊃ x2] ∨ x3   ¬x3 ⊃.x1 ⊃ x2))
         (<goal_t> ¬r(a)))
        ())
     (<goal_t> q(a)))
    ())
```

Equality

Here is the code for EQUALITY's *designer:*

```
(DEFFUNC EQUALITY-PLANGEN - (LST <plan_t>)                                    code
    (SEQUENT - sequent_t
     PREMS   - (LST <premise_t>)
     PARMS   - (LST sexp))
  (LET (PREM - <premise_t> EQTY (CONC (!_CONCLUSION SEQUENT)) - sexp)
    (DECL (PARMS - (LST (LST fixnum)))
      (WHEN (AND (>= (LIST-LENGTH PREMS) 1)
                 (>= (LIST-LENGTH PARMS) 1)
                 (MATCHQ (= ?() ?())
                   (SETQ EQTY (!_CONCLUSION (SETQ PREM (CAR PREMS)))))
                 (NON-INTERSECTING-PATHS PARMS))
        (LOOP FOR ((PATH IN PARMS) - (LST fixnum)
                   (IENVS (LIST (MAKE intenv NIL NIL))) - (LST intenv))
          RESULT
          (FOR (E IN (INTENVS-CVT IENVS))
               (SAVE (LET* (;;--we have to unzip to the side of the equality
                            ;;--because we don't want to rename variables that
                            ;;--may occur in (!_HOW PREM). it's ok if we rename
                            ;;--vars on CONC's side though.
                            (E (UNZIP E 'Right)) - env
                            (CONC (ENVSUBST CONC E 'Right)) - sexp
                            (EQTY (VARSUBST EQTY (!_RALIST E))) - sexp
                            (NEW (CADDR EQTY)) - sexp
                            (GOAL (IF (!_LALIST E) CONC (COPY-TREE CONC))))
                       (LOOP FOR ((PATH IN PARMS) - (LST fixnum))
                         (!= GOAL (*SUBTERM-REPLACE GOAL PATH NEW)))
                       (MAKE <plan_t> CONC 'Equality
                         (LIST (!_INSTANTIATE (!_HOW PREM) (!_RALIST E))
                               (MAKE <goal_t> GOAL))
                         '(,(EXTRACT-VAGUE-PATTERN GOAL PARMS) - ,@PARMS)))))
          WHILE (HAS-SUBTERM CONC PATH)
          RESULT NIL
          (!= IENVS (LAMBDA-SUB-UNIFY (SUBTERM CONC PATH)
                                      (CADR EQTY)
                                      (LVARS-ON-PATH CONC PATH NIL) NIL
                                      IENVS 'Left 'Right))
          WHILE IENVS
          RESULT NIL)))))
```

This is admitedly more complex, but not hopelessly so (unlike those plan generators dealing with quantifiers and λ-expressions). The condition of the when form is pure

paranoïa, and checks that premises and parameters are really what they are supposed to be: there must be one premise and it must be an equality; the parameters must be paths (denoting subterms), and these paths must not intersect (denoted subterms must be disjoint).

All the denoted subterms must be simultaneously unifiable with the expression on the left hand-side of the equality. The `loop` determines the corresponding substitution. Actually [see p76] there may be more than one resulting substitution: they are represented by the list **`ienvs`** of *internal environments.*

As its result, the `loop` computes a list of plan descriptions; one for each internal environment, a.k.a. substitution. All the denoted subterms must be replaced with the right hand-side expression of the equality, instantiated according to the substitution. The actual substitution is computed from the internal environment as explained on page 111.

A subloop computes the new `goal` by effecting the appropriate replacements, and a plan description is assembled whose first step is the instantiated plan description for inferring the equality, and whose second step is the new `goal`. The plan's parameters consist of the paths, so that the EQUALITY rule knows what subterms to replace when the validation is executed. It also contains an expression computed by:

```
(extract-vague-pattern goal parms)
```

This was a later addition made necessary by the Automatic Generalization mechanism. Since this mechanism may generalize certain subterms into free variables, there is no garantee that the subexpression denoted by a path will actually be there when the validation is executed. The *vague pattern* is a sort of expression-skeleton which will be applied at validation time to reestablish enough structure to make sure that the paths are meaningful and the validation will not bomb. More on this in Chapter 8.

Notice that plan descriptions know how to instantiate themselves with a substitution: this operation is performed by the type's `!_instantiate` method.

I will now illustrate how this code works on an example. Suppose the current goal is $p(f(a))$ and `foo` is an assertion of the form $q(x) \supset f(x) = h(x)$. The user invokes EQUALITY thus:

```
+plan equality f - foo
```

The plan generator's parser calls the detaching procedure to obtain a premise record for a formula of the form $f(a) = ??$, where ?? stands for an unknown term. This results in a list containing the following premise record:

```
⟨f(a) = h(a),
    (<plan_t> f(a) = h(a) modus-ponens
        (foo (<goal_t> q(a))) ())⟩
```

which the plan generator's designer uses to construct the following plan description:

```
(<plan_t> p(f(a)) equality
    ((<plan_t> f(a) = h(a) modus-ponens
         (foo (<goal_t> q(a))) ())
     (<goal_t> p(h(a))))
    (p(h(z)) - (1)))
```

6.5 Converting A Plan Description To A Plan Object

Once a list of plan descriptions has been obtained, the system must convert them into actual plan objects, perform various initialization and bookkeeping tasks, and return

a summary of the results in the form of a list of fates [see Section 5.3.4]. For each plan description, the process unfolds in 4 stages: automatic expansion, computation of the validation, construction of the plan object, initialization of the plan object.

6.5.1 Automatic Expansion

The process of automatic expansion has two functions. Firstly, it applies further transformations to the plan description. In particular, by default, it will replace conjunctive goals with plans, validated by AND-INTRO, with separate steps for each conjunct, and will similarly replace doubly-negated goals with plans, validated by TAUT-TRANS, whose sole step has the double-negation stripped off. The reason this phase is called Automatic Expansion is that these transformations are computed by calling the *designer* for plan generator EXPAND. The user may inhibit these transformations by setting `auto-expand*` to NIL.

The second function carried out by this phase is to propagate assumptions down to leaf `<goal_t>`'s. Remember that goal and plan descriptors may introduce assumptions through their ASSUMPTIONS slot. Therefore, a goal descriptor $\mathcal{G}$ which occurs within plan descriptor $\mathcal{P}_1$ which itself occurs as a step in plan descriptor $\mathcal{P}_2$ is such that its assumption set is really determined by the concatenation of $\mathcal{P}_2$'s assumptions, followed by $\mathcal{P}_1$'s assumptions, followed by $\mathcal{G}$'s own original set of assumptions.

It would be interesting to make it possible to specify more general transformations to be applied to plan descriptions. I have not looked into this. At present, the system simply expands conjunctions and negations (where applicable, i.e. whenever the principal symbol of the negated formula is another connective).

6.5.2 Computing The Validation

Each step of the plan must be denoted by an integer index, also called its *rank*. All assertion names (i.e. axioms/assumptions) are given negative step numbers. One reason for this is that it makes it easy to selectively hide them by setting `show-plan-negative-steps*` to NIL. We shall decrement the counter NEG to keep track of the last negative step number assigned. Goal descriptors, on the other hand, will be assigned positive step numbers. For this we increment the counter POS. Here is the algorithm:

validify [DESC]

  ⇒ POS := 0

  NEG := 1

  validify′ [DESC]

validify′ [NAME]

  ⇒ −−NEG

validify′ [(**<goal_t>** *p* ASSUMPTIONS RENAMED `nil`)]

  ⇒ ++POS

validify′ [(**<goal_t>** *p* ASSUMPTIONS RENAMED `t`)]

  ⇒ (* (++POS) (**weakened-proofs** asetify[ASSUMPTIONS][ASET]))

validify′ (**<plan_t>** *p* RULE STEPS PARAMETERS ASSUMPTIONS `nil`)

  ⇒ (RULE map[validify][STEPS] PARAMETERS)

validify′ (**<plan_t>** *p* RULE STEPS PARAMETERS ASSUMPTIONS `t`)

  ⇒ (* ((RULE map[validify][STEPS] PARAMETERS))

    (**weakened-proofs** asetify[ASSUMPTIONS][ASET]))

asetify [] [ASET′]

  ⇒ ASET′

asetify [FORMULA · REST] [ASET′]

  ⇒ asetify [REST] [make_assumption_set [FORMULA] [ASET′]]

In the above, ASET stands for the current assumption set. 'asetify' incrementally computes the assumption set ASET′ resulting from making the given ASSUMPTIONS on top of the current assumption set ASET. It is assumed that 'make_assumption_set' is a procedure which actually constructs an assumption set given the desired local assumption and the desired parent. As explained in Chapter 5, this procedure must *uniquify* its result: if the procedure is again invoked with the same local assumption and parent, it will return the previously computed assumption set instead of creating a new one.

6.5.3 Constructing The Plan Object

The plan generator was invoked at a goal node (or, occasionally, directly at a class node). The resulting plan objects purport to solve the corresponding class and are constructed by following the procedure below for each plan description:

1. First, the plan description's conclusion is unified with the class's pattern to determine an ALIST which indicates how the plan will instantiate the class's free variables. Some variables are thus revealed to become instantiated with terms whose shape is partially known; others are simply mapped to free variables in the plan. [see Chapter 5].
2. Then, all assertions are gathered which appear as steps in the plan description. Each one is converted to a solved step [Section 5.3.2] with the appropriate negative index [see previous section].
3. Similarly, all goal descriptors are gathered to be converted into goal objects. Here are the three major points of this conversion process:
   - The assumption set is computed from the goal descriptor's list of ASSUMPTIONS and the current assumption set ASET, as explained earlier (see function 'asetify' in previous section).

- The return variables are computed as follows: a free variable of the goal is considered a return variable iff either it corresponds to a return variable of the plan's class (either directly through the ALIST correspondence established above in item 1, or through the additional indirection of a renaming specified by the goal's RENAMED slot [page 201]), or it is mentioned in a sibling step.

- The goal is assigned the same positive index (a.k.a. rank) as when we computed the validation.

From this information a goal object may be constructed as documented in Chapter 5.

4. Finally, given the class, the ALIST, the solved steps, the other steps, and the validation, a plan object can be assembled, then initialized, as documented in Chapter 5.

6.6 Available Plan Generators

In this section, I list and document all plan generators available in LOGICALC.

6.6.1 Expand

The EXPAND plan generator handles all the obvious syntactic transformations. The plans it is capable of producing are justified in terms of TAUT-TRANS and sometimes AND-INTRO. Table 6.1 provides a summary of known expansions. In the first case, the current goal is a conjunction; expanding it produces a plan with one step for each conjunct. When the steps have been proven, their proofs are combined using inference rule AND-INTRO to produce a proof of the original conjunction.

Goal	Plan
$p_1 \wedge \ldots \wedge p_n$	$p_1, \ldots, p_n$ n steps
$p \supset q$	$\neg p \vee q$
$p_1 \vee \ldots \vee p_n$	$\neg \bigvee_{i \in A} p_i \supset \bigvee_{j \in B} p_j,\ A \cup B = \{1, \ldots, n\}$
$p_1 \vee \ldots \vee p_n$	$\bigvee_{i \in I} p_i,\ I \subseteq \{1, \ldots, n\}$
$\neg\neg p$	p
$\neg(p_1 \wedge \ldots \wedge p_n)$	$\neg p_1 \vee \ldots \vee \neg p_n$
$\neg(p_1 \vee \ldots \vee p_n)$	$\neg p_1 \wedge \ldots \wedge p_n$
$\neg(p \supset q)$	$p \wedge \neg q$

Table 6.1: Summary of Known Expansions

In the second case, the current goal is an implication; expanding it produces a plan with one step denoting the equivalent disjunction.

So far, invoking EXPAND required no arguments. In the third case, however, the current goal is a disjunction, and the object of the transformation is to obtain an equivalent implication. There are many ways of making an implication out of a disjunction, depending on which disjuncts we wish to put in the antecedent and which in the consequent. The user must specify by number which disjunct should be kept in the the consequent; all others are assigned to the antecedent. For example, `+plan expand 3 5` puts the 3rd and 5th disjuncts in the consequent, and all others in the antecedent.

In the fourth case, the current goal is also a disjunction, but the object is simply to drop one or more disjuncts and prove a stronger formula. The user must specify by number which disjuncts should be kept; all others are dropped. In order to distinguish this request from the previous one, the list of numbers should be preceded with `=`. For example, `+plan expand = 3 5` keeps the 3rd and 5th disjuncts and drops all the others.

The other cases do the obvious things.

6.6.2 Deduction Theorem

When the current goal is an implication, the natural way to prove it is to assume the antecedent and attempt to derive the consequent under this additional assumption. Therefore, for a goal of the form $A \Rightarrow p \supset q$, DEDUCTION-THEOREM constructs a plan whose sole step is $A; p \Rightarrow q$, where $A; p$ is a new assumption set whose parent is A and whose local assumption is p. The plan is validated by IF-INTRO which discharges the local assumption and infers the implication once again.

Unfortunately, the simple scheme outlined above no longer suffices when the antecedent p contains free variables. Indeed the naïve approach is no longer sound in that case because free variables occurring in the assumption set are *disconnected* from those occurring in the right hand side of the sequent. Much like in PROLOG, free variables occurring in an assertion are effectively renamed every time the assertion is used. I will now show that, because of this property, the naïve approach becomes unsound.

Consider the goal $A \Rightarrow p(x) \supset p(a)$. According to naïve DEDUCTION-THEOREM, we would proceed to look for a proof of $A; p(x) \Rightarrow p(a)$. From the local assumption $p(x)$ by LOGICALC's principal axiom, we derive $A; p(x) \Rightarrow p(u)$, where I have renamed x to u to emphacize the fact that the occurrence of the free variable on the right hand side of the sequent is not the same as the occurrence of x in the local assumption. Then, by SUBSTITUTION using $\{u \leftarrow a\}$, we can infer $A; p(x) \Rightarrow p(a)$. Finally discharging the assumption with IF-INTRO, we conclude $A \Rightarrow p(x) \supset p(a)$ which is clearly not valid (consider $\{x \leftarrow b\}$).

Three ways of dealing with this problem suggest themselves: (1) somehow maintain the connection with a free variable when the formula it occurs in is moved to the assumption set, (2) instantiate the free variables in the antecedent with arbitrary ground terms, (3) quantify the antecedent to capture the free variables within.

Maintaining the connection: This alternative is not practical with LOGICALC's approach. As intimated earlier, whenever an assertion is used to derive a conclusion, it is as if a new copy of the formula, with all free variables renamed, was used instead. Thus, different substitutions may apply in different inferences (e.g. $A, p(x) \Rightarrow p(a)$ and $A, p(x) \Rightarrow p(b)$).

In the case of DEDUCTION-THEOREM, when the antecedent contains free variables, the dynamic semantics that results in renaming the free variables when the assertion is retrieved from the database becomes unsound, as illustrated earlier.[1] To prevent this problem, we would need a scheme to annotate certain variables in such a way that they would not become implicitly renamed when an assertion mentioning them is retrieved from the database. Also, as further deductions proceed, these variables would become more and more instantiated. We would need a way to keep track of these substitutions.

What this proposal amounts to is to completely reify the sequent logic, and is not easily reconciled with an implementation of assumption sets as datapools. However, the mechanism for Automatic Generalization does take this approach [Chapter 8] because it is trivial to reify the deduction tree once it has been completely determined.

Instantiating with ground terms: The problem would not arise if there were no free variables in the antecedent. Therefore, a simple fix is to arbitrarily instantiate these variables with ground terms prior to moving the antecedent to the assumption set. The system can make up new skolem constants to serve as ground terms; if the user wants more flexibility in the particulars of the instantiation process, he should invoke SUBSTITUTION directly before DEDUCTION-THEOREM. This approach is not as drastic as it seems because Automatic Gen-

[1] Free variables are not actually renamed; the unifier constructs *internal environments* to keep track of assignments; each internal environment has a substitution for the left side and one for the right side, with a special notation to allow one side to reference a variable on the other side. An assertion's formula is taken to be on the left side, while the target formula provided by the generator's parser is considered to be on the right side. [see Chapter 3].

eralization will typically remove these ground terms and replace them with free variables when the validation is executed. On the other hand, if an arbitrary skolem constant will not do, and more specific terms are required, the user must guess these substitutions in advance and preestablish them with SUBSTITUTION before invoking DEDUCTION-THEOREM.

Quantifying the antecedent: The more general solution is to requantify the antecedent prior to moving it to the assumption set; e.g. for a goal of the form $A \Rightarrow p(x) \supset q$, first convert it to $A \Rightarrow (\forall y\, p(y)) \supset q$ using a skolemizing axiom (Section 3.5) $p(\alpha) \equiv \forall y\, p(y)$—which means that x becomes instantiated with the skolem constant α, and then, since the antecedent no longer contains free variables, apply DEDUCTION-THEOREM and produce the new goal $A; \forall y\, p(y) \Rightarrow q$. The latter's assumption set may be augmented with the formula $p(y)$ by one application of the inference generator SKOLEMIZE to the local assumption. Thus we get the convenience of free variables without the unsoundness (because x was instantiated with α).

Unfortunately, this approach works only for free variables occurring solely in the antecedent; it cannot apply to free variables occurring both in the antecedent and in the consequent, e.g. x in $A \Rightarrow p(x) \supset q(x)$. In this case, in order for the antecedent to be movable to the assumption set, the quantifier would have to be moved outside the sequent, thus yielding the reïfying approach of item 1, or x would have to be instantiated with ground terms as in item 2.

DEDUCTION-THEOREM examines the goal's formula. If there are free variables occurring both in the antecedent and in the consequent, it will make up new skolem constants to instantiate them. Then, if there are free variables remaining in the antecedent, it will capture them by quantification, adding the necessary skolemizing axiom to the database, and the validation will reflect the fact that a skolemization step is required to remove the quantifier in order to produce a proof of the original goal.

In order for the validation to successfully discharge the assumption resulting from assuming the antecedent, the proof of the consequent must have as its assumption set exactly this assumption set which was obtained by assuming the antecedent[2] (duh?). Consider the goal $A \Rightarrow p \supset \bullet q(a) \supset q(x)$. DEDUCTION-THEOREM produces the refinement $A; p \Rightarrow q(a) \supset q(x)$. Suppose that the user now effects the substitution $\{x \leftarrow a\}$. The new goal is $A; p \Rightarrow q(a) \supset q(a)$ which is tautological, therefore a proof is constructed which depends on no assumption: $\Rightarrow q(a) \supset q(a)$[3] Now it is time for the validation of the plan produced by DEDUCTION-THEOREM to be executed. The IF-INTRO inference rule is supposed to discharge the local assumption p, except that the proof's assumption set doesn't have p as its local assumption; in fact, it has no local assumption. In order to circumvent this problem, we must make sure to weaken the proof we are given before IF-INTRO attempts to perform its function. The hook mechanism is used to garantee that inference rule ASSUMPTION-INTRO is applied the right number of times to reestablish $A; p$ as the proof's assumption set. When that has been done, IF-INTRO can safely operate on $A; p \Rightarrow q(a) \supset q(a)$ to discharge p and derive $A \Rightarrow p \supset \bullet q(a) \supset q(a)$.

6.6.3 Modus Ponens

MODUS-PONENS implements the familiar back-chaining refinement. Given a goal $A \Rightarrow p_1$ and a major premise $A \Rightarrow q \supset p_2$, such that θ unifies p_1 and p_2 (i.e. $\theta p_1 \cong \theta p_2$), it produces the refinement $A \Rightarrow \theta q$. Typically it is invoked with the name of an assertion from which an appropriate implication can be detached, e.g. `+plan modus-ponens foo`; however, it can also be invoked with a conjecture to be set up as an auxiliary goal, e.g. `+plan modus-ponens =` *formula*. The code implementing this plan generator is short enough to be included here as an illustration:

[2] Translator's note: in order for the validation to discharge the assumption, the assumption has to be there.

[3] Actually, Automatic Generalization steps in and generalizes a away, producing $\Rightarrow p(y) \supset p(y)$.

code

```
(DEFFUNC PONENS-PARSE-PLANGEN - (LST (LRCD (LST <premise_t>)
                                           (LST sexp)))
    (SEQUENT - sequent_t PARMS - (LST sexp))
  (<# (\\ (P) (LRECORD (LIST P) NIL))
      (PROMPT-DETACHING (MAKE sequent_t
                          !>SEQUENT.ASET
                          '(IF ?* ,!>SEQUENT.CONCLUSION))
                        PARMS
                        (\\ () (OUT "Major premise? ")))))

(DEFFUNC PONENS-PLANGEN - (LST <plan_t>)
    (SEQUENT - sequent_t
     PREMS   - (LST <premise_t>)
     PARMS   - (LST sexp))
  (MATCH-COND !>(CAR PREMS).CONCLUSION
    (\? (IF ?P ?Q)
        (LIST (MAKE <plan_t> !>SEQUENT.CONCLUSION 'Modus-Ponens
                (LIST !>(CAR PREMS).HOW (MAKE <goal_t> P)) NIL)))
    (T NIL)))

(DEFFUNC PONENS-HELP-PLANGEN - void (X)
  (IGNORE X)
  (OUT 0 (&IV "MODUS-PONENS") " (Plan Generator)

    A=>q  --->  A=>p->q, A=>p

    Using an implication whose consequent unifies with the current
    goal, attempts to prove the corresponding instance of the
    implication's antecedent. In other words, it does one step of
    back-chaining. Typically, it is invoked as follows:
    ..."))

(MAKE plan_generator_t 'Modus-Ponens
  !'PONENS-HELP-PLANGEN
  !'PONENS-PARSE-PLANGEN
  !'PONENS-PLANGEN)
```

The NISP macro `<#` is a mapping construct much like `mapcar` in COMMON LISP, `\\` is an abbreviation for `lambda`, and `match-cond` is a pattern-matching conditional.

6.6.4 Modus Tollens

MODUS-TOLLENS is very similar to MODUS-PONENS. For a goal of the form $A \Rightarrow p_1$

and a major premise $A \Rightarrow \neg p_2 \supset q$, such that θ unifies p_1 and p_2, it produces the refinement $A \Rightarrow \neg\theta q$.

6.6.5 Resolution

Despite its name, this plan generator does not effect a resolution step; instead, it simply attempts to detach the current goal from the premises denoted by its arguments. The reason I called it RESOLUTION is that such a refinement bears a strong resemblance to what could be obtained by *unit non-clausal E-resolution* [see Chapter 7]. All the work is handled by the detaching procedure. The code is:

```
(DEFFUNC RESOLUTION-PARSE-PLANGEN - (LST (LRCD (LST <premise_t>)           code
                                               (LST sexp)))
    (SEQUENT - sequent_t PARMS - (LST sexp))
  (<# (\\ (P) (LRECORD (LIST P) NIL))
      (PROMPT-DETACHING SEQUENT PARMS)))

(DEFFUNC RESOLUTION-PLANGEN - (LST <plan_t>)
    (SEQUENT - sequent_t
     PREMS   - (LST <premise_t>)
     PARMS   - (LST sexp))
  (LET ((PREM (CAR PREMS)) - <premise_t>)
    (COND ((IS namddnode !>PREM.HOW)
           (LIST (MAKE <plan_t> !>PREM.CONCLUSION 'Identity
                   (LIST !>PREM.HOW) NIL)))
          ((IS <plan_t> !>PREM.HOW)
           (LIST (BE * !>PREM.HOW)))
          (T (EARROR RESOLUTION-PLANGEN NIL
               "PREM is a <goal_t>:" T PREM)))))

(DEFFUNC RESOLUTION-HELP-PLANGEN - void (X) ...)

(MAKE plan_generator_t 'Resolution
  !'RESOLUTION-HELP-PLANGEN
  !'RESOLUTION-PARSE-PLANGEN
  !'RESOLUTION-PLANGEN)
```

This plan generator is so frequently useful that the USE command stands for `+plan resolution`. Thus the user may simply type the more natural command `use foo` instead of `+plan resolution foo`.

6.6.6 Lemma

It is occasionally useful to prove a lemma on the side before proceeding with the rest of the proof. For a goal $A \Rightarrow p$ and a formula q provided as an argument, LEMMA produces the refinement $\{ A \Rightarrow q,\ A \Rightarrow p \}$. The validation simply uses TAUT-TRANS to drop the first conjunct. The formula to be proven as a lemma must be fully quantified with no free variables and specified as an argument. The code is:

code

```
(DEFFUNC LEMMA-PARSE-PLANGEN - (LST (LRCD (LST <premise_t>)
                                          (LST sexp)))
    (SEQUENT - sequent_t PARMS - (LST sexp))
  (LIST (LRECORD
         NIL
         (LIST (PROMPT-FOR-FORMULA
                (AND (CONSP PARMS) (CAR PARMS))
                (\\ () (OUT 0 "Type a fully quantified formula (containing"
                            T "no free variables) to serve as the lemma: ")))))))

(DEFFUNC LEMMA-PLANGEN - (LST <plan_t>)
    (SEQUENT - sequent_t
     PREMS   - (LST <premise_t>)
     PARMS   - (LST sexp))
  (LET ((CONC !>SEQUENT.CONCLUSION) - sexp)
    (LIST (MAKE <plan_t> CONC 'Taut-Trans
            (LIST (MAKE <plan_t>
                    `(AND ,(SKOLEMIZE (CAR PARMS)) ,CONC)
                    'And-Intro
                    (LIST (MAKE <goal_t> (GOAL-SKOLEMIZE (CAR PARMS)))
                          (MAKE <goal_t> CONC))
                    NIL))
            '((AND ?<P> ?<Q>) ?<Q>)))))

(DEFFUNC LEMMA-HELP-PLANGEN - void (X)
  (IGNORE X)
  (OUT 0 (&IV "LEMMA") " (Plan Generator)

   A=>p  ---> A=>lemma,A=>p

   Use this to prove a lemma on the side. Typically invoked by:

       +PLAN LEMMA lemma

   If the lemma has not been provided on the command line, the
   user will be prompted for it. The lemma should be a fully
```

```
    quantified formula, with no free variables whatsoever.
"))

(MAKE plan_generator_t 'Lemma
  !'LEMMA-HELP-PLANGEN
  !'LEMMA-PARSE-PLANGEN
  !'LEMMA-PLANGEN)
```

Note that, in many cases, it is preferable to specify a conjecture using the notation = *formula* rather than specify a lemma to be proven in this manner.

6.6.7 Substitution

SUBSTITUTION *alist* creates a plan whose sole step is the instance of the current goal created according to the substitution *alist*. The parser makes sure that the *alist* argument has the proper form (i.e. is a list of pairs (*var val*)) and has no circularities. It is validated by IDENTITY since unification is built into the logic.

6.6.8 Contradiction

The principle of a proof by contradiction is to assume the negation of the current goal and to use this assumption to derive a contradiction. Suppose the current goal is $A \Rightarrow p$, then the refinement is of the form $\{ A; \neg p \Rightarrow q,\ A; \neg p \Rightarrow \neg q \}$. From proofs of these two steps NOT-INTRO is able to conclude $A \Rightarrow p$. The formula q which serves as the contradiction must be specified as an argument.

In the earlier discussion of DEDUCTION-THEOREM, it was shown that we cannot move to the assumption set a formula containing free variables. The same is true here: if p contains free variables, they must first be captured by quantification, and the validation will have an extra skolemization step.

For example, if the current goal is $A \Rightarrow p(x)$, the skolemizing axiom $p(\alpha) \equiv \exists y\, p(y)$ is added to the database, x is instantiated with α, and the refinement has the two steps $\{ A; \neg\exists y\, p(y) \Rightarrow q,\ A; \neg\exists y\, p(y) \Rightarrow \neg q \}$. When the validation is executed, NOT-INTRO infers $A \Rightarrow \exists y\, p(y)$ from proofs of the two steps; then, from this last proof and the skolemizing axiom, SKOLEMIZE concludes $A \Rightarrow p(\alpha)$.

6.6.9 Equality

Plan generator EQUALITY produces refinements in which designated subterms of the goal have been replaced by other terms according to an equality provided as a premise. For example, if the current goal is $A \Rightarrow p(a)$ and `foo` is an assertion of the form $a = b$, then `+plan equality` a `- foo` will produce the refinement $A \Rightarrow p(b)$. The user should first specify the subterms to be replaced, using the notation discussed in section 6.2.2. Optionally he may also type a '-' followed by premise descriptors (see section 6.2.1) from which to obtain equalities.

The plan's validation invokes rule EQUALITY to effect the reverse replacements. The validation's parameters contain a list of paths to the subterms in question. As was discussed earlier, Automatic Generalization may generalize away some of the terms containing these subterms; in which case the corresponding subterms would no longer be present, and the validation could not perform successfully. To avoid unwarranted failure, it is necessary to reestablish enough structure to the proof's conclusion before EQUALITY attempts to perform its function. For this reason, the parameters now also include a skeleton-pattern which is unified with the proof's conclusion before EQUALITY begins its official task.

6.6.10 Case

In order to prove a goal by Case Analysis, the user must specify the cases in the form of an exhaustive disjunction. For example, `+plan case (or (p a) (not (p a)))` proposes exactly two cases: one in which `(p a)` is true, and one in which it is false. As a result, one step of the refinement will attempt to prove the goal under the additional assumption that `(p a)`, and another under the assumption that `(not (p a))`.

The disjunction does not have to be a tautology; it will be included as a step in the resulting plan. If it is outwardly a tautology, it will be revealed as such and converted to a solved step by the initialization procedure [Section 5.2.2].

Supposing the current goal is $A \Rightarrow q$ and the user proposes $p(a) \vee \neg p(a)$ as the disjunction of cases, it would be natural to expect a plan of the form:

1. $A \Rightarrow p(a) \vee \neg p(a)$
2. $A; p(a) \Rightarrow q$
3. $A; \neg p(a) \Rightarrow q$

However, the actual truth must perforce be somewhat more complex, as we shall see shortly. Let us further suppose that q is really $q(x)$, and that A contains the two assertions $p(a) \supset q(f(a))$ and $\neg p(a) \supset q(f(b))$. Obviously, I intend the first assertion to solve the first case, and the second one the second case, both through the mediation of MODUS-PONENS. However, if the plan looks like this:

1. $A \Rightarrow p(a) \vee \neg p(a)$
2. $A; p(a) \Rightarrow q(x)$
3. $A; \neg p(a) \Rightarrow q(x)$

as soon as the second step is solved, x becomes instantiated with $f(a)$, and the 3rd step, now $A, \neg p(a) \Rightarrow q(f(a))$ cannot besolved as expected because we have $q(f(a))$

instead of $q(f(b))$. As discussed earlier, the solution is to let x vary independently in either case: let it be renamed x_1 in the first case, and x_2 in the second case. The plan becomes:

1. $A \Rightarrow p(a) \vee \neg p(a)$
2. $A; p(a) \Rightarrow q(x_1)$
3. $A; \neg p(a) \Rightarrow q(x_2)$

Now the proofs will go through, and x_1 will become instantiated with $f(a)$, and x_2 with $f(b)$. At validation time we are handed:

1. $A \Rightarrow p(a) \vee \neg p(a)$
2. $A; p(a) \Rightarrow q(f(a))$
3. $A; \neg p(a) \Rightarrow q(f(b))$

The question is what to conclude from this, and, more specifically, what should x become instantiated with? The practical solution adopted here is to introduce a new constant α and conservatively extend the theory with the definitions: $p(a) \supset \alpha = a$ and $\neg p(a) \supset \alpha = b$.

Now, using MODUS-PONENS in the first case we can infer $\alpha = a$ from the first definition, and then, by one application of EQUALITY conclude $A, p(a) \Rightarrow q(f(\alpha))$ from the proof of step 2. Similarly in the second case we derive $A, \neg p(a) \Rightarrow q(f(\alpha))$. Now it becomes clear that the conclusion of Case Analysis must be $A \Rightarrow q(f(\alpha))$.

In general, the disjunction of cases has the form $p_1 \vee \ldots \vee p_n$; it must be exhaustive, but the cases are not necessarily disjoint. When a new constant α must be defined to be a_i in the ith case, we cannot naïvely define it with n axioms of the form $p_i \supset \alpha = a_i$. Doing so is unsound. To see why, suppose A contains the assertion $a_k \neq a_{k'}$ and that case k is not disjoint from case k' (for instance p_k might be equivalent to $p_{k'}$). In the intersection of case k and k' we can derive both $\alpha = a_k$ and $\alpha = a_{k'}$, and therefore

$a_k = a_{k'}$ which contradicts the assertion $a_k \neq a_{k'}$. Adding our definitional axioms caused the assumption set to become inconsistent.

To guarantee that this never happens, we must ensure that the cases are all pairwise disjoint. Instead of using the individual p_i's as the cases, we use p_1 for the 1st case, $\neg p_1 \wedge p_2$ as the second case, ..., $\neg p_1 \wedge \ldots \neg p_{n-1} \wedge p_n$ as the last case. Unfortunately, in the case of a simple tautology, such as $p(a) \vee \neg p(a)$, this policy has the effect of generating the plan:

1. $A \Rightarrow p(a) \vee \neg p(a)$
2. $A; p(a) \Rightarrow q(f(a))$
3. $A; \neg p(a) \wedge \neg p(a) \Rightarrow q(f(b))$

in which the local assumption for the 3rd step looks somewhat ludicrous.

Another point worth noting is that which conflicting subterms new α constants should be introduced for, and what their respective definitional axioms should be, cannot be determined in advance, but must be done at validation time. Once again, the hook mechanism must be called upon to intercept the proofs before the CASE rule gets hold of them, to make the necessary determinations, add all required definitional axioms, and effect the corresponding equality substitutions. When this is done, the conclusions in all cases have become identical and the CASE rule will successfully infer this common conclusion.

The hook calls function CASE-MAGIC which attempts to do the intuitively clever thing. One heuristic is to introduce a new constant only for innermost disagreeing subterms; by which I mean that for $q(f(a))$ and $q(f(b))$, α will be introduced to resolve the conflict a/b rather than, say, $f(a)/f(b)$.

Note that the conflicting terms may contain free variables $\overline{u}$, in which case the new skolem term introduced to resolve the conflict must also depend on these variables,

i.e. must have them as arguments: $\alpha(\overline{v})$, such that $\overline{u} \subseteq \overline{v}$. CASE-MAGIC attempts to minimize the number of free variables $\overline{v}$ by establishing a correspondence between free variables in the various cases.

Another point is to try not to introduce a skolem term to resolve a conflict between free variables, where a renaming would suffice. For example, it would be stupid to introduce a skolem term defined by $p_1 \supset \alpha(x) = x$ and $\neg p_1 \wedge p_2 \supset \alpha(y) = y$ to resolve the conflict x/y between $A; p_1 \Rightarrow q(x)$ and $A; \neg p_1 \wedge p_2 \Rightarrow q(y)$, where the renaming $y \rightarrow x$ applied to $q(y)$ would suffice.

Note that the rather complex scheme implemented here is all for the convenience of the user. Instead, before invoking the CASE plan generator, the user could capture all free variables in the goal by Existential Quantification, to be later followed by independent Skolemization in each case. Supposing we proceeded thus with the example used as an illustration in this section, we would first refine the goal $A \Rightarrow q(x)$ into $A \Rightarrow \exists x\, q(x)$ by means of QUANTIFY. Then we would invoke the CASE generator to produce the following plan:

1. $A \Rightarrow p(a) \vee \neg p(a)$
2. $A; p(a) \Rightarrow \exists x\, q(x)$
3. $A; \neg p(a) \wedge \neg p(a) \Rightarrow \exists x\, q(x)$

Then we could refine step 2 into $A; p(a) \Rightarrow q(x_1)$ by skolemization, derive $A; p(a) \Rightarrow q(f(a))$ as a proof, then conclude $A; p(a) \Rightarrow \exists x\, q(x)$ by quantification (as validation of the skolemization refinement), which would throw away all of the interesting substitution derived for x_1. Similarly for step 3. Case Analysis therefore yields the proof $A \Rightarrow \exists x\, q(x)$ and the validation of our early quantification step concludes $A \Rightarrow q(\alpha)$ by skolemization, where α is defined by the skolemizing axiom $q(\alpha) \equiv \exists x\, q(x)$.

As we can see, this last approach yields less information than the one implemented automatically by the CASE plan generator: contrast $q(\alpha)$ with $q(f(\alpha))$—much less information when all the cases turn out to be indentical, e.g. $q(\alpha)$ vs. $q(a)$, say.

6.6.11 Reduction

A redex is an expression representing the application of a λ-expression to arguments; it may denote either a proposition or a term. In LOGICALC, a redex is an expression of the form `(% (lambda` *vars body*`) .` *args*`)`. Reducing a redex consists of replacing the preceding expression with *body*[*args/vars*]; in other words, the redex is replaced by the body of the λ-expression in which actual parameters have been substituted for formal ones.

REDUCTION $i_1 \ldots i_n$ reduces the corresponding redexes in the current goal; where i_k denotes the i_k^{th} redex in the standard depth-first left-to-right enumeration.

REDUCTION's parser returns a list of paths to the redex subexpressions. These paths are then sorted so that the deeper redexes appear first. The reason is that a certain redex R_1 might be embedded within another redex R_2; if we reduced R_2 first, the path to R_1 would become invalid because 2 levels of structure are removed by reduction (the `%` level and the `lambda` level). By proceeding inside out, all remaining paths stay valid as the transformations are effected, which greatly simplifies the operation of the generator's designer.

Another problem to watch out for is the capture of free variables or constants by bound variables when formals are replaced by actuals. Suppose the current goal contains the expression:

```
(lambda (y) (% (lambda (x) (% (lambda (y) (f x y)) b)) y))
```

and that we wish to reduce the first redex `(% (lambda (x) ...) y)`. The naïve reduction would yield:

```
(lambda (y) (% (lambda (y) (f y y)) b))
```

in which the outer y has been captured by the binding for the inner y. The procedure effecting the substitution is careful to rename those variables of inner bindings which would otherwise capture an outer variable or a constant. Thus, REDUCTION actually produces:

```
(lambda (y) (% (lambda (y.1) (f y y.1)) b))
```

6.6.12 Abstraction

ABSTRACTION performs the function inverse to REDUCTION, i.e. it replaces a subexpression with a redex, such that reducing the redex yields the original expression again (modulo possible renamings of bound variables [see earlier]). The plan generator should be invoked using the following syntax:

+plan abstraction $redex_1$ -- $redex_2$ -- ... -- $redex_n$

where $redex_i$ describes a particular redex to be constructed and has the form:

subformula - var_1 $subterms_1$ - ... - var_m $subterms_m$

where *subformula* is a subterm descriptor denoting the subexpression of the current goal to be converted into a redex [see section 6.2.2]. Each var_j is the name of a variable for the redex's λ-expression; $subterms_j$ are subterms descriptors for occurrences of the expression to be abstracted as var_j. These occurrences should all be identical or unifiable. The $subterms_j$ descriptors are specified with respect to the *subformula*, not to the whole goal. For example, if the current goal is:

```
(p (f a a) b)
```

then '`+plan abstraction 1 - x 2`' causes the subterm `(f a a)` to be converted to a redex with the second occurrence of `a` abstracted as the variable `x`. The resulting refinement is:

```
(p (% (lambda (x) (f a x)) a) b)
```

Occasionally, it is convenient to introduce a variable that does not abstract a subexpression of the *formula*. In that case, it is necessary to provide the redex's actual parameter explicitly, and can be done using the notation var_j = $pattern_j$. Thus the command '`+plan abstraction 1 - x = (h c)`' for the same goal as above would produce the refinement:

```
(p (% (lambda (x) (f a a)) (h c)) b)
```

The notation can also be used in combination with subterm designators: var_j = $pattern_j$ $subterms_j$, in which case the designated subterms must be identical to, or unify with, $pattern_j$.

ABSTRACTION's parser is a nightmare (but nothing compared to QUANTIFY's). From the descriptions given on the command line, the parser constructs records describing the desired abstractions; each has the form ⟨PATH, ALIST⟩, where PATH is the path to a subexpression to be converted to a redex, and ALIST is a list of records describing the formal/actual parameters, each of which has the form ⟨VAR, PATHS, PATTERN⟩, where VAR is the name of the variable which is the formal parameter, PATTERN is the expression playing the rôle of the actual parameter, and PATHS are paths to subexpressions of the formula which are occurrences of PATTERN to be abstracted away by VAR.

Every path must be *normalized* to facilitate comparisons [page 195]. Unnecessary abstractions are removed, i.e. those of terms that will disappear anyway because some

larger term in which they occur will be abstracted away. Conflicting abstractions, such as of overlapping sequences, must be dealt with: the earliest one is kept, later ones are discarded. Then, the requested abstractions are sorted so that the deepest will be performed earliest; thus, as was the case for REDUCTION, the paths to the remaining subexpressions to be *redexified* will remain valid as the successive abstractions are carried out. However, the PATHS to the subterms within may not necessarily remain valid because new redexes may be introduced on the way. Consider the goal:

```
(p (f a (h a)) b)
```

and the command '**+plan abstraction h - x = c -- f - y a**'. The first abstraction yields:

```
(p (f a (% (lambda (x) (h a)) c)) b)
```

while the second one is meant to produce:

```
(p (% (lambda (y) (f y (% (lambda (x) (h y)) c))) a) b)
```

When the original command line is processed, the following 2 records are constructed: ⟨(1 2), (⟨x, (), c⟩)⟩, and ⟨(1),(⟨y ,((1) (2 1)), a⟩)⟩. Notice that, once the first abstraction has been carried out, the path (2 1), mentioned in the second abstraction record, and pointing to the second occurrence of a, is no longer valid because a new redex has appeared along the way. The correct path is now (2 1 2 1). In order for ABSTRACTION's designer to function correctly, these path corrections must be anticipated and performed in advance. I call this *path augmentation.* It is performed by the *parser* and amounts to inserting the sequence 1 2 at those strategic places where new redexes are expected to appear; the 1 selects the redex's λ-expression, while the 2 selects the λ-expression's body.

The validation for a plan produced by ABSTRACTION simply performs the corresponding reductions.

6.6.13 Skolemize

The point of Skolemization is to replace a quantified subformula with a similar formula where the quantifier has been removed and formerly quantified variables have been replaced by free variables or skolem terms as appropriate. Typically SKOLEMIZE is invoked thus:

+plan skolemize $i_1 \ldots i_n$

where i_k denotes the i_k^{th} occurrence of a quantified subformula in the traditional depth-first left-to-right enumeration. The occurrence descriptors are converted to paths, which are then sorted outermost first. In LOGICALC, it is not possible to skolemize a formula which occurs in the scope of another quantifier; this is the reason for sorting the desired skolemizations outermost first, and also justifies throwing out those that will remain in the scope of a quantifier.

Since skolemizations are performed outside-in, the paths to later subformulae to be skolemized may become invalid because quantifiers, which originally were on their paths, have been removed by earlier skolemizations. This is the opposite problem that we encountered with ABSTRACTION: there we had to augment paths by inserting the sequence `1 2` at strategic places, while here we must remove the index **2** from those places corresponding to quantifiers which we expect to be removed—the **2** served to pick the *body* from an expression of the form (**forall** *vars body*) or (**exists** *vars body*). These quantifiers are those in whose scope the formula appears, and they must all have been requested for skolemization in the user's command, otherwise the request for skolemizing the formula would have been discarded [see earlier]. The *parser* effects the necessary *path reductions.*

The quantifier of a formula to be skolemized may have either *universal* or *existential* force. A ∀ in a positive context has universal force, whereas it has existential force

in a negative context. A quantifier is in a positive (resp. negative) context if it is in the scope of an even (resp. odd) number of explicit or implicit negations. An explicit negation is expressed with the connective $\neg$. The antecedent of an implication is in an implicit negation because $p \supset q$ is equivalent to $\neg p \vee q$. It is well known that all quantifiers can be moved out of a formula by the process of conversion to prenex form [see e.g. [Man74]]; when this is done, quantifiers which had universal force will have been converted to $\forall$ in the prefix, whereas those which had existential force will now correspond to $\exists$.

When a quantifier has existential force, SKOLEMIZE simply removes it and replaces all corresponding (and formerly bound) variables with free variables. For example, if the current goal is $A \Rightarrow \exists x\, p(x)$, SKOLEMIZE will produce the refinement $A \Rightarrow p(x)$. Thus the free variable x is now allowed to become instantiated. When a proof, say $A \Rightarrow p(a)$, has been derived, the validation will reverse the refinement and reestablish the existential quantifier; in other words it will infer $A \Rightarrow \exists x\, p(x)$ from $A \Rightarrow p(a)$ by rule QUANTIFY.

When a quantifier has universal force, a skolemizing axiom [Section 3.5] is required for the refinement. For example, if the current goal is $A \Rightarrow \forall x\, p(x)$, the generator makes up a skolemizing axiom $p(\alpha) \equiv \forall x\, p(x)$ and adds it to the database, and then uses it to produce the refinement $A \Rightarrow p(\alpha)$. This new goal captures the intuition that if p can be shown to hold for a totally arbitrary object such as α, then surely it holds of any object x since the same proof would go through if x were substituted throughout for α. When a proof of $A \Rightarrow p(\alpha)$ has been derived, the validation, with the aid of the skolemizing axiom, infers $A \Rightarrow \forall x\, p(x)$.

Another way to invoke SKOLEMIZE is: `+plan skolemize` k `- foo`, where k denotes the kth quantified subformula (which had better not be in the scope of another quantifier), and `foo` is a premise designator [see section 6.2.1] presumably denoting a skolemizing axiom. In this case the generator does not invent a new skolemizing axiom since it was explicitly specified as an argument. Note that, since the detaching

procedure operates on the premise designator, it is a better idea to type `-raw foo` rather than simply `foo`.

If the current goal is:

```
(FORALL (X)
  (OR (P X)
      (FORALL (Y) (Q X Y))
      (EXISTS (Z) (R X Z))
      (EXISTS (U)
        (AND (P U) (FORALL (V) (S X V))))))
```

Then `+plan skolemize 1 4 5` produces the following plan

```
|- -1  (IF (S !.X !.V) (FORALL (V) (S !.X V)))
|-  0  (IF (OR (P !.X)
                (FORALL (Y) (Q !.X Y))
                (EXISTS (Z) (R !.X Z))
                (EXISTS (U)
                  (AND (P U)
                       (FORALL (V) (S !.X V)))))
           (FORALL (X)
             (OR (P !.X)
                 (FORALL (Y) (Q !.X Y))
                 (EXISTS (Z) (R !.X Z))
                 (EXISTS (U)
                   (AND (P U)
                        (FORALL (V) (S !.X V)))))))
     1  (OR (P !.X)
            (FORALL (Y) (Q !.X Y))
            (EXISTS (Z) (R !.X Z))
            (AND (P ?U) (S !.X !.V)))
```

6.6.14 Quantify

The point of Quantification is to replace a subformula with a quantified version of it, such that, from a proof of the refinement, a proof of the original goal can be

inferred by Skolemization. For example, it is possible to prove the goal $A \Rightarrow p(a)$ by first deriving the stronger proof $A \Rightarrow \forall x\, p(x)$, then $A \Rightarrow p(x)$ by Skolemization, and finally $A \Rightarrow p(a)$ by explicit substitution (typically omitted in favor of the same effect obtained implicitely through unification).

Introducing a quantifier with existential force requires a skolemizing axiom. For example, to produce a refinement for the goal $A \Rightarrow p(x)$, QUANTIFY must invent and add to the database the skolemizing axiom $p(\alpha) \equiv \forall y\, p(y)$. The new subgoal is $A \Rightarrow \forall y\, p(y)$ and x in the supergoal became instantiated with α. From a proof of the subgoal and the skolemizing axiom the validation ultimately derives $A \Rightarrow p(\alpha)$ by Skolemization. Note that this approach cannot apply to a goal of the form $p(a)$, where a is not a free variable because a won't unify with the new skolem term α.

Plan generator QUANTIFY should be invoked with the syntax:

+plan quantify $spec_1$ -- $spec_2$ -- ... -- $spec_n$

where $spec_i$ determines the quantification of a particular subformula of the current goal, and is typically of the form:

subformula - *quantifier* - var_1 $subterms_1$ - ... - var_m $subterms_m$

subformula is a subterm descriptor denoting the subformula to be quantified. *quantifier* is either **forall** or **exists**; it may also be **-universal** or **-existential**, in which cases the generator will determine, on the basis of the polarity of *subformula* in the goal, whether it should be **forall** or **exists**. Each var_i is the name of a variable to be bound by the *quantifier*, and $subterms_i$ are those subexpressions of *formula* to be quantified away by this variable.

When the quantification introduces a quantifier of existential force, the plan generator constructs and adds to the database a skolemizing axiom to be used to guide the

refinement and serve as a premise to SKOLEMIZE at validation time. As was the case in the case of SKOLEMIZE, here too it is possible to provide a premise descriptor (presumably denoting a skolemizing axiom) to avoid the creation and use of a new one. In this case, $spec_i$ should be of the form:

subformula - *quantifier* `-raw foo`

where `foo` is the name of an existing skolemizing axiom which applies to *subformula.*

The same remarks made earlier on ABSTRACTION's parser apply here too, and the code is essentially the same. Interpretation of the *specs*, however, is twice as complex because there are two distinct cases (universal and existential quantification) to consider instead of merely one. It should be noted that this code is written in such a way that it can also be used as the front end to inference generator QUANTIFY.

Chapter 7

Validations and Detachments

The refinement of a goal into a plan often requires that one or more premises be specified. In this chapter I describe a novel idea whereby a premise can be acquired through a non-clausal resolution-like process validated in terms of regular inference rules. This feature automates low-level inferential manipulations and relieves the user from the tedium of figuring out the details. It promotes and greatly enhances interactivity.

7.1 Introduction

In LOGICALC, the user will carry out a top-down analysis of the original goal by successive refinements of subgoals into plans. Each refinement is effected by invoking a plan generator [Chapter 6], and typically requires that one or more premises be specified. For example, Equality Substitution will replace occurrences of a term by another according to a given equality which must be provided as a parameter. If said equality is available in the assumption set as an assertion, then the assertion's name

can be used with no further ado. However, more often than not, the equality is not yet available as an individual assertion and must be derived.

Consider for instance the problem of establishing the truth of the following sequent:

$$q \supset a = b, q, p(a) \Rightarrow p(b)$$

A Natural Deduction proof for the above is:

1.	$q \supset a = b, q, p(a) \Rightarrow q \supset a = b$	Hypothesis
2.	$q \supset a = b, q, p(a) \Rightarrow q$	Hypothesis
3.	$q \supset a = b, q, p(a) \Rightarrow a = b$	Modus Ponens 1 2
4.	$q \supset a = b, q, p(a) \Rightarrow p(a)$	Hypothesis
5.	$q \supset a = b, q, p(a) \Rightarrow p(b)$	Equality 4 3

For an interactive approach to proof derivation, this linear limitation is very inconvenient because not only does it mean that you have to work out the proof before you can carry it out (instead of being able to adopt a more exploratory approach—which was the original intent), but also that you must derive the premises required by your refinements ahead of time as lemmas or perhaps forward inferences, thus negating the benefits of top-down refinement analysis.

In LOGICALC, if your goal is $p(b)$ and `foo` names the assertion $q \supset a = b$, then you may simply type:

```
+plan equality a - foo
```

to indicate that subterm a should be replaced according to an equality to be found in assertion `foo`. The system figures out what inferences are required to conclude the desired equality and what additional subgoals are necessary to carry these inferences out. I call this process "detaching." The system automatically determines how to detach $a = b$ from assertion `foo`.

Detaching $a = b$ from `foo`, i.e. $q \supset a = b$, can be achieved by Modus Ponens provided that we can show q. I say that $a = b$ is *detachable* from `foo` *modulo* q. q will automatically be inserted as an auxiliary step in the resulting plan. The principal step is, as expected, the one which is the result of substituting b for a in our goal, i.e. $p(a)$. Therefore, the plan will have two steps q and $p(a)$ which immediately match hypotheses.

Consider another example:

$$p \supset \centerdot q \supset r, p, \neg r \Rightarrow \neg q$$

This is precisely the sort of proof resolution is good at—especially if there are free variables involved—but a Natural Deduction proof must go through the following contortions:

1. $p \supset \centerdot q \supset r, p, \neg r \Rightarrow p \supset \centerdot q \supset r$	Hypothesis
2. $p \supset \centerdot q \supset r, p, \neg r \Rightarrow p$	Hypothesis
3. $p \supset \centerdot q \supset r, p, \neg r \Rightarrow q \supset r$	Modus Ponens 1 2
4. $p \supset \centerdot q \supset r, p, \neg r \Rightarrow \neg r$	Hypothesis
5. $p \supset \centerdot q \supset r, p, \neg r \Rightarrow \neg q$	Modus Tollens 3 4

Clearly, here too we should like the system to take care of these details. If `baz` names assertion $p \supset \centerdot q \supset r$, the user may type:

```
+plan resolution baz
```

and LOGICALC will automatically determine that p and $\neg r$ should be set up as auxiliary goals. I chose to name the plan generator **`resolution`** because it provides an apparent functionality close to that of (unit) non-clausal resolution, eventhough the validation [Section 7.3] is expressed in terms of regular inference rules (not resolution).

The idiom "**`+plan resolution`**" is so common that I defined the command **`use`** to mean the same thing. Instead of typing '**`+plan resolution baz`**', the user may simply say '**`use baz`**'.

7.2 Resolution and Natural Deduction

Resolution has some advantages which are relevant to interactive theorem proving, and some which are not. Relevant advantages include: (1) the use of unification to determine the required substitutions, and (2) the ability to pick any literal to resolve on in a clause (and not just, say, the first one). Because of the preliminary conversion to clausal form, the search space becomes very homogeneous, thus making it easy to build automatic theorem-provers. Also, resolution is refutation-complete. However, since my interest is in improving the user interface rather than in automating search, the last two properties are somewhat removed from the issue.

On the other hand, resolution has many disadvantages. I will ignore those related to automated search, and will concentrate instead on those which are of some import to interactive proof derivation.

- Resolution requires that all formulae undergo a preliminary conversion to clausal form. While this has the effect, noted earlier, of making the search space homogeneous, therefore better suited to automated search, it also eliminates a considerable amount of structural or pragmatic information encoded in the original, notationally richer, formulation (e.g. contrast $p \supset q$ with $\neg p \vee q$).
- This same conversion to clausal form introduces a lot of redundancy as a result of repeated applications of the distributive laws.
- People find clausal form difficult to understand. There are several reasons for that: (1) skolemization introduces new unfamiliar terms which become increasingly complex as new substitutions are performed—however, I have shown how the judicious use of abbreviations alleviates this problem—(2) the new representation is rather far removed from the original input and no longer possesses any kind of structure; (3) finally, the same one formula will typically result

in several clauses; thus, semantically related ensembles become scattered and difficult to grasp as a whole.

7.2.1 Non-Clausal Resolution

It is possible to remedy the problems listed above by extending the principle and permitting resolution to occur between formulae not in clause form. This procedure is known as non-clausal resolution. Non-clausal resolution operates on quantifier-free well-formed formulae. The procedure can create a resolvent from formulae containing atoms occurring with opposite polarity. Polarity is determined by the parity of the number of negations, explicit or implicit (as for the antecedent of an implication), in the scope of which the atom is located. The following definition is adapted from [Mur82]:

> *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be quantifier-free well-formed formulae such that the atom* $\mathbf{L}$ *occurs with opposite polarity in* $\theta\mathbf{A}$ *and* $\theta\mathbf{B}$, *where* θ *is a substitution. Let* $\{ L_1, \ldots, L_n \}$ *be the set of all atoms occurring in* $\mathbf{A}$ *and* $\mathbf{B}$ *such that* $\theta L_i \cong \mathbf{L}$. *The following formulae are non-clausal resolvents of* $\mathbf{A}$ *and* $\mathbf{B}$*:*
>
> $$\mathsf{S}^{\mathbf{L}}_{\mathrm{T}}\theta\mathbf{A} \vee \mathsf{S}^{\mathbf{L}}_{\mathrm{F}}\theta\mathbf{B}$$
> $$\mathsf{S}^{\mathbf{L}}_{\mathrm{F}}\theta\mathbf{A} \vee \mathsf{S}^{\mathbf{L}}_{\mathrm{T}}\theta\mathbf{B}$$
>
> *where* $\mathsf{S}^{\mathbf{L}}_{\mathrm{T}}\theta\mathbf{A}$ *denotes the result of substituting all occurrences of* $\mathbf{L}$ *by the constant* T *(i.e.* TRUTH*) in the formula* $\theta\mathbf{A}$ *(i.e. the formula obtained by applying substitution* θ *to* $\mathbf{A}$*).*

Typically, simplifications are applied to the resolvents to eliminate the truth constants T and F, e.g. $p \wedge \mathrm{T} \rightarrow p$.

Stickel [Sti82] also advocates the use of non-clausal resolution, in the context of a connection-graph driven procedure. However, the restriction that resolution should occur on atomic subformulae is not suitable for my purpose. My intention is to

extract premises for use in refinements, and those premises will often be non-atomic, e.g. Modus Ponens requires an implication.

This limitation can be overcome by extending non-clausal resolution to resolve on (non necessarily atomic) subformulae. In fact, this was already suggested in [Mur82].

7.2.2 Translating Proofs to Natural Deduction

Resolution, and other automated proof procedures, have the drawback that the representations they operate on as well as the presentation of their conclusions is neither intended nor adequate for human consumption. This deficiency has prompted a number of researchers to study ways of translating from proofs expressed in a language designed for automated search into formats better suited for inspection by people.

For example, Chester [Che76] looked at the translation of formal proofs into english. Andrews studied the translation of *matings* into Natural Deduction proofs [And81]. This idea of translating mechanical proofs to a Natural Deduction format brings up another point about automated search procedures and user involvement. Miller and Felty [MF86] make the following remark:

> *[...] On the other hand since the search in such a theorem prover is carried out in a space which is rather removed from a user's original input, it is difficult to get the user to interact with the search process.*

To address this issue, and following a result by Pfenning [Pfe84] that a resolution refutation can be converted to an expansion tree proof, they propose a system which explicitly represents both kinds of proofs and is able to translate between them.

7.2.3 Conclusions for LOGICALC

Since user involvement is a primary concern in LOGICALC, the detaching procedure should be both powerful and intuitive and should operate transparently. Non-clausal resolution has this to offer: (1) unification, and (2) the ability to perform an inference based on a subformula. Both these features are highly desirable and should be retained.

On the other hand, refutation proofs are in a format which is not suitable for human consumption; refinements based on such resolution steps are harder to understand and make it progressively more difficult for the user to interact with the system. For both these reasons, it is often desirable to elucidate a resolution proof (or step) by translating it into a Natural Deduction format. Since automated search is not the focus of this work, there is no need to actually represent resolvents, instead, I shall translate them imediately to Natural Deduction conclusions.

LOGICALC's detaching procedure (1) will be based on unification, (2) will allow the user to use a subformula as a premise in a refinement, (3) will use a Natural Deduction format. In this regime a premise is the description of a plan for inferring the desired formula; this plan is to be inserted into the corresponding refinement.

7.3 Premises and Validations

In LCF, a goal is analysed top-down, and the corresponding proof is then constructed bottom-up by composing validations, where a validation is a function which maps a tuple of theorems to a new theorem. Similarly in LOGICALC, goals are refined into plans and subgoals until the process bottoms out when all leaf goals have been matched with assumptions or axioms. Each plan includes a validation which, unlike LCF, is not a function but rather an expression representing the inferences to be

performed. When all steps of a plan have acquired answers, the inferences prescribed by the validation are carried out by a special interpreter. In both LCF and LOGICALC, executing a validation may fail to produce a conclusion if the conditions for the successful application of each inference rule are not all satisfied.

7.3.1 The Point of Validations

It is interesting to contrast this approach with backward-chaining systems such as PROLOG. The later do not incorporate a bottom-up validation phase, nor do they seem to require it, or are perceived as lacking in this respect. So what do we really need validations for? I expounded on this topic in Chapter 5, but it is probably a good idea to go over the main points here again anyway.

The following 4 items argue either for the necessity or the advantage of having validations and explicit proof objects:

double-check the validity of a refinement: Backward-chaining systems are entirely driven by rules such as Modus Ponens and And Introduction. Such a rule, when applied backwards, yields a refinement for which the validation is the forward application of the same rule. Obviously, for any given refinement obtained in this manner, the corresponding validation will always successfully produce a conclusion. Therefore, validations would provide a superfluous double-check for these systems. Similarly, PROLOG is driven by SLD resolution, which is sound.

In LOGICALC, however, refinements are obtained through the mediation of plan generators. A plan generator is essentially a function mapping a goal to a tuple of subgoals. Such a mapping may be arbitrarily complex or general and there is no reason to expect that an instance of the goal can be validly inferred from proofs of the subgoals. The validation makes sure that this inference is legal

by attempting to perform it. If it succeeds, it also determines the necessary substitution.

A plan generator corresponds more or less to a tactic in LCF. Here is what Paulson [Pau87] says on this topic:

> *There is no guarantee that the validation will prove the goal.* LCF *cannot prove a false statement, but a validation can raise an exception, run forever, or produce an irrelevant theorem. An invalid tactic is worst that useless: it can lead you down an incorrect path in a proof. The validation fails at the last moment, when the proof seems to be finished. As Milner says, 'Such dishonest tactics — those that promise more than can be performed — are to be avoided.'*

LOGICALC follows this advice, and validations are expected to succesfully produce conclusions for their respective plans. Failure is currently interpreted as a design flaw in the plan generator responsible for the refinement, and an error is signalled.

perform post-completion processing: Even after all steps of a plan have acquired answers, it may not be possible to immediately derive a conclusion. For instance, it may be known in advance that some further inference steps will have to be interposed, but the specifics cannot be determined until the steps have been solved because, say, they depend on the particular substitutions involved.

For instance, case analysis typically requires that the proofs of the steps be further transformed (by additional inferences) before they can be fed to the case rule; these transformations are effected by equality substitution. Which subterms must thus be replaced cannot be determined in advance because it depends on how free variables become instantiated in the course of the proof.

Validations provide a means to perform arbitary post-completion processing where necessary.

make conclusions available: (for example, to subsequent deductive operations).

If a conclusion has been derived once, there is no reason to force the user to

rederive it if he needs it again at some later point. This is an argument for having explicit proof objects, and therefore for having validations to produce them.

Having explicit proof objects also allows us to address the issues of subsumption and answer sharing [see Chapter 5].

Finally, without access to the corresponding complete proof tree, it would not be possible to generalize conclusions [see Chapter 8].

package conclusions for inspection: proofs should be available for human perusal. In particular, it should be possible to edit and prepare proofs for publication. For this it is necessary that each proof contain the complete trace of its derivation.

7.3.2 Representation of Validations

A validation is an expression that describes how to produce the conclusion of a plan given proofs of its steps. It is of the form:

(*inference_rule* (*premises...*) (*parameters...*)) .

inference_rule is the name of the rule to be invoked, *premises* are typically integers—where n denotes the proof of the plan's n^{th} step—and *parameters* further specify the particulars of the inference to be carried out (e.g. Equality Substitution requires that you specify which subterms are to be subtituted).

For example, if the current goal is q and `foo` is an assertion of the form $p \supset q$, then executing '**`+plan modus-ponens foo`**' will result in a plan with the following 2 steps:

0. `foo`
1. p

Step 0 is already solved since it is an assertion; it is nonetheless included in the plan because it will be needed when the validation is executed. Step 1 is the goal obtained by backward-chaining through `foo`. The validation has the following form:

```
(modus-ponens (0 1) ())
```

and it means to apply Modus Ponens to a proof of step 0 as major premise and a proof of step 1 as minor premise; no parameters are required, and therefore none are specified.

In general however, a validation's *premises* are allowed to be either integers or validations, to be recursively interpreted in a similar manner. This property of compositionality of validations make them a completely general tool for representing arbitrary inference trees, and permit the following:

- After a plan generator has produced the representation for a plan, and before this representation is effectively turned into an actual plan, further transformations may be applied to it and the validations thereof inserted within.

  For example, suppose the current goal is q and `foo` is an assertion of the form $p_1 \wedge p_2 \supset q$. The command '`+plan modus-ponens foo`' will result in the creation of a description for the plan:

  0. `foo`
  1. $p_1 \wedge p_2$

  Clearly, it would often be profitable to replace $p_1 \wedge p_2$ by the two individual steps p_1 and p_2. This further refinement can be determined by inspection of the above representation and effected on it directly:

  0. `foo`
  1. p_1
  2. p_2

Also, the validation is correspondingly transformed from `(modus-ponens (0 1) ())` to:

```
(modus-ponens (0 (and-intro (1 2) ())) ())
```

I dubbed this process "automatic goal expansion," and described it in Chapter 6.

- The ability to compose validations make it possible to write very "intelligent" plan generators to produce refinements which may be arbitrarily complex and require arbitrarily large inference trees to validate them.

  Except for Case Analysis, plan generators do not currently make much use of this capability because the "detaching" procedure generally obviates the need for it.

- The compositionality of validations also permits to plug in premises obtained by detachment. A detachment is really the representation of a plan. Often, it is the plan to derive as a conclusion a particular subformula of a given assertion and it may be inserted in the overall refinement where the premise was expected; the validations will be correspondingly composed.

7.3.3 Premises and Detachments

Plan generators often require premises as parameters. For example, to effect a refinement according to Modus Ponens, it is necessary to specify an implication as a premise. As long as the appropriate implication is available as an assertion in the assumption set, we can simply name it in order to use it. However, and more often than not, the desired formula is embedded within a larger assertion and can only be obtained through tedious additional inferences. Occasionally, it is not even known whether the desired premise is provable, but the user would like to make the refinement anyway, and only subsequently derive a proof for the postulated premise.

LOGICALC reconciles all these requirements by making premises be representations of how to derive the conclusion to be used as a parameter. I will introduce the various aspect of these representations through the study of several examples.

1. If the goal is $p(a)$ and `foo` is an assertion of the form $a = b$, then a refinement can be effected by the command:

   `+plan equality` a `- foo`

   which produces the following plan description:

   `(plan` $p(a)$ `equality (foo (goal` $p(b)$`)))`

   from the above description, a plan is finally constructed which has the structure shown below:

   Steps:
   [0. `foo`]
   1. $p(b)$
   Validation:
   `(equality (0 1) ((1)))`

   This example illustrate the simplest case, where the name of an assertion (e.g. `foo`) is used to represent a premise.

2. Now suppose that in fact there are no assertions to the effect that $a = b$. But we would like to make this conjecture anyway and perform the refinement, and only subsequently derive a proof of $a = b$. This can be achieved by the command:

   `+plan equality` a `- = (=` a b`)`[1]

[1]Sorry for mixing the standard mathematical notation $a = b$ with the actual LISP representation `(=` a b`)`, but it cannot be avoided here.

The first '=' indicates that the following specification is a formula to be used as a premise; but, since it does not come with a proof, it must also be set up as a subgoal. The plan description generated in this case is:

```
(plan p(a) equality ((goal (= a b)) (goal p(b))) ((1)))
```

Here, the premise is a goal description. The resulting plan is:

Steps:

1. $(= a\ b)$
2. $p(b)$

Validation:

```
(equality (1 2) ((1)))
```

3. Finally, suppose that `foo` is an assertion of the form $q \supset a = b$. We would like to perform the same refinement as before, and simply indicate to the system that the necessary equality $a = b$ should be inferred from assertion `foo`. To this end, the user may issue the following command:

```
+plan equality a - foo
```

The system automatically determines what inferences and subgoals are required to conclude $a = b$ from `foo`. This process is called "detaching." The resulting plan representation is:

```
(plan p(a) equality
    ((plan (= a b) modus-ponens (foo (goal q)))
     (goal p(b)))
  ((1)))
```

Here, the premise is a subplan description containing the subgoal q. The resulting plan is:

Steps:

[0. `foo`]

1. q
2. $p(b)$

Validation:

```
(equality ((modus-ponens (0 1) ()) 2) ((1)))
```

7.4 The Detaching Procedure

First, what does it mean exactly for a formula to be detachable from a premise?

> **P** *is detachable from* **A** *if its underlying unsigned formula occurs in* **A** *with the same polarity.*

The underlying unsigned formula is the formula you get when you strip off **P** all explicit top level negations. For example q_2 and $\neg p$ are both detachable from $p \supset q_1 \wedge q_2$. Furthermore, I say that q_2 is detachable "modulo" p because it is necessary to prove p in order to infer q_2 from $p \supset q_1 \wedge q_2$ by Modus Ponens followed by Taut-Trans. Similarly, $\neg p$ is detachable modulo $\neg . q_1 \wedge q_2$.

> **P** *is detachable from* **A** *modulo* $\{q_1, \ldots, q_n\}$ *if it is detachable from* **A** *and actually inferring* **P** *from* **A** *requires proofs of* $q_1 \ldots q_n$.

The notion of "modulo" is pretty intuitive. If I want to detach something from a conjunction, I can just infer the interesting conjunct and drop the others. No auxiliary goal is required in this case. On the other hand, if I want to detach something from a disjunction, I need to infer the interesting disjunct by proving the negations of the others; the negated other disjuncts constitute the modulo.

Another property of this notion is: if **P** is detachable from **A** modulo $\{q_1, \ldots, q_n\}$ then $\neg[q_1 \wedge \ldots \wedge q_n]$ is tautologically equivalent to the formula you get by replacing the occurrence in **A** of the underlying unsigned target by F if its polarity is positive, T otherwise, and simplifying [cf non-clausal resolution].

Although I defined detachability in terms of the underlying unsigned formula, the algorithm really uses the underlying simply signed formula which can be obtained by stripping as many explicit double-negations off **P** as possible. The reason this amounts to the same thing is that as **A** is being explored, implicit negations (such as in the antecedent of an implication) will be made explicit to the search process. The underlying simply signed formula is also called the "target" in the detaching procedure.

7.4.1 Detachment Records

The detaching procedure produces records which have the following structure:

$$\langle \text{FROM}, M, \text{PATH}, \theta \rangle$$

Where FROM is a premise in the style of Section 7.3.3, i.e. a description of a conclusion to detach from: it is either an assertion, the description of a goal, or the description of a plan. In practise, it is always either an assertion or a goal description. M is the number of double-negations that were stripped off the original formula to produce the target. PATH is a list of integers describing how to access the subformula in FROM's conclusion which this record purports to detach. Finally, θ is the substitution resulting from the unification of the target with the subformula denoted by PATH.

I will adopt the following convention: $i \cdot \text{PATH}$ denotes the list of integers whose first element is i and the rest is PATH. Conversely, $\text{PATH} \cdot i$ denotes a list whose last element is i and the rest (i.e. prefix) is PATH. Also, The notation PATH^{-1} stands for the list PATH reversed.

7.4.2 The Detaching Algorithm

We are given a formula 'P' and a premise 'FROM'. The object is to construct a list of detachment records representing possible detachments of this formula from this premise.

First, we determine the TARGET by stripping off the formula as many double-negations as possible. Let M be the number of double negations which were thus removed. Now, we are going to recursively explore the premise's conclusion for occurrences of TARGET, or rather for subformulae which unify with TARGET.

We are also given the boolean parameter DEEP which is false when auxiliary goals are disallowed. When we are detaching for input to a plan generator, then DEEP is true because auxiliary goals can be accommodated in the resulting premise. However, when we are detaching for input to an inference generator [page 55], it must be possible to carry out the inference immediately, therefore auxiliary goals cannot be allowed; in this case, DEEP is false.

The algorithm is expressed in a mixture of mathematical notation and functional programming with pattern matching. The entry point is 'detach'; 'detach′' implements the unification step which is tried for every subformula encountered during exploration; and 'detach″' performs a case analysis which keys recursive exploration off the shape of the current subformula.

Each function returns a set (list) of detachment records. Only 'detach′' creates new records. The construct '**when** ⟨*test*⟩: ⟨*result*⟩' returns ⟨*result*⟩ only when ⟨*test*⟩ succeeds, otherwise the empty set $\emptyset$. For each function, the first argument is a subformula of FROM's conclusion, and the second argument is the PATH to this subformula. We are looking for a subformula which will unify with TARGET.

detach [p] [PATH]

$\Rightarrow$ detach′ [p] [PATH] $\cup$ detach″ [p] [PATH]

detach′ [p] [PATH]

$\Rightarrow$ **when** θTARGET $\underset{\text{unify}}{\cong}$ θp: $\{\, \langle \text{FROM}, M, \text{PATH}^{-1}, \theta \rangle \,\}$[2]

detach″ [$p_1 \wedge \ldots \wedge p_n$] [PATH]

$\Rightarrow \bigcup_{i=1}^{n}$ detach [p_i] [$i \cdot$ PATH]

detach″ [$p_1 \vee \ldots \vee p_n$] [PATH]

$\Rightarrow$ **when** DEEP **or** $n = 1$: $\bigcup_{i=1}^{n}$ detach [p_i] [$i \cdot$ PATH]

detach″ [$p \supset q$] [PATH]

$\Rightarrow$ **when** DEEP: detach [$\neg p$] [$1 \cdot$ PATH] $\cup$ detach [q] [$2 \cdot$ PATH]

detach″ [$\neg\neg p$] [PATH]

$\Rightarrow$ detach [p] [$1 \cdot$ PATH]

detach″ [$\neg \centerdot p_1 \wedge \ldots \wedge p_n$] [PATH]

$\Rightarrow$ **when** DEEP **or** $n = 1$: $\bigcup_{i=1}^{n}$ detach [p_i] [$i \cdot$ PATH]

detach″ [$\neg \centerdot p_1 \vee \ldots \vee p_n$] [PATH]

$\Rightarrow \bigcup_{i=1}^{n}$ detach [p_i] [$i \cdot$ PATH]

detach″ [$\neg \centerdot p \supset q$] [PATH]

$\Rightarrow$ detach [p] [$1 \cdot$ PATH] $\cup$ detach [$\neg q$] [$2 \cdot$ PATH]

detach″ [$p(x, y)$] [PATH]

$\Rightarrow$ **when** p is symmetric: detach′ [$p(x, y)$] [$-3 \cdot$ PATH]

detach″ [$\neg p(x, y)$] [PATH]

$\Rightarrow$ **when** p is symmetric: detach′ [$\neg p(x, y)$] [$-3 \cdot$ PATH]

7.4.3 Converting a Detachment into a Premise

A detachment is not directly usable by a generator; it must first be converted to the appropriate representation. For a plan generator, it should be expressed as a premise,

[2] Because of segment variables [ref], unification may produce several unifiers θ_j for $1 \leq j \leq m$. In this case, detach′ should return $\bigcup_{j=1}^{m} \{\, \langle \text{FROM}, M, \text{PATH}, \theta_j \rangle \,\}$.

e.g. the representation of a plan in the style introduced and illustrated earlier. For an inference generator, it should be expressed as a proof. In this section I will present the algorithm for converting a premise record into the representation of a plan, to serve as premise to a plan generator.

It should be noted that "find mode" will let the user browse through candidate detachments, and display and select those which are deemed of interest. This requires yet another conversion scheme: from a record to a format suited for presentation to a user.

The entry point is 'premify' whose argument is a detachment record. The recursive conversion is carried out by 'premify'' whose first argument is the subformula detached so far, second argument is a PATH into its first argument to the subformula to be ultimately detached, and its third argument is a premise describing how to infer the first argument—this is typically a plan description.

premify $\langle$FROM, M, PATH, $\theta\rangle$

    ⇒ **let** FORMULA = conclusion of FROM **in**
        let PREMISE = premify′ [θFORMULA] [PATH] [FROM] **in**
            while $M > 0$ **do**
                let CONC = conclusion of PREMISE **in**
                    PREMISE := `(plan` ¬¬CONC `taut-trans (`PREMISE`)`
                        `(`x `(not (not` x`))))`
                $M := M - 1$
            return PREMISE

premify′ [p] [] [HOW]

    ⇒ HOW

premify′ [$p_1 \wedge \ldots \wedge p_n$] [$i \cdot$ PATH] [HOW]

    ⇒ premify′ [p_i] [PATH] [`(plan` p_i `taut-trans (`HOW`) ((and` x_1 ... x_n`)` x_i`))`]

premify′ [$\vee p$] [$1 \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [p] [PATH] [(plan p taut-trans (HOW) ((or x) x))]

premify$'$ [$p_1 \vee \ldots \vee p_n$] [$i \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [p_i] [PATH] [(plan p_i modus-ponens

((plan $\bigwedge_{1 \le j \le n}^{j \ne i} \neg p_j \supset p_i$ taut-trans (HOW)

((or x_1 ... x_n)

(if (and ... (not x_j) ...)$_{j \ne i}$ x_i)))

(goal (and ... (not x_j) ...)$_{j \ne i}$)))

premify$'$ [$p \supset q$] [$1 \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [$\neg p$] [PATH] [(plan $\neg p$ modus-tollens (HOW (goal $\neg q$)))]

premify$'$ [$p \supset q$] [$2 \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [q] [PATH] [(plan q modus-ponens (HOW (goal p)))]

premify$'$ [$\neg\neg p$] [$1 \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [p] [PATH] [(plan p taut-trans (HOW) ((not (not x)) x))]

premify$'$ [$\neg \wedge p$] [$1 \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [$\neg p$] [PATH] [(plan $\neg p$ taut-trans (HOW) ((not (and x)) (not x)))]

premify$'$ [$\neg \centerdot p_1 \wedge \ldots \wedge p_n$] [$i \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [$\neg p_i$] [PATH] [(plan $\neg p_i$ modus-ponens

((plan $\bigwedge_{1 \le j \le n}^{j \ne i} p_j \supset \neg p_i$ taut-trans (HOW)

((not (and x_1 ... x_n))

(if (and ... x_j ...)$_{j \ne i}$ (not x_i))))

(goal (and ... x_j ...)$_{j \ne i}$)))

premify$'$ [$\neg \centerdot p_1 \vee \ldots \vee p_n$] [$i \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [$\neg p_i$] [PATH] [(plan $\neg p_i$ taut-trans (HOW)

((not (or x_1 ... x_n)) (not x_i)))

premify$'$ [$\neg \centerdot p \supset q$] [$1 \cdot$ PATH] [HOW]

$\Rightarrow$ premify$'$ [p] [PATH] [(plan p taut-trans (HOW) ((not (if x y)) x))]

premify$'$ [$\neg \centerdot p \supset q$] [$2 \cdot$ PATH] [HOW]

$\Rightarrow$ premify′ [$\neg q$] [PATH] [`(plan` $\neg q$ `taut-trans (HOW) ((not (if` x y`)) (not` y`)))`]

premify′ [$p(x, y)$] [$-3 \cdot$ PATH] [HOW]

$\Rightarrow$ premify′ [$p(y, x)$] [PATH] [`(plan` $p(y, x)$ `symmetry (HOW))`]

premify′ [$\neg p(x, y)$] [$-3 \cdot$ PATH] [HOW]

$\Rightarrow$ premify′ [$\neg p(y, x)$] [PATH] [`(plan` $\neg p(y, x)$ `symmetry (HOW))`]

7.4.4 Converting a Detachment into a Proof

Inference generators [page 55] can also benefit from the detaching mechanism. However, since subgoals do not make sense when conclusions are derived in a forward manner, only detachments which require no auxiliary goals should be allowed. This requirement is enforced by requiring FROM to be an assertion (or more generally a proof), and by setting DEEP to false in the detaching algorithm. Thus, all constructed detachments have modulo $\emptyset$.

When assertion **A** *detaches* **P** *modulo* $\emptyset$*, we say that* **A** *"asserts"* **P**.

The algorithm to convert a detachment satistying the above requirements into a proof is straightforward and involves only the two inference rules 'taut-trans' and 'symmetry'.

The entry point is 'proofify'. Remember that FROM is a proof. The recursive inference of the formula to be detached is carried out by proofify′ whose first argument is the formula concluded so far, second argument is a PATH into its first argument to the subformula to be ultimately concluded, and its third argument is a proof of its first argument.

proofify $\langle$FROM, M, PATH, $\theta\rangle$

$\Rightarrow$ **let** FORMULA = conclusion of FROM **in**

let PROOF = proofify′ [θFORMULA] [PATH] [FROM] **in**

while $M > 0$ **do**

PROOF := taut-trans [PROOF] [x `(not (not` x`))`]

$M := M - 1$

return PROOF

proofify′ [p] [] [PROOF]

$\Rightarrow$ PROOF

proofify′ [$p_1 \wedge \ldots \wedge p_n$] [$i \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [p_i] [PATH] [taut-trans [PROOF] [`(and` x_1 ... x_n`)` x_i]]

proofify′ [$\vee p$] [$1 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [p] [PATH] [taut-trans [PROOF] [`(or` x`)` x]]

proofify′ [$\neg\neg p$] [$1 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [p] [PATH] [taut-trans [PROOF] [`(not (not` x`))` x]]

proofify′ [$\neg \wedge p$] [$1 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [$\neg p$] [PATH] [taut-trans [PROOF] [`(not (and` x`))` `(not` x`)`]]

proofify′ [$\neg . p_1 \vee \ldots \vee p_n$] [$i \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [$\neg p_i$] [PATH] [taut-trans [PROOF] [`(not (and` ... x_i ...`))` `(not` x_i`)`]]

proofify′ [$\neg . p \supset q$] [$1 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [p] [PATH] [taut-trans [PROOF] [`(not (if` x y`))` x]]

proofify′ [$\neg . p \supset q$] [$2 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [$\neg q$] [PATH] [taut-trans [PROOF] [`(not (if` x y`))` `(not` y`)`]]

proofify′ [$p(x, y)$] [$-3 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [$p(y, x)$] [PATH] [symmetry [PROOF]]

proofify′ [$\neg p(x, y)$] [$-3 \cdot$ PATH] [PROOF]

$\Rightarrow$ proofify′ [$\neg p(y, x)$] [PATH] [symmetry [PROOF]]

7.5 Detaching Modulo Equalities

Now that the detaching technique enables us to "extract" a subformula from a premise for use as input to a plan or inference generator, I will extend it further by introducing a provision for inexact matching. The general idea is this: given a goal, it is often clear to the user that it should be solved using a particular assertion (say `foo`), but attempts at producing detachments fail because some subterms in the goal do not unify with the corresponding subterms of the targeted subformula in `foo`. In such a case, we should like the system to allow mismatches and supplement the resulting detachment with the necessary equality substitutions; the equality formulae required for these substitutions will have to appear as auxiliary goals.

Such a mechanism is available in LOGICALC and can be requested by means of the options **`+mismatches`** and **`-mismatches`** n. The introductory example of Chapter 1 illustrated this feature.

Several ideas have been proposed in the past to deal with equalities. Most notably E-Resolution, which is certainly the more natural extension that can be grafted to the Non-Clausal Resolution rule;[3] and congruence closure-based systems which are a rather attractive alternative if the size of equality classes can somehow be controlled [see Focused Binding in ONTIC [McA87]].

7.5.1 E-Resolution

Theories with equality offer an additional challenge to theorem-proving programs. Since reasoning from the equality axioms is difficult, requires very long derivations, and typically causes many redundant or irrelevant conclusions to be generated, the *paramodulation* rule was proposed [RW69] to permit more direct equality substitutions.

[3]The actual search procedure, e.g. level saturation, is not the issue here.

From two clauses $\mathbf{A}[\alpha]$ *and* $\alpha' = \beta' \vee \mathbf{B}$ *such that* $\theta\alpha \cong \theta\alpha'$ *where* θ *is a mgu, infer* $\theta\mathbf{A}[\beta'] \vee \theta\mathbf{B}$.

Unfortunately, paramodulation is still a very undirected process and E-Resolution was proposed instead [Sib69,Mor69]: paramodulating decisions should be directed by unification attempts and corresponding disagreement sets.

From two clauses $\mathbf{A} \vee P[t_1]$ *and* $\neg P[t_2] \vee \mathbf{B}$ *conclude* $\mathbf{A} \vee \mathbf{B} \vee t_1 \neq t_2$

More generally:

From two clauses $\mathbf{A} \vee P(t_1, \ldots, t_n)$ *and* $\neg P(t'_1, \ldots, t'_n) \vee \mathbf{B}$ *such that* θ *is the most general partial unifier (mgpu)*[4] *of* $P(t_1, \ldots, t_n)$ *and* $P(t'_1, \ldots, t'_n)$ *and* $\{ \langle s_1, s'_1 \rangle, \ldots, \langle s_k, s'_k \rangle \}$ *are the irreducible (innermost) disagreements, conclude* $\theta\mathbf{A} \vee \theta\mathbf{B} \vee s_1 \neq s'_1 \vee \ldots \vee s_k \neq s'_k$.

Digricoli [DH86] further extends this approach with his notion of Resolution by Unification and Equality (RUE). He defines the following two rules:

RUE Rule of Inference (Open Form): The RUE resolvent of $P(s_1, \ldots, s_n) \vee A$ and $\neg P(t_1, \ldots, t_n) \vee B$, is $\sigma A \vee \sigma B \vee D$, where σ is a substitution and D is a disjunction of inequalities specified by a disagreement set of the complementary literals $\sigma P(s_1, \ldots, s_n)$, $\sigma \neg P(t_1, \ldots, t_n)$.

NRF Rule of Inference (Open Form): The Negative Reflective Function (NRF) resolvent of the clause $t_1 \neq t_2 \vee A$ is $\sigma A \vee D$, where σ is a substitution and D is a disjunction of inequalities specified by a disagreement set of σt_1, σt_2.

[4]The most general partial unifier is obtained by performing regular unification while skipping over irreducible disagreements.

7.5.2 Congruence Closure

David McAllester implements a different approach in his interactive system ONTIC [McA87]. He proposes the idea of equality constraint graphs. All the terms in his language are compiled into nodes of a graph. A labelling process colors these nodes according to the equivalence classes (of the equality relation) they respectively belong to. This computation is based on a congruence closure algorithm [see e.g. [DST80]]. Nodes with the same color are considered equal.

Although very attractive, his approach requires compilation to a special purpose representation. Also, ONTIC's inference mechanisms were designed for a different paradigm of interactive proof derivation, and are not easily reconciled with the principle of a Natural Deduction.

7.5.3 Extensions to Handle Mismatches

I believe that E-Resolution offers the alternative which can be more naturally retrofitted in the framework of a Non-Clausal Resolution inspired detaching procedure as presented earlier.

Consider: It would often be very convenient if, from the goal $p(b)$ and an assertion `foo` of the form $q \supset p(a)$, one could directly produce the refinement:

Steps:

[0. `foo`]

1. q
2. $a = b$

Validation:

```
(equality (2 (modus-ponens (0 1) ())) ((1)))
```

In order to support this capability, I extended the unification and detaching procedures. The resulting functionality is a reflection of Unit Non-Clausal E-Resolution into Natural Deduction.

Extension to Unification

I define the most general partial unifier of A and B to be a pair $\langle\theta, \{\text{PATH}_1, \ldots, \text{PATH}_n\}\rangle$ where θ is a substitution constructed by carrying out regular unification while skipping over irreducible disagreements, and $\{\text{PATH}_1, \ldots, \text{PATH}_n\}$ represents precisely the set of those irreducible disagreements. Each PATH_i is a list of integers denoting the subexpressions of θA and θB which disagree.

The actual implementation must be prepared to handle disagreements arising during unification with segment variables [Section 2.6.2]. Segment variables have several nasty effects on unification: (1) there may be more than 1 mgu, (2) in fact there may be infinitely many and unification cannot be complete, (3) the path to a subterm (or sequence of subterms) on one side may be different from the path to the corresponding subterm (or sequence of subterms) on the other side. Thus, the disagreement set of an mgpu must be represented by pairs of paths. For the purpose of the present exposition, I will ignore this complication.

The new boolean parameter MISMATCHES is defined, and is set to true when partial unification is desired. The unification procedure proceeds in the same recursive fashion as before, but if a step fails[5] and MISMATCHES is true an alternative is attempted: a procedure is invoked to decide whether to allow this disagreement; if the procedure answers "no" then this recursive unification step utterly fails; however, if it answers "yes," a new disagreement is recorded in the partial unifier under construction—note

[5] Except for the case of failure to unify a variable to a term containing an occurrence of this variable.

that in order for this to be possible, the unification procedure has to be extended to keep track of the current "path" into both formulae.

Management of the disagreement set could be made a lot more sophisticated. In particular, no check is currently done to see if a new disagreement is identical—in terms of the subexpressions involved—to one already recorded in the partial unifier. Also, no attempt is being made to take transitivity of equality into account. In practice, these shortcomings have little incidence on the technique.

The decision procedure which is invoked in order to determine whether to allow a disagreement is parametrized as follows:

E-RESOLUTION-MAX*:

This may be NIL or a positive integer.. In the later case, if the current partial unifier already contains this many disagreements, the new one is disallowed, and consequently unification fails. The default value is 1; it can be dynamically changed through the premise specification '`-mismatches` n' which will reset it to n for the duration of this particular detachment [page 194].

E-RESOLUTION-FCN*:

This may be NIL or another decision procedure. In the later case, this other procedure is called to make the final determination. The user may program an arbitrarily complex procedure to suit his own needs; the system's default has the following behavior:

- If the disagreement is about a functor,[6] then fail. Remember that if this recursive unification step fails utterly, then its parent step will fail too and similarly will attempt to record a disagreement into the partial unifier. The net effect is that an attempt to unify $p(f(a))$ and $p(g(a))$ will fail when f and g are tentatively unified, but no disagreement will be allowed

[6] I.e. something in functional position.

at that level, thus unification of $f(a)$ and $g(a)$ also fails and this time the disagreement will be permitted—corresponding to the equation $f(a) = g(a)$.

- If disagreements have been limited to expressions whose main symbol is in the list E-RESOLUTION-AUTHORIZED-FUNCTORS*, and neither expression is of this form, then fail.[7] The default value is NIL; it can be dynamically changed through a premise specification; e.g. '`-mismatches (f g)`' will only allow mismatches on terms whose main symbols are either `f` or `g`.
- If the disagreeing subexpressions are propositions, then fail. In this logic, equality between propositions is not meaningful. For a different approach see for instance Andrews' $\mathcal{Q}_0$ system [And86].
- Otherwise succeed—meaning that the disagreement will be allowed and dutifully recorded in the partial unifier.

Extending The Detaching Procedure

Now that unification has been extended to return most general partial unifiers as defined in the previous section, detachment records too must be similarly extended to include the disagreement set. Therefore a detachment record is now a structure of the form:

$$\langle \text{FROM}, M, \text{PATH}, \theta, \{ \text{PATH}_1, \ldots, \text{PATH}_n \} \rangle$$

'detach′' must be redefined as follows:

detach′ $[p]$ [PATH]

$$\Rightarrow \textbf{when } \theta\text{TARGET} \underset{\text{e-unify}}{\cong} \theta p \quad (\text{mod } \{ \text{PATH}_1 \ldots, \text{PATH}_n \}):$$
$$\{ \langle \text{FROM}, M, \text{PATH}^{-1}, \theta, \{ \text{PATH}_1 \ldots, \text{PATH}_n \} \rangle \}$$

[7] Actually, the code is little more intricate because it matters for purpose of interactivity to take into account whether the expression in question is in the target or in the premise from which a detachment is being extracted.

Note that, if MISMATCHES is false, the behavior is identical to that of the earlier version and $n = 0$.

Similarly, 'premify' must also be modified to take the disagreements into account and insert appropriate equality substitution steps. I will use the notation $p[\text{PATH}_i]$ to represent the subexpression denoted by PATH_1 in proposition p; and $p[t/\text{PATH}_i]$ will stand for the formula obtained from p by substituting t for the subexpression denoted by PATH_i.

premify ⟨FROM, M, PATH, θ, { $\text{PATH}_1 \ldots, \text{PATH}_n$ }⟩
  ⇒ **let** FORMULA = conclusion of FROM **in**
    let PREMISE = premify′ [θFORMULA] [PATH] [FROM] **in**
      for $i := 1$ **to** n **do**
        let CONC = conclusion of PREMISE **in**
        let TERM_1 = TARGET[PATH_i] **in**
        let TERM_2 = CONC[PATH_i] **in**
          PREMISE := (plan CONC[$\text{TERM}_1/\text{PATH}_i$] equality
                ((goal (= TERM_2 TERM_1)) PREMISE) (PATH_i))
      while $M > 0$ **do**
        let CONC = conclusion of PREMISE **in**
          PREMISE := (plan ¬¬CONC taut-trans (PREMISE)
                (x (not (not x))))
          $M := M - 1$
      return PREMISE

7.6 Conclusion

In this chapter I presented a uniform mechanism whereby premises to plan (and inference) generators may be assertions or user specified formulae or subformulae of

same. This I called the Detaching Procedure, and an implementation was presented. I further extended it to allow mismatches during unification with targeted subformulae. I characterize the resulting functionality as a reflection of Unit Non-Clausal E-Resolution into Natural Deduction; Non-Clausal because (1) clausal conversion is not required and (2) the target may be a complex formula rather than a literal; Unit because one of the formula, the target, is resolved on, while the other is resolved into; E-Resolution because mismatches are permitted and result in auxiliary equality goals.

This mechanism must now be interfaced inwardly with the rest of the system (e.g. generators and the database) and outwardly with the user. The user interface consists primarily in the language of premise specifications which is documented in Chapter 6; it also includes a special purpose browser called Find Mode, which allows the user to browse throught detachments and select those of interest [Appendix B].

Each plan generator must parse its own arguments; therefore it must also interpret the premise descriptors which the user specified on the command line. This can be achieved by calling:

$$\text{prompt-detaching } [A \Rightarrow p]\ [\langle \mathit{descriptor} \rangle]\ [\langle \mathit{prompt} \rangle]$$

where A is the current assumption set and p is the desired formula to be detached, $\langle \mathit{descriptor} \rangle$ is the user specified premise specification, and $\langle \mathit{prompt} \rangle$ is a function providing context specific prompting should further prompting turn out to be necessary (such as when $\langle \mathit{descriptor} \rangle$ is empty). This call returns a list of premises.

Similarly, for inference generators:

$$\text{prompt-asserting } [A \Rightarrow p]\ [\langle \mathit{descriptor} \rangle]\ [\langle \mathit{prompt} \rangle]$$

returns a list of proofs. Both 'prompt-detaching' and 'prompt-asserting' are implemented using the more general:

$$\text{prompt-for-premises } [A \Rightarrow p]\ [\langle \mathit{descriptor} \rangle]\ [\text{DEEP}]\ [\langle \mathit{prompt} \rangle]$$

which returns a list of detachments. 'prompt-detaching' sets DEEP to true and converts the detachments using 'premify', while 'prompt-asserting' sets DEEP to false and converts the results with 'proofify'.

The last necessary bit of functionality is an interface to the database that enables 'prompt-for-premises' to quickly look up candidate assertions to detach from in the assumption set. This is implemented by the function 'consult-index' which is described in Chapter 2.

Chapter 8

Automatic Generalization

In this chapter I explain LOGICALC's mechanism for the automatic generalization of proofs. I discuss the advantage of having such a mechanism and show that it is closely related to Explanation-Based Generalization. However, EBG's regression procedure is not sound within LOGICALC's original framework and I propose a reification of the logic to get around this problem. Finally the algorithm is presented.

8.1 Introduction

In LOGICALC, the preferred representation of a proposition—whether a goal or an assertion—is as a skolemized formula. For example, to prove $\forall x\, p(x)$, the system would set up a goal of the form $A \Rightarrow p(\alpha)$, where α is a new skolem term. Unfortunately, if a proof is derived for this goal, it too will be of the form $A \Rightarrow p(\alpha)$ rather than the more general $A \Rightarrow p(x)$, even though it is clear, since α was not mentioned in the original theory, that the proof would have gone through for any term substituted throughout for α. Obviously, explicit manipulations of quantifiers can get us around this problem by specifically deriving a proof of $A \Rightarrow \forall x\, p(x)$ and then manually adding

$p(x)$ to the assumption set by forward application of the skolemization rule to the newly derived conclusion. However, the point of chosing skolemized formulae as the default representation was precisely to avoid such manipulations in the first place.

Within a framework based on skolemization, it is still possible to get much the same effect as explicit quantifier manipulations without actually requiring that they be carried out. In a skolemized formula, occurrences of skolem terms implicitly carry the original quantification structure. It is possible to recover this structure through a procedure known as *reverse skolemization*, which can be automated [BB78,CP80]. For instance, from the proof $A \Rightarrow p(\alpha)$, we might recover $A \Rightarrow \forall z\, p(z)$ and then further conclude $A \Rightarrow p(z)$.

I argue, however, that this is simply the wrong way to look at, or even characterize, the problem. Consider the tautology $p(a) \supset p(a)$: clearly, its status as a tautology does not depend on the identity of the term a. $p(b) \supset p(b)$ is a tautology for precisely the same reason: the complete trace of the verification procedure is identical in both cases, although the data operated on is different. It would be desirable to produce the more general conclusion $p(x) \supset p(x)$, where x is a free variable.

Why should we attempt to derive more general conclusions? Aside from the aesthetic statisfaction of the purist, there is a more practical reason: whenever a proof is derived for a goal, LOGICALC makes this proof available so that it might serve to solve other goals or be used as a premise in subsequent refinements. This is done through two mechanisms: firstly the proof is asserted in the database (this is how it can subsequently serve as a premise), and secondly it is *broadcast* to related goals [see Chapter 5] so that they may examine the proof to see if it can solve them too.

Clearly, the more general the proof, the more widely applicable too. Thus a larger number of goals can be solved with no additional work required from the user (or the system for that matter). The particular sense of generality used here is the one implied by unification: a conclusion p is more general than p' if p' is an instance of

p. To generalize a conclusion means to replace some of its subexpressions with free variables in such a way that the new formula thus obtained is also a valid inference.

The purpose of the reverse skolemization technique proposed earlier was to allow us to replace certain skolem terms with free variables, but it did not cover the case of generalizing over non-skolem terms. The characterization of generalization formulated in the previous paragraph, however, treats skolem terms and other terms alike; furthermore, unlike reverse skolemization, it does not require a preliminary step to recover quantifiers.

The generalized conclusion must be valid. In fact, for various reasons such as "expectations of pending validations," it must be valid in precisely the sense that it can be derived through the same proof structure (i.e. inference tree) as the original conclusion. The object of generalization is to find the least instantiated *skeleton* of the latter for which there exists a proof tree isomorphic to the one we were given. This determination can be made by regressing the conclusion's skeleton through the proof structure in the manner of Explanation-Based Generalization.

8.2 Explanation-Based Generalization

The object of Explanation-Based Generalization (henceforth EBG) is to formulate general concepts from specific training examples by explaining, with the aid of a domain theory, why the training examples are members of the concepts being learned. In [MKKC86], the EBG *problem* is defined as follows:

Given:

- *Goal Concept:* A concept definition describing the concept to be learned (it is assumed that this concept fails to satisfy the Operationality Criterion).

- *Training Example:* An example of the goal concept.
- *Domain Theory:* A set of rules and facts to be used in explaining how the training example is an example of the goal concept.
- *Operationality Criterion:* A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

Determine:

- A generalization of the training example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion.

Then the EBG *method* is defined thusly:

1. *Explain:* Construct an explanation in terms of the *domain theory* that proves how the *training example* satisfies the *goal concept* definition.
   - This explanation must be constructed so that each branch of the explanation structure terminates in an expression that satisfies the *operationality criterion.*
2. *Generalize:* Determine a set of sufficient conditions under which the explanation structure holds, stated in terms that satisfy the *operationality criterion.*
   - This is accomplished by regressing the *goal concept* through the explanation structure. The conjunction of the resulting regressed expressions constitutes the desired concept definition.

In [KCM87], EBG is presented as an augmentation of SLD-resolution for Horn Clause Logic, and an implementation is provided in the form of a PROLOG meta-interpreter. A simple PROLOG meta-interpreter for pure PROLOG is:

```
prolog(A)      :- clause(A,true).                                    prolog
prolog((A,B)) :- prolog(A),prolog(B).
prove(A)       :- clause(A,B),prove(B).
```

In EBG, a variabilized version of the original query is regressed through the same proof tree, and the generalization is formed by propagating rule substitutions but ignoring fact substitutions. Using the meta-interpreter technique, regression through the proof tree can be performed as the tree is being constructed, i.e. concurrently with PROLOG execution. This can be realized by adding, to the meta-interpreter, an argument which is a (less instantiated) reflection of the goal and with which we perform the same back-chaining steps as with the actual goal, but skip unification with facts (more precisely, we skip the proofs of operational subgoals). The desired generalization consists in the set of the images of all operational subgoals in the resulting proof tree: in order to collect them, we need to add a 3rd argument to the meta-interpreter.

```
ebg(true,true,[]).                                                   ebg
ebg(A,GenA,[GenA]) :- operational(A),!,A.
ebg((A,B),(GenA,GenB),Op) :-
        ebg(A,GenA,OpA),
        ebg(B,GenB,OpB),
        append(OpA,OpB,Op).
ebg(A,GenA,OpA) :-
        clause(GenA,GenB),
        copy((GenA:-GenB),(A:-B)),
        ebg(B,GenB,Op).
```

The meta-interpreter would typically be invoked by:

```
ebg(p(f(a),g(b)),p(X,Y),OpDef)
```

where `p(f(a),g(b))` is the training example, `p(X,Y)` is the skeleton to be regressed and is the variabilized image of the training example,[1] and `Opdef` is a list of op-

[1] Although this implementation requires that the user specify the skeleton to be regressed, the latter could easily be computed automatically instead.

erational subgoals defining the concept, and which can be used, e.g., to efficiently recognize instances of it.

8.3 Application to Proof Generalization

The problem of generalizing a conclusion can be construed as a restricted case of EBG, in which the `operational` predicate is always false, and the result we care about is the 2nd argument of `ebg/3` rather than the operational definition collected in the 3rd argument (empty in this case). In fact, since we are given the entire proof tree (a.k.a. explanation structure), on the face of it the task should be simpler; however, there are some complications.

Firstly, EBG's algorithm was developed for explanation structures constructed with the aid of back-chaining rules. Any practical system of Natural Deduction, however, offers a set of inference rules that is much richer and goes beyond mere back-chaining (a.k.a. Modus Ponens). Since each rule may require arbitrarily complex computations, regressing through an inference step will typically be correspondingly harder [see e.g. Equality].

Another source of complications is the ability to make assumptions using the Deduction Theorem in the course of a proof. EBG presupposes a fixed, flat domain theory. The ability to make assumptions means that the effective domain theory of a subproof may be a superset of the original domain theory. As we shall see shortly, the straightforward extension of EBG to allow for assumptions no longer preserves soundness. More generally:

> *The regression technique of EBG is sound only if the system has the property that all inferences involving ground terms remain valid when these terms are replaced with free variables.*

LOGICALC's logical system does not satisfy the above requirement because of the built-in *copy on fetch* effect similar to that of PROLOG. In PROLOG, whenever a rule of the form `p(X) :- q(X).` is used, a new `X` is created so that multiple uses of this rule in the course of the same program do not conflict; it is as if each time `X` was renamed differently.

Similarly in LOGICALC, the typical axiom schema is $A, p(x) \Rightarrow p(t)$; that is, from any hypothesis, we are licensed to infer an instance of it. In particular, we may derive $A, p(x) \Rightarrow p(y)$ where the variable y is free for x in $p(x)$; this is what I meant by *copy on fetch*; the free variable x in the hypothesis remains unaffected. Unfortunately, in general, discharging an hypothesis containing a free variable is not sound: from hypothesis $p(x)$ I may conclude $p(a)$ according to the axiom sequent $A, p(x) \Rightarrow p(a)$; but naïvely discharging said hypothesis yields $A \Rightarrow p(x) \supset p(a)$ which is not valid (consider e.g. $x \leftarrow b$).

The idealized characterization of EBG's algorithm applied to the domain of logical proofs is: take the conclusion to be generalized, construct the most variabilized skeleton corresponding to its propositional shape, regress the latter through the proof structure. The object is to determine the most general substitution of the skeleton's variables such that each inference step of the proof structure remains valid. Because of the limitation discussed above, the intended generalization cannot always be carried out within the original logical framework.

Consider for example the following Natural Deduction proof:

1.	$A, a = b \Rightarrow z = z$	axiom
2.	$A, a = b \Rightarrow f(a) = f(a)$	from **1**
3.	$A, a = b \Rightarrow a = b$	assumption
4.	$A, a = b \Rightarrow f(a) = f(b)$	from **2** and **3** by equality
5.	$A \Rightarrow a = b \supset f(a) = f(b)$	discharging assumption $a = b$

Clearly this proof does not depend on the identity of a or b. Obviously, the proper generalization is of the form $A \Rightarrow x = y \supset f(x) = f(y)$, where x and y are free

variables. EBG would attempt to regress the skeleton $u_1 = u_2 \supset f(u_3) = f(u_4)$[2] through the proof structure, which would result in:

1.	$A, u_1 = u_2 \Rightarrow z = z$	axiom
2.	$A, u_1 = u_2 \Rightarrow f(u_3) = f(u_3)$	from **1**
3.	$A, u_1 = u_2 \Rightarrow u_3 = u_4$	assumption
4.	$A, u_1 = u_2 \Rightarrow f(u_3) = f(u_4)$	from **2** and **3** by equality
5.	$A \Rightarrow u_1 = u_2 \supset f(u_3) = f(u_4)$	discharging assumption $u_1 = u_2$

This is clearly wrong. One source of the problem is the *copy on fetch* which occurs in step 3; another is the invalid discharge of an assumption containing free variables in step 5.

The actual root of the problem is a conflation of intended meanings for free variables. Let us consider $u_1 = u_2$ and, in particular, the meaning which it is ascribed by the logical system when it is inserted in the assumption set. Since, from an hypothesis we are licensed to infer any instance of it, from the asssumption $u_1 = u_2$ we may conclude that any term is equal to any other term. Clearly, that is not at all what we had in mind. What we wanted was to find a substitution for $u_1 \ldots u_4$ such that every inference step in the proof structure remained valid: $u_1 \ldots u_4$ were meant to be existentially quantified not at the level of the skeleton, but over the whole proof tree. In other words, we want to consider the proof tree as a term over which to quantify: we must reify proofs as well as the logic that serves to validate them.

8.4 A Reified Sequent Logic For Horn Clauses

In this section, I will illustrate the technique extending EBG to handle assumptions by providing a PROLOG meta-interpreter in the manner of Kedar-Cabelli and McCarty

[2] Actually, it would start out with $u_1 = u_2 \supset u'_3 = u'_4$, and regression through step 4 would determine the assignments $u'_3 \leftarrow f(u_3)$ and $u'_4 \leftarrow f(u_4)$. For the moment, we can ignore this point of detail.

[KCM87]. The program extends PROLOG to accommodate the notion of assumptions in a very simple way: Firstly, goals are redefined to be of the form $A \Rightarrow p$, where A is a list of assumptions expressed as Horn clauses. Secondly, goals of the form $A \Rightarrow p{:-}q$ are allowed and the interpreter implements the following strategy to solve them: q is assumed (in some well-defined way to be explicitated later) and p is tentatively solved under this additional assumption; in other words, $A \Rightarrow p :- q$ is turned into $A, q \Rightarrow p$ modulo a few details. Note that extending PROLOG with hypothetical implications is not a novel idea, see e.g. N-PROLOG [CR84] for an in-depth discussion.

The first thing to realize is that this *copy on fetch*, which one gets for free in PROLOG, is nothing but skolemization in disguise. Every clause in the database can be understood as having a implicit prefix of universal quantifiers capturing all of its free variables. Fetching a clause from the database essentially effects an Alphabetic Change of Bound Variables using brand-new names, then strips away the useless prefix.

In this new framework where assumptions are allowed, we need to be able to make the distinction between those free variables for which new copies must be made every time the corresponding clause is fetched from the assumption set, and those which are meant to be existentially quantified over the whole computation. These two kinds of free variables are (1) free variables in the object logic, and (2) free variables at the meta-level. On the right hand-side of a sequent both kinds may be represented by PROLOG variables. Whenever an assumption is made, however, it must be processed so that free variables at the object-logic level are all identified and new copies can be created for them at fetch-time, while meta-variables remain unchanged.

First, we begin by defining `=>` as an infix operator to represent a sequent. Its left argument is a list of assertions of the form `assertion(`*head*`,`*body*`,`*vars*`)`, where *vars* are the meta-variables. Its right argument is the query.

```
:- op(1150,xfx,'=>').
```

Next, we must define a few auxiliary predicates. **member** succeeds if its 1st argument can be unified with some element of the list which is its second argument. **member_exact** is similar except that it checks for exact identity and does not attempt unification; it is particularly useful with lists of variables. copy copies its first argument to its second argument while renaming all variables.

```
member(X,[Y|L]) :- X = Y ; member(X,L).
member_exact(X,[Y|L]) :- X == Y ; member_exact(X,L).
copy(X,Y) :- assert('copy marker'(X)),retract('copy marker'(Y)).
```

To prove a query is to prove the sequent, with no assumptions, where this query appears as the conclusion. While a query is being *executed*, new skolem terms may have to be created. To guarantee uniqueness, I use a monotonically increasing global counter to generate new skolem ids: I could have implemented it as a distinguished assertion in the global database, but instead I chose to pass its current value as an additional argument to the meta-interpreter. Therefore, prove/3 has three arguments: the first one is the sequent to be proven, the second one is the input counter value, and the third one, the output counter value.

```
prove(Query) :- prove(([]=>Query),0,_).
```

Now let's take a look at the meta-interpreter itself. **true** always succeeds.

```
prove((A=>true),C,C).
```

For any other goal, first try to find a clause in the sequent's assumption set such that its head unifies with the goal. Each element of the assumption set is of the form assertion(*head*,*body*,*vars*). A copy operation is necessary to create new copies of object-logic free variables: H is copied to P′ which is unified with P, and similarly T is copied to T′ which is unified with Q. However meta-level variables must remain unchanged, which is why, after V is copied to V′, we unify V′ with V thus identifying the new copies with the corresponding originals.

```
prove((A=>P),Ci,Co) :-
  member(assertion(H,T,L),A),
  copy(assertion(H,T,V),assertion(P,Q,V)),
  prove((A=>Q),Ci,Co).
```

Otherwise, we should try clauses in the global database.

```
prove((A=>P),Ci,Co) :- clause(P,Q),prove((A=>Q),Ci,Co).
```

A conjunction is handled by proving each conjunct successively.

```
prove((A=>P,Q),Ci,Co) :- prove((A=>P),Ci,Cm),prove((A=>Q),Cm,Co).
```

Finally, an implication is handled as follows: variables shared by both the antecedent (body) and the consequent (head) are considered meta-variables. All other free variables in the antecedent must be captured by quantification. Suppose the goal is of the form $A \Rightarrow p(X) :- q(X,Y)$, then the system will attempt to solve $[\text{assertion}(q(X,Z), \text{true}, [X])|A] \Rightarrow p(X)$, where X is considered a meta-variable.[3] However, in order to preserve soundness when the assumption is discharged, Y must be instantiated with a skolem term that depends on X. In essence, the system turns the goal into $\exists X \centerdot A \Rightarrow p(X) :- \forall Y\, q(X,Y)$ and the skolem term is introduced by the skolemizing axiom $q(X, \text{sk}(n,[X])) \equiv \forall Y\, q(X,Y)$ which lets us reinsert the universal quantifier.

We determine the free variables L1 of Q, and L2 of P, then we split L1 into L3 = L1∩L2 and L4 = L1 − L2. As announced earlier, to preserve soundness, we must instantiate every variable in L4 with new skolem terms depending on the variables in L3; however, we also want all the variables of L1 to remain free in the assumption: for this reason we make a copy R of Q, and we shall use R as the assumption instead; but, since the meta-variables L3 must remain unchanged, L3 appears on both sides of the copy operation.

[3] LOGICALC's DEDUCTION-THEOREM plan generator would have to instantiate them with arbitrary constants.

```
prove((A=>(P:-Q)),Ci,Co) :-
  vars_of(Q,L1), vars_of(P,L2), split_exact(L1,L2,L3,L4),
  copy((Q,L3),(R,L3)), skolify(L4,L3,Ci,Cm),
  assertify(R,L3,A,B), prove((B=>P),Cm,Co).
```

Collecting the free variables occurring in a term is done by recursive case analysis on the input term. `member_exact` serves to weed out duplicates.

```
vars_of(X,L) :- vars_of(X,[],L).
vars_of(X,Li,Lo) :-
  atomic(X) -> Li=Lo;
  var(X)     -> (member_exact(X,Li) -> Li=Lo; Lo=[X|Li]);
  X=[H|T]    -> vars_of(H,Li,Lm), vars_of(T,Lm,Lo);
  X =.. L, vars_of(L,Li,Lo).
```

In `split_exact`, the 1st and 2nd arguments are input lists of unbound variables, the 3rd argument is their intersection, and the 4th argument is the list of all these variables of the 1st list which are not in the 2nd list (i.e. their difference).

```
split_exact([],_,[],[]).
split_exact([X|L1],L2,L3,L4) :-
  (member_exact(X,L2) ->
    L3=[X|LL3],L4=LL4 ;  L3=LL3,L4=[X|LL4])
  split_exact(L1,L2,LL3,LL4).
```

`skolify` is given a list of unbound variables and unifies each one with a new skolem term. The 2nd argument is the list of free variables that this skolem term must depend on (typically: the meta-variables).

```
skolify([],V,Ci,Ci).
skolify([sk(Ci,V)|L],V,Ci,Co) :- Cm is 1+Ci, skolify(L,V,Cm,Co).
```

`assertify` takes 3 inputs and returns 1 output. Inputs are: 1) the body of a clause, 2) the list of meta-variables occurring within, 3) the assumption set to add the new

assertions to. Output is: the augmented assumption set. Typically the body of a clause consists in a sequence of conjuncts separated by commas: each conjunct will result in a separate clause.

```
assertify(X,_,_,_) :- var(X), !, fail.
assertify((P,Q),L,Ai,Ao) :- !,assertify(P,L,Ai,Am),assertify(Q,L,Am,Ao).
assertify((P:-Q),L,A,[assertion(P,Q,L)|A]) :- !.
assertify(P,L,A,[assertion(P,true,L)|A]).
```

The technique for augmenting the above meta-interpreter to support EBG proper is documented in [KCM87] and was briefly illustrated earlier. Also, as argued in [vHB88], the training example is a convenience, not an absolute requirement:

> *EBG is usually described as needing a training example for the reformulation of the goal concept, but this training concept is not really necessary for the execution of EBG. The only purpose it serves is to restrict the number of possible reformulations of the goal concept by guiding the search through all possible rules from the domain theory.*

In the following, I will describe LOGICALC's generalization procedure which is designed very much along the lines of the meta-interpreter above.

8.5 LOGICALC's Generalization Procedure

In this section, I describe the representation of the reified system, then I present the regression algorithm proper, and finally I discuss various problems.

8.5.1 Reified Representation

There are three levels of reification: (1) object-logic free variables in assumptions must be distinguishable from meta-level variables, (2) assumption sets can no longer

be represented by datapools, but must instead become terms, (3) sequents must also become terms.

Assumption Sets

In the course of the derivation of a proof, an assumption is typically made when the user invokes the DEDUCTION-THEOREM plan generator [Section 6.6.2, p217]. Let me briefly recapitulate what happens in the general case. The current goal is of the form $A \Rightarrow p(x,y) \supset q(x)$. We cannot simply move $p(x,y)$ to the assumption set for two reasons:

Firstly, the very process of asserting it in the datapool severs the connection between the occurrence of x in the antecedent and its occurrence in the consequent. In order to maintain this connection (1) either we would have to move the implicit existential quantifier outside the sequent, yielding $\exists x_\bullet A \Rightarrow p(x,y) \supset q(x)$, which can't be done in the original logic, (2) or we must instantiate x with a ground term α (thus artificially removing the problem) yielding $A \Rightarrow p(\alpha,y) \supset q(\alpha)$

Secondly, for reasons of soundness, if we want to interpret y as a free variable in the assumption, then we must explicitly capture it with a universal quantifier around the antecedent prior to moving this antecedent to the assumption set. There is a skolemizing axiom [Section 3.5, p98] $p(\alpha,\beta) \equiv \forall z\, p(\alpha,z)$ which we can use to generate the subgoal $A \Rightarrow \forall z\, p(\alpha,z) \supset q(\alpha)$ and the substitution $\{y \leftarrow \beta\}$.

Finally, we may move the antecedent to the assumption set and, by skolemization, remove the newly introduced quantifier: $A, p(\alpha,z) \Rightarrow q(\alpha)$.

Note that making up new skolem constants to instantiate shared variables (e.g. x in the above example) is equivalent to solving the stronger goal where the existential quantifier outside the sequent (see above) is changed to a universal quantifier, then skolemized away.

In the reified system, the problem is turned around: we no longer have to instantiate shared variables with constants because the connection between variables occurring on the left hand side of the sequent and those on the right hand side is maintained (they are part of the same meta-term). In fact, it is now the effect of *copy on fetch* which we no longer get for free [see below].

For reified assumption sets, we do not need such mechanisms as indexing, which are provided by datapools, because we know exactly what assumptions we need and where: such information is recorded in the proof tree. This makes reification of assumption sets very simple: whereas the ASSERTIONS slot of a normal assumption set holds a list of names denoting assertions in the assumption set's datapool, the same slot of the reified assumption set directly holds the corresponding list of formulae.

Just as was the case with assumption sets, reified assumption sets are *uniquified.* In other words, the operation which maps an assumption set to its reification always returns the same value given the same input. An associative cache is maintained for the duration of a complete regression.

Free Variables

When an assumption is made during the course of a proof, all free variables are captured either by arbitrary instantiation or by explicit universal quantification, i.e. the local assumption has the form $\forall z\, p(\alpha, z)$ [see above]. This local assumption is then processed to yield a more useful set of assertions: for instance, if it has the form of a conjunction, then each conjunct is asserted separately; also, unless `lc-skolemize*` is false, skolemization will be applied automatically.

During generalization, if the assumption contains free variables, these must have been introduced by the generalization procedure; they do not have to be captured; instead,

they should be considered meta-variables. In this implementation, I represent them by ordinary free variables and I simply leave them alone.

But what of these free variables which are introduced by the process of skolemizing an assumption? Clearly these are object-logic free variables and should exhibit the *copy on fetch* effect. In order to distinguish them from meta-variables (which should not be copied), I represent them differently: whereas ordinary variables are represented by terms of the form `(\? ⟨name⟩)`, they are represented by `(*xmvar ⟨name⟩)`. Note that this distinction is only necessary on the left hand side, i.e. in reified assumption sets; on the right hand side, meta-variables and object-logic variables are both represented by ordinary variables.

Whenever a (reified) assertion is fetched from a reified assumption set, the *copy on fetch* effect must be explicitly simulated: all ***xmvar**-type variables are replaced (in the copy) by newly created ordinary variables; meta-variables are untouched.

Sequents

Originally, I thought that a reified sequent should carry along the corresponding reified assumption set. However, it turns out that doing so is completely unnecessary since the latter can be recovered from the proof [training example] that is being passed along (i.e. by mapping the proof's assumption set to its reification). Therefore reified sequents are pairs of the form $\langle \text{CONC}, \theta \rangle$, where CONC is the conclusion and θ is the corresponding substitution. For practical reasons, CONC must be kept as instantiated as possible—i.e. $\theta\text{CONC} = \text{CONC}$. However, this does not mean that θ is superfluous. Indeed it is needed to instantiate the reified tree of assumption sets which, as was just mentioned, is not carried along.

8.6 The Regression Algorithm

8.6.1 Regressing A Proof

The procedure **regress-proof** is given a proof PROOF and a substitution θ specifying assignments to meta-variables in the reified tree of assumption set corresponding to the PROOF's assumption set. **regress-proof** returns a list of reified sequents, i.e. of elements of the form $\langle \text{CONC}, \theta' \rangle$ where CONC unifies with and subsumes the proof's conclusion and $\theta \subseteq \theta'$.

PROOF is an axiom

Let CONC be the axiom's formula in which all free variables have been renamed to new variables (cf. copy on fetch). Return a list consisting of the single element $\langle \text{CONC}, \theta \rangle$.

PROOF is an assumption

Let ASET be PROOF's assumption set. PROOF is the nth element in ASET's ASSERTIONS slot. Let ASET$'$ be the reification of ASET, and FMLA be the nth formula in ASET$'$'s ASSERTIONS slot. Now, let FMLA$'$ be the result of explicitly simulating *copy on fetch* by replacing all ***xmvar**-type variables with new ordinary variables. Return a list consisting of the single element $\langle \text{FMLA}', \theta \rangle$.

PROOF is a tautology

Invoke the special tautology regressor [Section 8.6.3, p290].

PROOF is the result of DUCK backward-chaining rules

As explained elsewhere, it is possible to build-in theories by encoding decision procedures as DUCK backward-chaining rules. At this time, LOGICALC does not preserve a full trace of the computation, which makes it impossible to regress

through such a derivation (since we don't know what it is). See later for a discussion of this problem. The regression procedure *punts*. The precise meaning of punting will be provided shortly.

PROOF is an inference

First, we must regress the steps. We do not regress the steps independently because, supposing there are n PROOF's steps, we would be left with n lists $\cup_{k=1}^{n_k} \langle p_{i_k}, \theta_{i_k} \rangle$ for $i = 1$ to n. Then we would have to somehow *merge* these back together, e.g. by picking one element in each list and determining whether the substitutions were compatible and how they should be collapsed. Instead, we proceed incrementally:

- If there is only 1 step, we regress it normally, thus collecting a list of elements of the form $\langle p, \theta' \rangle$. Then we transform each element into $\langle [p], \theta' \rangle$, i.e. we replace the conclusion with a list whose sole element is the conclusion.
- Otherwise, the list of steps is PROOF_k for $k = j$ to n. We recursively regress $\text{PROOF}_{j+1} \dots \text{PROOF}_n$ in the manner which is presently being described and obtain a list of elements of the form $\langle [p^i_{j+1} \cdots p^i_n], \theta^i \rangle$ for $i = 1$ to m. Then, for each i we regress PROOF_j using θ^i as the substitution and collect a list of reified sequents $\langle p_j^{\ell_i}, \theta^{\ell_i} \rangle$ for $\ell = 1$ to ν^i_j. As the result we return the list $\langle [p_j^{\ell_i}, p^i_{j+1} \cdots p^i_n], \theta^{\ell_i} \rangle$ for $\ell = 1$ to ν^i_j and $i = 1$ to m.

In other words, the algorithm above proceeds right to left and returns lists of pseudo-reified sequents of the form $\langle [p_1 \dots p_n], \theta' \rangle$. I say "pseudo" because instead of a single conclusion, it has a list of the regressed conclusions of the steps it started with.

Then, the rule's *regressor* is invoked with 3 arguments: (1) the proof, (2) the list of regressed steps $[p_1 \dots p_n]$, (3) the substitution θ'. The regressor is essentially identical to the procedure implementing the inference rule, but operates on the

reified representation and returns a list of reified sequents. It would be nice if the exact same code could serve both purposes; this is discussed later.

The lists obtained by successively invoking the regressor for each one of the pseudo-reified sequents computed above are concatenated and returned as the result.

Otherwise

If the proof does not have a regressor, or, for some reason, this regressor fails to return anything, then punt [see below].

8.6.2 Punting

When all else fails, this is the default regression strategy to fall back on. The idea is simply to use the proof's conclusion as its own generalization, i.e. to do nothing. Unfortunately, the substitution is now out of synch because it does not reflect all the assignments which would be necessary to actually make this very instantiated conclusion on the basis of the reified representation. Soundness goes out the window because it is possible to augment the substitution with assignments that would not be possible (i.e. in contradiction) with the true substitution.

Consider the example where $p(f(x)) \supset q(f(x))$ is an axiom, $p(f(a))$ is an assumption, and $q(f(a))$ is derived from them by MODUS-PONENS. Suppose the reified representation has $p(u)$ as the assumption (with u a meta-variable) and $q(v)$ as the conclusion to be regressed, and the substitution θ does not so far assign u or v. If I decide to punt naïvely, and merely unify $q(v)$ with $q(f(a))$, I will correctly derive the additional assignment $v \leftarrow f(a)$ but u will remain unassigned (which in itself is unsound), or even could later on be assigned a term which does not unify with $f(a)$.

What this analysis suggests is that all the meta-variables participating in the derivation of the proof on which we punt should be fully instantiated. What does this

mean? It means that we should recursively inspect the proof on which we punt and for each assumption which occurs somewhere in its proof tree we should unify the assumption with its reified version in order to instantiate the meta-variables. Performing this rather drastic operation is a sufficient condition to guarantee soundness. Why? Because the reified proof tree (which is not explicitly represented) has in some sense been unified with the actual proof tree: all the generalization involved in the reified derivation has been removed, the leaves of the reified proof tree are identical (modulo variable renamings) to the leaves of the actual proof tree, therefore the same conclusion falls out and we have determined the substitution that guarantees it. It might be possible to do better than this, but it is precisely because the more clever algorithm gave up or failed that we had to resort to punting in the first place.

The actual punting procedure works as described in the preceding paragraph.

8.6.3 Regressing A Tautology

In the case of a tautology, we are given a proof PROOF and a substitution θ representing the assignments to meta-variables. The object is to produce a variabilized skeleton of PROOF's conclusion instantiated only as much as required to make it a tautology. Since PROOF's conclusion is a tautology, this problem is guaranted to have at least one solution.

Let FMLA be the variabilized skeleton of PROOF's conclusion, i.e. it has the same propositional shape, but all terms have been replaced with variables. If FMLA is not a tautology, then there exists a (consistent) assignment of truth values to its constituent literals that falsifies it.

Since PROOF's conclusion is a tautology, the corresponding assignment would be illegal for it; that is, it would assign opposite truth values to different occurrences of the same literal. The reason the assignment was legal for FMLA is that the corresponding

two occurrences of these literals looked different as a result of variabilization. In order to rule out this falsifying assignment, we must unify the two variabilized literals to constrain them to be identical.

Consider all pairs of variabilized literals which have been assigned opposite truth values in the assignment falsifying FMLA. For each pair, if they unify resulting in the substitution θ' (where $\theta \subset \theta'$), then we can attempt to falsify θ'FMLA again and be certain that this particular assignment will be ruled out.

By repeatedly proceeding thus, we can determine all the ways to minimally instantiate FMLA so that it becomes a tautology.

Finally, we filter out irrelevant generalization, i.e. those which do not subsume PROOF's conclusion. For example, if PROOF is $A \Rightarrow p(a) \vee \neg p(a) \vee \neg p(b)$, the regression procedure will start out with the skeleton $p(u) \vee \neg p(v) \vee \neg p(w)$ and will return the following two answers:

$$p(x) \vee \neg p(x) \vee \neg p(y)$$
$$p(x) \vee \neg p(z) \vee \neg p(x)$$

Only the first one subsumes the proof's conclusion; the second one is discarded as being irrelevant.

8.6.4 Where Automatic Generalization Fits

Automatic Generalization was a rather late addition to LOGICALC, and its integration with the rest of the system is not seamless. In the current scheme, the regression mechanism is invoked by the constructor for proof data-structures: (1) the proof structure is consed up and initialized, (2) then it is handed over to the regression procedure which returns a list of formulae generalizing the proof's conclusion, and

(3) the system picks the first generalization and destructively replaces the proof's conclusion with it.

It might be a better idea to have Generalization mascarade as some sort of inference rule: given a proof, it would return a list of proofs which can be derived by *generalization.* Note however that Generalization is not an inference rule (at the object-logic level) because it takes a proof *tree* as input rather than premises.

The advantage of the current scheme is its simplicity: you cons up a new proof and as a side-effect it is generalized. In the proposed improvement, consing up a proof would return a list of generalizations and much of the code would have to be rewritten to accommodate this extension.

8.7 Regressors

Each inference rule has a *regressor* which is just like the procedure implementing the inference rule, but operates on the reified representation instead. A regressor takes 3 arguments: the proof to be regressed, the list of formulae representing the regressed steps of the proof, and a substitution carrying along the assignments to meta-variables; it returns a list of reified sequents (of type `xsequent_t`).

I am going to illustrate the implementation with 3 examples. In each case, I shall provide both the regressor and the inference rule for comparison. Here is the code for AND-INTRO:

```
(DATAFUN REGRESS AND-INTRO                                          and-intro
  (DEFFUNC - (LST xsequent_t)
      (PROOF - proof_t STEPS - (LST sexp) ALIST - (LST bdg))
    (LIST (MAKE xsequent_t '(AND ,@(VARSUBST STEPS ALIST)) ALIST))))

(DEFFUNC AND-INTRO-RULE - (LST sequent_t)
    (PREMS - (LST sequent_t) PARMS - (LST sexp))
```

```
  (LIST (MAKE sequent_t
          (WEAKEST-ASET-OF-SEQUENTS PREMS)
          '(AND ,@(<# !_CONCLUSION PREMS)))))
```

The similarity is most obvious in this example because AND-INTRO is one of the more trivial inference rules. The regressor is slightly simpler because it doesn't have to deal explicitly with the assumption set.

Let us now take a look at MODUS-PONENS. Here too the same remarks apply:

modus-ponens

```
(DATAFUN REGRESS MODUS-PONENS
  (DEFFUNC - (LST xsequent_t)
      (PROOF - proof_t STEPS - (LST sexp) ALIST - (LST bdg))
    (MATCH-COND STEPS
      (\? ((IF ?P ?Q) ?R)
          (FOR (E IN (SAME-UNIFY P R ALIST))
               (SAVE (MAKE xsequent_t
                       (VARSUBST Q (!_LALIST E))
                       (!_LALIST E)))))
      (T NIL))))

(DEFFUNC MODUS-PONENS-RULE - (LST sequent_t)
    (PREMS - (LST sequent_t) PARMS - (LST sexp))
  (MATCH-COND !>(CAR PREMS).CONCLUSION
    (\? (IF ?P ?Q)
        (LET ((ENVS (UNIFY !>(CADR PREMS).CONCLUSION P NIL)) - (LST env))
          (AND ENVS (LIST (MAKE sequent_t
                            (WEAKEST-ASET-OF-SEQUENTS PREMS)
                            (ENVSUBST Q (CAR ENVS) 'LEFT))))))
    (T NIL)))
```

Finally, here is the regressor for EQUALITY. For reference, the code implementing the inference rule is also included. Comparison of the two will again reveal the tight correspondence. The interesting feature of this example is that the regressor looks up the parameters in the proof's validation. Here it is necessary to find out the paths to the subterms to be replaced.

equality

```
(DATAFUN REGRESS EQUALITY
  (DEFFUNC - (LST xsequent_t)
```

```
      (PROOF - proof_t STEPS - (LST sexp) ALIST - (LST bdg))
    (MATCH-COND STEPS
      (\? ((= ?NEW ?OLD) ?PAT)
          (LET (TARGET - sexp PATHS - (LST (LST fixnum))
                (PARMS (!_PARAMETERS (!_VALIDATION PROOF))) - (LST sexp)
                (VARS NIL) - (LST symboid))
            (OR (MATCHQ (?((BE * TARGET)) - . ?((BE * PATHS))) PARMS)
                (PROG1 '#T
                  (!= PATHS (BE * PARMS))
                  (!= TARGET (EXTRACT-VAGUE-PATTERN
                              (!_CONCLUSION (CADR (!_STEPS PROOF)))
                              PATHS))
                  (!= VARS (VARS-OF TARGET))))
            (LOOP FOR ((PATH IN PATHS) - (LST fixnum)
                       (XSEQUENTS (FOR (E IN (UNIFY PAT TARGET ALIST))
                                    (SAVE (LET ((ALIST (ENV-ALIST E 'LEFT)))
                                            (MAKE xsequent_t
                                              (VARSUBST PAT ALIST)
                                              ALIST)))))
                       - (LST xsequent_t))
              WHILE XSEQUENTS
              RESULT (IF VARS
                         (LOOP FOR ((XS IN XSEQUENTS))
                           (!= (!_ALIST XS) (X-CLEAN-ALIST *-* VARS))
                           RESULT XSEQUENTS)
                         XSEQUENTS)
              (!= XSEQUENTS (<! (\\ (XS) (XPATH-DEMODULATE2 XS OLD NEW PATH)) *-*)))))
     (T NIL))))

(DEFFUNC EQUALITY-RULE - (LST sequent_t)
    (PREMS - (LST sequent_t) PARMS - (LST sexp))
  (LET* ((ONE (CAR  PREMS)) - sequent_t
         (TWO (CADR PREMS)) - sequent_t
         (PATTERN !>TWO.CONCLUSION) - sexp
         (ASET (WEAKEST-ASET-OF-SEQUENTS PREMS)) - aset_t)
    (MATCH-COND !>ONE.CONCLUSION
      (\? (= ?NEW ?OLD)
         (MATCH-COND PARMS
           (\? (?TARGET - . ?REST)
               (LOOP FOR ((PATH IN (BE * REST)) - (LST fixnum)
                          (PATTERNS (<# (\\ (E - env)
                                             (VARSUBST PATTERN (ENV-ALIST E 'Left)))
                                         (UNIFY PATTERN TARGET NIL))) - (LST sexp))
                 WHILE PATTERNS
                 RESULT (<# (\\ (PAT) (MAKE sequent_t ASET PAT)) PATTERNS)
                 (!= PATTERNS (<! (\\ (PAT)
                                     (PATH-DEMODULATE2 PAT OLD NEW PATH))
                                PATTERNS))))
```

```
        (T (LOOP FOR ((PATH IN (BE * PARMS)) - (LST fixnum)
                      (PATTERNS (LIST PATTERN)) - (LST sexp))
              WHILE PATTERNS
              RESULT (<# (\\ (PAT) (MAKE sequent_t ASET PAT)) PATTERNS)
              (!= PATTERNS (<! (\\ (PAT)
                                    (PATH-DEMODULATE2 PAT OLD NEW PATH))
                                PATTERNS))))))
    (T NIL))))
```

Regressors are usually simpler than the corresponding inference rules not only because they don't have to explicitly deal with assumption sets, but also because they don't have to be as paranoid; they can trust in the proof tree.

8.8 Discussion Of Problems

In this section I discuss some problems with the Generalization mechanism. Mostly, these are due to the fact that generalization was a late addition and is not seamlessly integrated with the rest of the system.

8.8.1 Disappearing Terms

The very purpose of generalization is to replace certain terms of a newly derived conclusion with free variables. However, if an existing validation implicitly assumed that it would be able to find and manipulate these terms, when said validation is executed, it fails because the terms whose presence it expected have been variabilized away. Consider the following successive refinements:

1.	$\Rightarrow a = b \supset . p(f(a)) \supset p(f(b))$	
2.	$a = b \Rightarrow p(f(a)) \supset p(f(b))$	`(if-intro (0 1) ())`
3.	$a = b \Rightarrow p(f(a)) \supset p(f(a))$	`(equality (0 1) ((2 1 1)))`

The right-most column indicates the validation to infer the preceding (parent) goal. The goal numbered **3** will be recognized as a tautology: a proof is constructed and handed over to the generalization procedure which yields:

$$a = b \Rightarrow p(u) \supset p(u)$$

Now, the validation `(equality (0 1) ((2 1 1)))` attempts to produce an inference from $a = b$ and $p(u) \supset p(u)$, but path `(2 1 1)`, which used to denote the location for b, is now bogus because `(2 1)` is u and you you can't go any deeper than this. The term denoted by `(2 1 1)` has been generalized away and the validation fails.

When I added automatic generalization to LOGICALC, I had not anticipated the above problem, and it caused some rather puzzling bugs until I finally realized what was going on. The fix I came up with was to slightly complicate the parameters of these inference rules which look for terms in their premises: the idea is to give them sufficient information to restore the syntactic structure which they are counting on. Typically, a skeleton pattern is added to the parameters, and the rule is supposed to first unify this skeleton with the corresponding premise to derive a substitution that sufficiently instantiates the premise and thereby permits the desired inference step to proceed successfully.

Thus, the EQUALITY validation of the above example is really of the form:

(**equality** (0 1) ($x \supset p(f(y))$ - (2 1 1)))

Note how the skeleton has only just enough structure to restore term `(2 1 1)`. When this validation is executed, the EQUALITY rule, first unifies the conclusion $p(u) \supset p(u)$ with the skeleton $x \supset p(f(y))$ and constructs the resulting instance $p(f(v)) \supset p(f(v))$.[4] Now, the equality substitution of term `(2 1 1)`, i.e. the 2nd occurrence of v, can proceed successfully assigning a to v and concluding $p(f(a)) \supset p(f(b))$.

[4]The skeleton's variables are renamed to avoid possible confusion with the conclusion's variables: this is the reason why I wrote v rather than y in the resulting instance. Actually, the real trick is to use 2-sided pseudo-substitutions [page 111].

I am rather dissatisfied with this fix because it complicates the inference rules. In retrospect, it might have been better to extend the validation language and interpreter to allow expressions of the form (@ ⟨*arg*⟩ *skeleton*⟩) to appear as arguments of validations. The validation interpreter would determine the proof corresponding to ⟨*arg*⟩ and when it packages it as a sequent (type `sequent_t`) for the corresponding rule, it would unify the conclusion with ⟨*skeleton*⟩ in order to instantiate it further to meet the implicit expectations built into the desired inference step:

(equality (0 (@ 1 $x \supset p(f(y))$)) ((2 1 1)))

8.8.2 Built-In Theories

Conclusions derived by the office of built-in theories are stumbling blocks for the regression procedure. Built-in theories are coded in the form of DUCK back-chaining rules, and are designed to relieve the user from the tedium of having to explicitly and painstakingly derive proofs for the more obvious subgoals which pop up all the time when one works within a particular theory.

For example, consider the task of proving that a term is an element of a set specified by enumeration, e.g. that `c` is an element of `{a b c}` (whose internal representation is `(lset [a b c])`). Clearly, this task is both trivial and uninteresting, but the actual derivation of the proof is a small nightmare because it requires 3 refinements through MODUS-PONENS and 3 refinements to drop a disjunct; that's a minimum of 6 operations. If the set had n elements instead of 3, a minimum of $2n$ refinements would be needed (if the element to be checked for membership occurs at the end).

In an interactive system, the more obvious subgoals should be taken care of automatically, so that the user may concentrate on the more interesting/difficult parts of the proof. For example, LOGICALC defines the rule:

```
(AXIOM LSET-SIMPLE-DEF
  (<- (ELT ?X (LSET ?L)) (MEMBER ?X ?L)))
```

which automatically takes care of the above example.

On the face of it, it would seem that regressing through a proof obtained by back-chaining should be simpler than with proofs derived through bona-fide inference rules. Unfortunately, there are two reasons why this is not so.

Firstly, LOGICALC does not currently save a trace of the back-chaining computation. Consequently there is nothing to regress through. This shortcoming could be fixed, but not without considerable hassle. Much work would have to be invested in modifying, not only LOGICALC, but also the underlying DUCK system.

Secondly, DUCK rules may invoke arbitrary *extra-logical* operations. The `lisp` and `call` "predicates" make it possible to invoke a LISP procedure to perform part or all of the work. For instance, LOGICALC define equality of tuples in two ways: axiom `tuple-equality` provides the logical axiomatization, while `tuple-simple-equality` is the back-chaining rule that takes care of the simpler cases:

tuple equality

```
(AXIOM TUPLE-EQUALITY
  (IFF (= [?X1 !?L1] [?X2 !?L2])
       (AND (= ?X1 ?X2) (= ?L1 ?L2))))

(AXIOM TUPLE-SIMPLE-EQUALITY
  (<- (= ?L1 ?L2)
      (AND (IS-LIST ?L1)
           (IS-LIST ?L2)
           (SAME-LENGTH ?L1 ?L2)
           (FORALLIN/2 [?X1 ?X2] ?L1 ?L2 (= ?X1 ?X2)))))

(DEFPRED (IS-LIST ?X - obj)
  (CALL (IS-LIST ?X)))

(DEFFUNC IS-LIST - void (X - sexp) ...)
```

The problem, of course, is that there is just no general way to regress through an arbitrary computation, and, at present, regression *punts* when it encounters a proof obtained through back-chaining.

8.8.3 Regressors vs. Inference Rules

Regressors and inference rules perform identical operations on different representations. Inference rules operate on sequents, while regressors do so on their reification. It would be desirable to eliminate such duplication of code by folding the latter into the former.

An ordinary sequent is represented by a structure $\langle \text{ASET}, \text{CONC} \rangle$ of type `sequent_t`. A reified sequent is represented by a structure $\langle \text{CONC}, \theta \rangle$ of type `xsequent_t`; the reified assumption set is not included because it can be recovered from the proof which is being passed along, but it could be. These two representations can easily be merged together and suggest a data-structure of the form $\langle \text{ASET}, \text{CONC}, \theta \rangle$, where ASET may be either an ordinary assumption or its reification depending on whether this is an ordinary sequent or a reified sequent.

With a suitable and unifying interface to both ordinary assumption sets and reified assumption sets, I believe it would be possible to use the very same code both for regular inferences and for regression.

8.8.4 Regression vs. Inferencing

Despite my arguments to the effect that regression and inferencing can be made to share the same mechanisms, regression should not be mistaken for an inference rule in disguise. Inference rules map premises to conclusions. Regression maps proof trees to conclusions; in a sense, it acts as a meta-inference rule.

When regression operates, it must inspect the whole proof tree and recapitulate the inferences therein. Obviously, this entails a lot more work than performing a single inference. A legitimate question is: is generalization always worth it, and how can we keep the cost down?

When regression runs, it must instantiate the reified proof all the way down to axioms and non-discharged assumptions. Consequently, the real payoff happens when an assumption is being discharged by IF-INTRO. This remark suggests a course of action: we might want to invoke generalization only when handed a proof whose top inference is validated by IF-INTRO (actually, we also want to regress tautologies). No such policy is implemented at the moment, but the code is very flexible and will let the user experiment easily with various ideas. In particular, the procedure mapping a proof tree to generalized formulae is held in variable **proof-generalize***, whose default value is **#'lc-regress**. The user is free to specify any kind of filtering scheme instead. Also, variable **ebg-regress-fail-test*** may be set to a predicate which decides whether to punt or not for any particular inference node in the proof. At present, no such predicate is in effect and the value of the variable is NIL.

In any case, when generalization is requested, it must regress throughout the whole tree. The fact that the steps of the top inference node have already been generalized does not help for two reasons: firstly, if the top inference happens to discharge an assumption, then more generalizations may be able to take place. Secondly, we still have to go all the way down to occurrences of assumptions in order to get the bindings right [see punting earlier].

Even though my implementation of generalization is extremely sub-optimal, and conses like there is no tomorrow, I have found that in practice it has rather good performance. This is encouraging and validates the idea.

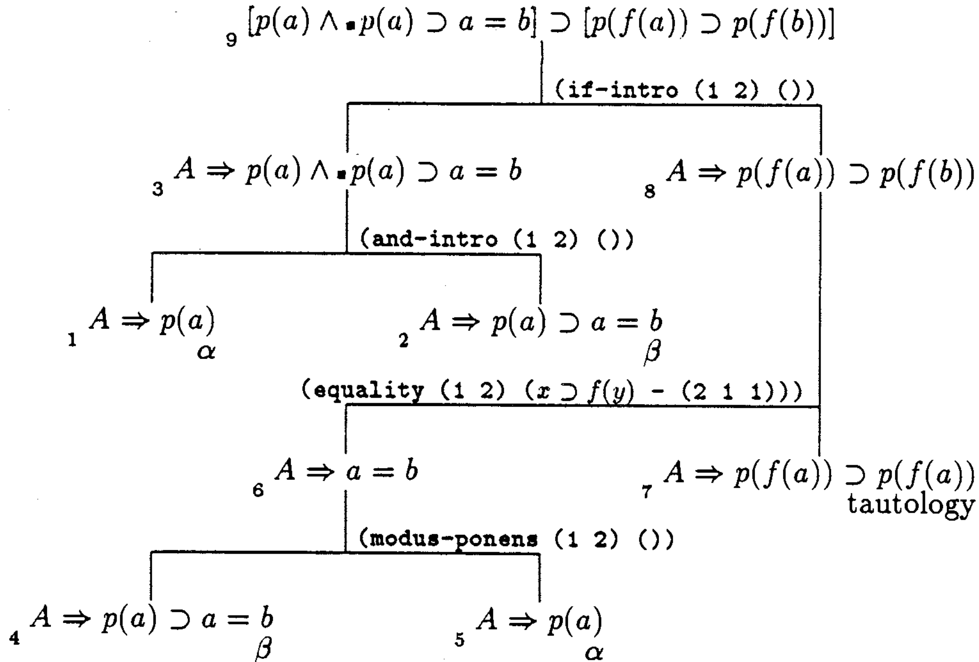

Figure 8.1: Generalization Example

8.9 Generalization In Action

In this section, I will illustrate the functioning of the generalization procedure by going step by step through a complete example. We are given a proof of $[p(a) \wedge \blacksquare p(a) \supset a = b] \supset [p(f(a)) \supset p(f(b))]$ whose proof tree is displayed in figure 8.1. To avoid cluttering the pictorial representation, A stands for the assumption set in which $p(a) \wedge \blacksquare p(a) \supset a = b$ is the local hypothesis and the corresponding local assertions are α: $p(a)$ and β: $p(a) \supset a = b$.

The leaf-nodes of the tree have been labeled with their justification: α or β if they are assumptions, or 'tautology' if they were proven to be tautologous.

Each proof node in figure 8.1 is also labeled with a number. The numbers reflect the order in which these nodes are regressed through. To regress through an inference node, the procedure first regresses its steps (in left-to-right order).

The first node for which something happens is the one labeled 1. It is an assumption. In order to regress an assumption, the procedure must obtain the reified assumption set. The latter is constructed and the variabilized local assertions are $p(u_1)$ and $p(u_2) \supset u_3 = u_4$. Since node 1 is justified by local assertion α, the regression procedure returns $[\langle p(u_1), \{\}\rangle]$.

Node 2 is similarly justified by assertion β, and the regression returns $[\langle p(u_2) \supset u_3 = u_4, \{\}\rangle]$. As described earlier, the procedure regressing sibling steps here returns $[\langle [p(u_1), p(u_2) \supset u_3 = u_4], \{\}\rangle]$. The regressor for AND-INTRO returns $[\langle p(u_1) \wedge p(u_2) \supset u_3 = u_4, \{\}\rangle]$ as the result of regressing through node 3.

Now, we go down the other branch, all the way to node 4. Node 4 and 5 are respectively justified by β and α, and, proceeding as above, the regression of these steps produces $[\langle [p(u_2) \supset u_3 = u_4, p(u_1)], \{\}\rangle]$. Now the regressor for MODUS-PONENS unifies $p(u_1)$ with $p(u_2)$, effects the inference and yields $[\langle u_3 = u_4, \{u_2 \leftarrow u_1\}\rangle]$ for node 6.

Then, we must regress node 7 using $\{u_2 \leftarrow u_1\}$ as the current substitution. Since this proof is a tautology, we must invoke the tautology regressor described in section [ref]. The regressor is handed the variabilized conclusion $p(u_5) \supset p(u_6)$ and finds that it can be falsified with the interpretation $\{p(u_5) \leftarrow \text{TRUE}, p(u_6) \leftarrow \text{FALSE}\}$. The only way to rule this out is to unify $p(u_5)$ with $p(u_6)$. The regression returns $[\langle p(u_5) \supset p(u_5), \{u_2 \leftarrow u_1, u_6 \leftarrow u_5\}\rangle]$.

Regression of steps 6 and 7 returns $[\langle [u_3 = u_4, p(u_5) \supset p(u_5)], \{u_2 \leftarrow u_1, u_6 \leftarrow u_5\}\rangle]$, and the EQUALITY regressor is invoked. First, it renames[5] the variables in the skeleton pattern $x \supset p(f(y))$ thus obtaining $x' \supset p(f(y'))$, then performs unification with $p(u_5) \supset p(u_5)$ to restore the desired structure. The substitution is $\{u_2 \leftarrow u_1, u_6 \leftarrow u_5 \leftarrow f(y')\}$, and the second premise becomes $p(f(y')) \supset p(f(y'))$. Now u_3 (equality's left-hand side) is unified with the second occurrence of y' and said occurrence is

[5] Actually, see earlier footnote p8.8.1.

replaced with u_4 (equality's right-hand side). The regression yields $[\langle p(f(u_3)) \supset p(f(u_4)), \{u_2 \leftarrow u_1, u_6 \leftarrow u_5 \leftarrow f(y'), u_3 \leftarrow y'\}\rangle]$.

Regression of steps 3 and 8 returns $[\langle [p(u_1) \wedge \blacksquare p(u_1) \supset u_3 = u_4, p(f(u_3)) \supset p(f(u_4)), \{u_2 \leftarrow u_1, u_6 \leftarrow u_5 \leftarrow f(y'), u_3 \leftarrow y'\}\rangle]$—notice how the 1st premise, which was the result of regressing through node 3, has been further constrained by the substitution's assignment $u_2 \leftarrow u_1$—and the IF-INTRO regressor yields $[\langle [p(u_1) \wedge \blacksquare p(u_1) \supset u_3 = u_4] \supset [p(f(u_3)) \supset p(f(u_4))], \{u_2 \leftarrow u_1, u_6 \leftarrow u_5 \leftarrow f(y'), u_3 \leftarrow y'\}\rangle]$.

8.10 Comparison With ONTIC

ONTIC [cite McAllester87] includes *Automatic Universal Generalization* as one of its inference mechanisms. It applies when the following conditions are met:

- g is a generic individual of type τ.
- The system has labeled the node for a formula $\Phi(g)$ true.
- No assumptions have been made about the individual g other than the assumption that it is an instance of type τ.
- No free variable of $\Phi(g)$ has a type that depends on g.

In this case, the system infers the universal closure:

$$\forall x{:}\tau\ \Phi(x)$$

Unfortunately, universal generalization is limited in the following way:

> *This inference mechanism does not construct new formulas or add new graph structure. In order for this inference mechanism to be applied, all of the formulas involved must already be compiled into nodes in the graph structure.*

This is in contrast with LOGICALC, where generalizations are added to the database (and thus become available for subsequent refinements/inferences) and are also broadcast throughout the existing network of goal classes.

In addition, whereas ONTIC is limited to generalize only on generic individuals, LOGICALC's mechanism applies to all terms.

Chapter 9

Proof Editing And Formatting

In this Chapter I discuss the formatting of proofs, with particular attention given to the automatic generation of documents intended for publication. LOGICALC includes a *proof editor* to assist the user in this task. The default heuristics usually produce satisfactory results and no user involvement is required. Architecture, heuristics, and display policies are explained, and a few glimpses at the implementation are offered.

9.1 Introduction

Interactive proof development is typically an exploratory enterprise, and the facts that the graph encoding the on-going work often grows to large proportions, that both the user and the system contribute to it, and that the focus of attention moves rapidly from one location in the graph to another, make it difficult to keep track of the big picture.

Therefore, when a (sub)proof has been derived, it may be useful to pause and study the details of its derivation in order to better understand its true articulation and

its structure. Since proofs are represented by nodes in the graph, it is possible to manually navigate through them. However, this method of inspection provides limited context since only one inference node at a time can thus be observed; and the big picture remains elusive.

What is needed is a tool that provides a global view of a proof in a format which is easy to grasp. A possibility is to generate a graphical representation reflecting the proof structure, e.g. using the ISI grapher. LOGICALC does not at present offer such a facility. Also, although a graphical representation is often preferable, it may not always be practical because of the size of the proof tree.

Another alternative is to generate a linear representation inspired of the classical format of Natural-Deduction proofs which consists in a sequence of lines, each one representing a conclusion. Every line is numbered and is given a justification which either states that it is a hypothesis, or an instance of an axiom schema, or is derived from earlier lines by some inference rule; in the latter case, the inference rule and the premises' line numbers are listed. Furthermore each line lists (by line number) the hypotheses which it depends on. The general form of a line is:

$k.$	$[i_1 \dots i_n]$	$\vdash p$	RULE $j_1 \dots j_m$
line number	**hypotheses**	**conclusion**	**justification**

For example, here is a simple derivation expressed in this notation:

1.		$\vdash a = b$	hypothesis
2.		$\vdash f(a) = f(a)$	axiom
3.	[1]	$\vdash f(a) = f(b)$	equality 1 2
4.		$\vdash a = b \supset f(a) = f(b)$	if-intro 1 3

An advantage of such a representation is that it is familiar to everyone with an interest in logic. It is also simple and intuitive enough that a neophyte may quickly achieve a reasonable level of fluency reading material presented in this fashion. This property is particularly important in light of my next point.

Another reason for producing a proof summary is to communicate it to someone else. After having invested time and effort in the derivation of a proof, the user may want to prepare it for inclusion in a document for publication. Formatted proofs are also of interest in the classroom: for the student, to answer a homework, for the tutor, to produce solutions and slides (e.g. see the introductory example of Chapter 1). Also, the user may simply want such a summary for his own benefit, as the printed version of a proof is easier to study than the graph itself, and may serve as a reference.

Since a proof summary is specifically intended for human consumption, it should be digested and prepared to make it easier to read. In particular, people are interested in the highlights of the proof, i.e. those inferences which embody the essential articulation of the proof; trivial or incidental subparts should be omitted or otherwise condensed in non-obtrusive ways. Here are a few cases in point:

- *Uninteresting Conclusions.* In a given domain of interest, there will be predicates useful for the axiomatization, but whose rôle is not essential. In AI applications, type predication performed by the `IS` predicate is typically not of great relevance to what is truly going on. Lines concluding propositions of the form `(IS` ⟨*type*⟩ ⟨*object*⟩`)` as well as those participating in their derivation only clutter and obscure the overall proof summary without contributing anything to the user's understanding.

- *Uninteresting Inferences.* There are categories of inferences which are so obvious that there is no point in actually showing them. If we don't show them, the user will supply the actual inferences without even being aware of doing so and the clutter in the proof summary will be further reduced, thus enhancing its readability. For example, any formula concluded by IDENTITY, or AND-INTRO, or SYMMETRY.

- *Uninteresting Derivations.* Some derivations are so simple that it is possible to get away with not showing them without causing confusion in the reader.

MODUS-PONENS is not a trivial inference rule, and therefore is not covered by the previous item. On the other hand, when it is given trivial premises, it is often reasonable to *not* allocate a line for the conclusion, and, whenever it is mentioned, abbreviate it *in-line* by listing the premises instead. The reader easily supplies the missing inference from the context.

Built-in theories [page 60] are often designed to take care of the more obvious queries. They are meant to factor out various tedious aspects of a particular domain. Consequently, it is often inappropriate to go into the details of conclusions obtained through their mediation.

LOGICALC provides a tool whose heuristics take the above points into account and can turn a large proof into a reasonable summary. It is quite flexible, and different policies can be implemented by ajusting a multitude of parameters. This tool is shelled within a special purpose editor which allows the user to interact with the summary, edit and improve it, change or override defaults, insert comments, etc...

9.2 The Proof Editor

The proof editor can be entered from a proof node by issuing the `edit` command. At that point the entire proof tree rooted at this node is digested and one line is created for every subproof, assumption, and axiom. Thus the proof tree is effectively flattened and presented in a linear fashion. This linearized presentation is packaged as another type of node, known as a *proof view*, and the user interacts with it in the same manner as with other nodes, i.e. through the graph editing mechanism mediated by LOGICALC's shell. The introductory example of Chapter 1 showed an example of a proof view which I include again here:

```
[33] w> EDIT                                                transcript
```

Proof View

```
  12 (LEFT !.OTHER-HAND)                                                      [THUMBS 1 2 !:RIGHT331]
  13 (LEFT !.ONE-HAND)                                                        [11 12]
  14 (IF (AND (RIGHT !.ONE-HAND) (NOT (LEFT ?X))) (LEFT !.ONE-HAND))          [!:RIGHT331 !:NOT332 13]
  15 (LEFT !.ONE-HAND)                                                        [6 7 14]
  16 (RIGHT !.OTHER-HAND)                                                     [THUMBS 1 2 15]
  17 (RIGHT !.ONE-HAND)                                                       [11 16]
  18 (NOT (LEFT !.ONE-HAND))                                                  [LEFT-OR-RIGHT 17]
  19 (NOT (= !.ONE-THUMB !.OTHER-THUMB))                                      [15 18]
=>20 (IF (AND (IS CREATURE ?Y) (IS THUMB ?X) (PART ?X ?Y)) (AND (IS THUM...   [!:PART280 5 19]
```

A proof view is mainly a sequence of lines, after the manner of Natural Deduction proofs. Since there are typically many lines, only a small window is displayed by default; the size of this window is controlled by parameter `edit-context`. Within the sequence of lines, the proof editor maintains the notion of the current line—indicated by the arrow—and centers the context window around that line. In the above illustration, the focus is on the last line, which explains why only the preceding context is non-empty.

Each line is numbered; the corresponding conclusion is prettyprinted (by default, it is prettyprinted on a single line and only a limited prefix is displayed—see, e.g. line 20—but other policies can be chosen by setting various parameters); it is followed by a list of references to others lines which this one depends on (typically, these are the premises of the corresponding inference step).

A line is usually referenced by its line number. Axioms and assumptions, however, are referenced by name (by default). Line 18 concludes that `!.ONE-HAND` is not a left hand, and the list of dependencies shows that this conclusion uses axiom `left-or-right`, which states that a hand must be either a left hand or a right hand, but not both, and line 17 which asserts that `!.ONE-HAND` is a right hand.

A proof view also carries a number of editable properties which influence formatting heuristics and summary generation. The user may implement different policies, or change the visual appearance of the summary by altering their values with the `option` command.

9.2.1 The Notion of Users

Even though a complete proof is usually visualized as a tree, in practice, it is often a (acyclic) graph. Through the various mechanisms of answer sharing, answer broadcasting, proof uniquification, and proof insertion in the database, it is possible for the very same proof object to participate as a premise in multiple inferences, and therefore to occur more than once in the overall proof *tree*.

When the proof graph is converted to a proof view, for each line therein, we keep track of all the other lines for which this one is required as a premise: they constitute this line's users.

Why keep track of *users*? The answer is twofold. Firstly, the (human) user may take advantage of this information to better understand the dependencies, and also may naviguate along these links from one part of the proof view to another. The reverse links are of course the *steps* of a proof line, and the user may naviguate along those as well.

Secondly, as I have alluded to before, it is possible to omit parts of the proof from the summary. If all of a line's users have been omitted, then clearly the line is not required by anyone and should be omitted too. Therefore, the list of users serves a rôle in the computation of a line's *visibility*—a notion which I shall introduce presently.

9.2.2 The Notion of Visibility

In every proof tree there are inference steps which are of no interest to the user because the conclusion does not play an essential rôle in the theory. In typical AI axiomatizations, type propositions of the form `(IS` ⟨*type*⟩ ⟨*object*⟩`)` do not carry a great deal of information. They can typically be omitted from a proof summary with no loss of clarity. In fact their omission reduces the clutter and helps clarify the

summary. Therefore, the editor's heuristics recognize such lines and arrange for them to be hidden instead of visible.

Other lines correspond to inferences which are so trivial that it is unnecessary to display them. For example, if line ℓ' is inferred from line ℓ by IDENTITY, it would be rather silly to show both ℓ and ℓ'. In such a case, the editor's heuristics will arrange for ℓ' to be *in-lined*. In other words, ℓ' is not displayed, and wherever it is referenced as a premise, the editor will display a reference to ℓ instead. When reading the summary, the user is quite unaware that he is supplying a missing inference.

The same technique applies to SYMMETRY. If line ℓ is $a = b$ and line ℓ' is $b = a$ derived from ℓ by SYMMETRY, ℓ' can be in-lined with no loss of clarity: it is obvious from the context which side of the equality is used to replace what.

A slightly more complex case is that of conjunctions produced by AND-INTRO. For example, let's say line ℓ_i is p_i for $i = 1$ to n, line ℓ' is $p1 \wedge \ldots \wedge p_n$ obtained by AND-INTRO from the ℓ_i's, and line ℓ'' is $p_1 \wedge \ldots \wedge p_n \supset q$; from ℓ' and ℓ'', MODUS-PONENS infers q (line ℓ'''). Clearly, line ℓ' does not contribute much to the summary, except that it takes up space with a trivial inference. Instead, the default heuristics will arrange for ℓ' to be in-lined; which means that the justification of line ℓ''' (the conclusion from MODUS-PONENS) will outwardly reference lines ℓ_i for $i = 1$ to n instead of line ℓ', and say something like "*From $\ell_1 \ldots \ell_n$ and ℓ''.*" Once again, the missing inference should be quite transparent to someone reading the summary.

This last example brings up interesting questions. Remember that the reference to line ℓ' is the concatenation of references to lines ℓ_i for $i = 1$ to n. Typically, this would produce the corresponding sequence of line numbers. However, suppose line ℓ_k for some k, $1 \leq k \leq n$, is also in-lined. Then, its own list of references would have to be *spliced* in where its line number would otherwise have appeared. As a result, the list of references making up the in-line representation of line ℓ' could grow to very large proportions and could no longer be considered transparent. Therefore, the size

of the in-line reference must be taken into account when deciding whether to in-line or not.

Also, if an in-lined conclusion is used in several places (has multiple users), then its longuish reference text will have to appear repeatedly in various justifications. It might be better to not in-line it and be able to reference it simply by its line number. There is clearly a tradeoff between the number of lines which are actually displayed and the size of individual justifications.

To support the sort of functionality outlined above, each line has a *visibility* property which can take one of three values: *normal*, *in-line*, *hide*. A normal line is just what you would expect, and it is referred to by its line number (or maybe by its name if it is an axiom or an assumption). An in-lined line is not displayed and is referenced by concatenating the references to its steps. A hidden line is neither displayed nor referred to (its reference text is empty).

The editor's heuristics assign appropriate visibility to each line. These decisions may be explicitly overriden by the user, and can also be altered by tweaking various parameters.

9.2.3 Editing Operations

In this section I will give a brief synopsis of editing operations supported by the proof editor.

motion commands: to change the current line (`previous`, `next`, `line`), or go back to the proof whose view we are presently editing (`proof`).

bury/unbury: to control what lines are actually shown for editing purposes. This has nothing to do with which lines are shown in the summary, but merely allows

the user to restrict the display of the proof view to those lines relevant to the editing task which he is presently engaged in.

line ordering: lines must be ordered such that none comes after any of its users. This policy only specifies a partial ordering. By default, the ordering respects the proof structure: an inference comes after its steps and each step comes after its left sibling. The `float` command allows the user to move lines upwards or downwards in the sequence of lines, as long as this motion remains consistent with the partial ordering.

dump: produces a summary of the proof. In particular, `dump -document -tex` *filename* will output the summary to file *filename* in a format suitable for processing by LaTeX.

parameters: many aspects of the editor, of its heuristics, and the visual appearance of the summary, are controlled by parameters, also called *options*. The command `option -set` *name value* may be used to set the proof view's *name* option to *value*.

editing a line: from a proof view, it is possible to move to a representation of a line with the `edit` command. Here is what you can do to a line:

1. Turn it on/off. This is somewhat like hidding it.

2. Turn some of its steps on/off. Steps are represented by *sublines* which point to the actual lines. This extra level of indirection makes it possible to associate properties with the subline to override those of the corresponding line without altering the latter. For instance, it is possible to turn off a step (i.e. it will not be mentioned in the justification of this line) without turning off the corresponding line (which may still be mentioned in other justifications).

3. Add a comment. It is occasionally a good idea to intersperse comments with the lines of the summary to explain what is going on. The summary generator knows that comments should be formatted as text.

4. Set the line's *simplicity*. The editor's heuristics take into account the simplicity of a line to decide whether said line should be in-lined. The simplicity of a line is also computed by heuristics, but this determination may be explicitly overridden by the user.

5. Set the line's *visibility* to explicitly override the default computation.

6. Name a line. Typically useful to assign a more meaningful name to an assumption. For example, if assumption `!:=309` is $a = b$, it might be a good idea to rename it **`a-equals-b`** to improve the readability of the summary.

7. Move to other nodes: **`user`**, **`step`**, **`proof`**, **`view`**, **`hset`**.

9.3 Construction of a Proof View

In this section I will describe the data-structures underlying the implementation, outline the process of initialization, and explicitate the algorithm computing a line's visibility.

9.3.1 Data-Structures

Proof View

A proof view is essentially a doubly-linked list of lines (see below), together with a set of properties specifying heuristic policies and visual appearance. For practical

purposes it is also a good idea to maintain a list of the axioms occurring therein, as well as a list of all the lines representing assumptions. A list of *hsets* representing all the assumption sets is similarly maintained. A proof view is implemented by a structure of the form:

$$\langle \text{PROOF, LINES, AXIOMS, ASSUMPTIONS, HSETS, PROPS} \rangle$$

Constructing a proof view has the side-effect of consing up a line for the proof's conclusion; this, in turn, causes lines to be recursively constructed for all subproofs.

Line

Each proof in the proof tree is converted to a line. Lines are maintained as a doubly-linked list that reflects their natural sequencing in the Natural Deduction summary. Furthermore, each line includes a list of steps (which are the lines used as premises by this inference step) and a list of users (those other lines which require this one as a premise). There is pointer to the *hset* (see below) representing the proof's assumption set, and also a set of properties (e.g. *visibility* and *simplicity*). A line is implemented by a structure of the form:

$$\langle \text{PROOF, PREV, NEXT, HSET, STEPS, USERS, KIND,}^{1} \text{ VIEW, PROPS} \rangle$$

Construction of a line object first requires constructing lines for the proof's steps, then packaging these lines into sublines (see below). An *hset* must also be obtained to represent the proof's assumption set. Finally, the new line object is appended to the view's doubly-linked list of lines (appending it guarantees that it appears *after* the lines representing its steps).

Note that lines are uniquified in the sense that, if the same proof occurs several times in the proof tree, the same line object will be used for all these occurrences.

[1] A symbol—one of `axiom`, `assumption`, `deduction`—which indicates what kind of a line this is.

Subline

A subline represents the *use* of a line as a premise to infer another line. It is implemented by a pair ⟨LINE, PROPS⟩. When a line ℓ is packaged as a subline (step) for line ℓ', ℓ' must be added to ℓ's *users* list. At present, the only useful properties are `pgm-off`, which is true when heuristics have determined that the step should be turned off, and `usr-off`, which is used to indicate an overriding decision by the user to turn the step off in any case.

Hset

An hset is the representation of an assumption set, and plays a rôle during the dumping of a proof summary: assumptions are not displayed for each line, instead, whenever two consecutive lines do not depend on the same set of assumptions, a description of the second assumption set is inserted between the two lines; this description remains in effect until overriden by another. An hset is implemented by a structure of the form:

⟨ASET, ASSUMPTIONS, PARENT, VIEW, PROPS⟩

which closely resembles the structure of the assumption set ASET. ASSUMPTIONS is the list of lines coresponding to the local assertions; PARENT is the parent hset; VIEW is the proof view; and PROPS are the hset's properties.

Hsets are uniquified in the sense that the hset constructor, when invoked repeatedly for the same assumption set, will return the same hset object rather than cons up a new one each time. This is important since, whenever a proof is converted to a line, its assumption set is also converted to an hset, and proofs may share the same assumption set; therefore, lines should similarly share the same hsets.

To convert an assumption set to an hset, the parent assumption set must first be converted to an hset. Then, the local assertions must be transformed into lines.

As far as the user is concerned, the only interesting property of an hset is `usr-name`, whose value is a string that names the hset and overrides the default naming scheme.

9.3.2 Initialization

After the proof view has been constructed, the following initialization steps must be performed:

1. Set up the proof view's default properties. Typically, this involves copying global defaults to local properties.
2. Establish the doubly-linked list of lines.
3. Sort axioms, sort assumptions. The default policy for a summary is to not display axioms as ordinary lines, but rather to list them all first, then refer to them by name in the body of the summary. To make it easier to look up an axiom, they are sorted by alphabetical order.
4. Compute each line's visibility.
5. Assign line numbers.
6. Assign hset numbers. An hset is typically named by a string consisting of the prefix `H` followed by a number. The user may override this default naming scheme.

9.3.3 Computing a Line's Visibility

The algorithm presented in this section determines whether a line should be normal, in-lined, or hidden. It consists of 2 passes: the first one makes the determination on the basis of local information, while the second pass merely turns back on certain lines because of other inferences where they are involved as premises.

First Pass

The first pass is guided by the following principles:

- *Hide system axioms.* This is principally of interest for built-in theories which are implemented using back-chaining rules. The justifications for deductions arrived at through this mechanism should usually be hidden from the user. The following declaration:

  (hide-from-lc '⟨*name*⟩)

  will cause assertion ⟨*name*⟩ to not be mentioned in a summary. The line corresponding to assertion ⟨*name*⟩ is always hidden.

- *Show axioms and assumptions.*

- *Hide simple deductions obtained from hidden premises.* If a line is trivially derived from premises all hidden, then it probably doesn't have to be visible either. The deduction is considered trivial here if the rule is either one of the **hide-rules** or one of the **in-line-rules**.

  hide-rules is a property of the view which consists of a list of inference rules. Inference steps validated by one of these rules should be in-lined.

  in-line-rules is similar, except that the corresponding inferences should be in-lined only when simple enough.

- *Hide lines discharging an assumption on a hidden conclusion.* If line ℓ' corresponds to $A \Rightarrow p \supset q$ and is derived from line ℓ, corresponding to $A; p \Rightarrow q$, by IF-INTRO, then, if line ℓ is hidden, line ℓ' should be hidden too.

- *Identify trivial discharges.* This is intended to catch cases such as line ℓ', representing $A \Rightarrow p(a) \supset p(a)$, derived from line ℓ, representing $A; p(a) \Rightarrow p(a)$, by IF-INTRO. In this case the sublines of ℓ' are turned off because they are not necessary to understand the conclusion.

  Note that when a subline is turned off, a message is sent to the corresponding line to check on its users; If all users have been turned off, the line should turn itself off too.

- *Hide unessential propositions.* If the conclusion's principal symbol is one of the **hide-predicates**, then the line should be hidden.

  hide-predicates is a property of the view which consists of a list of predicates which are considered to state unessential propositions. At present, the list has only one element: **is**, which denotes type predication.

- *Show literals.* Conclusions which are literals are the basic blocks supporting the whole deduction; it is generally a good idea to display them as lines regardless of other (in-lining) heuristics. This policy is controlled through the view's **show-literals** property.

  If **show-literals** is true, and the conclusion is a literal, and it is not a trivial deduction from another literal (e.g. by IDENTITY or SYMMETRY), then the line should have normal visibility.

- *In-line Trivial Propositions.* Conjunctions are typically not very interesting as conclusions. There are two common cases: (1) a conjunction is inferred by AND-INTRO to serve as a minor premise for MODUS-PONENS, (2) a conjunction is inferred by MODUS-PONENS and then one of the conjuncts is inferred by

TAUT-TRANS. In either case, it is often unnecessary to actually display the line corresponding to the conjunction itself.

If the conclusion's principal symbol is one of the `in-line-predicates`, then, an in-line abbreviation is computed as follows: for each step that is not turned off, a list of references is computed. The list consist of the line number if the step has normal visibility (or the name, for an axiom or an assumption); for an in-lined step, it is recursively computed (and cached) in the manner which I am presently describing. The resulting lists are concatenated (and duplicate references are removed).

If the abbreviation thus computed does not exceed the size specified by the view's **`max-in-line-size`** property, then the line may be assigned visibility "in-line."

`in-line-predicates` is a property of the view which consists of a list of predicates. At present it has a single element: **and**.[2]

- *In-line Trivial Inferences.* This case is similar to the preceding item, but the triggering condition is that the inference rule validating this line's conclusion be one of **`hide-rules`** [see above]. The same decision procedure applies.
- *In-line Simple Deductions.* If the line is deemed simple, it should be in-lined. Simplicity is computed as follows:
  - It may be explicitly set by the user.
  - Axioms and assumptions are simple.
  - A conclusion derived trivially (see **`hide-rules`** and `in-line-rules`) from simple premises, and such that its in-line abbreviation does not exceed the maximum permitted size, is simple.

[2]For the purpose of summary preparation, we shall not distinguish between connectives and predicates. All that interests us is that either may appear as the principal symbol of a formula.

  - Other cases are not simple.

- All other lines are normal by default.

Second Pass

The point of this second pass is to change the visibility of certain lines from *in-line* to *normal*. This is done when a line plays an important rôle as a premise in some inference step; if it remained in-lined, the result might be a little confusing. For example, it is a good idea to make sure that the equality involved in an EQUALITY substitution inference appears as a line. Similarly for the disjunction of cases of the CASE rule. etc...

The visibility may be changed from *in-line* to *normal* only if it has not been overriden by the user.

If the line is a trivial inference (e.g. by IDENTITY or SYMMETRY) from a single premise, instead of changing the visibility of the line itself, the program makes sure that the line corresponding to said premise has visibility "normal."

When the visibility of a line has thus been changed, the visibility of its users must be recomputed.

9.4 Generating a Summary

A summary is generated when the user issues the **dump** command. In particular, **dump -tex -document** *filename* outputs the summary to file *filename* as a self-contained document ready to be processed by LaTeX.

In this section I will describe in greater detail the format of a summary; then I will outline the various algorithms involved in its generation and I will illustrate my data-driven implementation with snipets of code.

9.4.1 The Format of a Summary

A summary typically consists of 3 sections: abbreviations, axioms, and the proof itself, i.e. a sequence of lines.

Abbreviations

The first section of the summary consists in a list of definitions for the abbreviations used in the proof. An abbreviation [Section 4.4] is a skolem term used for notational purposes only, and which stands for some other term. The list makes it clear what each abbreviation stands for, and is intended to serve as a reference when reading the proof summary. This section may be explicitly omitted with the `-noabbreviations` option to the **dump** command.

The example from the Introduction Chapter had the following abbreviations section:

ABBREVIATIONS

```
!.ONE-THUMB  = (SK ONE-THUMB 7)
!.C          = (SK C 6)
!.OTHER-HAND[2](?ONE-HAND ?C)
             = (SK OTHER-HAND 2 ?ONE-HAND ?C)           -- THUMBS
!.OTHER-THUMB[3](?ONE-HAND ?C)
             = (SK OTHER-THUMB 3 ?ONE-HAND ?C)          -- THUMBS
!.ONE-HAND[5](?C ?T)
             = (SK ONE-HAND 5 ?C ?T)                    -- CREATURE-THUMB
!.ONE-HAND[5](!.C !.ONE-THUMB)
             = (SK ONE-HAND 5 !.C !.ONE-THUMB)          -- CREATURE-THUMB
!.OTHER-THUMB[3](!.ONE-HAND !.C)
             = (SK OTHER-THUMB 3 !.ONE-HAND !.C)        -- THUMBS
```

```
!.OTHER-HAND[2](!.ONE-HAND !.C)
            = (SK OTHER-HAND 2 !.ONE-HAND !.C)         -- THUMBS
!.HAND-OF-THUMB[4](?T)
            = (SK HAND-OF-THUMB 4 ?T)                  -- ONE-HAND-PER-THUMB
!.HAND-OF-THUMB[4](!.ONE-THUMB)
            = (SK HAND-OF-THUMB 4 !.ONE-THUMB)         -- ONE-HAND-PER-THUMB
!.OTHER-THUMB[3](!.ONE-HAND[5](?Y ?X) ?Y)
            = (SK OTHER-THUMB 3 !.ONE-HAND(?X ?Y) ?Y) -- THUMBS
```

An abbreviation and the term it stands for are displayed on either sides of an equality sign. Notice that only the top-level of the term is expanded; subterms may still be represented by abbreviations (e.g. `!.C` and `!.ONE-THUMB` in the 6th definition) which may be looked up in this same list.

Also, when a skolem term originated in an axiom, the name of that axiom is listed so that it may be easily looked up. The first two abbreviations above lack a corresponding axiom because they came from the skolemization of the original goal.

Axioms

The second section of the summary consists of a list of all the axioms to be explicitly referenced in the proof, and is also intended to serve as a reference when reading the proof. To make it easier to look up an axiom, the list is arranged in alphabetical order. This section may be explicitly omitted with the `-noaxioms` option to the `dump` command.

The axioms section from the example in the Introduction Chapter begins thus:

AXIOMS

```
CREATURE-THUMB
          (IF (AND (IS CREATURE ?C) (IS THUMB ?T) (PART ?T ?C))
              (AND (IS HAND !.ONE-HAND[5](?C ?T))
                   (PART ?T !.ONE-HAND[5](?C ?T))
                   (PART !.ONE-HAND[5](?C ?T) ?C)))
```

```
LEFT-OR-RIGHT
          (IF (IS HAND ?H)
              (AND (OR (LEFT ?H) (RIGHT ?H))
                   (NOT (AND (LEFT ?H) (RIGHT ?H)))))
```

Each axiom is named, and its corresponding formula is prettyprinted.

Proof

The third section of the summary is the Natural Deduction proof proper, and consists of a sequence of lines. The format of a line is typically:

$$\underbrace{\qquad\qquad}_{\langle indentation \rangle}\langle line\ number \rangle \text{———} \langle justification \rangle$$
$$\langle formula \rangle$$

The ⟨*indentation*⟩ reflects the level of assumption: the more assumptions the inference depends on, the larger the ⟨*indentation*⟩. The exact increment is determined by the view's `hset-indent` property. ⟨*indentation*⟩ is defined as the product of the assumption set's number of ancestors with this increment.

Of course, indentation alone is insufficient. We also need to state specifically what assumptions are being made. I devised a contextual scheme to track and report on the current set of assumptions as the proof unfolds. The general principle is that, if the assumption set of line $n+1$ is the same as that of line n, then there is no need to repeat a description of it; line n is simply displayed with the same ⟨*indentation*⟩ as line $n+1$, and the reader should interpret this as indicating that the same assumption set applies.

If line $n+1$ makes different assumptions from line n, the program finds the first ancestor A common to both assumption sets (by following parent links) and outputs descriptions of all the sets from A down to the assumption set of line $n+1$.

In particular, this means that if the assumption set for line $n+1$ is an ancestor of the one for line n, no description is output: we are simply exiting levels of assumptions and the reduced ⟨*indentation*⟩ tells us exactly what level of assumption. In practice, however, it may be difficult to correctly interpret the ⟨*indentation*⟩ without a ruler, especially across pages. Therefore, in this case a reminder of the form ⟨*hset descriptor*⟩ ⊢ is inserted just before line n's ⟨*formula*⟩. See, e.g. line 10 of the introductory example:

```
8 ——— From ONE-HAND-PER-THUMB and 1 Conclude:
  H1 ≡ !:PART280 ⊢
  (= !.HAND-OF-THUMB !.ONE-HAND)
```

Whenever the description of an assumption set not encountered before must be inserted, the program outputs a line of the form:

$$\underbrace{\qquad\qquad}_{\langle indentation \rangle} \textbf{Assume } \mathrm{H}_n \equiv a_1\ a_2 \ldots a_k + \mathrm{H}_m$$

Where ⟨*indentation*⟩ reflects the new level of assumptions, H_n is the name of this assumption set, H_m is the name of its parent, and the a_i's are the names of the local assertions. Each a_i is then defined by producing its name followed by its formula. For example:

```
Assume H2 ≡  !:=305 + H1
   !:=305 (= !.ONE-THUMB !.OTHER-THUMB)
```

When a description of the same assumption set must be inserted at some later point in the proof, the word **Assume** is replaced with **Assuming**, and the definitions of the a_i's are omitted. For example:

```
Assuming H2 ≡  !:=305 + H1
```

The ⟨*justification*⟩ is a short piece of text that explains this inference; the premises are referred to by line number (or by name for axioms and assumptions). Remember that a premise may be abbreviated in-line, in which case it is represented by the list of references to the lines it was derived from.

In this design, I made the choice to keep justifications simple and non-distracting. A justification seldom mentions the inference rule because either (1) it is quite clear how the conclusion follows from the premises, or (2) because, as a result of in-lining, it would be rather arbitrary and misleading to designate this rule as the significant one. Frequent templates are:

From ... Obviously
From ... and ... Conclude
By equality substitution of ... in ... Conclude

9.4.2 Implementation

Dumping a Summary

The algorithm for producing a summary has the following form:

1. *Preamble.* If the `-document` option was specified, a LaTeX preamble is inserted so that the resulting summary may be processed as a document.
2. *Abbreviations.* Unless printing of abbreviations has been disabled with the `-noabbreviations` option, a reference list of all abbreviations used in the proof is inserted at this point. This requires inspecting all axioms, assumptions, and lines which are to be displayed eventually in the summary.
3. *Axioms.* Unless printing of axioms has been disabled with the `-noaxioms` option, a reference list of all axioms explicitly mentioned in the proof is inserted at

this point. Note that this list is produced only if the proof view's `list-axioms` property has value **`first`**; that is, if the user wants axioms to be listed first. The alternative is for axioms to be listed in the proof.

4. *Assumptions.* Unless printing of assumptions has been disabled with the `-noassumptions` option, a reference list is produced in the same manner as above. This is controlled by the **`list-assumptions`** property. By default, assumptions are not listed first, but rather are introduced by descriptions of assumption sets interspersed with the lines of the proof.

5. *Assumption Sets.* If property **`list-hsets`** requires that *hsets* be listed **`first`**, a list of their respective descriptions is inserted at this point. By default, hsets are not listed first, and their descriptions are appropriately interspersed with the lines of the proof.

6. *Lines.* The lines with normal visibility are now "dumped" in their natural order. I shall say more about this in the next section.

7. *Postamble.* If the **`-document`** option was specified, it is necessary to insert some LaTeX code to end the document.

Dumping a Line

If the line is an axiom (resp. assumption), and property **`list-axioms`** (resp. **`list-assumptions`**) has value **`yes`** (in particular, it is not **`first`**, which means that axioms (resp. assumptions) where not listed first, before the proof), then a line is output specifying name and formula.

Otherwise, the line is a deduction and the following procedure executes:

1. The algorithm keeps track of the last assumption set. If the current assumption set is the same as the last one, we can go on to the next step. Otherwise, it

may be different in either of 3 ways. Let A be the last assumption set and B be the current one.

(a) $A \subset B$. The new assumption set B makes strictly more assumptions than the last one, A, i.e. A is an ancestor of B in the hierarchy of assumption sets. Descriptions of all the assumption sets between A (not included) and B (included) are inserted.

(b) $B \subset A$. The new assumption set B is an ancestor of the old one, A. This means we are simply popping out of further levels of assumptions. No descriptions need be inserted.

(c) A and B are unrelated. We must find the first ancestor common to both A and B and output descriptions of all the assumption sets between that ancestor and B.

The last assumption set is updated to be B for the next iteration.

2. Now a description of the line itself must be output. This involves indenting according to the level of assumptions; the deeper the assumption set is in the hierarchy, the more indented the line must be (increment specified by property `hset-indent`).

   Then the line's reference is inserted; Typically this is the line's number, but the user may want to give names to certain lines.

   A horizontal rule is added, followed by a justification of the line's inference. The synthesis of a justification is discussed in the next section.

   If we popped out of a few levels of assumptions, a brief reminder of the assumption set is inserted here, followed by a turnstile $\vdash$.

   Finally the line's conclusion is prettyprinted.

3. If the line has a comment associated with it, it is inserted at this point. When LaTeX output is being produced, the corresponding text is typeset in a `\parbox` and delimited by horizontal rules.

Dumping a Justification

The synthesis of a justification is driven by the corresponding inference rule. Each inference rule is associated with a *justification procedure*, which is applied to the line and is responsible for outputing text interspersed with references to the premises in a manner that makes intuitive sense to the reader.

For example, the justification procedure for inference rule CASE is coded as follows:

```
(DATAFUN DUMP-LINE-JUSTIF CASE                                  case analysis
  (DEFPROC - void (LINE - line_t)
    (OUT (&rm "By case analysis")
         (E (DUMP-STEPS !>LINE.STEPS '#T))
         1 (&rm "Conclude:"))))
```

A slightly more complex example is MODUS-PONENS, because it must check and behave differently according to whether either/both of its steps are off/on.

```
(DATAFUN DUMP-LINE-JUSTIF MODUS-PONENS                          modus ponens
  (DEFPROC - void (LINE - line_t)
    (COND ((<& !_OFF !>LINE.STEPS)
           (OUT (&rm "Obviously:")))
          ((!_OFF (CAR !>LINE.STEPS))
           (OUT (&rm "From minor premise")
                (E (DUMP-STEP (CADR !>LINE.STEPS) '#F))
                1 (&rm "Obviously:")))
          ((!_OFF (CADR !>LINE.STEPS))
           (OUT (&rm "From major premise")
                (E (DUMP-STEP (CAR !>LINE.STEPS) '#F))
                1 (&rm "Obviously:")))
          (T (OUT (&rm "From")
                  (E (DUMP-STEP (CAR !>LINE.STEPS) '#F))
                  1 (&rm "and")
                  (E (DUMP-STEP (CADR !>LINE.STEPS) '#F))
                  1 (&rm "Conclude:"))))))
```

9.4.3 Example

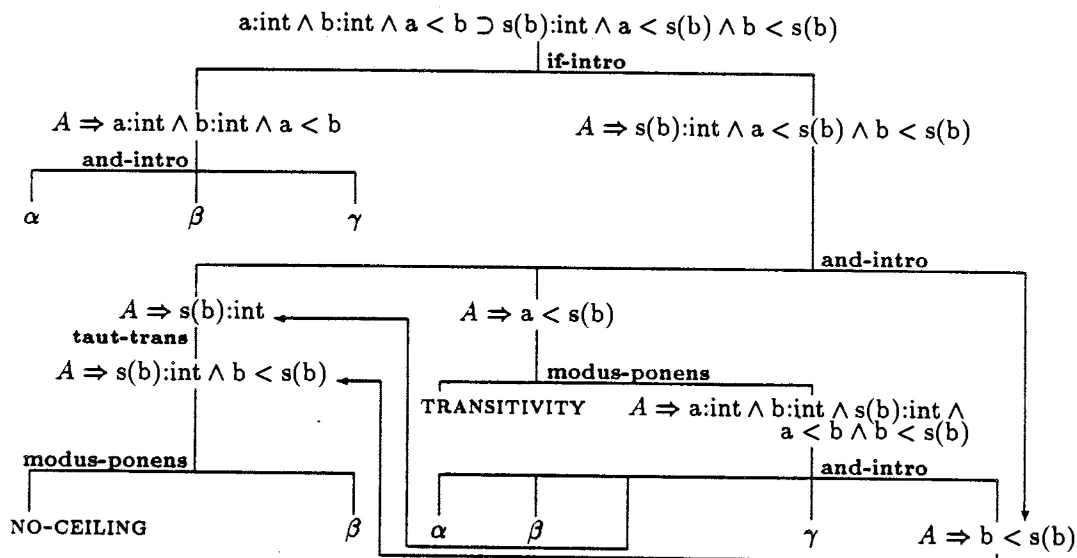

Figure 9.1: Proof Structure

In this section, I will produce an example to illustrate the generation of a summary. This example is purposedly trivial; the size of a non-trivial example would overwhelm the reader and would prohibit a graphical display of the proof structure. Also, proof generalization was not applied so as not to confuse the issue with irrelevant details.

The proof provides an answer to the goal of showing that for any two integers a and b, such that a < b, there exists an integer u which is greater than both of them. The following two axioms are required:

NO-CEILING $\forall n{:}\text{int}\,\exists s{:}\text{int}\, n < s$
TRANSITIVITY $\forall x, y, z{:}\text{int}\centerdot x < y \wedge y < z \supset x < z$

where the notation x:int represents the proposition asserting that x is of type int (i.e. integer). Skolemization transforms the axioms into the following quantifier-free formulae:

NO-CEILING n:int $\supset$ s(n):int $\wedge$ $n <$ s(n)
TRANSITIVITY x:int $\wedge$ y:int $\wedge$ z:int $\wedge$ $x < y \wedge y < z \supset x < z$

The notation s(n) is meant to convey the idea of the successor of n.

The proof structure is displayed in Figure 9.1, where A is used to abbreviate the following assumption set:

$$A = \{\alpha\text{: a:int},\ \beta\text{: b:int},\ \gamma\text{: a} < \text{b}\}$$

For each node in the proof structure, a line is constructed. The natural order of these lines is determined by a depth-first, left-to-right traversal of the structure:

1. α
2. β
3. γ
4. $A \Rightarrow$ a:int $\wedge$ b:int $\wedge$ a $<$ b
5. NO-CEILING
6. $A \Rightarrow$ s(b):int $\wedge$ b $<$ s(b)
7. s(b):int
8. TRANSITIVITY
9. $A \Rightarrow$ b $<$ s(b)
10. $A \Rightarrow$ a:int $\wedge$ b:int $\wedge$ s(b):int $\wedge$ a $<$ b $\wedge$ b $<$ s(b)
11. $A \Rightarrow$ a $<$ s(b)
12. $A \Rightarrow$ s(b):int $\wedge$ a $<$ s(b) $\wedge$ b $<$ s(b)
13. a:int $\wedge$ b:int $\wedge$ a $<$ b $\supset$ s(b):int $\wedge$ a $<$ s(b) $\wedge$ b $<$ s(b)

By default, axioms and assumptions are not displayed as lines: axioms are listed before the proof, and assumptions appear in the descriptions of assumption sets. Therefore, lines 1, 2, 3, 5, and 8 will not appear in the summary.

Line 4 is small conjunction of assumptions; therefore, it is trivial and should be in-lined.

Line 6 is a trivial inference by MODUS-PONENS from axiom NO-CEILING and assumption β; it should be in-lined.

Line 7 concludes that s(b) is an integer. Since, by default, type propositions are considered uninteresting, line 7 will be hidden.

Line 9 is a literal, and the default heuristic is to show lines concluding literals.

Line 10 is concluded by AND-INTRODUCTION from 5 premises. Such a conclusion should be in-lined if the justification is short enough. Since 3 of the premises are type propositions, they will not be shown. For the remaining premises: a $<$ b is γ, and b $<$ s(b) is line 9 which is displayed and can be referred to by number. Consequently, the in-line justification is of size 2, which is small enough and warrants in-lining line 10.

Line 11 is a literal and must be displayed (see line 9).

Line 12 is a conjunction with justification of size 2 (see line 10): it is in-lined.

Line 13 is the ultimate conclusion and is necessarily displayed.

On request, the system produces the following summary where the lines actually displayed are given consecutive reference numbers (i.e. line 9 is assigned the number 1, line 11 is assigned 2, and line 13 is assigned 3):

ABBREVIATIONS

```
!.A          = (SK A 3)
!.B          = (SK B 4)
!.S[2](?N)   = (SK S 2 ?N)
!.S[2](!.B)  = (SK S 2 !.B)
```

AXIOMS

```
NO-CEILING (IF (IS INTEGER ?N) (AND (IS INTEGER !.S[2](?N)) (< ?N !.S[2](?N))))

TRANSITIVITY
          (IF (AND (IS INTEGER ?X)
                   (IS INTEGER ?Y)
                   (IS INTEGER ?Z)
                   (< ?X ?Y)
```

```
          (< ?Y ?Z))
      (< ?X ?Z))
```

PROOF

```
Assume H1 ≡  !:IS245 !:IS246 !:<247
  !:IS245
        (IS INTEGER !.A)
  !:IS246
        (IS INTEGER !.B)
  !:<247 (< !.A !.B)

1 ——— From NO-CEILING Obviously:
  (< !.B !.S[2](!.B))

2 ——— From TRANSITIVITY and !:<247 1 Conclude:
  (< !.A !.S)

3 ——— From 2 1 Discharging !:<247 Conclude:
 H0 ⊢
 (IF (AND (IS INTEGER !.A) (IS INTEGER !.B) (< !.A !.B))
     (AND (IS INTEGER !.S) (< !.A !.S) (< !.B !.S)))
```

9.5 Conclusion

McAllester [McA87] defines the notion of *expansion factor* thusly:

> *One way of measuring the performance of a verification system is to compare the length of a natural argument with the length of a corresponding machine readable proof. The ratio of the length of a machine readable proof to the length of the corresponding natural argument is called the* expansion factor *for that proof.*

In this chapter, I have described a framework and given algorithms/heuristics for the dual problem which is to produce a human readable summary from a machine represented proof. Thus far, the system only outputs Natural-Deduction summaries, and it is unclear how to characterize the notion of *expansion factor*; however, Chapter 10 provides some data points: the first proof presented therein required 55 invocations

of plan generators to derive, involves 71 axioms or assumptions and 130 distinct inference nodes (some of them used multiple times), and resulted in a summary 33 lines long. Another interesting approach would be to translate proofs into English [Che76].

Chapter 10

An Axiomatization of Qualitative Physics

In this chapter, I describe an axiomatization of Qualitative Physics inspired by Kuipers's theory of Qualitative Simulation. There are several reasons for presenting this particular work.

Firstly, it is interesting in its own right. Kuipers' theory is more formal than most work in Qualitative Physics, and, as we shall see, can be converted straightforwardly into an axiomatic theory which permits deductive envisionment.

Secondly, it serves as an illustration of the sort of problems which can be tackled with LOGICALC.

Thirdly, it shows how LOGICALC can easily be extended with additional rules and plan generators to facilitate the study of certain theories. Since the use of the situation calculus revealed itself to be somewhat inconvenient for certain operations, I decided to experiment with some ideas. As a consequence, this is more recent work than I originally intended to present. I wrote specialized inference rules for the situation

calculus, and found them to greatly facilitate theorem proving work in the situation calculus, and to permit a more concise axiomatization. It took me only two days to write the additional inference rules, plan generators, and generalizers, formulate a cleaner axiomatization, and rederive the proof for the U-Tube example.

In this exposition, I will not axiomatize time or the logic of fluents, as this level of detail will not be needed. I will study two problems in qualitative physics and present the corresponding proofs derived and prepared with the aid of LOGICALC. The second problem calls for a slight refinement of the axiomatization which follows presently.

10.1 Qualitative Simulation

In 1986, B. Kuipers described a theory of Qualitative Simulation [Kui86], with the intention of (1) providing a precise ontology, (2) clarifying and formalizing the qualitative mathematics behind the prediction of behavior from qualitative constraint equations, (3) presenting a complete algorithm generalizing the best features of existing algorithms.

My purpose here is to borrow from his ontology and his algorithm to derive a logical axiomatization appropriate and sufficient for carrying out proofs. Therefore, I will only describe his approach in so far as it is required to understand the derivation of my axiomatization.

The mandate of a qualitative simulation algorithm is this: from a description of the structure of the system, and given an initial state, produce a graph of the possible future states of the system (a directed edge between state A and state B indicates that state B is a possible immediate successor to state A).

The structure of a system is described by a set of (physical) parameters (quantities) and a number of constraint equations between them. Each parameter is taken to

Figure 10.1: U-Tube

be a continuously differentiable real-valued function of time. At any point in time, the value of a parameter is specified qualitatively in terms of its relationship with a totally ordered set of landmark values (whose essential properties are their ordinal relationships). The qualitative state of a parameter is determined by its ordinal relationship with landmark values and its direction of change. There is a totally ordered set of distinguished time-points: the current time is either at or between distinguished time-points.

In order to illustrate this theory, we now turn to Kuipers' example of the U-tube, which will later serve as one of the application examples for which a proof was derived using LOGICALC: two partially filled tanks are connected at the bottom by a thin tube. In the initial state, there is more water in tank A than in tank B. QSIM determines that water flows from tank A to tank B until an equilibrium is reached. Figure 10.1 depicts the initial state and mentions the quantities involved: HA and HB are the

water-levels in tank A and tank B; initially, they are respectively at HA0 and HB0. The respective pressures at the bottom of each tanks are PA and PB, and FAB is the flow of water from A to B. The following table recapitulates the quantities together with their initial values and corresponding set of landmark values:

Quantity	Initial Value	Landmark Values
HA	HA0	$\{0, \mathsf{HA0}, +\infty\}$
HB	HB0	$\{0, \mathsf{HB0}, +\infty\}$
PA	PA0	$\{0, \mathsf{PA0}, +\infty\}$
PB	PB0	$\{0, \mathsf{PB0}, +\infty\}$
PAB	PAB0	$\{-\infty, 0, \mathsf{PAB0}, +\infty\}$
FAB	FAB0	$\{-\infty, 0, \mathsf{FAB0}, +\infty\}$

These quantities satisfy a number of constraints, e.g. the pressure PA at the bottom of tank A is a monotonically increasing function of the water-level HA—which is expressed by the constraint $\mathsf{PA} = M^+(\mathsf{HA})$. More specific constraints, such as DERIV, ADD, MULT, and MINUS, are available and can be used to encode more complex relationships and qualitative differential equations.

These constraints are supplemented with sets of corresponding values. A set of corresponding values is typically an *interesting* solution for a particular constraint, where a solution is considered interesting when the values are landmarks. For example, the pressure PA in tank A becomes 0 when the water-level HA becomes 0. This is expressed by the set of corresponding values $\{\mathsf{HA} = 0, \mathsf{PA} = 0\}$.

The table below summarizes a possible choice of constraints and corresponding values for the U-Tube problem:

Constraint	Corresponding Values
$PA = M^+(HA)$	$\{HA = 0, PA = 0\}$ $\{HA = HA0, PA = PA0\}$ $\{HA = +\infty, PA = +\infty\}$
$PB = M^+(HB)$	$\{HB = 0, PB = 0\}$ $\{HB = HB0, PB = PB0\}$ $\{HB = +\infty, PB = +\infty\}$
$PA = \text{ADD}(PAB, PB)$	
$FAB = M^+(PAB)$	$\{PAB = 0, FAB = 0\}$ $\{PAB = -\infty, FAB = -\infty\}$ $\{PAB = +\infty, FAB = +\infty\}$ $\{PAB = PAB0, FAB = FAB0\}$
$FAB = \text{DERIV}(HB)$	
$FAB = \text{DERIV}(HA)$	

The initial state is then specified by listing the qualitative state for each quantity at the beginning of the experiment:

Quantity	Qualitative State
HA	$\langle HA0, \text{DEC} \rangle$
HB	$\langle HB0, \text{INC} \rangle$
PA	$\langle PA0, \text{DEC} \rangle$
PB	$\langle PB0, \text{INC} \rangle$
PAB	$\langle PAB0, \text{DEC} \rangle$
FAB	$\langle FAB0, \text{DEC} \rangle$

With QSIM, it is not necessary to provide explicitly all initial qualitative states and sets of corresponding values because, given sufficient information, the algorithm can propagate this information through the network of constraints, and infer the missing pieces.

The QSIM algorithm proceeds to infer the closure of the 'next state' relation, starting from the initial state. This process of inference is controlled, guided, and pruned in various ways.

First, a state is either at a distinguished time-point or between two adjacent distinguished time-points. Let's call the former a *point situation* and the latter an *interval*

situation. These two types of situations must alternate: this is the first constraint on permissible transitions. A transition from a point situation to an interval situation is called a *P-transition,* while a transition from an interval situation to a point situation is called an *I-transition.*

Secondly, since quantities are assumed to be continuously differentiable, the Intermediate Value Theorem and the Mean Value Theorem restrict the way they can change from one qualitative state to the next. Table 10.1, copied from [Kui86], summarizes the permissible transitions—ℓ_i and ℓ_{i+1} are adjacent landmark values, and ℓ_* is a new landmark introduced by the algorithm, and placed between ℓ_i and ℓ_{i+1}. From this

P-transitions	State	Next
P1	$\langle \ell_i$,STD$\rangle$	$\langle \ell_i$,STD$\rangle$
P2	$\langle \ell_i$,STD$\rangle$	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$
P3	$\langle \ell_i$,STD$\rangle$	$\langle (\ell_{i-1},\ell_i)$,DEC$\rangle$
P4	$\langle \ell_i$,INC$\rangle$	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$
P5	$\langle (\ell_i.\ell_{i+1})$,INC$\rangle$	$\langle (\ell_i.\ell_{i+1})$,INC$\rangle$
P6	$\langle \ell_i$,DEC$\rangle$	$\langle (\ell_{i-1},\ell_i)$,DEC$\rangle$
P7	$\langle (\ell_{i-1},\ell_i)$,DEC$\rangle$	$\langle (\ell_{i-1}.\ell_i)$,DEC$\rangle$
I-transitions		
I1	$\langle \ell_i$,STD$\rangle$	$\langle \ell_i$,STD$\rangle$
I2	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$	$\langle \ell_{i+1}$,STD$\rangle$
I3	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$	$\langle \ell_{i+1}$,INC$\rangle$
I4	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$
I5	$\langle (\ell_i,\ell_{i+1})$,DEC$\rangle$	$\langle \ell_i$,STD$\rangle$
I6	$\langle (\ell_i,\ell_{i+1})$,DEC$\rangle$	$\langle \ell_i$,DEC$\rangle$
I7	$\langle (\ell_i,\ell_{i+1})$,DEC$\rangle$	$\langle (\ell_i,\ell_{i+1})$,DEC$\rangle$
I8	$\langle (\ell_i,\ell_{i+1})$,INC$\rangle$	$\langle \ell_*$,STD$\rangle$
I9	$\langle (\ell_i,\ell_{i+1})$,DEC$\rangle$	$\langle \ell_*$,STD$\rangle$

Table 10.1: Legal Transitions

table, and given a state description, QSIM can generate a set of candidates for the next state, and then uses the constraints to filter out those states which are inconsistent.

10.2 Situations

The desired axiomatization must properly capture the notion of states and of transitions between them. This is precisely what the situation calculus, introduced by McCarthy [McC57], has to offer.

There are various formulations of the situation calculus, but they all include the ability to express that certain propositions hold in a given state. For my purposes, I will require the second-order predicate **during**. The notion that *formula* holds in *situation* is expressed by the proposition:

(**during** *situation formula*)

Like Kuipers, we are committed to the idea of distinguished time-points, and to the notion that the system is either at a distinguished time point (point situation), or between two adjacent distinguished time-points (interval situation). Thus we define two types:

```
(DEFDUCKTYPE timepoint obj)
(DEFDUCKTYPE situation obj)
```

A point situation is denoted by `(AT` *t*`)`. For example, in the U-Tube example, the initial situation is `(AT T0)`. The situation following *s* is denoted by `(NEXT` *s*`)`.

```
(DEFFUN situation (AT   ?T - timepoint))
(DEFFUN situation (NEXT ?S - situation))
```

As announced earlier, the second-order predicate `during` allows us to state that a certain formula is true in a given situation.

```
(DEFPRED (DURING ?S - situation ?P - prop))
```

In the situation calculus, actions and events are often construed as functions from one state to the next—which results in the well-known limitation that the situation calculus is not well-suited to capturing domains with simultaneous actions, or concurrent processes, or where an explicit ordering (threading) of events is not possible or desirable.

The ontology required by Qualitative Simulation calls neither for actions nor for events, but for transitions. In any given situation, the set of candidate transitions is entirely determined by Kuipers' table. We shall not consider transitions to be functions from state to successor state. Instead, we choose to formulate everything hypothetically and say *"if transition t occurs at the end of situation s, then proposition p holds is the next situation (successor of s)."*

For convenience, we define point situations and interval situations as two specialized subtypes of situations.

```
(DEFDUCKTYPE p-situation situation)
(DEFDUCKTYPE i-situation situation)
```

In order to relieve the user from the tedious duty of having to prove that a situation is of one of these types, we write a short logic program:

```
(ASSERTION-GROUP SITUATION-TYPES
  (<- (IS p-situation (AT ?T)) (IS timepoint ?T))
  (<- (IS p-situation (NEXT ?S))
      (AND (NOT (VARS ?S ALL))
           (IS i-situation ?S)))
  (<- (IS i-situation (NEXT ?S))
      (AND (NOT (VARS ?S ALL))
           (IS p-situation ?S))))
```

For good measure, we include the corresponding axiomatization:

```
(AXIOM AT-SITUATION
  (FORALL (T - timepoint)
    (IS p-situation (AT T))))

(AXIOM NEXT-OF-P-SITUATION
  (FORALL (S - p-situation)
    (IS i-situation (NEXT S))))

(AXIOM NEXT-OF-I-SITUATION
  (FORALL (S - i-situation)
    (IS p-situation (NEXT S))))
```

10.3 Quantities

Each physical parameter is represented by a quantity.

```
(DEFDUCKTYPE quantity obj)
```

10.3.1 Landmarks

A landmark is a distinguished value for a quantity, therefore it is a number.

```
(DEFDUCKTYPE landmark number)
```

Each quantity is associated with a set of landmark values. The basic idea is to say something of the form:

$$\texttt{(during } s \texttt{ (landmarks } q \texttt{ [}\ell_1\ \ell_2\ \ldots\ \ell_n\texttt{]))}$$

to indicate that q has this set of landmark values in situation s. In addition, the list argument makes ordinal relationships explicit: landmark values are listed in increasing order.

However, because of transitions I8 and I9, we need to be able to insert new landmark values in this list. In order to facilitate these insertions, we split the list in two: one part below the current value of the quantity, one part above:

```
(DEFPRED (LANDMARKS ?Q - quantity ?LEFT ?RIGHT - (LST landmark)))
```

The precise interpretation is this: if we are steady or increasing at a landmark, that landmark is the first one on the RIGHT. If we are decreasing at a landmark, it is the last one on the LEFT. If we are between two landmarks, the smaller one is last on the LEFT, and the larger one is first on the RIGHT.

We introduce the predicate landmarks< to reason about the ordinal relationship between landmarks.

```
(DEFPRED (LANDMARKS< ?Q - quantity ?X ?Y - landmark))

(AXIOM LANDMARKS-ORDER
  (IF (DURING ?S (LANDMARKS< ?Q ?X ?Y))
      (DURING ?S (< ?X ?Y))))
```

and we write a short logic program to automatically prove landmarks< goals by looking up the set of landmark values associated with the quantity in the given situation:

```
(ASSERTION-GROUP LANDMARKS-ORDER-DEF
  (<- (DURING ?S (LANDMARKS< ?Q ?X ?Y))
      (OR (DURING ?S (LANDMARKS ?Q [!?L1 ?X !?L2 ?Y !?L3] ?L4))
          (DURING ?S (LANDMARKS ?Q ?L4 [!?L1 ?X !?L2 ?Y !?L3]))
          (DURING ?S (LANDMARKS ?Q [!?L1 ?X !?L2] [!?L3 ?Y !?L4])))))
```

10.3.2 Qualitative States

In Kuipers' formulation, the *qualitative state* of a parameter expresses its ordinal relationship with landmark values and its direction of change. A parameter's landmark values are totally ordered; therefore a parameter is either *at* a landmark value or *between* two adjacent landmark values. Thus, Kuipers takes a qualitative state to be a pair $\langle \ell_i, dir \rangle$ or $\langle (\ell_i, \ell_{i+1}), dir \rangle$, where ℓ_i and ℓ_{i+1} are adjacent landmark values, and *dir* denotes the direction of change and is one of: STD (steady), INC (increasing), DEC (decreasing).

We characterize the state of quantity q in situation s, with a formula of the form:

(during s (state q d v))

where d is a direction (one of INC, DEC, STD), and v is either a list [ℓ_i] of one landmark if the quantity is at this landmark, or a list [ℓ_i ℓ_{i+1}] of two adjacent landmarks if the quantity is between them.

```
(DEFDUCKTYPE direction obj)
(DUCLARE STD INC DEC - direction)
(DEFDUCKTYPE value (LST landmark))
(DEFPRED (STATE ?Q - quantity ?D - direction ?V - value))
```

I shall postpone presenting the axiomatization of state changes until after transitions have been explained.

10.3.3 Values of Quantities

A physical parameter is a real-valued function of time, and it is often convenient to discuss the value of a quantity. We introduce the function `value-of` for this purpose.

```
(DUCLARE VALUE-OF - (FCN number [quantity]))
```

Note that this 'function' really denotes a fluent. When the quantity q varies between two values, there is no one single number denoted by the expression (value-of q). This should be interpreted as mere notational convenience; fluents are further discussed in the next section.

If a quantity is at a landmark, then its value is equal to that landmark.

```
(AXIOM AT-VALUE
  (IF (DURING ?S (STATE ?Q ?D [?L]))
      (DURING ?S (= (VALUE-OF ?Q) ?L)))))
```

If a quantity is between two landmarks, then all we can say is that its value is neither less than the smaller one, nor greater than the larger one.

```
(AXIOM BETWEEN-VALUES
  (IF (DURING ?S (STATE ?Q ?D [?L1 ?L2]))
      (AND (DURING ?S (NOT (< (VALUE-OF ?Q) ?L1)))
           (DURING ?S (< (VALUE-OF ?Q) ?L2)))))
```

Here are two additional axioms (lemmas) which I found useful during the derivation of the proof for the U-Tube problem. The first one states that if a quantity is increasing and greater than x, then it is still greater than x in the next situation.

```
(AXIOM NEXT-OF-INC
  (IF (AND (DURING ?S (STATE ?Q INC ?V))
           (DURING ?S (< ?X (VALUE-OF ?Q))))
      (DURING (NEXT ?S) (< ?X (VALUE-OF ?Q)))))
```

The second one states that if a quantity is between two landmarks, then, in the next situation, it is neither less than the smaller one, nor greater than the larger one.[1]

[1] Because of continuity.

```
(AXIOM NEXT-OF-BETWEEN
  (IF (DURING ?S (STATE ?Q ?D [?L1 ?L2]))
      (AND (DURING (NEXT ?S) (NOT (< (VALUE-OF ?Q) ?L1)))
           (DURING (NEXT ?S) (NOT (< ?L2 (VALUE-OF ?Q)))))))
```

10.4 Fluents and The Situation Calculus

As noted earlier, the value of a quantity often changes during an interval situation, and, therefore, writing something like:

(during *s* **(< (value-of** *q*) 0))

where we pretend that **value-of** is of type (FCN number [quantity]), is a little odd.

The truth is that the correct type for during's 2nd argument ought to be (PRD [timepoint]) rather than **prop**; i.e. it should be a *fluent* proposition. The meaning of a 'during' expression would be that the fluent proposition is true at each time point in the given situation.

Let me outline briefly what a correct axiomatization might look like. First, we would define during to take a fluent proposition as its 2nd argument:

```
(DEFPRED (DURING ?S - situation ?F - (PRD [timepoint])))
```

Now we should explicate the meaning of a 'during' expression: a fluent is true during a situation iff it evaluates to true at each and every timepoint in this situation.

```
(DEFPRED (WITHIN ?T - timepoint ?S - situation))

(AXIOM DURING-DEF
```

```
(FORALL (S - situation F - (PRD [timepoint]))
  (IFF (DURING S F)
       (FORALL (T - timepoint)
         (IF (WITHIN T S) (% F T))))))
```

Then we would define **value-of** to take an additional time point argument.

```
(DUCLARE VALUE-OF (FCN number [quantity timepoint]))
```

Finally, we would be able to write:

(during *s* (\\ (*t*) (< (value-of *q* *t*) 0)))

to express the notion that the value of q remains negative throughout situation s.

However, for my purposes, I never need to exploit the precise meaning of during, and the timepoint argument is always superfluous. Therefore, as a matter of notational convenience, I will always omit this timepoint argument, and the syntactic form of during's 2nd argument should be intepreted as a short hand for the corresponding fluent expression.

Another way of looking at this, is to assume that, by convention, inside of a during, type t is really type (**FCN** *t* **[timepoint]**). For example, **numbers** are actually real-valued functions of time; and, when I write 0, I really mean the constant fluent which is identically zero at every time-point.

When axiomatizing situated logic and situated arithmetic, we must keep in mind that we are specifying the properties of fluents. For instance, it is **not** the case that:

(or (during *s* *p*) (during *s* (not *p*)))

10.5 Transitions

In Kuipers' formulation, transitions are pairs of qualitative states—legal transitions are summarized in Table 10.1. In the context of the situation calculus, however, it seems more appropriate to think of a transitions as an event occurring at the end of a situation, thus causing the next situation to happen.

Transitions fall into 6 categories:

Continue: The qualitative state of the quantity remains the same.

Start-Up: The quantity was steady, and begins increasing.

Start-Down: The quantity was steady, and begins decreasing.

Pass: The quantity was increasing at ℓ_i (resp. between ℓ_i and ℓ_{i+1}) and is now increasing between ℓ_i and ℓ_{i+1} (resp. at ℓ_{i+1}). Similarly when decreasing.

Stop: The quantity was increasing between ℓ_i and ℓ_{i+1}, and becomes steady at ℓ_{i+1}. Similarly when decreasing.

Stop-New: The quantity was increasing between ℓ_i and ℓ_{i+1}, and becomes steady at the new landmark ℓ_* such that $\ell_i < \ell_* < \ell_{i+1}$. Similarly when decreasing.

To express the notion that q undergoes transition Start-Up at the end of situation s, we write:

(during s (start-up q))[2]

Here are the predicates for each of the 6 possible transition types.

[2] On reflection, this encoding is a little strange. (start-up q s) might have been a better choice. However, it makes the notation more uniform and, since the nature of fluents is not explicitated or exploited here, this bit of dubious syntax is of little consequence.

```
(DEFPRED (CONTINUE   ?Q - quantity))
(DEFPRED (START-UP   ?Q - quantity))
(DEFPRED (START-DOWN ?Q - quantity))
(DEFPRED (PASS       ?Q - quantity))
(DEFPRED (STOP       ?Q - quantity))
(DEFPRED (STOP-NEW   ?Q - quantity))
```

Naturally, not all transitions are possible in every situation. For example, when a quantity is steady, it can either remain steady, or start increasing, or start decreasing. No other alternative is possible.

```
(AXIOM P-TRANSITIONS-STD
  (FORALL ((REACHABLE ??S) - p-situation Q - quantity V - !!value)
    (IF (DURING S (STATE Q STD V))
        (OR (DURING S (CONTINUE   Q))
            (DURING S (START-UP   Q))
            (DURING S (START-DOWN Q))))))
```

In a point-situation, if a quantity is increasing at ℓ_i, it can only *pass*, i.e. become increasing between ℓ_i and ℓ_{i+1}. Similarly, if it is already increasing between ℓ_i and ℓ_{i+1}, it can only *continue* to do so.

```
(AXIOM P-TRANSITIONS-INC
  (FORALL ((REACHABLE ??S) - p-situation Q - !!quantity
           L1 L2 - !!landmark)
    (AND (IF (DURING S (STATE Q INC [L1]))    (DURING S (PASS     Q)))
         (IF (DURING S (STATE Q INC [L1 L2])) (DURING S (CONTINUE Q))))))
```

The remaining cases are easily obtained in like fashion from Table 10.1.

```
(AXIOM P-TRANSITIONS-DEC
  (FORALL ((REACHABLE ??S) - p-situation Q - !!quantity
           L1 L2 - !!landmark)
    (AND (IF (DURING S (STATE Q DEC [L1]))    (DURING S (PASS     Q)))
         (IF (DURING S (STATE Q DEC [L1 L2])) (DURING S (CONTINUE Q))))))
```

```
(AXIOM I-TRANSITIONS-STD
  (FORALL ((REACHABLE ??S) - i-situation Q - !!quantity V - !!value)
    (IF (DURING S (STATE Q STD V)) (DURING S (CONTINUE Q)))))

(AXIOM I-TRANSITIONS-INC
  (FORALL ((REACHABLE ??S) - i-situation Q - !!quantity V - !!value)
    (IF (DURING S (STATE Q INC V))
        (OR (DURING S (CONTINUE Q))
            (DURING S (PASS     Q))
            (DURING S (STOP     Q))
            (DURING S (STOP-NEW Q))))))

(AXIOM I-TRANSITIONS-DEC
  (FORALL ((REACHABLE ??S) - i-situation Q - !!quantity V - !!value)
    (IF (DURING S (STATE Q DEC V))
        (OR (DURING S (CONTINUE Q))
            (DURING S (PASS     Q))
            (DURING S (STOP     Q))
            (DURING S (STOP-NEW Q))))))
```

10.6 Qualitative State Changes

We are now in possession of the necessary formal apparatus required to successfully capture the information embodied in Table 10.1.

Continue. Thanks to the very convenient 'Continue' transition, we can summarize transitions P1, P5, P7, I1, I4 and I7 with just one axiom. This axiom simply states that, if a quantity undergoes the 'Continue' transition, then both its qualitative state and its set of landmarks remain unchanged in the next situation.

```
(AXIOM CONTINUE-NEXT
  (FORALL (S - situation Q - !!quantity D - !!direction V - !!value
           LEFT RIGHT - !!(LST landmark))
```

```
(IF (AND (DURING S (STATE Q D V))
         (DURING S (LANDMARKS Q ?LEFT ?RIGHT))
         (DURING S (CONTINUE Q)))
    (AND (DURING (NEXT S) (STATE Q D V))
         (DURING (NEXT S) (LANDMARKS Q ?LEFT ?RIGHT)))))
```

Start-Up (P2). When a quantity is steady at ℓ_i and undergoes the 'Start-Up' transition, it becomes increasing between ℓ_i and ℓ_{i+1} in the next situation. ℓ_i was the first element in its RIGHT set of landmarks; in the next situation, it is transferred to the end of the LEFT set of landmarks.

```
(AXIOM START-UP-NEXT
  (FORALL (S - p-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q STD [L1]))
             (DURING S (LANDMARKS Q LEFT [L1 L2 !?RIGHT]))
             (DURING S (START-UP Q)))
        (AND (DURING (NEXT S) (STATE Q INC [L1 L2]))
             (DURING (NEXT S) (LANDMARKS Q [!?LEFT L1] [L2 !?RIGHT]))))))
```

Start-Down (P3). When a quantity is steady at ℓ_i and undergoes the 'Start-Down' transition, it becomes decreasing between ℓ_{i-1} and ℓ_i in the next situation. ℓ_{i-1} is determined by looking up the last element in the LEFT set of landmarks. The set of landmarks remain unchanged.

```
(AXIOM START-DOWN-NEXT
  (FORALL (S - p-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q STD [L2]))
             (DURING S (LANDMARKS Q [!?LEFT L1] [L2 !?RIGHT]))
             (DURING S (START-DOWN Q)))
        (AND (DURING (NEXT S) (STATE Q DEC [L1 L2]))
             (DURING (NEXT S) (LANDMARKS Q [!?LEFT L1] [L2 !?RIGHT]))))))
```

Pass-Up (P4,I3). When a quantity is increasing at a landmark value ℓ_i in a point situation and undergoes the 'Pass-Up' transition, it becomes increasing between ℓ_i and ℓ_{i+1} in the next situation. ℓ_{i+1} is determined by looking up the element following ℓ_i in the RIGHT set of landmarks.

```
(AXIOM P-PASS-UP-NEXT
  (FORALL (S - p-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q INC [L1]))
             (DURING S (LANDMARKS Q ?LEFT [L1 L2 !?RIGHT]))
             (DURING S (PASS Q)))
        (AND (DURING (NEXT S) (STATE Q INC [L1 L2]))
             (DURING (NEXT S) (LANDMARKS Q [!?LEFT L1] [L2 !?RIGHT]))))))
```

Conversely, when a quantity is increasing between ℓ_i and ℓ_{i+1} during an interval situation, and undergoes the 'Pass-Up' transition, it becomes increasing at ℓ_{i+1} in the following point situation. The sets of landmarks remain unchanged.

```
(AXIOM I-PASS-UP-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q INC [L1 L2]))
             (DURING S (PASS Q)))
        (AND (DURING (NEXT S) (STATE Q INC [L2]))
             (IF (DURING S (LANDMARKS Q LEFT RIGHT))
                 (DURING (NEXT S) (LANDMARKS Q LEFT RIGHT)))))))
```

Pass-Down (P6,I6). When a quantity is decreasing at a landmark value ℓ_i in a point situation and undergoes the 'Pass-Down' transition, it becomes decreasing between ℓ_{i-1} and ℓ_i in the next situation. ℓ_i is the last element in the LEFT set of landmarks, and ℓ_{i-1} is determined by looking the element that precedes it. In the following situation ℓ_i is moved to the RIGHT set of landmarks.

```
(AXIOM P-PASS-DOWN-NEXT
  (FORALL (S - p-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q DEC [L2]))
             (DURING S (LANDMARKS Q [!?LEFT L1 L2] ?RIGHT))
             (DURING S (PASS Q)))
        (AND (DURING (NEXT S) (STATE Q DEC [L1 L2]))
             (DURING (NEXT S) (LANDMARKS Q [!?LEFT L1] [L2 !?RIGHT]))))))
```

Conversely, when a quantity is decreasing between ℓ_i and ℓ_{i+1} during an interval situation, and undergoes the 'Pass-Down' transition, it becomes decreasing at ℓ_i in the following point situation. The sets of landmarks remain unchanged.

```
(AXIOM I-PASS-DOWN-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(lst landmark))
    (IF (AND (DURING S (STATE Q DEC [L1 L2]))
             (DURING S (PASS Q)))
        (AND (DURING (NEXT S) (STATE Q DEC [L1]))
             (IF (DURING S (LANDMARKS Q LEFT RIGHT))
                 (DURING (NEXT S) (LANDMARKS Q LEFT RIGHT)))))))
```

Stop (I2,I5). When a quantity is increasing between ℓ_i and ℓ_{i+1} during an interval situation, and undergoes the 'Stop' transition, it becomes steady at ℓ_{i+1} in the next situation. The sets of landmarks remain unchanged.

```
(AXIOM STOP-UP-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q INC [L1 L2]))
             (DURING S (STOP Q)))
        (AND (DURING (NEXT S) (STATE Q STD [L2]))
             (IF (DURING S (LANDMARKS Q LEFT RIGHT))
                 (DURING (NEXT S) (LANDMARKS Q LEFT RIGHT)))))))
```

Conversely, when a quantity is decreasing between ℓ_i and ℓ_{i+1} during an interval situation, and undergoes the 'Stop' transition, it becomes steady at ℓ_i in the next situation. ℓ_i was the last element in the LEFT set of landmarks and is moved to the front of the RIGHT set of landmarks in the following situation.

```
(AXIOM STOP-DOWN-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q DEC [L1 L2]))
             (DURING S (STOP Q)))
        (AND (DURING (NEXT S) (STATE Q STD [L1]))
             (IF (DURING S (LANDMARKS Q [!?LEFT L1] [L2 !?RIGHT]))
                 (DURING (NEXT S) (LANDMARKS Q LEFT [L1 L2 !?RIGHT])))))))
```

Stop New (I8,I9). When a quantity is increasing between ℓ_i and ℓ_{i+1} during an interval situation, and undergoes the 'Stop-New' transition, it becomes steady at a new landmark value ℓ_* which we must insert between ℓ_i and ℓ_{i+1} in the corresponding set of landmarks.

```
(AXIOM STOP-NEW-UP-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q INC [L1 L2]))
             (DURING S (LANDMARKS Q LEFT RIGHT))
             (DURING S (STOP-NEW Q)))
        (EXISTS (L - landmark)
          (AND (DURING (NEXT S) (STATE Q STD [L]))
               (DURING (NEXT S) (LANDMARKS Q LEFT [L !?RIGHT])))))))
```

Similarly when decreasing.

```
(AXIOM STOP-NEW-DOWN-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
```

```
(IF (AND (DURING S (STATE Q DEC [L1 L2]))
         (DURING S (LANDMARKS Q LEFT RIGHT))
         (DURING S (STOP-NEW Q)))
    (EXISTS (L - landmark)
      (AND (DURING (NEXT S) (STATE Q STD [L]))
           (DURING (NEXT S) (LANDMARKS Q LEFT [L !?RIGHT]))))))
```

10.7 Constraints

Constraints were directly lifted from Kuipers' formulation. Here is a table summarizing briefly the notations and intended meanings:

(M+ q_1 q_2)	$q_1 = H(q_2), q_2 = G(q_1), \dot{H} > 0, \dot{G} > 0$
(M- q_1 q_2)	$q_1 = H(q_2), q_2 = G(q_1), \dot{H} < 0, \dot{G} < 0$
(DERIV q_1 q_2)	$q_2 = \dot{q}_1$
(ADD q_1 q_2 q_3)	$q_1 + q_2 = q_3$
(MULT q_1 q_2 q_3)	$q_1 \times q_2 = q_3$

A constraint is really just a proposition:

```
(DEFDUCKTYPE constraint prop)
(DUCLARE M+ M- DERIV - (FCN constraint [quantity quantity]))
(DUCLARE ADD MULT    - (FCN constraint [quantity quantity quantity]))
```

Each situation is associated with a set of constraints. What set of constraint is in effect in any given situation is problem specific and is determined by operating regions (ranges of values for physical parameters).

```
(DEFPRED (CONSTRAINTS ?CL - (SET_OF constraint)))
```

In order to establish what the current set of constraint is, we must be able to derive a proposition of the form:

```
(DURING situation (CONSTRAINTS set_of_constraints))
```

It is up to the problem statement to provide the necessary axioms to make deductions of this sort possible. The simplest possibility is when the set of constraints never changes, in which case an axiom of the following form suffices:

```
(FORALL (S - situation) (DURING S (CONSTRAINTS set_of_constraints)))
```

The axiomatization must capture the essential properties of these constraints. For example, if quantities q_1, q_2, and q_3 are related by an ADD constraint in situation s, then the (fluent) value of q_3 is equal to the sum of the (fluent) values of q_1 and q_2 throughout s.

```
(AXIOM ADD-CONSTRAINT
  (FORALL (S - !!situation Q1 Q2 Q3 - !!quantity
           CL - !!(SET_OF constraint))
    (IF (AND (DURING S (CONSTRAINTS CL))
             (ELT (ADD Q1 Q2 Q3) CL))
        (DURING S (= (+ (VALUE-OF Q1) (VALUE-OF Q2)) (VALUE-OF Q3))))))
```

10.8 Corresponding Values

We depart somewhat from Kuipers in our treatment of *corresponding values*. If $\mathcal{C}(q_1, \ldots, q_n)$ is a constraint on quantities q_1 through q_n, a n-tuple of corresponding values for this constraint is an element $\langle \ell^1, \ldots, \ell^n \rangle$ in the cartesian product of the corresponding sets of landmarks such that they satisfy the constraint.

In our view, the purpose of *corresponding values* is often to *qualitatively* rule out transitions which lead to situations which are *quantitatively* inconsistent with respect to certain constraints.

For example, suppose that quantities q_1, q_2, and q_3 are related by the constraint (ADD q_1 q_2 q_3) (meaning $q_1 + q_2 = q_3$) and that 1 is a landmark value for q_1, and 2 a landmark value for q_2, then the problem statement should indicate that 3 is a landmark value for q_3 and that $\langle 1,2,3 \rangle$ is a triple of corresponding values for the constraint.

In QSIM, corresponding values are treated specially, presumably to better serve the filtering needs of the simulation algorithm. A logical axiomatization such as ours has no such requirement. Therefore, we elect to represent corresponding values just like we do other constraints:

```
(DUCLARE CORRESPOND - (FCN constraint [(SET_OF (LRCD quantity number))]))
```

The usage is (CORRESPOND $\{[q_1 v_1] \ldots [q_n v_n]\}$), where q_i is a quantity and v_i a value for that quantity.

Note that the definition of corresponding values given above allows two n-tuples of corresponding values for the same constraint that differ only in one place. In order to allow more useful conclusions, we shall require a stronger assumption: namely that, given a constraint $C(q_1, \ldots, q_n)$ and a n-uple of corresponding values for this constraint either at least 2 quantities (from the set of n q_i's) are *not* at their corresponding values, or they are *all* at their corresponding values.

```
(AXIOM CORRESPOND-DEF
  (FORALL (S  - !!situation
           CL - !!(SET_OF constraint)
           L  - !!(SET_OF (LRCD quantity number)))
    (IF (AND (DURING S (CONSTRAINTS CL))
             (ELT (CORRESPOND L) CL))
        (OR (FORALL (Q - quantity V - number)
              (IF (ELT [Q V] L) (DURING S (= (VALUE-OF Q) V))))
            (EXISTS (Q1 Q2 - quantity V1 V2 - number)
              (AND (NOT (= Q1 Q2))
```

```
(ELT [Q1 V1] L)
(ELT [Q2 V2] L)
(NOT (DURING S (= (VALUE-OF Q1) V1)))
(NOT (DURING S (= (VALUE-OF Q2) V2))))))))
```

10.9 Situated Arithmetic

In order to reason about the values of quantities, we need to formalize our knowledge of arithmetic—more specifically, the arithmetic of fluents (see Section 10.4). The axioms will all be about propositions of the form (during s p), where p is some arithmetic formula. This is the reason I entitled the present section 'Situated Arithmetic'. In order to carry out useful proofs, we only require a few elementary axioms.

Equation Manipulation. A very useful axiom is the one that lets us move a term from one side of an equation to the other.

```
(AXIOM EQUATION-MANIPULATION
  (IFF (DURING ?S (= ?X (+ ?Y ?Z)))
       (DURING ?S (= (- ?X ?Z) ?Y))))
```

Inequation Manipulation. Similarly, here is the corresponding axiom that enables us to move a term from one side of an inequation to the other.

```
(AXIOM INEQUATION-MANIPULATION
  (IFF (DURING ?S (< ?X (+ ?Y ?Z)))
       (DURING ?S (< (- ?X ?Z) ?Y))))
```

Identity Equation. Zero (which, as you may recall from Section 10.4, should be interpreted as a notational abbreviation for the fluent which is identically 0 at every time point) is the identity element for addition.

```
(AXIOM PLUS-ZERO
  (DURING ?S (= ?X (+ 0 ?X))))
```

Other axioms can be added in the obvious manner, but, since they won't be required in this exposition, I will spare the reader and stop here.

10.10 Situated Inference Rules

In this section I present the more recent and experimental aspects of this reworking of my axiomatization.

When I first started working with my previous axiomatization, I soon realized that there were some manipulations which could not be performed easily. For example, 'Equality Substitution' is defined to be:

$$\frac{\begin{array}{c} A \Rightarrow e_1 = e_2 \\ A \Rightarrow p[e_1] \end{array}}{A \Rightarrow p[e_2]}$$

When $p[e_1]$ is really of the form (**during** s $q[e_1]$), it is often the case that instead of a first premise of the form $A \Rightarrow e_1 = e_2$, we would rather have something like $A \Rightarrow$ (**during** s $e_1 = e_2$).

It is possible to achieve this effect with the help of 2nd order schemas such as:

```
(AXIOM TIMELESS
  (IF (% ?P) (DURING ?S (% ?P))))

(AXIOM SITUATED-MODUS-PONENS
  (IF (AND (DURING ?S (IF ?P ?Q))
           (DURING ?S ?P))
      (DURING ?S ?Q)))
```

However, it is clearly painful to do so. In particular, we lose most of the benefits which the 'detaching' mechanism was supposed to grant us. The problem, of course, is that LOGICALC was developed for first order logic, while, here, we apply it to a reification of first order logic (the situation calculus).

One way to work around this limitation is to contrive the axiomatization so that problematic manipulations never have to be effected. In essence, this requires something akin to partially evaluating the situated axiomatization to make useful lemmas readily available (modulo some conditionalization). Also, we must eschew functions in favor of predicates to avoid the need for equality substitutions.

My previous work was developed according to the approach outlined above, and resulted in an axiomatization that was often unnatural and overly verbose.

In agreement with the principle that 'there should be a simple way to do simple things,' the new improved approach introduces a few additional inference rules especially designed for carrying out inferences in the situation calculus. Eventhough they are not as well integrated with the detaching mechanism as the regular rules, I believe it is a credit to LOGICALC's extensibility that I was able to write the code and graft it onto the existing system, all in a day's work.

For the purpose of solving the U-Tube problem, I needed 'Situated Equality Substitution', 'Situated Symmetry', and 'Situated Negation'. For good measure, I also added the 'Timeless' rule which states that, if a proposition is always true, then, in particular, it is true throughout a given situation.

10.10.1 Situated Negation

There was no need to resort to an inference rule for this one, a simple axiom sufficed.

```
(AXIOM SITUATED-NEGATION
```

```
(IF (DURING ?S (NOT ?P))
    (NOT (DURING ?S ?P))))
```

Note that the converse is not true because ?P may change truth value during ?S, thus failing to hold throughout ?S while still not succeeding in being false throughout it either.

10.10.2 Timeless Truths

If a proposition is always true, then, a fortiori, it is true throughout a given situation. Formulating this as an axiom is a little trickier than it was for negation because we can no longer pretend that the proposition is merely another funny term. In other words, we are not allowed to write:

```
(IF ?P (DURING ?S ?P))          (Γ)
```

because FOL does not permit free variables where a formula is expected. Instead we must write:

```
(IF (% ?P) (DURING ?S (% ?P)))
```

And perform explicit β-reductions.

Why was this not a problem with 'Situated Negation'? Because situated logic is reified in FOL. The 2nd argument of `during` is a proposition of situated logic; but, when we axiomatize this logic in FOL, it is represented by a term of FOL. It is merely a convenient abuse of notation which allows us to identify the two.

Of course, if all you care about is to get the job done, you *can* write (Γ), and it *will* work as expected; but the theoretical basis of this practical sleight of hand is a little shaky.

The better way to overcome these problems is to drop FOL and switch to HOL. Unfortunately, doing so would require a change of representation (e.g. adding type annotations) which would not be easily accommodated by our existing tools (in particular: DUCK).

The 'Timeless' rule can be depicted thus:

$$\frac{\begin{array}{l} A \Rightarrow p \\ A \Rightarrow (\text{IS situation } s) \end{array}}{A \Rightarrow (\text{during } s\ p)}$$

Here is the code implementing the inference rule, the plan generator, and the regressor:

code

```
(DEFFUNC TIMELESS-RULE - (LST sequent_t)
    (PREMS - (LST sequent_t) PARMS - (LST sexp))
  (AND (= (LENGTH PREMS) 2)
       (MATCH-COND (!_CONCLUSION (CADR PREMS))
         (\? (IS situation ?S)
             (LIST (MAKE sequent_t
                     (WEAKEST-ASET-OF-SEQUENTS PREMS)
                     '(DURING ,S ,(!_CONCLUSION (CAR PREMS))))))
         (T NIL))))

(DEFFUNC TIMELESS-PLANGEN - (LST <plan_t>)
    (SEQUENT - sequent_t
     PREMS   - (LST <premise_t>)
     PARMS   - (LST sexp))
  (IGNORE PREMS PARMS)
  (MATCH-COND (!_CONCLUSION SEQUENT)
    (\? (DURING ?S ?P)
        (LIST (MAKE <plan_t> '(DURING ,S ,P) 'Timeless
                (LIST (MAKE <goal_t> P)
                      (MAKE <goal_t> '(IS situation ,S)))
                NIL)))
    (T NIL)))

(DATAFUN REGRESS TIMELESS
  (DEFFUNC - (LST xsequent_t)
      (PROOF - proof_t STEPS - (LST sexp) ALIST - (LST bdg))
    (IGNORE PROOF)
    (MATCH-COND STEPS
      (\? (?P (IS ?TY ?S))
          (FOR (E IN (SAME-UNIFY TY 'situation ALIST))
```

```
            (SAVE (MAKE xsequent_t
                    (VARSUBST '(DURING ,S ,P) (!_LALIST E))
                    (!_LALIST E)))))
    (T NIL))))
```

10.10.3 Situated Equality Substitution

This is the interesting rule. If e_1 and e_2 are equal throughout situation s and $p[e_1]$ is true throughout s, then we are licensed to infer that $p[e_2]$ also holds throughout s. The rule can be depicted thus:

$$\frac{\begin{array}{c} A \Rightarrow (\text{during } s\ e_1 = e_2) \\ A \Rightarrow (\text{during } s\ p[e_1]) \end{array}}{A \Rightarrow (\text{during } s\ p[e_2])}$$

Very cleverly, 'Situated Equality' simply invokes regular Equality Substitution with $A \Rightarrow e_1 = e_2$ and $A \Rightarrow$ (during s $p[e_1]$) as arguments, and lets it do the bulk of the work.[3]

The plan generator cannot quite piggy-back on the plan generator for regular Equality Substitution in the same fashion. However the code is very similar: I simply copied the original code and modified it here and there to achieve the desired effect.

Similarly for the corresponding parser. Here, however, I quickly run into a limitation of the interface when I tried to use the plan generator. When it is invoked to substitute a subterm t, the plan generator (actually its *parser*) looks for a premise of the form:

$$(\text{during } s\ (= t\ ?\text{X}))$$

but it won't find it if the relevant detachable formula is of the form:

[3]With an additional unification step between the situation argument in the 1st premise and the situation argument in the 2nd premise.

(during s (= ?X t))

For regular FOL, the 'detaching' procedure automatically removes this obstacle with its knowledge of symmetric predicates (when it encounters an symmetric predicate, it tries the arguments both ways). Here again, the reification introduced by the situation calculus gets in the way. My temporary solution is not very elegant, but it works well enough in practice: first, the parser tries to detach:

(during s (= t ?X))

If that fails, it tries it the other way:

(during s (= ?X t))

This is inelegant because we have to have to go through the detaching interface (and therefore search the database) twice, instead of just once, taking care of trying symmetric predicates both ways on the fly. Unfortunately, there is no way to fix this problem short of building some knowledge of the situation calculus into the detaching procedure.

10.10.4 Situated Symmetry

In the previous description, I failed to indicate in what way I justified looking for a situated equality with the arguments switched. The answer is 'Situated Symmetry', which can be depicted thus:

$$\frac{A \Rightarrow (\text{during } s \; p(e_1, e_2))}{A \Rightarrow (\text{during } s \; p(e_2, e_1))} \quad p \text{ is symmetric}$$

When, for Situated Equality Substitution, I detached situated equality premises with the arguments reversed, I would then wrap an extra description around each one requiring an inference using 'Situated Symmetry' to reestablish the arguments in the proper order (see Chapters 6 and 7).

The implementation of Situated Symmetry is trivially derived from regular Symmetry.

10.10.5 Generalization

Generalization too was developed with FOL in mind. Extending it to work properly for FOL *and* the Situation Calculus simply requires treating the 2nd argument of `during` as propositions in their own right rather than as mere terms.

In practice, here is what this means: there is function which turns a proposition into a skeleton where connectives and predicates remain, but all other terms have been replaced with free variables. This function must be extended to treat the 2nd argument of a `during` as a proposition (and turn it into a skeleton) rather than a term (in which case it would have been turned into a free variable). Implementing this extension required adding 2 lines of code.

You may recall that in Chapter 8, Section 8.8.1 p295, we remarked that certain terms may disappear, as a result of the generalization mechanism, whose presence is required for the successful execution of later validations (e.g. for Equality Substitution). My (admitedly unsatisfactory) fix for this problem was to annotate those validations having such a requirement with a skeleton pattern: unifying the conclusion with this skeleton prior to executing the validation guarantees that enough structure is reestablished by instantiation for the validation to operate successfully.

We have exactly the same problem with 'Situated Equality', and it can be fixed in the same fashion. Implementing this fix required adding 4 lines of code to the procedure responsible for extracting the skeleton annotation.

10.11 The U-Tube Problem

The U-Tube problem was introduced earlier in this chapter. The setup consists of two tanks, A and B, connected by a thin tube at the bottom (see Figure 10.1). Initially, the water-level is higher in tank A than in tank B. We should like to prove that the system reaches an equilibrium, with the water-levels settling at some intermediate height.

In the interest of simplicity, we will prove a less ambitious goal instead:

- Firstly, we shall not try to show that the respective water levels in each tank settle at the same height (in fact, they might not if they are subjected to different (constant) atmospheric pressures). Instead, we shall simply show that the water level in tank A becomes 'Steady'.

- Secondly, we shall make the simplifying assumption that the water-level `HA` in tank A does not undergo the 'Continue' transition at the end of the interval situation which follows the initial situation point situation `(AT T0)`.

  If `HA` *continues* at the end of the second situation, then all other quantities must also *continue*. Naturally, a comprehensive axiomatization of the Situation Calculus must exclude such a possibility.[4] Typically, this might be done with an axiom stating that either all quantities *continue* for ever (i.e. we enter an interval situation whose far end is at $+\infty$), or there exists one quantity which does not *continue*.

  Ruling out the 'Continue' transition for `HA` would involve proving that, if it were to occur, *all* quantities would similarly continue, thus resulting in a contradiction with the axiom enunciated above. It is possible to take this approach, and

[4] Because otherwise there would be envisionments in which the world grinds down to a halt, producing sequences of situations during which nothing changes. It seems only right to require that *something* must change from one situation to the next.

we shall do so for the second problem. For the moment, however, we shall make the above simplifying assumption.

10.11.1 Axiomatization

Here is the short axiomatization which suffices to derive the intended conclusion:[5]

```
(DUCLARE TO - timepoint)
(AXIOM INITIAL-SIT (INITIAL (AT TO)))

(DUCLARE PA PB HA HB PAB - quantity)
(DUCLARE PAO PBO HAO HBO PABO +INF - landmark)

(AXIOM HA-INITIAL-LANDMARKS
  (DURING (AT TO) (LANDMARKS HA [0 HAO] [])))

(AXIOM HA-INITIAL-DEC
  (DURING (AT TO) (STATE HA DEC [HAO])))

(AXIOM HB-INITIAL-LANDMARKS
  (DURING (AT TO) (LANDMARKS HB [0] [HBO +INF])))

(AXIOM HB-INITIAL-INC
  (DURING (AT TO) (STATE HB INC [HBO])))

(AXIOM PA-INITIAL-LANDMARKS
  (DURING (AT TO) (LANDMARKS PA [0 PAO] [])))

(AXIOM PA-INITIAL-DEC
  (DURING (AT TO) (STATE PA DEC [PAO])))

(AXIOM PB-INITIAL-LANDMARKS
  (DURING (AT TO) (LANDMARKS PB [0] [PBO +INF])))

(AXIOM PB-INITIAL-INC
  (DURING (AT TO) (STATE PB INC [PBO])))
```

[5] In fact, the axioms about HB and PA are not needed in the proof.

```
(AXIOM PAB-INITIAL-LANDMARKS
  (DURING (AT T0) (LANDMARKS PAB [0 PAB0] [])))

(AXIOM PAB-INITIAL-DEC
  (DURING (AT T0) (STATE PAB DEC [PAB0])))

(AXIOM THE-CONSTRAINTS
  (FORALL (S - situation)
    (DURING S (CONSTRAINTS {(ADD PAB PB PA)
                            (CORRESPOND {[HA 0] [PA 0]})
                            (CORRESPOND {[HA HA0] [PA PA0]})
                            (CORRESPOND {[HB 0] [PB 0]})
                            (CORRESPOND {[HB HB0] [PB PB0]})}))))
```

Axiom THE-CONSTRAINTS is problem specific and indicates that the set of constraints remains the same in all situations. From the specified constraints, only the first two are needed in the proof.

The target theorem is:

```
(IF (NOT (DURING (NEXT (AT T0)) (CONTINUE HA)))
    (DURING (NEXT (NEXT (AT T0))) (STATE HA STD [?HA1])))
```

10.11.2 Proof Outline

In the initial situation, the water level HA in tank A must 'Pass Down' because it is decreasing at the landmark value HA0 in the point situation (AT T0) (Transition P6).

Therefore, in the second (interval) situation, HA is decreasing between 0 and HA0. By axiom I-TRANSITIONS-DEC, HA may now either (1) Continue, (2) Pass (Down), (3) Stop (at 0), (4) Stop at a New intermediate value.

The 'Continue' transition is ruled out by our simplifying assumption.

Pass (Down) and Stop can be ruled out by contradiction: If either of these transitions occurs at the end of the second situation, then HA must be 0 in the third (point) situation. By corresponding values, we infer that PA must also be 0, and, since PAB = PA − PB, we conclude that PAB < 0. However, in the initial situation PAB > 0, therefore (by continuity) it must be > 0 in the 2nd situation and ≥ 0 in the 3rd.

Thus, we have ruled out every transition but 'Stop New', which yields the desired result.

10.11.3 Proof Summary

I include here the proof summary generated by LOGICALC (see Chapter 9). The derivation of this proof required 55 invocations of plan generators. It involves 71 axioms and assumptions. A number of these axioms correspond to logic programs. They are considered low-level and uninteresting (because they implement obvious deductions), and are not explicitly mentioned in the proof summary. The proof tree contains 130 distinct inference nodes.[6] 26 of these nodes occur more than once in the tree (which, as a consequence, is not a tree but a DAG)—the corresponding deductions were derived once, but used several times.

The default heuristics produce a summary 62 lines long. I used the editor's commands to change the visibility of 7 lines, and FLOATed 1 line downward and 4 lines upward to improve readability. This resulted in the following 33 lines long summary. I offer a line-by-line commentary after the proof itself.

For brevity, the lists of abbreviations and axioms used in the proof are not included here.

[6] Conclusions obtained by backward-chaining through logic programs only contribute 1 node (rather than 1 per backward-chaining step).

1 ——— From `P-TRANSITIONS-DEC` and `HA-INITIAL-DEC` Conclude:
```
HO ⊢
(DURING (AT TO) (PASS HA))
```

2 ——— From `P-PASS-DOWN-NEXT HA-INITIAL-DEC HA-INITIAL-LANDMARKS 1` Obviously:
```
(DURING (NEXT (AT TO)) (STATE HA DEC [0 HAO]))
```

3 ——— From `P-PASS-DOWN-NEXT HA-INITIAL-DEC HA-INITIAL-LANDMARKS 1` Obviously:
```
(DURING (NEXT (AT TO)) (LANDMARKS HA [0] [HAO]))
```

4 ——— From `ADD-CONSTRAINT` and `THE-CONSTRAINTS` Conclude:
```
(DURING (NEXT (NEXT (AT TO)))
        (= (+ (VALUE-OF PAB) (VALUE-OF PB)) (VALUE-OF PA)))
```

5 ——— From `EQUATION-MANIPULATION` and `4` Conclude:
```
(DURING (NEXT (NEXT (AT TO)))
        (= (- (VALUE-OF PA) (VALUE-OF PB)) (VALUE-OF PAB)))
```

6 ——— From `P-TRANSITIONS-INC` and `PB-INITIAL-INC` Conclude:
```
(DURING (AT TO) (PASS PB))
```

7 ——— From `P-PASS-UP-NEXT PB-INITIAL-INC PB-INITIAL-LANDMARKS 6` Obviously:
```
(DURING (NEXT (AT TO)) (STATE PB INC [PBO +INF]))
```

8 ——— From `AT-VALUE` and `PB-INITIAL-INC` Conclude:
```
(DURING (AT TO) (= (VALUE-OF PB) PBO))
```

9 ——— From major premise `LANDMARKS-ORDER` Obviously:
```
(DURING (AT TO) (< 0 PBO))
```

10 ——— By situated equality substitution of `8` in `9` Conclude:
```
(DURING (AT TO) (< 0 (VALUE-OF PB)))
```

11 ——— From `NEXT-OF-INC` and `PB-INITIAL-INC 10` Conclude:
```
(DURING (NEXT (AT TO)) (< 0 (VALUE-OF PB)))
```

12 ——— From `NEXT-OF-INC` and `7 11` Conclude:
```
(DURING (NEXT (NEXT (AT TO))) (< 0 (VALUE-OF PB)))
```

Assume `H1 ≡ !:NOT421`
```
!:NOT421
      (NOT (DURING (NEXT (AT TO)) (CONTINUE HA)))
```
Assume `H2 ≡ !:DURING449 + H1`
```
!:DURING449
      (DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF HA) 0))
```

13 ——— From major premise `!:DURING449 THE-CONSTRAINTS CORRESPOND-DEF` Obviously:

(DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF PA) 0))

14 ——— By situated equality substitution of 13 in 12 Conclude:
(DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF PA) (VALUE-OF PB)))

15 ——— By situated equality substitution of PLUS-ZERO in 14 Conclude:
(DURING (NEXT (NEXT (AT TO)))
(< (VALUE-OF PA) (+ 0 (VALUE-OF PB))))

16 ——— From INEQUATION-MANIPULATION and 15 Conclude:
(DURING (NEXT (NEXT (AT TO)))
(< (- (VALUE-OF PA) (VALUE-OF PB)) 0))

17 ——— By situated equality substitution of 5 in 16 Conclude:
(DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF PAB) 0))

18 ——— From 17 Discharging !:DURING449 Conclude:
H1 ≡ !:NOT421 ⊢
(IF (DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF HA) 0))
(DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF PAB) 0)))

19 ——— From P-TRANSITIONS-DEC and PAB-INITIAL-DEC Conclude:
HO ⊢
(DURING (AT TO) (PASS PAB))

20 ——— From P-PASS-DOWN-NEXT PAB-INITIAL-DEC PAB-INITIAL-LANDMARKS 19 Obviously:
(DURING (NEXT (AT TO)) (STATE PAB DEC [O PABO]))

21 ——— From NEXT-OF-BETWEEN 20 Obviously:
(DURING (NEXT (NEXT (AT TO))) (NOT (< (VALUE-OF PAB) 0)))

22 ——— From SITUATED-NEGATION and 21 Conclude:
(NOT (DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF PAB) 0)))

Assuming H1 ≡ !:NOT421
Assume H6 ≡ !:DURING613 + H1
!:DURING613
(DURING (NEXT (AT TO)) (PASS HA))

23 ——— From I-PASS-DOWN-NEXT 2 !:DURING613 Obviously:
(DURING (NEXT (NEXT (AT TO))) (STATE HA DEC [O]))

24 ——— From AT-VALUE and 23 Conclude:
(DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF HA) 0))

25 ——— From 18 and 24 Conclude:
(DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF PAB) 0))

```
   26 ——— By contradiction 25 22 Conclude:
     H1 ≡ !:NOT421 ⊢
     (NOT (DURING (NEXT (AT TO)) (PASS HA)))

       Assume H7 ≡ !:DURING623 + H1
         !:DURING623
                (DURING (NEXT (AT TO)) (STOP HA))

       27 ——— From STOP-DOWN-NEXT 2 !:DURING623 Obviously:
         (DURING (NEXT (NEXT (AT TO))) (STATE HA STD [0]))

       28 ——— From AT-VALUE and 27 Conclude:
         (DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF HA) 0))

       29 ——— From 18 and 28 Conclude:
         (DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF PAB) 0))

   30 ——— By contradiction 29 22 Conclude:
     H1 ≡ !:NOT421 ⊢
     (NOT (DURING (NEXT (AT TO)) (STOP HA)))

   31 ——— From I-TRANSITIONS-DEC 2 and !:NOT421 26 30 Conclude:
     (DURING (NEXT (AT TO)) (STOP-NEW HA))

   32 ——— From STOP-NEW-DOWN-NEXT 2 3 31 Obviously:
     (DURING (NEXT (NEXT (AT TO))) (STATE HA STD [!.L]))

33 ——— From 32 Discharging !:NOT421 Conclude:
  HO ⊢
  (IF (NOT (DURING (NEXT (AT TO)) (CONTINUE HA)))
      (DURING (NEXT (NEXT (AT TO))) (STATE HA STD [!.L])))
```

Here is the promised commentary:

[1] Since HA is initially decreasing at HA0, it must Pass.

[2] and therefore becomes decreasing between 0 and HA0.

[4] From the definition of ADD and the constraint (ADD PAB PB PA), infer $\underline{PAB} + \underline{PB} = \underline{PA}$.[7]

[5] from [4], $\underline{PAB} = \underline{PA} - \underline{PB}$.

[7] I will use the notation $\underline{Q}$ to represent `(value-of Q)`.

[6] Since PB is initially increasing at PB0, it must Pass.

[7] and consequently becomes increasing between PB0 and $+\infty$.

[8] Initially $\underline{PB} = PB0$.

[9] And we know that $PB0 > 0$.

[10] Therefore, initially, $\underline{PB} > 0$.

[11] Since, PB is increasing. $\underline{PB} > 0$ must also be true in the following situation.

[12] and, by continuity, it must still be true in the point situation after that (the 3rd situation).

[13] Assuming that $\underline{HA} = 0$ in the 3rd situation, we conclude that $\underline{PA} = 0$ as well, since these are corresponding values for these two quantities.

[14] By equality substitution into [12], i.e. $\underline{PB} > 0$, we conclude $\underline{PB} > \underline{PA}$.

[17] Therefore, $\underline{PAB} < 0$ (see [5]).

[18] Discharging the assumption, we conclude that **if** $\underline{HA} = 0$ in the 3rd situation, **then** $\underline{PAB} < 0$ as well.

[19] Since PAB is initially decreasing at PAB0, it must Pass.

[20] and becomes decreasing between 0 and PAB0 in the 2nd situation.

[21,22] Since $0 \leq \underline{PAB} \leq PAB0$ in the 2nd situation, it cannot suddenly jump below 0 in the 3rd situation. Therefore it cannot be the case that $\underline{PAB} < 0$ in the 3rd situation.

[23] Suppose that HA Passes in the 2nd situation. Then it must become decreasing at 0 in the 3rd situation.

[24] Therefore $\underline{HA} = 0$.

[25] and, from [18], it follows that $\underline{\text{PAB}} < 0$.

[26] This is in contradiction with [22], therefore HA cannot Pass in situation 2.

[27–30] Similarly for Stop.

[31] Therefore, assuming that HA does not Continue in situation 2, we have ruled out all but the Stop-New transition.

[32] Thus, HA must become Steady at a new landmark value.

[33] Finally, discharging the assumption, we conclude that **if** HA does not Continue in situation 2, then it must become Steady at a new landmark value in situation 3.

The inference on line [13] is rather obvious to the reader, but its derivation is quite lengthy. Axiom `correspond-def` states that, for all quantities for which there exists a constraint of corresponding values, either they all are at their corresponding values, or at least two of them are not.

Here, the corresponding values are $\underline{\text{HA}} = 0$, $\underline{\text{PA}} = 0$. We have to show that we can't pick two quantities (one of HA or PA) such that they are not at their corresponding values (i.e. 0). This is obvious, since HA *is* at the "corresponding" value by hypothesis, there remains only PA to chose from. However, the proof must go through every way of picking 2 quantities from the set of quantities involved in the constraint, and show that at least one of them is at the "corresponding" value. In the present case, there are only two ways of picking two quantities from a set of 2, yet the proofs are lengthy and took up 24 lines in the default summary (prior to editing).

10.12 The Case of The Leaking Tank

In this section we shall consider the problem of a tank with a hole in its side, and which is being filled at a constant rate by a steady in-flow of water (see Figure 10.2). Initially, the tank is empty, and it is known that the leaking capacity of the hole is greater than the in-flow. We wish to prove that the water-level becomes steady at the hole. As announced earlier, we no longer want to make the simplifying assumption

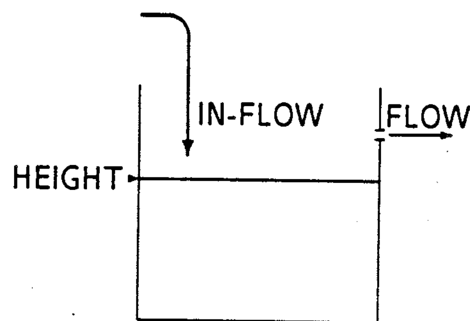

Figure 10.2: Leaking Tank

ruling out the Continue transition as we did in the case of the U-Tube. Instead, we will add an axiom stating that either there is some quantity which does not Continue, or the situation goes on forever (i.e. we never get to the next situation).

10.12.1 Reachable Situations

The reader will have noticed that the axiomatization used earlier contained occurrences of formulae of the form `(reachable` *sit*`)`, although I didn't specify at the time precisely what this meant.

A situation is ***reachable*** (given a certain set of assumptions) if it can be attained from the initial situation through a sequence of permissible transitions. In the earlier proof

I simply considered all situations of the form (**next**^{n} s_0) to be reachable, where s_0 was the initial situation.

Now, we shall be more careful and spell out exactly in what circumstances a situation is reachable.

Firstly, what I mean when I say that the next situation is not reachable is that the current situation goes on forever. Obviously this is not possible for a point-situation. Therefore, the interval-situation following a reachable point-situation is always reachable.

```
(AXIOM P-REACHABLE
  (FORALL (S - p-situation)
    (IF (REACHABLE S) (REACHABLE (NEXT S)))))
```

On the other hand, a reachable interval-situation may well extend indefinitely into the future. This happens precisely when the system achieves a permanent regime, i.e. all quantities Continue.

```
(AXIOM I-REACHABLE
  (FORALL (S - i-situation L - !!(SET_OF quantity))
    (IF (AND (REACHABLE S)
             (DURING S (QUANTITIES L)))
        (IFF (REACHABLE (NEXT S))
             (EXISTS ((ELT ??Q L))
               (NOT (DURING S (CONTINUE Q))))))))
```

Notice that, in order to use this axiom, we need to be able to identify the set of quantities in the given situation, and the problem statement must provide the means for this.[8]

```
(DEFPRED (QUANTITIES ?L - (SET_OF quantity)))
```

[8]It is conceivable that, in some problems, the set of quantities may not be the same in all situations.

The notion of a (non) reachable situation allows us to capture one aspect of time, namely the possibility that certain intervals may extend indefinitely into the future, without the need to introduce heavy formal machinery to reason about time, and also ties this neatly with the notion of a system achieving a stable regime.

10.12.2 Constants And Fluents

Similarly, although we will not develop here a complete axiomatic treatment of fluents, we can easily manage to capture one particularly important notion: namely, that of a constant.

```
(DEFPRED (CONSTANT ?X - number))
```

Constants, when combined through the usual arithmetic operations, yield constants:

```
(AXIOM CONSTANT-DEF
  (FORALL (S - !!situation X Y - !!number)
    (IF (AND (DURING S (CONSTANT X))
             (DURING S (CONSTANT Y)))
        (AND (DURING S (CONSTANT (+ X Y)))
             (DURING S (CONSTANT (- X Y)))
             (DURING S (CONSTANT (* X Y)))
             (DURING S (CONSTANT (/ X Y)))))))
```

Also, if ℓ is a constant, then it is equivalent to say that a quantity is steady at ℓ or that its value is ℓ throughout the situation.

```
(AXIOM I-STD
  (FORALL (S - i-situation Q - !!quantity L - !!landmark)
    (IF (DURING S (CONSTANT L))
        (IFF (DURING S (STATE Q STD [L]))
             (DURING S (= (VALUE-OF Q) L))))))
```

The reason for requiring that ℓ be constant is that, even though the fluent value of a quantity can be equal to another (varying) fluent, the quantity can only be said to be steady at a constant.

10.12.3 Derivatives

Obviously, the sign of the derivative of the value of a quantity determines how the latter varies, i.e. whether it is steady, increasing, or decreasing.

```
(AXIOM DERIV-DEF
  (FORALL (S     - !!situation
           F DF - !!quantity
           CL    - !!(SET_OF constraint)
           D     - !!direction
           V     - !!value)
    (IF (AND (DURING S (CONSTRAINTS CL))
             (ELT (DERIV F DF) CL)
             (DURING S (STATE F D V)))
        (AND (IF (DURING S (< (VALUE-OF DF) 0)) (= D DEC))
             (IF (DURING S (< 0 (VALUE-OF DF))) (= D INC))
             (IF (DURING S (= 0 (VALUE-OF DF))) (= D STD))))))
```

Naturally, we must make it plain that the 3 directions of variation are mutually exclusive. It suffices to say that they are distinct.

```
(ASSERTION-GROUP DIRECTIONS
  (NOT (= DEC INC))
  (NOT (= DEC STD))
  (NOT (= INC STD)))
```

Finally, we need to state the following very important property: if the current situation extends indefinitely into the future and the derivative of a quantity is greater than some positive constant throughout the situation, then there is no constant such

that the value of the quantity does not eventually exceed it. We need the positive constant to avoid asymptotic limits.

```
(AXIOM POSITIVE-DERIV
  (FORALL (S - i-situation F DF - !!quantity CL - !!(SET_OF constraint))
    (IF (AND (REACHABLE S)
             (NOT (REACHABLE (NEXT S)))
             (DURING S (CONSTRAINTS CL))
             (ELT (DERIV F DF) CL)
             (EXISTS ((DURING S (CONSTANT ??C)))
               (AND (DURING S (< 0 C))
                    (DURING S (< C (VALUE-OF DF))))))
        (NOT (EXISTS ((DURING S (CONSTANT ??L)))
               (DURING S (< (VALUE-OF F) L)))))))
```

10.12.4 Arithmetic Revisited

For this problem, arithmetic needed to be fleshed out a little more.

```
(AXIOM INEQUATION-MANIPULATION
  (AND (IFF (DURING ?S (< ?X (+ ?Y ?Z)))
            (DURING ?S (< (- ?X ?Z) ?Y)))
       (IFF (DURING ?S (< (+ ?X ?Z) ?Y))
            (DURING ?S (< ?X (- ?Y ?Z))))))

(AXIOM INEQUATION-MANIPS
  (IF (DURING ?S (< 0 ?Z))
      (AND (IFF (DURING ?S (< ?X (/ ?Y ?Z)))
                (DURING ?S (< (* ?X ?Z) ?Y)))
           (IFF (DURING ?S (< (/ ?X ?Z) ?Y))
                (DURING ?S (< ?X (* ?Y ?Z)))))))

(AXIOM TIMES-ZERO
  (DURING ?S (= 0 (* 0 ?X))))
```

The axiom of density says that for any two numbers $x < y$, you can pick a number z such that $x < z < y$. This is true of fluents, but I wanted to be able to say that in

the case where x and y are constants, then you can pick z constant too. So, I only stated the axiom for constants.

```
(AXIOM DENSITY
  (FORALL (S - !!situation)
    (FORALL ((DURING S (CONSTANT ??(X Y))) - !!number)
      (IF (DURING S (< X Y))
          (EXISTS ((DURING S (CONSTANT ??Z)) - !!number)
            (AND (DURING S (< X Z))
                 (DURING S (< Z Y))))))))
```

10.12.5 The Problem Statement

```
(DUCLARE TO - timepoint)
(AXIOM INITIAL-SIT (INITIAL (AT TO)))

(DUCLARE HEIGHT   SPEED FLOW - quantity
         IN-FLOW  SURFACE    - number
         OUT-FLOW HOLE       - landmark)
```

The set of quantities remains the same throughout the experiment.

```
(AXIOM THE-QUANTITIES
  (DURING ?S (QUANTITIES {HEIGHT SPEED FLOW})))
```

The out-flow capacity of the hole is known to be greater than the in-flow of water.

```
(AXIOM IN-OUT (DURING ?S (< IN-FLOW OUT-FLOW)))
```

Initially, the tank is empty and the water-level is increasing at 0.

```
(AXIOM HEIGHT-INITIAL-LANDMARKS
  (DURING (AT TO) (LANDMARKS HEIGHT [] [0 HOLE +INF])))

(AXIOM HEIGHT-INITIAL-INC
  (DURING (AT TO) (STATE HEIGHT INC [0])))
```

During this experiment, there is a constraint that changes: namely that which determines the flow of water through the hole. When the water level is below the hole, no water goes through the hole, whereas when the level is above the hole, the flow rate through the hole is constant and equal to `out-flow`.

We have a choice of whether to state this as a constraint, in which case the set of constraints will be different below and above the hole, or as couple of separate axioms. It is more convenient to use the second approach.

```
(AXIOM FLOW-BELOW-HOLE
  (IF (DURING ?S (< (VALUE-OF HEIGHT) HOLE))
      (DURING ?S (= (VALUE-OF FLOW) 0))))

(AXIOM FLOW-ABOVE-HOLE
  (IF (DURING ?S (< HOLE (VALUE-OF HEIGHT)))
      (DURING ?S (= (VALUE-OF FLOW) OUT-FLOW))))
```

Notice that this axiomatization cleverly avoids saying anything about what happens *at* the hole.

Initially, there is no water leaking through the hole, and the corresponding flow is steady at 0.

```
(AXIOM FLOW-INITIAL-LANDMARKS
  (DURING (AT T0) (LANDMARKS FLOW [] [0 OUT-FLOW])))

(AXIOM FLOW-INITIAL-STD
  (DURING (AT T0) (STATE FLOW STD [0])))
```

We also have the choice of whether to express the relationship among the various quantities as a set of qualitative equations, or as an arithmetic equation between fluents. For convenience again we choose the latter.

The speed of the water-level is related to the difference between the in-flow and the flow out of hole in inverse proportion to the horizontal area of the tank, which we take to be constant from top to bottom and equal to `surface`.

```
(DUCLARE SURFACE - number)

(AXIOM SURFACE-POSITIVE (DURING ?S (< 0 SURFACE)))

(AXIOM IN-FLOW-POSITIVE (DURING ?S (< 0 IN-FLOW)))

(AXIOM FLOW-EQUATION
  (DURING ?S (= (VALUE-OF SPEED) (/ (- IN-FLOW (VALUE-OF FLOW)) SURFACE))))
```

The only significant constraint which remains to be expressed is that `speed` is the derivative of `height`. This set of constraints remains the same throughout the experiment.

```
(AXIOM THE-CONSTRAINTS
  (DURING ?S (CONSTRAINTS {(DERIV HEIGHT SPEED)})))
```

Finally, we must identify the constants.

```
(ASSERTION-GROUP THE-CONSTANTS
  (DURING ?S (CONSTANT 0))
  (DURING ?S (CONSTANT IN-FLOW))
  (DURING ?S (CONSTANT OUT-FLOW))
  (DURING ?S (CONSTANT SURFACE))
  (DURING ?S (CONSTANT HOLE)))
```

The target theorem is:

```
(DURING (NEXT (NEXT (AT T0))) (STATE HEIGHT STD [HOLE]))
```

10.12.6 Proof Outline

We must show that `height` Stops in the second situation. By axiom `i-transitions-inc`, the only possiblities are Continue, Pass, Stop, and Stop-New. We shall rule out every transition but Stop by contradiction, i.e. by assuming that the transition takes place and showing that this leads to a contradiction.

If `height` Continues, then it remains below the hole, therefore the `speed` remains constant and positive (Continues), and the flow out of the hole remains 0 (Continues). Since all quantities continue, situation 2 must extend indefinitely into the future and situation 3 is unreachable.

However, since `speed` is constant and positive and is the derivative of `height`, `height` must eventually exceed the altitude of the hole. This is in contradiction with the conclusion that `height` will remain forever below the hole.

If `height` Passes, then it is increasing at the hole in point-situation 3 and must Pass again; which means it must be increasing between the hole and $+\infty$ in interval situation 4. But since `height` is above the hole, its derivative, which is proportional to `in-flow` $-$ `out-flow` must be negative. Therefore it must be decreasing too. Contradiction.

Finally, if `height` Stops-New, it becomes steady at an altitude below the hole. However, its derivative is still positive (see above), therefore it should be increasing. Contradiction again.

10.12.7 Proof Summary

Here is the proof summary generated by LOGICALC. The derivation of this proof required 65 invocations of plan generators (many tedious inferences are in fact required

to derive line 13 and to manipulate arithmetic formulae and equations). It involves 68 axioms and assumptions. The proof tree containt 163 distinct inference nodes, 46 of which occur more than once in this graph.[9]

The default heuristics produce a summary 77 lines long. By editing the visibility of 11 lines and FLOATing another upwards, I impoved slightly on appearance and legibility.

For brevity, the list of axioms used in the proof is not included here.

ABBREVIATIONS

```
!.Q[2](?S ?L)
            = (SK Q 2 ?S ?L)                                  -- I-REACHABLE
!.Q[2]((NEXT (AT TO)) {HEIGHT SPEED FLOW})
            = (SK Q 2 (NEXT (AT TO)) {HEIGHT SPEED FLOW})   -- I-REACHABLE
!.Z[9](?Y ?X ?S)
            = (SK Z 9 ?Y ?X ?S)                               -- DENSITY
!.Z[9]((VALUE-OF SPEED) O (NEXT (AT TO)))
            = (SK Z 9 (VALUE-OF SPEED) O (NEXT (AT TO)))      -- DENSITY
!.L[3](?RIGHT ?LEFT ?Q ?S)
            = (SK L 3 ?RIGHT ?LEFT ?Q ?S)                     -- STOP-NEW-UP-NEXT
!.L[3]([HOLE +INF] [O] HEIGHT (NEXT (AT TO)))
            = (SK L 3 [HOLE +INF] [O] HEIGHT (NEXT (AT TO)))  -- STOP-NEW-UP-NEXT
```

PROOF

1 ——— From INITIAL-SIT Obviously:
HO ⊢
(REACHABLE (AT TO))

2 ——— From P-TRANSITIONS-INC 1 and HEIGHT-INITIAL-INC Conclude:
(DURING (AT TO) (PASS HEIGHT))

3 ——— From P-PASS-UP-NEXT HEIGHT-INITIAL-INC HEIGHT-INITIAL-LANDMARKS 2 Obviously:
(DURING (NEXT (AT TO)) (STATE HEIGHT INC [O HOLE]))

4 ——— From 1 Obviously:
(REACHABLE (NEXT (AT TO)))

[9]See section 10.11.3 for for relevant commentary.

```
5 ——— From BETWEEN-VALUES 3 Obviously:
  (DURING (NEXT (AT TO)) (< (VALUE-OF HEIGHT) HOLE))

    Assume H1 ≡ !:DURING510
      !:DURING510
           (DURING (NEXT (AT TO)) (CONTINUE HEIGHT))
        Assume H2 ≡ !:%528 + H1
          !:%528 (% {HEIGHT SPEED FLOW} !.Q)

        6 ——— From !:%528 Obviously:
          (OR (= !.Q HEIGHT) (= !.Q SPEED) (= !.Q FLOW))

            Assume H4 ≡ !:=531 + H2
              !:=531 (= !.Q HEIGHT)

            7 ——— By equality substitution of !:=531 in !:DURING510 Conclude:
              (DURING (NEXT (AT TO)) (CONTINUE !.Q))

        8 ——— From 7 Discharging !:=531 Conclude:
          H2 ≡ !:%528 H1 ⊢
          (IF (= ?X HEIGHT) (DURING (NEXT (AT TO)) (CONTINUE ?X)))

9 ——— From FLOW-BELOW-HOLE and 5 Conclude:
  HO ⊢
  (DURING (NEXT (AT TO)) (= (VALUE-OF FLOW) 0))

10 ——— From CONSTANT-DEF THE-CONSTANTS-2 THE-CONSTANTS-1 Obviously:
  (DURING ?X.1 (CONSTANT (- IN-FLOW 0)))

11 ——— From CONSTANT-DEF 10 THE-CONSTANTS-4 Obviously:
  (DURING ?X.2 (CONSTANT (/ (- IN-FLOW 0) SURFACE)))

12 ——— By situated equality substitution of 9 in 11 Conclude:
  (DURING (NEXT (AT TO)) (CONSTANT (/ (- IN-FLOW (VALUE-OF FLOW)) SURFACE)))

13 ——— From I-STD 12 and FLOW-EQUATION Conclude:
  (DURING (NEXT (AT TO))
          (STATE SPEED STD [(/ (- IN-FLOW (VALUE-OF FLOW)) SURFACE)]))

14 ——— From I-TRANSITIONS-STD and 4 13 Conclude:
  (DURING (NEXT (AT TO)) (CONTINUE SPEED))

    Assuming H1 ≡ !:DURING510
         Assuming H2 ≡ !:%528 + H1
              Assume H5 ≡ !:=533 !:NOT534 + H2
                !:=533 (= !.Q SPEED)
                !:NOT534
                       (NOT (= !.Q HEIGHT))
```

15 ——— By equality substitution of `!:=533` in **14** Conclude:
`(DURING (NEXT (AT T0)) (CONTINUE !.Q))`

16 ——— From **15** Discharging `!:=533 !:NOT534` Conclude:
`H2 ≡ !:%528 H1 ⊢`
```
(IF (AND (= ?Z SPEED) (NOT (= ?Y ?X)))
    (DURING (NEXT (AT T0)) (CONTINUE ?Z)))
```

17 ——— From `I-STD THE-CONSTANTS-1` and **9** Conclude:
`H0 ⊢`
`(DURING (NEXT (AT T0)) (STATE FLOW STD [0]))`

18 ——— From `I-TRANSITIONS-STD` and **4 17** Conclude:
`(DURING (NEXT (AT T0)) (CONTINUE FLOW))`

Assuming `H1 ≡ !:DURING510`
Assuming `H2 ≡ !:%528 + H1`
Assume `H6 ≡ !:=536 !:NOT537 !:NOT534 + H2`
```
!:=536 (= !.Q FLOW)
!:NOT537
        (NOT (= !.Q SPEED))
!:NOT534
        (NOT (= !.Q HEIGHT))
```

19 ——— By equality substitution of `!:=536` in **18** Conclude:
`(DURING (NEXT (AT T0)) (CONTINUE !.Q))`

20 ——— From **19** Discharging `!:=536 !:NOT537 !:NOT534` Conclude:
`H2 ≡ !:%528 H1 ⊢`
```
(IF (AND (= ?V FLOW) (NOT (= ?U ?Z)) (NOT (= ?Y ?X)))
    (DURING (NEXT (AT T0)) (CONTINUE ?V)))
```

21 ——— By case analysis **6 8 16 20** Conclude:
`(DURING (NEXT (AT T0)) (CONTINUE !.Q))`

22 ——— From **21** Discharging `!:%528` Conclude:
`H1 ≡ !:DURING510 ⊢`
`(IF (% {HEIGHT SPEED FLOW} ?X) (DURING (NEXT (AT T0)) (CONTINUE ?X)))`

23 ——— From `I-REACHABLE` **4** `THE-QUANTITIES` and **22** Conclude:
`(NOT (REACHABLE (NEXT (NEXT (AT T0)))))`

24 ——— By situated equality substitution of `FLOW-EQUATION` in **12** Conclude:
`H0 ⊢`
`(DURING (NEXT (AT T0)) (CONSTANT (VALUE-OF SPEED)))`

25 ——— By situated equality substitution of `PLUS-ZERO` in `IN-FLOW-POSITIVE` Conclude:

```
(DURING ?X.1 (< (+ 0 0) IN-FLOW))
```

26 ——— From INEQUATION-MANIPULATION and 25 Conclude:

```
(DURING ?X.1 (< 0 (- IN-FLOW 0)))
```

27 ——— By situated equality substitution of 9 in 26 Conclude:

```
(DURING (NEXT (AT TO)) (< 0 (- IN-FLOW (VALUE-OF FLOW))))
```

28 ——— By situated equality substitution of TIMES-ZERO in 27 Conclude:

```
(DURING (NEXT (AT TO)) (< (* 0 ?_859) (- IN-FLOW (VALUE-OF FLOW))))
```

29 ——— From INEQUATION-MANIPS SURFACE-POSITIVE and 28 Conclude:

```
(DURING (NEXT (AT TO)) (< 0 (/ (- IN-FLOW (VALUE-OF FLOW)) SURFACE)))
```

30 ——— By situated equality substitution of FLOW-EQUATION in 29 Conclude:

```
(DURING (NEXT (AT TO)) (< 0 (VALUE-OF SPEED)))
```

31 ——— From DENSITY THE-CONSTANTS-1 24 30 Obviously:

```
(DURING (NEXT (AT TO)) (CONSTANT !.Z))
```

32 ——— From DENSITY THE-CONSTANTS-1 24 30 Obviously:

```
(DURING (NEXT (AT TO)) (< 0 !.Z))
```

33 ——— From DENSITY THE-CONSTANTS-1 24 30 Obviously:

```
(DURING (NEXT (AT TO)) (< !.Z (VALUE-OF SPEED)))
```

Assuming H1 ≡ !:DURING510

34 ——— From POSITIVE-DERIV and 33 32 31 THE-CONSTRAINTS 23 4 Conclude:

```
(NOT (AND (DURING (NEXT (AT TO)) (CONSTANT ?_868))
          (DURING (NEXT (AT TO)) (< (VALUE-OF HEIGHT) ?_868))))
```

35 ——— From 34 and THE-CONSTANTS-5 Conclude:

```
(NOT (DURING (NEXT (AT TO)) (< (VALUE-OF HEIGHT) HOLE)))
```

36 ——— By contradiction 5 35 Conclude:

```
HO ⊢
(NOT (DURING (NEXT (AT TO)) (CONTINUE HEIGHT)))
```

37 ——— From P-PASS-UP-NEXT HEIGHT-INITIAL-INC HEIGHT-INITIAL-LANDMARKS 2 Obviously:

```
(DURING (NEXT (AT TO)) (LANDMARKS HEIGHT [0] [HOLE +INF]))
```

Assume H7 ≡ !:DURING630

```
!:DURING630
      (DURING (NEXT (AT TO)) (PASS HEIGHT))
```

38 ——— From I-PASS-UP-NEXT 3 !:DURING630 Obviously:

(DURING (NEXT (NEXT (AT TO))) (STATE HEIGHT INC [HOLE]))

39 ——— From I-PASS-UP-NEXT 3 !:DURING630 and 37 Conclude:
(DURING (NEXT (NEXT (AT TO))) (LANDMARKS HEIGHT [O] [HOLE +INF]))

40 ——— From I-REACHABLE 4 THE-QUANTITIES and 36 Conclude:
HO ⊢
(REACHABLE (NEXT (NEXT (AT TO))))

Assuming H7 ≡ !:DURING630

41 ——— From P-TRANSITIONS-INC 40 and 38 Conclude:
(DURING (NEXT (NEXT (AT TO))) (PASS HEIGHT))

42 ——— From P-PASS-UP-NEXT 38 39 41 Obviously:
(DURING (NEXT (NEXT (NEXT (AT TO)))) (STATE HEIGHT INC [HOLE +INF]))

43 ——— From BETWEEN-VALUES 42 Obviously:
(DURING (NEXT (NEXT (NEXT (AT TO)))) (< HOLE (VALUE-OF HEIGHT)))

44 ——— From FLOW-ABOVE-HOLE and 43 Conclude:
(DURING (NEXT (NEXT (NEXT (AT TO)))) (= (VALUE-OF FLOW) OUT-FLOW))

45 ——— By situated equality substitution of PLUS-ZERO in IN-OUT Conclude:
HO ⊢
(DURING ?X.1 (< IN-FLOW (+ O OUT-FLOW)))

46 ——— From INEQUATION-MANIPULATION and 45 Conclude:
(DURING ?X.1 (< (- IN-FLOW OUT-FLOW) O))

47 ——— By situated equality substitution of TIMES-ZERO in 46 Conclude:
(DURING ?X.1 (< (- IN-FLOW OUT-FLOW) (* O ?_1428)))

48 ——— From INEQUATION-MANIPS SURFACE-POSITIVE and 47 Conclude:
(DURING ?X.1.1 (< (/ (- IN-FLOW OUT-FLOW) SURFACE) O))

Assuming H7 ≡ !:DURING630

49 ——— By situated equality substitution of 44 in 48 Conclude:
(DURING (NEXT (NEXT (NEXT (AT TO))))
(< (/ (- IN-FLOW (VALUE-OF FLOW)) SURFACE) O))

50 ——— By situated equality substitution of FLOW-EQUATION in 49 Conclude:
(DURING (NEXT (NEXT (NEXT (AT TO)))) (< (VALUE-OF SPEED) O))

51 ——— From DERIV-DEF THE-CONSTRAINTS 42 and 50 Conclude:
(= INC DEC)

52 ——— From DIRECTIONS-1 Obviously:
HO ⊢
(NOT (= INC DEC))

53 ——— By contradiction 51 52 Conclude:
(NOT (DURING (NEXT (AT TO)) (PASS HEIGHT)))

Assume H8 ≡ !:DURING698
!:DURING698
(DURING (NEXT (AT TO)) (STOP-NEW HEIGHT))

54 ——— From STOP-NEW-UP-NEXT 3 37 !:DURING698 Obviously:
(DURING (NEXT (NEXT (AT TO))) (STATE HEIGHT STD [!.L]))

55 ——— From AT-VALUE and 54 Conclude:
(DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF HEIGHT) !.L))

56 ——— From STOP-NEW-UP-NEXT 3 37 !:DURING698 Obviously:
(DURING (NEXT (NEXT (AT TO))) (LANDMARKS HEIGHT [0] [!.L HOLE +INF]))

57 ——— From LANDMARKS-ORDER and 56 Conclude:
(DURING (NEXT (NEXT (AT TO))) (< !.L HOLE))

58 ——— By situated equality substitution of 55 in 57 Conclude:
(DURING (NEXT (NEXT (AT TO))) (< (VALUE-OF HEIGHT) HOLE))

59 ——— From FLOW-BELOW-HOLE and 58 Conclude:
(DURING (NEXT (NEXT (AT TO))) (= (VALUE-OF FLOW) 0))

60 ——— By situated equality substitution of TIMES-ZERO in 26 Conclude:
HO ⊢
(DURING ?X.1 (< (* 0 ?_1636) (- IN-FLOW 0)))

61 ——— From INEQUATION-MANIPS SURFACE-POSITIVE and 60 Conclude:
(DURING ?X.1.1 (< 0 (/ (- IN-FLOW 0) SURFACE)))

Assuming H8 ≡ !:DURING698

62 ——— By situated equality substitution of 59 in 61 Conclude:
(DURING (NEXT (NEXT (AT TO)))
(< 0 (/ (- IN-FLOW (VALUE-OF FLOW)) SURFACE)))

63 ——— By situated equality substitution of FLOW-EQUATION in 62 Conclude:
(DURING (NEXT (NEXT (AT TO))) (< 0 (VALUE-OF SPEED)))

64 ——— From DERIV-DEF THE-CONSTRAINTS 54 and 63 Conclude:
(= STD INC)

65 ——— From `DIRECTIONS-3` Obviously:
`HO ⊢`
`(NOT (= STD INC))`

66 ——— By contradiction **64 65** Conclude:
`(NOT (DURING (NEXT (AT TO)) (STOP-NEW HEIGHT)))`

67 ——— From `I-TRANSITIONS-INC` **4 3** and **36 53 66** Conclude:
`(DURING (NEXT (AT TO)) (STOP HEIGHT))`

68 ——— From `STOP-UP-NEXT` **3 67** Obviously:
`(DURING (NEXT (NEXT (AT TO))) (STATE HEIGHT STD [HOLE]))`

10.12.8 Proof Commentary

[1,4] The initial situation and the one that follows it are both reachable.

[2] Since HEIGHT is initially increasing in a point-situation, it must Pass.

[3] and becomes increasing between 0 and HOLE.

[5] and $\underline{\text{HEIGHT}} < 0$.

Assuming that HEIGHT continues in the 2nd situation, we are going to show that all quantities must Continue.

[6] Assuming quantities are in the set {HEIGHT SPEED FLOW}, then a quantity must be equal to one of these 3 elements.

[7] Assuming the quantity is HEIGHT, then, by hypothesis, it Continues.

[8] from which we obtain the generalized conclusion that any quantity equal to HEIGHT Continues.

[9] Given [5], we know that $\underline{\text{FLOW}} = 0$ by axiom `flow-below-hole`. 0 is a constant.

[10–12] We thus show that the arithmetic expression representing $\underline{\text{SPEED}}$ is constant.

[13] Therefore SPEED becomes steady at said constant value.

[14] and, since this is an interval-situation, it has no other choice but to Continue.

[15,16] Therefore, if a quantity is equal to SPEED it must Continue.

[17] From [9] we know that $\underline{\text{FLOW}} = 0$, which, since 0 is a constant, is equivalent to saying that FLOW is steady at 0.

[18] Thus, like SPEED, FLOW has no other choice but to Continue.

[19,20] We conclude that if a quantity is equal to FLOW it must Continue.

[21–22] Therefore, by case analysis based on the disjunction of cases specified by [6], from [8], [16], and [20] we conclude that all quantities must Continue.

[23] and that the 3rd situation is not reachable, i.e. the 2nd situation extends indefinitely into the future.

[24] From [12] we infer that $\underline{\text{SPEED}}$ is constant.

[25–30] By arithmetic manipulations, we also conclude that $\underline{\text{SPEED}} > 0$.

[31] Since we know that 0 and $\underline{\text{SPEED}}$ are both constants and $\underline{\text{SPEED}} > 0$, by density, we can pick a constant Z such that $0 < Z < \underline{\text{SPEED}}$.

[32–34] and, since SPEED is the derivative of HEIGHT, by `positive-deriv` from [23] we infer that there can be no constant such that $\underline{\text{HEIGHT}}$ doesn't eventually exceed it.

[35] In particular, this is true of HOLE; i.e. it is not the case that $\underline{\text{HEIGHT}}$ remains below the hole throughout situation 2.

[36] By contradiction with [5], we conclude that HEIGHT will not Continue.

Now, assuming that HEIGHT Passes we derive another contradiction.

[38] If Height Passes, then it must become increasing at the hole.

[40] Since we know by [36] that HEIGHT does not Continue, we know that the next situation is reachable.

[41] Therefore, by axiom **p-transitions-inc**, HEIGHT has no choice but to Pass.

[42] and to become increasing between the hole and $+\infty$ in situation 4.

[43,44] Consequently, $\underline{\text{FLOW}}$ = OUT-FLOW.

[45–48] (IN-FLOW – OUT-FLOW)/SURFACE < 0.

[49,50] Therefore $\underline{\text{SPEED}} < 0$.

[51] [42] indicates that HEIGHT is increasing whereas [50] implies that it is decreasing.

[53] By contradiction conclude that HEIGHT cannot Pass.

Now assuming that HEIGHT Stops-New we derive another contradiction.

[54] By hypothesis, HEIGHT becomes steady at the new landmark value L,

[56,57] such that L < HOLE.

[58] Therefore $\underline{\text{HEIGHT}}$ < HOLE.

[59] and $\underline{\text{FLOW}} = 0$.

[60–63] From which we derive that $\underline{\text{SPEED}} > 0$.

[64] [54] specifies that HEIGHT is steady, whereas [63] implies that it is increasing.

[66] By contradiction conclude that HEIGHT cannot Stop-New.

Finally the desired conclusion.

[67] By elimination, only the Stop transition remains.

[68] Therefore, HEIGHT must become steady at the hole.

10.13 Conclusion

In this chapter, I have described an axiomatization of qualitative physics that captures Kuiper's theory of Qualitative Simulation in the framework of first-order logic. I also demonstrated that it is feasible to carry out formal proofs in this axiomatization.

All the mechanisms provided by LOGICALC have proven useful in the derivation of the two proofs presented here. The detaching mechanism played a major rôle, principally through the office of the USE command. Equivalence classes and the answer sharing/broadcasting meachanism also contributed, as evidenced by a number of multiply occurring inference nodes.

I have shown that LOGICALC can easily be extended with new rules and new plan generators to facilitate theorem-proving work with certain representations or axiomatizations. I illustrated this by implementing additional tools specifically for the situation calculus.

The proof editor was also exercised to produce the proof summaries included in this chapter. The default heuristics performed a satisfactory job of condensing the proof and of providing an initial formatting of its summary. Then, through the editor, I effected minor cosmetic alterations to enhance appearance and legibility; few are normally needed, if any.

Chapter 11

Conclusion

What have we wrought?

We have a program that generalizes backward-chaining to a rich system of natural deduction and frees the user from many uninteresting details.

For the AI researcher, and, we believe, for others as well, the default reliance on skolemization and the implicit ubiquitousness of unification are both convenient and natural. They relieve the user from having to worry about the details of substitutions and quantifier eliminations. There is no need to guess the proper instantiations in advance. Free variables allow us to delay these guesses, and unification supplies the appropriate values when they are needed.

The system manages the proof state as a graph, but also supplies default motions through this graph to follow the current thread of reasoning. It encourages an exploratory attitude whereby the user may try things out, and, in particular, need not apriori form a definitive outline of the proof, but may let the evolving context suggest ways to proceed. For example, the system explicitly represents alternative proof

plans. Most importantly, it facilitates searches of the database guided by contextual requirements (e.g. the shape of the desired premise) and provides an interactive browser that lets the user examine and select candidates.

Not only does the system facilitate searching the database, but it also takes over the management of inferential details. In particular, the *detaching* mechanism encourages the user to reach into the database and pull out formulae with a shape appropriate for the proposed refinement. The system will automatically determine what auxiliary goals you need to prove to infer such a formula from the database, thereby validating its use in the proof plan.

Thanks to this mechanism, a large fraction of most proofs can be constructed in a resolution-like fashion by issuing commands of the form '`USE` *name*', where *name* refers to an assertion in the assumption set. The resulting subgoals are those needed to infer the current goal from this assertion. In other words, the user merely points to an assertion and tells the system: *"figure out how to use this to refine the current goal."*

Furthermore, the system tries very hard to minimize the amount of work required of the user. For example, it will inform the user if a proof derived for an earlier goal is applicable to a later one. The user may then select this existing answer rather than attempt a new proof. In fact, proofs are automatically generalized to broaden their applicability (i.e. their chances of being reused) as much as possible.

One singularly attractive contribution of LOGICALC is the manner in which plan generators are interfaced with the user. Unlike many modern systems where refinements must be effected by calling procedures with well-defined arguments, here these arguments may be provided in a (mostly) free-form fashion. Each plan generator has its own parsing procedure to make sense of the user's input. Moreover, our interface provides for implicit and explicit searches with contextual constraints, for implicit

detaching, and for the ability to fire up an interactive browser to help fill in an argument.

Our own experience with LOGICALC has been very positive, especially while investigating the formalization of qualitative physics. The ability to explore the proof space freely and naturally was crucial in assessing the strengths and the limitations, as well as the occasional omissions, of our axiomatization. Many features now included in the system were born from practial experience. For example, the `-skolem` option, together with the indexing scheme that makes its operation possible, was added because a fair number of goals turned out to be *about* specific skolem terms originating from the database, and we wanted an easy way to locate relevant assertions *about* them. The system also proved to be easy to extend, as we demonstrated by implementing additional rules and plan generators to facilitate reasoning in the situation calculus. Finally, the proof summaries generated by our heuristics are surprisingly readable, and can be directly included in typeset documents.

In the following, we will present a summary of the system, emphasizing its contributions. Then we will discuss related work, acknowledge influences and point out differences. Finally, we will outline future directions of research.

11.1 Summary Of The System

LOGICALC is a system for the interactive derivation of proofs which was developed from an AI perspective and with the specific intention of facilitating work with formal AI problems. These circumstances as well as our motivating interests are reflected in the choice of logic and the architecture of the system which, as a result, is poised at an interesting juncture between classical logic and traditional AI practice.

The Logic

Researchers in AI typically specify formal problems using some variant of first-order logic, and, more often than not, the quantifier-free predicate calculus is adequate. Furthermore, it is normally expected that the search for a proof will be conducted with the aid of unification. In LOGICALC, we chose to meet these expectations while retaining at the same time the full power of first-order logic.

Our logic is first-order, but all formulae (axioms and goals) are skolemized by default. Occasionally, automatic skolemization becomes undesirable: for example when the antecedent of a goal is universally quantified and we intend to assume it in a proof using the Deduction Theorem. In such a case, skolemization can be explicitly disallowed.

Working with a skolemized representation has several advantages. Firstly, we are spared the tedium of explicit quantifier manipulations. Secondly, we can rely on unification as a fundamental operation, both for generating refinements and inferences, as well as for searching the database (the set of assumptions). Thirdly, skolemized assumptions can be indexed, so as to to increase the speed and specificity of database searches, in ways that a fully quantified representation does not permit.

A skolemized representation also has disadvantages: skolem terms can easily become inscrutable, and conclusions are often phrased in terms of specific skolem terms when in fact they are true in general. We proposed an abbreviation mechanism that eliminates the first problem for both input and output. We also implemented a proof generalization algorithm which generalizes equally over all terms, and, *a fortiori*, remedies the second shortcoming.

Our logical system is based on a system of Natural Deduction for a variant of classical first-order sequent logic. We write $A \Rightarrow p$ for the sequent expressing that proposition p logically follows from the set A of assumptions. The twist in our approach is that A

is implemented by a datapool. There can be no connection between the free variables appearing in A and those in p. In fact, looking up an assertion in A automatically gives us a copy with fresh variables just like it would in a logic programming language such as PROLOG. Datapools, therefore assumption sets, are organized hierarchically, but the underlying database is truly a flat space of assertions managed by an ATMS: only those assertions are visible which are 'in' the currently selected datapool.

This system was shown to be sound and complete for first-order logic.

The Architecture

LOGICALC allows proofs to be constructed in a *forward* manner, by applications of inference rules mediated by *inference generators*. However, since this is not the preferred mode of operation, it was not described in this thesis.

Primarily, proofs are developed *backwards*, by refining goals into plans with subgoals through the assistance of *plan generators*. The latter are reminiscent of LCF tactics, but are not so easily composable (but see p11.3). I discuss them further below. Let it simply be said that plans, as in LCF, consist primarily of a list of subgoals together with a validation.

The initial goal, i.e. the theorem to be proven, is analyzed in a top-down fashion. The process of backward refinement is repeated interactively until satisfactory answers have been found for all leaf goals, usually by unification with axioms or assumptions. Then a proof is synthesized bottom-up by combining the proofs of a plan's steps using the plan's validation.

LOGICALC provides several mechanisms and architectural devices to make this approach more attractive and more productive.

Firstly, goals are grouped in equivalence classes. The primary motivation for introducing this notion is to avoid having to repeat the same proof for goals which are

identical to, or alphabetical variants of, one another. When a proof of an instance of a goal is found, it is recorded in the goal's class and all members of this class can benefit from it. Furthermore, we also maintain links between related classes. New answers are broadcast along these links and are also recorded in those classes where they are found to be applicable (i.e. they unify with the class's pattern).

Secondly, when a new equivalence class is created, the system automatically looks for answers to it in four ways: (1) by attempting to borrow and adapt answers already recorded in related classes, (2) by checking to see if the class's pattern is a tautology, (3) by attempting to match the class's pattern against assertions in the database, (4) by running logic programs.

Each plan created as a result of invoking a plan generator is recorded with the class it is attempting to prove. In contrast with most systems, in LOGICALC, each class may have several plans simultaneously under investigation. There are two main reasons for allowing this: firstly, two different goal occurrences that are members of the same class may require distinct analyses resulting in two distinct instantiations of their free variables. Secondly, the user may wish to pursue alternative proof attempts. Other systems often allow the latter only through backtracking; in which case switching to an earlier proof attempt means you have to redo all the work again. In contrast, LOGICALC retains all proof attempts at the cost of more memory.

Each goal sees all the answers recorded in its class. When a satisfactory answer is found for a particular goal occurrence, that answer may be followed, thus resulting in a successor plan; that is, a plan obtained by making a copy of the one in which the goal was a step, removing that step, and instantiating the others with the answer's substitution. When all steps have been eliminated, their proofs are gathered, passed as arguments to the plan's validation: a proof is produced for the plan's class. This proof is generalized, then recorded in the class and broadcast along links to related classes.

The proof generalization algorithm is another contribution of our system, which is inspired by Explanation Based Generalization but extends it to our system of natural deduction for first-order logic. Unlike in traditional EBG where meta-variables (i.e. quantified over the explanation structure) and object variables (i.e. free variables of the object logic) may conveniently be conflated (both being represented, say, by PROLOG variables), our technique must maintain the distinction to preserve soundness in the presence of (skolemized) assumptions. Our algorithm generalizes equally over all terms.

In summary, our architecture is organized around the notion of a graph of goals and plans. We improve on this basic design by grouping goals in equivalence classes, primarily to promote the sharing of answers. Since answers are broadcast across the network of classes, and their proofs are asserted in the database, the more general they are, the wider their applicability, and the less work we need to repeat. Therefore, in order to make the most of each proof, we generalize them before they are released system-wide.

The Graph Browser And Editor

Since our architecture is based on the careful management of a complex graph, where nodes are goals, plans, classes, proofs and answers, it is only appropriate to design the user interface as a graph browser and editor. This browser maintains the notion of a current node (or focus of attention), for which it provides an informative display, and lets the user issue commands to act on it, or to move the focus of attention to another node (usually by following some sort of link between them).

Communication with the user is effected through a line-oriented shell, with csh features such as the ability to recall earlier commands, and to perform substitutions. Considerable effort was invested in the integration of a facility for providing interac-

tive help. This help is essentially of two kinds: (1) documentation, and (2) argument elaboration.

The first kind of help may be requested by the keyword `-help`. Everywhere a command or an argument is expected, this keyword may be issued instead. In response, the system prints appropriate documentation.

The second kind of help may be requested by the keyword `-find`. It is a way for the user to ask the system *"help me fill in this argument."* Typically, the system responds by initiating a subsession with a specialized interactive browser which lets the user examine possible candidates (or search for them, if the system lacked sufficient information to make an initial selection) and choose those which are deemed of interest. This functionality is implemented by the *Find Mode* package, and is briefly documented in Appendix B. When the interactive subsession is terminated, all selected candidates are returned to the command, to be used as arguments.

The shell is implemented in a modular fashion using object-oriented techniques. Each type of node is equipped with its own display function and command processor. Such an implementation is easily extensible. In particular, it is easy to define new commands for any given command processor.

As I mentioned earlier, we provide a mechanism for abbreviations. It was originally motivated by the desire to improve the appearance of skolem terms, and to facilitate their input. This mechanism was subsequently extended to allow abbreviation of any term, as well as of commands. We contribute an algorithm for selecting the most specific abbreviation pattern applicable to a given term.

Plan Generators And Detaching

The graph can be extended in two ways: either by creating new plans through the invocation of plan generators, or by constructing a successor plan to an existing plan

by following an answer to one of its steps.

Plan generators are clearly related to LCF tactics: they are meant to obtain plans consisting of subgoals and equipped with a validation. However, we strived to make the interactive interface easier, simpler, more helpful and more powerful. As a result, our design distinguishes three layers: (1) parsing user input, (2) assembling plan descriptions, and (3) realizing these descriptions.

User input is simplified in the following ways: first, through the use of a flat command line instead of the function calls required by LCF type systems. Secondly, by allowing commands and options to be abbreviated by a prefix. Thirdly, by offering powerful notations for premise and subterm arguments; in particular, the ability to invoke *Find Mode*, to fill in some arguments interactively, by specifying the option `-find`.

A major contribution of our interface is that all premise arguments are obtained through the *detaching* mechanism. This technique is remarkably similar in spirit to non-clausal E-resolution, but, rather than a resolvent, it produces plan descriptions appropriate for our system of natural deduction. Thanks to the detaching mechanism, the user can simply name a relevant assertion and the system will figure out automatically how to use it in this context, setting up auxiliary goals if necessary.

Since our framework expects skolemization to play a major role, we endeavored to make proving formulae about skolem terms as painless as possible. Therefore, for the user's convenience, we maintain an indexing scheme which lets us quickly find those assertions mentioning a particular skolem term. This selection scheme can be activated by the `-skolem` option. Thus, to prove a property of a skolem term, it is often sufficient to supply `-skolem` as the premise argument and let the system figure it out.

Each plan generator has a "designer" component which takes a list of plan descriptions as arguments (presumably obtained through the detaching mechanism), as well as a

list of parameters, and assembles new plan descriptions from these premises. Such a design is very uniform and easily composable, but, so far, we do not provide a convenient way for the user to take programmatic advantage of it. LOGICALC itself does not, at present, make use of this composability except, modestly, for "automatic goal expansion."

Finally, these descriptions are realized; i.e. they are transformed into real plans whose validations are also extracted from these very same descriptions.

Occasionally, a validation cannot be completely determined at plan creation time. For such situations, we proposed the notion of validation hooks. They allow us to gather information and determine what inferences to make at validation time.

Proof Editing And Formatting

In LOGICALC, proofs are actual objects and can be inspected. We provide a mechanism to prepare summarized representations of a proof tree in a variant of the traditional natural deduction format. Such a summary can be output in a variety of ways, including as TEX and LATEX code for inclusion in typeset documents.

We claim that a good proof summary must strive to achieve the right level of detail. Some parts of a proof are trivial or uninteresting, and they should be hidden or abbreviated. Some inferences are sufficiently obvious that they don't have to be stated explicitly. Too many details, and we loose sight of the overall structure. Too few details, and the proof no longer conveys sufficient information to be considered a satisfactory explanation.

To support the construction of summaries that satisfy the above criterion, we introduced the notion of a line's *visibility*, and we contribute an algorithm for computing their default values according to a set of heuristics controlled by user-settable parameters.

The proof editor prepares a summary according to these heuristics. Then, the user may interactively further edit this summary to improve on the presentation and achieve better legibility.

11.2 Related Work

Resolution

Resolution has held a preeminent position in automated theorem proving ever since its introduction by Robinson in 1965 [Rob65]. It relies on a powerful matching operation, namely *unification*, but requires that all formulae be expressed in clausal form. The latter requirement is often considered a disadantage because the conversion to clausal form introduces a lot of redundancy (by distributivity), and the result is difficult to understand due to skolemization, loss of structure (everything is flattened out), and fragmentation (one axiom may result in many clauses). On the other hand the search space is very homogeneous. However, that too causes problems because it is difficult to maintain some sense of direction when searching for a proof.

For this reason, many restrictions and modifications to the basic resolution rule have been proposed, such as linear resolution, set of support strategy, or model elimination [FLSY74, Lov78], or the idea of connection graphs introduced by Kowalski [Kow75,Bib81]. Also, SLD-Resolution on Horn clauses is used as the basis for logic programming (cf. PROLOG).

Non-clausal resolution was described by Murray [Mur82]. Another resolution-like procedure that does not require clausal conversion is mating search proposed by Andrews [And81] and implemented in the TPS theorem prover. The connection graph method was also adapted to non-clausal resolution by Stickel [Sti82].

Another extension of resolution has to do with the proper treatment of equalities. Wos and Robinson introduced paramodulation which is a generalization of equality substitution, and uses an equality literal from one clause to rewrite a term in another clause [RW69]. Unfortunately, this process is very undirected creating many useless resolvents. Instead, Morris proposed E-Resolution [Mor69] where paramodulations are effected only to repair a failed unification of the basic resolution procedure. Digricoli further extends this approach with his notion of resolution by unification and equality (RUE) [DH86].

The unification procedure and the resolution rule were extended to higher-order logic based on the typed λ-calculus by Pietrzykowski [Pie73,JP76] and Huet [Hue72,Hue75].

LOGICALC also uses unification as a fundamental operation. At present, it is only a slightly extended version of the first-order procedure; however, this is not a fundamental commitment.

LOGICALC's subtratum supports the notion of logic programming; in fact it provides both backward chaining and forward chaining flavors. This facility is often used to implement decision procedures. For example, the system includes a non-trivial logic program to automatically solve goals denoting type constraints.

LOGICALC also borrows from non-clausal resolution the idea of being able to perform an inference using a subformula of an hypothesis. While in resolution theorem-proving there is only one single rule of inference, LOGICALC generalizes this idea to a system of natural deduction: the *detaching* mechanism provides a way to pull out subformulae from assertions in the database and use them as premises for refinements.

The metaphor here is of reaching inside a hypothesis, grabbing a subformula, and pulling it out; in the process, the original hypothesis is partially turned inside-out, and, when we have pulled the subformula all the way out, usually, there will remain a few other pieces of the hypothesis still hanging off it: these are the auxiliary goals

which have to be proven in order to infer the selected subformula from the hypothesis (in other words, they constitute the part of the resolvent that comes from the hypothesis).

Although this mechanism is inspired by non-clausal resolution, it is implemented entirely on top of natural deduction. As a further extension in the spirit of E-Resolution and RUE, the detaching mechanism can also allow mismatches and automatically repair failed unifications by introducing additional equality goals.

TPS

As explained earlier, the conversion to clausal form in resolution-based methods has many disadvantages such as the proliferation of literals resulting from the repeated application of distributive laws. Andrews proposed the notion of "matings" to carry out refutations [And76], and described a corresponding proof procedure [And81].

Formulae are represented in matricial form, where the horizontal direction corresponds to disjunction, and the vertical direction to conjunction. $P \vee [Q \wedge R]$ is written:

$$\left[P \vee \left[\begin{array}{c} Q \\ R \end{array}\right]\right]$$

A mating for a matrix G is a set of pairs of literal occurrences in G such that they can all be simultaneously resolved; i.e. there exists a substitution θ such that, for each mated pair (P, Q), $\theta P = \neg \theta Q$.

A refutation mating is one such that G is false for any interpretation consistent with the pairing, and can be recognized by the criterion of path-acceptability: a mating is acceptable iff every vertical path through the matrix contains a mated pair.

The matrix is in negation normal form and skolemized. The search for an acceptable mating may involve duplicating some quantifiers, i.e. replacing a subformula $\forall x\, P$ by $[\forall x\, P] \wedge [\forall x\, P]$, in order to obtain distinct universal instantiations.

TPS is essentially a system for carrying out natural deduction proofs in the simply-typed λ-calculus. The user may apply inference rules directly (forward/inference, backward/refinement), or invoke tactics, or initiate mating search. In the later case, a tactic is required to translate the resulting expansion tree proof back into natural deduction format.

Rewriting Techniques

Rewriting techniques are another widely used approach to automated theorem proving.

One of the more successful programs of this kind is the Boyer-Moore theorem prover [BM79]. It is used to prove equations between terms in pure LISP. The formal theory accounts for inductively constructed objects such as natural numbers and finite sequences, and allows recursive definitions provided that a measure and a well-founded relation can be found such that the measure of each argument in a recursive call is smaller than the measure of the corresponding input argument according to the well-founded relation. It is from this condition that the Boyer-Moore theorem prover derives its real power: namely the ability to perform induction proofs (e.g. by induction on the length of a list: the corresponding recursive definition simply CDR's down the list, and the length of the list decreases monotonically for each recursive call).

The Boyer-Moore theorem prover does no search. It uses heuristics to determine what axiom or lemma to apply as a rewrite rule and where, and repeats the process until the original equation is proven (i.e. has been rewritten to TRUE), or no rewrite rule is applicable. This single-minded deterministic approach makes it quite fast, but, occasionally, its heuristics will select a path that does not lead to a successful proof. It is the responsibility of the user to supply appropriate lemmas to guide the rewriting

process to success. Matt Kaufmann [Kau90] extended this framework to introduce the notion of *tactics* as well as allow lower-level interaction by the user.

Ketonen's EKL [KW84, Ket84] is an interactive proof-checker for a sorted logic with quantification and abstraction. A new proof line is usually obtained either by introducing an axiom, or an assumption, or by rewriting an existing line, or by rewriting a user specified term or formula. In the later case, if it is a term t which rewrites to t', the line $t = t'$ is added, otherwise it is a formula p that rewrites to p'; if p' is the constant TRUE, then p is added as a line else $p \equiv p'$.

Rewriting is carried out by interpreting earlier lines as rewrite rules. This interpretation is more involved than with the Boyer-Moore theorem prover: rewrite rules maybe quantified; furthermore, in addition to equations there are negated formulae ($\neg A$ rewrites A to FALSE), conjunctions (each conjunct is interpreted as a separate rewrite rule), implications ($A \supset B$ rewrites B to TRUE in places were A must hold). Other ways of deriving new lines are by universal instantiation (specialization) and IF introduction (discharging assumptions).

EKL uses Huet's algorithm for higher-order unification, and handles associative operators cleverly. The direction of rewriting can be explicitly controlled by the user. The system can be extended through its meta-theory. In this respect, it was influenced by Weyrauch's FOL system, its principle of reflection, and the idea of semantic attachments.

The Knuth-Bendix procedure is a powerful method for converting systems of equations into sets of rewrite rules, but it will not be discussed here.

LOGICALC does not feature a rewriting mechanism of the form described above. On the other hand, it inherits forward-chaining rules from the DUCK system. A forward-chaining rule $p \rightarrow q$ is triggered when a fact matching p is added to the database: q is then also added to the database, or executed if it is an executable predicate. Currently, this facility is used only to automatically generate entries for various indexing schemes.

FOL and GOAL

FOL is a proof checker for first-order predicate calculus with equality and sorted variables, developed at Stanford by McCarthy and Weyhrauch [FW76,Wey78,Wey80]. Proofs are constructed using Prawitz's system of natural deduction.

A distinguishing feature of this program is its ability to perform semantic simplifications by attaching LISP functions to operators and predicates of FOL. This trick permits the rewriting of ground terms and formulae by direct evaluation. Thus $1 + 2$ can be simplified to 3 if the LISP function PLUS is attached to the FOL operator $+$.

Traditional rewriting is also supported. The user specifies a set of (quantified) equations or equivalences, and the system interprets them as left to right rewrite rules.

FOL supports simultaneously the notions of theory and metatheory, and a way to interface between them though the principle of reflection and the operators $\uparrow$ (quote) and $\downarrow$ (eval). For example, an axiomatization of well-formedness can be found in [Wey78] (see [HHP87] for a more recent approach to the problem of encoding in a formal system the syntax and rules of another logical system).

GOAL written by Bulnes-Rozas [BR79] provides a facility for goal-oriented, top-down proof construction in FOL. The user may invoke matchers, tactics, and strategies. A matcher attempts to prove a goal directly without reducing it further (e.g. by matching against an axiom, or by recognizing it as a tautology). A tactic reduces a goal into subgoals. Each tactic is the inverse of an inference rule. The reduced goal keeps the necessary information so that the appropriate inference rule is called when all its subgoals have been proven. This process of constructing a proof of a goal from proofs of its subgoals is called unwinding. The information that makes it possible is called a reason and can be viewed as a simplified version of a LCF's validation. A strategy is simply a procedure consisting of a sequence of applications of tactics and matchers.

GOAL inherits the simplification and rewriting mechanisms of FOL.

LOGICALC takes a rather similar approach, but extends it in several directions. It has a more complex and more powerful graph structure, plan generators can create plans whose validations involve simultaneously many inference rules, the detaching mechanism makes the process of refining goals into subgoals orders of magnitude easier, and various browsers (such as the main graph editor, and find mode) facilitate interaction with the system.

LCF

LCF stands for "a Logic for Computable Functions" and is based on the model of computability proposed by Dana Scott in 1969. The original system was developed at Stanford by Milner in 1971–72 and functioned as a proof checker. Milner then decided to provide a meta-language (ML) with which he implemented Edinburgh LCF [Mil79a]. ML is a higher-order functional programming language with a polymorphic type system.

Theorems are instances of the datatype `thm`, and inference rules are functions which compute theorems. The type discipline guarantees that derived inference rules (i.e. functions expressed in terms of existing inference rules) also return theorems. Assuming that the primitive inference rules are sound and that their implementation is faithful, then it is impossible to derive non-theorems.

LCF popularized the notions of tactics and tacticals. Tactics are primarily used to refine goals into subgoals, whereas tacticals are higher-order functions that serve to combine tactics and are often written as infix operators. For example, t_1 `THEN` t_2 represents a composite tactic that applies tactic t_1 first, and then invokes t_2 on the result.

ML has a mechanism for raising and handling exceptions which is very convenient to express partial functions. If a tactic determines that it is not applicable on its given arguments, then it raises a failure exception. The tactic ORELSE handles a failure signaled by its 1st argument by invoking its 2nd argument; thus, t_1 ORELSE t_2 is a tactic that is just like t_1, except when t_1 is inapplicable, in which case it is just like t_2. Similarly, REPEAT t applies t repeatedly until it is no longer applicable.

However, the technique of programming with tactics and tacticals has a wider range of applications than mere partial subgoaling methods. For example, TPS uses them also to translate expansion trees to natural deduction proofs.

Paulson's Cambridge LCF [Pau87] is also based on PPλ (Polymorphic Predicate λ-calculus—the logic of LCF), but extends it with $\vee$, $\exists$, $\Longleftrightarrow$, predicates, additional inference mechanisms, and rewriting.

The Synthesizer Generator and IPE

The Synthesizer Generator [RT87] is a generalization of an earlier system, the Cornell Program Synthesizer [RT81]. From a formal language specification, the Generator creates a display editor for manipulating sentential objects in this language. In particular, the editor is well-suited to the generation of structure editors that enforce the syntax and static semantics of a particular language.

In 1984, Reps and Alpern [RA84] described a system for interactive proof checking implemented with the Synthesizer Generator, and based on the notion of an attribute grammar that expresses the rules of Gentzen's sequent calculus [Gen35]. The proof tree is represented by a consistently attributed derivation tree; e.g. the set of A_i is called antecedent attribute, and the set of B_i the succedent attribute. The user can interactively edit this proof tree. If a particular subtree is altered or restructured, attributes are incrementally reevaluated.

A node that does not satisfy all the checks and attribute equations from the language specification is inconsistent and noted as such, but, in contrast to a system like LCF, is nonetheless permissible. This allows the user to incrementally fix/develop a proof.

Ritchie's Interactive Proof Editor IPE [Rit87, RT88] is a continuation of the work of Reps and Alpern, and explores the feasibility of "proof by clicking." Its domain is intuitionistic first-order logic defined as a natural deduction system. Proofs are generated top-down, through a graphical mouse-driven interface.

Nuprl

The NUPRL system [C+86] is a display-based interactive proof development environment for constructive type theory and was inspired by the Synthesizer Generator. It takes a strongly foundational stance and interprets constructive proofs as programs. The intuition is that a program specification typically has the form $\forall x{:}A\, \forall y{:}B\; R(x, y)$, where x is the input and y the output. A constructive proof provides a method for obtaining a y given an x.

NUPRL borrows from Martin-Löf's framework, in particular the notion of propositions as types, as well as from LCF, and operates according to Bates' *Refinement Logic* [Bat79]. A proof is a finite tree whose nodes are pairs consisting of a subgoal, in the form of a sequent, and a rule name or a place holder. Invoking an applicable rule at a terminal node with a place holder results in the latter being replaced by the rule's name and zero or more children nodes being attached to this one. Thus the logic explicitly represents partial proofs. A proof tree with no remaining place holders is complete.

The underlying language of NUPRL is the λ-calculus with dependent types. Types are arranged in a cumulative hierarchy of universes. The atomic types are `int`, `atom`, and `void` (the empty type). Additional types can be constructed by means of type

constructors. Thus, A **list** is the type of lists whose elements are of type A, $A\#B$ is the cartesian product, $A|B$ the disjoint union, $A \to B$ the type of functions from A to B. There are also constructors for dependent types: $x{:}A\#B$ the dependent product (like $\Sigma x{:}A.B(x)$), $x{:}A \to B$ the dependent function type (like $\Pi x{:}A.B(x)$), and also $\{x{:}A|B\}$ for sets and $(x, y) : A//B$ for quotient types (equivalence classes). Each type comes with an equivalence relation for equality notated $a = b \in A$ ($a = a \in A$ is abbreviated $a \in A$).

In the propositions-as-types paradigm, propositions are encoded by type terms, and proving a proposition amounts to showing that the type is inhabited by exhibiting an element of this type, i.e. by deriving a judgement $t : A$ where A is the type encoding the proposition, and t is an element of this type.

When a proof has been constructed, a term representing its computational content can be extracted by means of the **term_of** operation. Thus a program can be synthesized from a constructive proof of its specification.

NUPRL allows the user to define new notations with templates, has facilities to support libraries, and, like LCF uses ML as its metalanguage and tactics to help in the proof development process.

Automath

The pioneering work of de Bruijn on the AUTOMATH project [dB80] has spawned and continues to inspire very active research in logical frameworks. The project, conceived in 1966, was aimed at developing a system for writing mathematical theories with sufficient precision to allow their correctness to be verified automatically. Its framework was intended to be general enough to allow all kinds of mathematical notions to be captured easily, while making as few commitments as possible. Thus, it could simultaneously serve multiple schools of mathematics with distinct foundational

ideas. Probably the most notorious accomplishment of the AUTOMATH project was the systematic translation (encoding) of Landau's *Grundlagen* by Jutting [Jut77].

The Calculus of Construction

The calculus of constructions (henceforth CC) is another foundational approach, proposed by Coquand and Huet [CH85b,CH85a,CH86] and inspired both by Martin-Löf theory of types and by Girard's F system. It is a higher-order formalism for constructing proofs in natural deduction style. The basic language extends the polymorphic λ-calculus: terms and types are expressions of the same nature.

There is one constant '$*$' representing the universe of all types. $[x{:}M]N$ denotes quantification, $(\lambda\, x{:}M)N$ abstraction, $(M\ N)$ application. The calculus makes 3 distinctions: contexts, propositions, and proofs. A context is a sequence of declarations of the form $[x_1{:}M_1]\ldots[x_n{:}M_n]$. The corresponding type $[x_1{:}M_1]\ldots[x_n{:}M_n]*$ is also called a context. A proposition is an object of type a context, and a proof is an object of type a proposition. The calculus is expressed in terms of sequents where the left-hand side is a context and the right-hand side is either a context or a typing.

From a programming language perspective, contexts, propositions and proofs can be identified respectively with declarations, specifications and algorithms.

The system described in [CH85b] allows the user to give names to constructed objects (this is typically represented by the notation $[x = M]$ in a context), and also to introduce new mixfix syntax. Implicit arguments have been introduced to reduce the burden of polymorphic instantiation when some argument types can be obtained from the 'interesting' arguments.

A 2 register machine (2 sides of sequent) for carrying out constructions and proofs is described in [CH85a]. Pollack's LEGO system lets the user construct proofs top-down

by incremental refinements [LPT89, Tay89]. Also, CC is a convenient framework for developing programs which are certified to terminate and meet a logical specification. Paulin-Mohring describes an algorithm to extract the computational content of a CC proof [PM89].

Isabelle

Paulson remarked that the LCF framework was being used to construct theorem provers for various logics, and that much of the same work had to be duplicated for each one. To remedy this situation, he proposed to develop a generic theorem prover called ISABELLE [Pau86].

ISABELLE's meta-logic is intuitionistic higher-order logic with implication, universal quantifiers, and equality (resp. $\phi \Longrightarrow \psi$, $\Lambda x.\phi$, and $a \equiv b$). With it, you can encode your own logic. For example, the rule of and-introduction can be represented by:

$$\Lambda P.\ \Lambda Q.\ P \Longrightarrow (Q \Longrightarrow P \wedge Q)$$

Similarly, $\forall x.P$ can be encoded by $\mathrm{All}(\lambda x.P)$ where 'All' is a new constant (of type, say, $(i \rightarrow o) \rightarrow o$).

Thus, you can customize your own logic by encoding its signature and theory using ISABELLE's meta-logic, and by extending the parser/printer with new syntactic rules (mixfix definitions).

ISABELLE does not construct proof trees. Instead, the process of searching for a proof is viewed as the elaboration of derived rules of inference using higher-order unification, where the encoding of inference rules generalizes PROLOG's Horn clauses.

The *goal* package manages the proof construction and maintains a proof state which consists of a number of subgoals. Applying a tactic to the current state results in a

lazy stream of new states. The *goal* package presents us with the first state in this stream. We can backtrack either (1) to an ancestor state (by discarding everything done since), or (2) chronologically to examine other solutions in a lazy stream of states.

ISABELLE is distributed with encodings for first-order logic, constructive type theory, classical first-order sequent calculus, and higher-order logic.

λ-Prolog

λ-PROLOG is a similar enterprise for encoding logics using a simple intuitionistic logic as meta-language: in this case the higher-order theory of hereditary harrop formulae. The distinguishing feature of this approach is that it presents itself as an extension of PROLOG with higher-order functions, λ-terms, higher-order unification, modules and secure abstract datatypes [NM88]. It also permits quantification and implication in queries and the bodies of clauses.

Felty [Fel89,FM90] explains how to encode various logics in λ-PROLOG, and describes the implementation of tactic based theorem provers using logic programming techniques.

The Logical Framework

LF is yet another attempt at providing a general framework for encoding logics, but, instead of simple intuitionistic HOL used by ISABELLE and λ-PROLOG, it is based on a dependent-type λ-calculus which makes it more like AUTOMATH.

Again, the idea is that the user need only specify the logic and automatically obtains an editor that provides parsing and pretty printing, rules, tactics, definitions, abbreviations, libraries etc...

The language is a three-level λ-calculus with Π-types that distinguishes objects, types and families of types, and kinds. A logic is specified by its signature which includes declarations of constants and rules (constants of higher-order types).

Felty has shown that LF and λ-PROLOG have essentially the same expressive power [Fel89, FM90].

The Environment for Formal Systems

Griffin's EFS [Gri87b] is an interactive environment, implemented with the Cornell Synthesizer Generator, which supports both Edinburgh LF and the Calculus of Constructions.

It is heavily inspired by NUPRL and borrows its notion of refinement rules, but it also pursues and extends Reps and Alpern's idea of using an attribute grammar to manage proof construction and maintain proof consistency through incremental editing operations.

The environment permits the interactive definition of the syntax and rules of a formal system, of abbreviations, supports both top-down and bottom-up styles of proof, allows user-defined refinement rules, and has a facility for carrying out $\beta\eta\delta$-reductions.

Unfortunately, the EFS lacks a programmable meta-language à la ML, and, therefore, does not allow the user to write general tactics.

Elf

ELF is Pfenning's [Pfe90] endeavour to produce a general meta-language that unifies logic definition in the style of LF with logic programming as in λ-PROLOG. This is

realized by giving types an operational interpretation much like PROLOG does with Horn clauses.

ELF is based on the $\lambda_{\Pi\Sigma}$ calculus which extends LF's λ_{Π} with a Σ type constructor, and exploits Elliott's unification algorithm fora λ-calculus with dependent types [Ell89].

Signatures are transformed and indexed by type families to allows fast PROLOG-style backchaining. ELF's search process automatically constructs representations of object-logic proofs; this is in contrast with other logic programming languages where either these proof are implicit (PROLOG call tree) or it is the responsibility of the programmer to construct them explicitly (as Felty did in λ-PROLOG).

ONTIC

McAllester's ONTIC [McA87] is a system for verifying mathematical arguments. In spirit, it has a mandate similar to that of AUTOMATH: a mathematical argument can be expressed in a very high-level fashion, and the system automatically verifies its validity and supplies the missing parts.

ONTIC relies on forward-chaining. In order to avoid combinatorial explosion, this process is guided by user-specified focus objects. Mathematical proofs often contain lines stating: *"let A be a τ"* where τ is a type, e.g. a mathematical concept such as a set or a group. In ONTIC, the user proceeds similarly by entering the command `(let-be` A τ`)`. ONTIC's database consists of statements about generic objects: the previous command equates A with an unused generic object of type τ.[1]

ONTIC is implemented with Truth Maintenance techniques in the spirit of McAllester's ealier work [McA78,McA80]. The mathematical library is compiled into a graph with nodes for terms, formulae, type expressions, etc... All the inference mechanisms used

[1]If they are all in use, a new generic object is created, and everything known about objects of type τ is duplicated for it.

in ONTIC manipulate labellings of this graph structure. A truth labelling indicates what follows from the current assumptions. A color labelling reflects the current equivalence relation (e.g. due to inferred or assumed equalities): nodes colored similarly are considered equal. The labelling is computed by congruence closure (which replaces unification).

ONTIC has a mechanism for automatic generalization: if g is a generic individual of type τ, and $\Phi(g)$ is labelled true, and no assumptions have been made about g other than it is an instance of type τ, then $\forall x{:}\tau\, \Phi(x)$ can be labelled true as well. This mechanism is more limited than LOGICALC's, in particular because it requires that the formula $\forall x{:}\tau\, \Phi(x)$ be already compiled in the graph structure.

11.3 Future Work

Graphical Interface

Currently, LOGICALC only supports TTY-like interaction. We intend to develop a full-blown graphical interface that will allow mouse-driven interaction. For example, the user must frequently examine the current display, choose an item representing another (presumably neighbouring) node in the graph, and issue a command to move the focus of attention to it. It should be possible to effect these motions by clicking.

Another issue is the proper interfacing of plan generators with the graphical front-end. For example, equality substitution requires that the user indicate which subterm, in the current goal, is to be replaced. LOGICALC provides a powerful and concise notation for specifications of these kind, but some users will prefer to effect the selection using the mouse on the current goal display.

Finally, it would be desirable to have a graphical representation of the graph (of goals and plans, etc...) constructed so far. Such a map would make it easier to visualize

the current state of the proof and would help remind the user of the big picture. Furthermore, the current path from the root (as embodied in the browser's stack) should be highlighted and large motions in the graph could be effected by clicking.

Higher-Order Framework

At present, LOGICALC operates in an extended first-order framework. λ-terms are tolerated as a convenient notational device, but they do not rest on solid ground. Moving to a higher-order framework, such as the λ_Π calculus, will provide a firm theoretical foundations for λ-terms. Furthermore, β-reductions will no longer have to be carried out by hand as it is the case now. We anticipate no difficulty in replacing our unification algorithm (which already may return more than one unifier) with a higher-order procedure such as Huet's or Elliott's version for dependent types.

Graduation to a higher-order framework will let us investigate LOGICALC's adequacy for mathematical research and algorithm development.

Detaching

A legitimate question—even more so now that we have proposed to port LOGICALC to a higher-order framework—is: *"would the architecture and the mechanisms proposed and implemented in* LOGICALC *be equally useful (or even meaningful) for other frameworks?"* First, I will consider the case of "detaching."

The basic idea of the "detaching" mechanism is quite simple. Certain refinements require premises. These premises may not be available as hypotheses and may have to be inferred. The point of "detaching" is to (1) automatically determine the plan required to infer the premises, and (2) merge it with the plan resulting from the requested refinement. The principle of this technique does not depend on the formal system used.

In ISABELLE, the process of proof search is carried out by stringing inference rules together using higher-order resolution. In such a system, detaching can be viewed as the extraction of derived inference rules by exhaustive inspection of a formula. It is trivial to write a tactic that returns the list of all the derived rules that can be extracted from a given formula. For example, if we are given a theorem `foo` whose conclusion is a conjunction, we can produce a derived rule whose conclusion is the first conjunct by executing `(conjunct1 RES foo)`. Similarly, if `foo` is an implication, we can produce a derived rule whose conclusion is the consequent by executing `(mp RES foo)`, and that has the implication's antecedent as an additional premise.

The particular transformations available to the detaching process depend on the formal system being used. For example, in order to detach (from) the antecedent of an implication, Modus Tollens must be a (derived) rule of inference.

Generalization

Concerning generalization, the issue is twofold: how can it be done and how can it help (i.e. why bother)? First, let us address the second question. Generalization is beneficial only if proofs derived earlier during an interactive session can be used to solve goals occurring later in this same session. Once again, let us cast this requirement in terms of ISABELLE.

Since proving theorems in ISABELLE consists of producing derived inference rules, the idea would be to record intermediate rules obtained in this manner and use them as templates to be automatically matched against pending goals. Implementing this idea would require an extension to ISABELLE's GOAL package.

What about feasibility of generalization in the first place? The reason we want to generalize is that, when a proof is developed by top-down analysis, every goal is about

specific objects, and the proofs which are found for them are also stated about these objects. It is often the case that these proofs are instances of more general proofs. The point of generalization is to recover this generality after the fact.

If the system maintains a representation of the proof tree, then we may generalize the premises (i.e. leaves of the tree) and play back the inferences recorded in the tree (from the bottom up) in order to produce a generalized conclusion. A method for doing this in the presence of skolemized assumptions was presented in this thesis.

But, if the system (e.g. ISABELLE) does not maintain a representation of the proof tree, are we doomed? No! Instead, we can take advantage of a technique used for PROLOG-based EBG [KCM87, vHB88]. We keep two versions of the current goal: one is the ordinary specific instance, the other one is a skeletal version of it. Every resolution that is carried out on the first one, must be exactly reflected on the other.[2] The idea is to take advantage of the specificity of the first version to guide the search process, while, at the same time, using the second version to record the most general conclusion that can be obtained through the particular sequence of inferences which we have established so far.

Graph Structure

Editing a graph structure is a natural way to engage in top-down goal-oriented theorem proving. LOGICALC not only makes it easy to navigate this graph, but it also puts it to work for you. For example, by grouping goals into equivalence classes and by broadcasting answers along links established between related classes, LOGICALC is able to guarantee that an answer may benefit as many goals as possible. This mechanism in conjunction with proof generalization helps make the most of each answer.

[2] When limited to first-order unification, this is a trivial requirement. Higher-order unification will undoubtedly complicate the matter.

ISABELLE does not explicitly represent the proof tree. At all times, we are presented with a list of all active subgoals (i.e. what would correspond to the fringe of the non-closed branches of the tree). A structured approach such as LOGICALC's goal/plan elaboration is more helpful when working interactively on complex proofs: it maintains the distinction between separate sub-problems and allows the system to recognize and handle recurring subgoals. Also, LOGICALC allows the user to develop concurrently multiple alternative proof plans.

On the other hand, maintaining a large graph structure has the disadvantage that it takes up a lot of memory space, and ISABELLE's simple but homogenous representation facilitates automated processing.

In order to implement the network of links between related classes, we need to be able to determine whether a formula is an instance of another. In a higher-order framework, we can take advantage of Huet's pre-unification procedure (or an extension thereof). In fact, since the distinction between more and less general classes has turned out to be of little practical use, we may simply abolish it and opt for a weaker criterion based on pre-unifiability.

Proof Summaries

One particularly nice feature of LOGICALC is that it is able to provide an edited account of a completed proof. In ONTIC, a proof consists of the sequence of commands issued by the user. A traditional mathematical exposition can often be translated into such a sequence in fairly natural way. However, although you can easily get a shorter account by deleting some commands, you cannot obtain a greater level of detail for non-obvious inferences as there is not record of how they were obtained. Also, ONTIC proofs have no intermediate result, which makes it hard to see where they are going.[3]

[3]This is the reason for the `push-goal` command, which has no effect on the system, but merely annotates the proof for ease of reading.

Since ISABELLE doesn't maintain a proof tree, we have no access to the justification for a newly obtained derived inference rule. Therefore, no summary can be produced. Actually, the proof state records some of this information and could be extended to record more. However, the basic problem is that a tactic does not produce a trace of its activity. It is possible to treat the sequence of tactic applications as some sort of proof, but there is simply not enough information available to produce a more intelligible account.

The techniques and heuristics presented in this thesis for the generation of proof summaries require that theorems be explicitly validated by a proof (tree). On the other hand, in a tactic-driven system, we could incorporate information about which tactic was used to construct a particular (partial) proof to arrive at a better summary. For example, certain tactics may implement decision procedures for simple theories. Often, this is an indication that the corresponding subproof is rather simple and should not be expanded in the summary.

Tactics

A paradigm which has acquired considerable momentum is the use of tactics to aid in the theorem proving task. We have claimed earlier that LOGICALC's plan generators are related to LCF tactics. In this section I wish to investigate further the nature of this relationship, and establish that our extent implementation can accommodate tactics and tacticals for the elaboration of refinement steps.

An LCF tactic is a partial subgoaling function, i.e. it maps a goal to a plan, or else fails (signals an error condition). A plan generator takes some user input and proposes a set of plans for the current goal. As I explained earlier, this task is carried out in three phases: (1) user input is parsed and packaged, (2) for each argument record produced by phase 1, the plan generator's *designer* function constructs a set of plan descriptions, (3) finally these descriptions are realized into actual plans by the system.

Tactics correspond more specifically to the designer functions called in phase 2. Each designer function takes a list of premises and a list of parameters and constructs a list of plan descriptions. Like tactics, they can be composed because input premises are also expressed in the language of plan descriptions. In other words, a designer function combines a list of plan descriptions (steps) to produce one or more larger plan descriptions.

We now substantiate this correspondance by dicussing a prototype implementation of tactics and tacticals that was developed with an afternoon's worth of work.

A **tactic_t** is a function that maps a **plandesc_t** to a list of **plandesc_t**. We do not have the notion of signalling failure because we do not require it. In LCF, when a tactic is inapplicable, or cannot produce a plan for whatever reason, there is no natural value to return instead. If one was chosen arbitrarily, every tactic would have to check for it in its input. It is much more convenient to signal a special error condition and trap it at strategic choice points. In LOGICALC, we return lists of plans; failing is handled simply by returning an empty list.

A **plandesc_t** is more than just an expression in the language of plan descriptions; it is a pair of a plan description and a substitution. A tactic may be applied to a part of a larger plan description. While transforming this subpart, it may also instantiate certain variables. We need the corresponding substitution to instantiate all occurrences of these variables in other parts of the larger plan.

The fact that our tactics return lists of plan descriptions is a generalization of LCF's subgoaling methods, and has the advantage of requiring no special mechanism for handling failures. However, our tacticals must now be prepared to handle lists of **plandesc_t** and map tactics over them.

Also, what I have dubbed 'the language of plan descriptions' consists of three sorts of entities: names of assertions, goal representations, and plan representations (whose

steps are also plan descriptions). Many tactics are meaningful only when applied to goal representations, and should be prepared to check this condition on their input.

The following tactics and tacticals have been implemented:

(**TacPlangen** *designer prems parms*)

Invokes the *designer* function, using the tactic's argument as the current goal (description), *prems* as the premises, and *parms* as the parameters.

(**TacOr** $tac_1 \ldots tac_n$)

Invokes tac_1, and, if it returns an empty list, then tries invoking the next tactic, etc...

(**TacAnd** $tac_1 \ldots tac_n$)

Invokes tac_1, then maps tac_2 over the results, etc...

(**TacMap** *tac*)

Invokes *tac* on the goal representations occurring in the argument of the resulting tactic's. This is meant to facilitate the application of goal-oriented tactics to all subgoals occurring in a plan description.

(**TacRepeat** *tac*)

Applies *tac*, then maps it over the results repeatedly, until no further transformations can be achieved.

(**TacUse** *specs*)

If the tactic's argument is a goal description, attempts to detach it from the *specs*. The *specs* is a list of expressions similar to the argument to a `USE` command. For example, it may be a list of assertion names. It may also include the `-find` option to invoke Find Mode.

TacHyp

If the tactic's argument is a goal description, attempts to match it against

axioms and assumptions. When it succeeds, the goal description is replaced by the name of the corresponding assertion.

In order to allow the use of these tactics, I extended the user interface with the new command `TACTIC`.

`TACTIC` *tac*

Simply invokes *tac* on the current goal. This is much like invoking a plan generator, except that *tac* is an arbitary NISP expression denoting a tactic.

`TACTIC` *tac plangen args...*

This is like '`+PLAN` *plangen args...*', except that tactic *tac* is then mapped over the resulting plan descriptions.

For example, we can turn off automatic goal expansion by setting `auto-expand*` to NIL, and yet obtain the same effect by repeatedly mapping plan generator EXPAND's designer function over plan descriptions using the following tactic:

```
(TacRepeat (TacMap (TacPlangen #'expand-plangen nil nil)))
```

As a matter of fact, by default, the `TACTIC` command will wrap `(TacRepeat (TacMap ...))` around its tactic argument because that is almost always what you want, and doing it automatically saves typing.

In practice, `#'expand-plangen` is applicable to more cases than we really want to transform. The solution is to write a tactic, say `ExpandFilter`, that filters only those goal descriptions we want to expand. The correct tactic to use is then:

```
(TacAnd #'ExpandFilter (TacPlangen #'expand-plangen nil nil))
```

We have shown that the tactic-based paradigm of interactive proof construction can easily be added to LOGICALC. Our tactics operate on plan descriptions which are expressions in the intermediate language used by plan generators. For every plan generator, we can produce a tactic by applying tactical **TacPlangen** to its designer function. Furthermore, by this method, tactics can benefit from other mechanisms provided by LOGICALC, such as interactive help, detaching, and find mode.

An alternative design for LOGICALC would be to designate tactics as the more primitive notion, and to obtain plan generators by composing them with user interface functions.

11.4 Conclusion

In this thesis, we have described a system that was designed for, and is well suited to, the interactive development of proofs in formal AI domains. It is based on an interesting logic which combines the sequent calculus with skolemization and unification, yet implements assumptions sets with an ATMS.

The system is based primarily on the notion of top-down analysis by successive refinements, and embodies this principle in its architecture: proof construction is viewed as graph editing.

We have proposed several mechanisms that take advantage of this architecture to reduce the work involved in the development of a proof. Recurring subgoals are handled by goal classes. Furthermore, every new answer is broadcast along a network of links so as to benefit as many goals as possible.

This last effect is further amplified by the preliminary application of proof generalization. To this end, we proposed a procedure extending EBG to a system of natural deduction with skolemized assumptions.

Our architecture lends itself well to an exploratory style of proof, which is further encouraged by the "detaching" mechanism. The latter is transparently integrated with plan generators and may also be invoked through "find mode." We maintain several indexings into the database to speed up searches.

We have also described a method for the semi-automated preparation of proof summaries. Several examples served to illustrate the system formatting heuristics and editing capabilities.

Finally, we have contributed a first-order axiomatization based on Kuipers' theory of Qualitative Simulation, and we have demonstrated the capabilities of the system as well as its extensibility by deriving two proofs in this formalism.

We have established that our approach is compatible with the use of tactics by exhibiting a prototype implementation of the latter. In the future, we plan to provide a graphical interface and to graduate to a higher-order framework.

Bibliography

[AB70] Robert Anderson and W. W. Bledsoe. A linear format for resolution with merging and a new technique for establishing completeness. *Journal of the Association for Computing Machinery*, 17(3):525–534, July 1970.

[Aba87] Martin Abadi. *Temporal Logic Theorem Proving.* Technical report 1151, Stanford University, March 1987.

[AHM87] Arnon Avron, Furio A. Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, LFCS, Dept. of Computer Science, University of Edinburgh, July 1987.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AINP88a] P. Andrews, S. Issar, D. Nesmith, and F. Pfenning. ETPS user's manual. Technical report, Carnegie Mellon University, June 1988.

[AINP88b] P. Andrews, S. Issar, D. Nesmith, and F. Pfenning. Grader manual. Technical report, Carnegie Mellon, June 1988.

[AINP88c] P. Andrews, S. Issar, D. Nesmith, and F. Pfenning. TPS user's manual. Technical report, Carnegie Mellon University, June 1988.

[AINP88d] P. Andrews, S. Issar, D. Nesmith, and F. Pfenning. TPS3 facilities guide short version. Technical report, Carnegie Mellon University, June 1988.

[AM86a] Martin Abadi and Zohar Manna. Modal theorem proving. Technical Report 1100, Department of Computer Science, Stanford University, May 1986.

[AM86b] Martin Abadi and Zohar Manna. A timely resolution. Technical Report 1106, Department of Computer Science, Stanford University, April 1986.

[And76] Peter B. Andrews. Refutations by matings. *IEEE Transactions on Computers*, (C–25):801–807, 1976.

[And81] Peter B. Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, April 1981.

[And86] P.B. Andrews. *An Introduction To Mathematical Logic And Type Theory: To Truth Through Proof.* Computer Science And Applied Mathematics. Academic Press, Department of Mathematics, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1986.

[And88] Peter B. Andrews. On connections and higher-order logic. Technical Report 30, Department of Mathematics, Carnegie Mellon University, September 1988.

[Bas89] David A. Basin. Building theories in nuprl. In *Logic at Botik '89*, number 363 in Lecture Notes in Computer Science, 1989.

[Bat79] J.L. Bates. *A Logic for Correct Program Development.* PhD thesis, Cornell University Department of Computer Science, 1979.

[BB74] W. W. Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Journal of Artificial Intelligence*, (5):51–72, 1974.

[BB77] A. Michael Ballantyne and W. W. Bledsoe. Automatic proofs of theorems in analysis using nonstandard techniques. *Journal of the Association for Computing Machinery*, 24(3):353–374, July 1977.

[BB78] W. W. Bledsoe and A. Michael Ballantyne. Unskolemizing. Technical Report 41A, Department of Mathematics and Computer Sciences, University of Texas at Austin, July 1978.

[BBD+82] Michael Ballantyne, W. W. Bledsoe, Jon Doyle, Robert C. Moore, Richard Pattis, and Stanley J. Rosenschein. Automatic deduction. Technical Report 937, Department of Computer Science, Stanford University, October 1982.

[BC81] J.L. Bates and R.L. Constable. Definition of micro-prl. Technical Report 82-492, Cornell University Department of Computer Science, October 1981.

[BC82] J.L. Bates and R.L. Constable. Proofs as programs. Technical Report 82-530, Cornell University Department of Computer Science, 1982.

[BC84] J.L. Bates and R.L. Constable. The nearly ultimate pearl. Technical Report 83-551, Cornell University Department of Computer Science, January 1984.

[BC85] J.L. Bates and R.L. Constable. Proofs as programs. *ACM transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[Bib80] W. Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14:243–261, 1980.

[Bib81] Wolfgang Bibel. On matrices with connections. *Journal of the Association for Computing Machinery*, 28(4):633–645, October 1981.

[Bib82] Wolfgang Bibel. A comparative study of several proof procedures. *Journal of Artificial Intelligence*, (18):269–293, 1982.

[Bis67] E. Bishop. *Foundation of Constructive Analysis.* McGraw Hill, 1967.

[BJ80] George S. Boolos and Richard C. Jeffrey. *Computability And Logic.* Cambridge University Press, second edition, 1980.

[Ble77] W. W. Bledsoe. Non-resolution theorem proving. *Journal of Artificial Intelligence*, (9):1–35, 1977.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, 1979.

[BM84] Robert S. Boyer and J. Strother Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the Association for Computing Machinery*, 31(3):441–458, July 1984.

[Bow82] K.A. Bowen. Programming with full first-order logic. In J.E. Hayes, D. Michie, and Y-H. Pao, editors, *Machine Intelligence*, volume 10, chapter 21, pages 421–440. Ellis Horwood, 1982.

[BR79] J.B. Bulnes-Rozas. *GOAL: a goal oriented command language for interactive proof construction.* PhD thesis, Stanford University Department of Computer Science, 1979.

[Bri79] D.S. Bridges. *Constructive Functional Analysis.* Pitman, London, 1979.

[Bur86] R.M. Burstall. Research in interactive theorem proving at edinburgh university. Technical Report ECS-LFCS-86-12, Laboratory for Foundations of Computer Science, October 1986.

[C+86] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Cas86] Ross Casley. A proof editor for propositional temporal logic. Technical Report 1109, Department of Computer Science, May 1986.

[CH85a] T. Coquand and G. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. Technical Report 463, INRIA, France, December 1985.

[CH85b] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. Technical Report 401, INRIA, France, May 1985.

[CH86] Thierry Coquand and Gérard Huet. The calculus of constructions. Technical Report 530, INRIA, May 1986.

[CH87] R.L. Constable and D.J. Howe. Nuprl as a framework for defining logics. constable-howe-87, February 1987.

[Cha70] C. L. Chang. The unit proof and the input proof in theorem proving. *Journal of the Association for Computing Machinery*, 17(4):698–707, October 1970.

[Che76] Daniel Chester. The translation of formal proofs into english. *Journal of Artificial Intelligence*, (7):261–278, 1976.

[Chu56] Alonzo Church. *Introduction To Mathematical Logic*, volume 1 of *Princeton Mathematical Series.* Princeton University Press, 2 edition, 1956.

[CKB85] R.L. Constable, T.B. Knoblock, and J.L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Computer Science and Applied Mathematics. Academic Press, Academic Press, Inc. 111 Fifth Avenue, New York, N.Y. 10003, 1973.

[Coh78] Avra Cohn. High level proof in LCF. Technical Report CSR-35-78, University of Edinburgh, November 1978.

[Con71] R.L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of IFIP Congress, Ljubljana*, pages 229–233, 1971.

[Con83] R.L Constable. Constructive mathematics as a programming logic 1: Some principles of theory. In *Topics in the Theory of Computation.* Foundation of Computation Theory, 1983.

[Coq86] Thierry Coquand. An analysis of girard's paradox. Technical Report 531, INRIA, May 1986.

[CP80] P. T. Cox and T. Pietrzykowski. On reverse skolemization. Technical Report 01, Faculty of Mathematics, University of Waterloo, January 1980.

[CR84] D. Cabbay and U. Reyle. N-Prolog: An extension to Prolog with hypothetical implications i. *Journal of Logic Programming*, 1(4):319–355, 1984.

[CS71] C. L. Chang and J. R. Slagle. Completeness of linear refutation for theories with equality. *Journal of the Association for Computing Machinery*, 18(1):126–136, January 1971.

[CS87] R.L. Constable and S.F. Smith. Partial objects in constructive type theory. Technical Report 822, Cornell University Department of Computer Science, March 1987.

[Dav86] Ernest Davies. A logical framework for solid object physics. draft, September 1986.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, pages 381–392. North-Holland, 1972.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[Des88] Joëlle Despeyroux. THEO: An interactive proof development system. Technical Report 887, INRIA, Sophia-Antipolis, 2004 route des Lucioles, F-06565 Valbonne Cedex, France, August 1988.

[DH86] Vincent J. Digricoli and Malcolm C. Harrison. Equality-based binary resolution. *Journal of the Association for Computing Machinery*, 33(2):253–289, April 1986.

[dK84] J. de Kleer. Choices without backtracking. In *Proc. Fourth National Conference on Artificial Intelligence*, pages 79–85. Morgan Kaufmann, 1984.

[dK87] J. de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28(2):127–162, 1987.

[dK88] J. de Kleer. A general labeling algorithm for assumption-based truth maintenance. In *Proc. Seventh National Conference on Artificial Intelligence*, pages 188–192. Morgan Kaufmann, 1988.

[DM88] D. Duchier and D.V. McDermott. LOGICALC: An environment for interactive proof development. In Lusk and Overbeek [LO88], pages 121–130.

[Doy79] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the Association for Computing Machinery*, 27(4):758–771, October 1980.

[Duc88] Denys Duchier. The LOGICALC manual. Technical Report 660, Yale University, Department of Computer Science, August 1988.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques*, pages 121–136. Springer-Verlag LNCS 355, April 1989.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990.

[EP91] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In *Advanced Language Implementation*, 1991.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, August 1989.

[FLSY74] S. Fleisig, D. Loveland, A. K. Smiley, and D. L. Yarmush. An implementation of the model elimination proof procedure. *Journal of the Association for Computing Machinery*, 21(1):124–139, January 1974.

[FM90] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In G. Huet and G. Plotkin, editors, *First Workshop on Logical Frameworks*, pages 231–244, 1990.

[FW76] Robert E. Filman and Richard W. Weyhrauch. An FOL primer. Technical Report STAN-CS-76-572, Computer Science Department, Stanford University, September 1976.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, (39):176–210, 405–431, 1935.

[Gil70] P. C. Gilmore. An examination of the geometry theorem machine. *Journal of Artificial Intelligence*, 1:171–187, 1970.

[Gil78] David A. Giles. The theory of lists in LCF. Technical Report CSR-31-78, University of Edinburgh, October 1978.

[Gin84] Matthew L. Ginsberg. Counterfactuals. Technical Report 1029, Department of Computer Science, Stanford University, December 1984.

[GL85] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *Journal of the Association for Computing Machinery*, 32(2):280–295, April 1985.

[GMM+77] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. Technical Report CSR-16-77, University of Edinburgh, September 1977.

[GMW77] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. Technical Report CSR-11-77, University of Edinburgh, September 1977.

[GR84] D. Gabbay and U. Reyle. N-Prolog: An extension to Prolog with hypothetical implications. *Journal of Logic Programming*, 1(4):319–355, 1984.

[Gri87a] Timothy G. Griffin. An environment for formal systems. Technical report, Cornell University Department of Computer Science, June 1987.

[Gri87b] Timothy G. Griffin. An environment for formal systems. Technical Report ECS-LFCS-87-34, Laboratory for Foundations of Computer Science, August 1987.

[Har88] Robert Harper. An equational formulation of LF. Technical Report ECS-LFCS-88-67, LFCS, Dept. of Computer Science, University of Edinburgh, October 1988.

[Has87] Laurent Hascoët. Un constructeur d'arbres de preuve dirigé par des tactiques. Technical Report 770, INRIA, Sophia-Antipolis, 2004 route des Lucioles, F-06565 Valbonne Cedex, France, December 1987.

[HC72] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* University Paperbacks. Methuen & Co, Ltd, second edition, 1972.

[Hen79] L. J. Henschen. Theorem proving by covering expressions. *Journal of the Association for Computing Machinery*, 26(3):385–400, July 1979.

[Hen80] Gary G. Hendrix. KLAUS: a system for managing information and computational resources. Technical Note 230, SRI International, October 1980.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Technical Report ECS-LFCS-87-23, Laboratory for Foundations of Computer Science, March 1987.

[Hin55] K.J.J. Hintikka. Two papers on symbolic logic. *Acta Philosophica Fennica*, 8:1–115, 1955.

[Hir88] Haym Hirsh. Reasoning about operationality for explanation-based learning. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 214–220. Morgan Kaufmann, June 1988.

[How87] Douglas J. Howe. The computational behaviour of girard's paradox. Technical Report 820, Department of Computer Science, Cornell University, March 1987.

[HP90] G. Huet and G. Plotkin, editors. *Proceedings Of The First Workshop on Logical Frameworks*, May 1990.

[HR78] Malcolm C. Harrison and Norman Rubin. Another generalization of resolution. *Journal of the Association for Computing Machinery*, 25(3):341–351, July 1978.

[HST89] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structure and representation in LF. Technical Report ECS-LFCS-89-75, LFCS, Dept. of Computer Science, University of Edinburgh, March 1989.

[Hue] G.P. Huet. A mechanization of type theory. *??*, pages 139–146, ??

[Hue72] G.P. Huet. *Constrained Resolution: A Complete Method For Higher Order Logic.* PhD thesis, Case Western Reserve University, August 1972.

[Hue75] G.P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue88] Gérard Huet. A uniform approach to type theory. Technical Report 795, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, February 1988.

[HW74] L. Henschen and L. Wos. Unit refutations and horn sets. *Journal of the Association for Computing Machinery*, 21(4):590–605, October 1974.

[JP76] D.C Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, (3):123–171, 1976.

[Jut77] L.S. Jutting. *Checking Landau's Grundlagen in the AUTOMATH System.* PhD thesis, Eindhoven University, Eindhoven, The Netherlands, 1977.

[Kaj] Norbert Kajler. Building graphic user interfaces for computer algebra systems. In *Design and Implementation of Symbolic Computation Systems*, number 429 in Lecture Notes in Computer Science.

[Kal87] Marc Kaltenbach. Computer representation and animation of mathematical proofs with dynaboard. Technical Report 700, INRIA, July 1987.

[Kau90] Matt Kaufmann. Demo of the Boyer-More theorem prover and some of its extensions. In G. Huet and G. Plotkin, editors, *First Workshop on Logical Frameworks*, pages 299–322, 1990.

[KCM87] Smadar T. Kedar-Cabelli and L. Thorne McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389. Morgan Kaufmann, June 1987.

[Ket84] Jussi Ketonen. EKL — a mathematically oriented proof checker. In R.E. Shostak, editor, *7th International Conference on Automated Deduction, LNCS 170*, pages 65–79. Springer-Verlag, 1984.

[KK71] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Journal of Artificial Intelligence*, (2):227–260, 1971.

[Kle67] Stephen Cole Kleene. *Mathematical Logic.* John Wiley & Sons, Inc., 1967.

[Kon84] K. Konolige. *A Deduction Model of Belief and its Logics.* PhD thesis, Stanford University Department of Computer Science, 1984.

[Kow75] Robert Kowalski. A proof procedure using connection graphs. *Journal of the Association for Computing Machinery*, 22(4):572–595, October 1975.

[Kre86] Christoph Kreitz. Constructive automata theory implemented with the nuprl proof development system. Technical Report 779, Department of Computer Science, Cornell University, September 1986.

[Kui86] Benjamin Kuipers. Qualitative simulation. *Journal of Artificial Intelligence*, 29:298–388, 1986.

[KW84] J. Ketonen and J.S. Weening. EKL—an interactive proof checker user's reference manual. Technical Report 1006, Stanford University, 1984.

[Laf87] Yves Lafont. De la deduction naturelle à la machine catégorique. Technical Report 670, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, May 1987.

[Lee72] Richard C. T. Lee. Fuzzy logic and the resolution principle. *Journal of the Association for Computing Machinery*, 19(1):109–119, January 1972.

[LO84] E.L. Lusk and R.A Overbeek. The automated reasoning system ITP. Technical Report 27, Argonne National Laboratory, 1984.

[LO88] Ewing Lusk and Ross Overbeek, editors. *9th International Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science. Springer-Verlag, May 1988.

[Lov78] Donald W. Loveland. *Automated Theorem Proving: a logical basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland, 1978.

[LPT89] Z. Luo, R.A. Pollack, and P. Taylor. LEGO user manual. Technical report, LFCS, Dept. of Computer Science, University of Edinburgh, 1989.

[LR81] D. W. Loveland and C. R. Reddy. Deleting repeated goals in the problem reduction format. *Journal of the Association for Computing Machinery*, 28(4):646–661, October 1981.

[LS88] J. Lambek and P. J. Scott. *Introduction To Higher Order Categorical Logic*, volume 7 of *Cambridge Studies In Advanced Mathematics*. Cambridge University Press, 1988.

[Mal86] Yonathan Malachi. *Nonclausal Logic Programming*. Technical report 1127, Stanford University, March 1986.

[Man74] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[McA78] David McAllester. A three-valued truth maintenance system. AI Memo 473, MIT, AI Lab, 1978.

[McA80] David McAllester. An outlook on truth maintenance. AI Memo 551, MIT, AI Lab, 1980.

[McA87] D.A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, May 1987.

[McC57] John McCarthy. Situations, actions and causal laws. AI-Memo 1, Artificial Intelligence Project, Stanford University, 1957.

[McD79] D.V. McDermott. Contexts and data dependencies: A synthesis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3):237–246, May 1979.

[McD83] D.V. McDermott. The NISP manual. Technical Report 274, Yale University Department of Computer Science, June 1983.

[McD85] D.V. McDermott. The DUCK manual. Technical Report 399, Yale University Department of Computer Science, June 1985.

[McD88] D.V. McDermott. Revised NISP manual. Technical Report 642, Yale University, Department of Computer Science, August 1988.

[McD89] D.V. McDermott. A general framework for reason maintenance. Technical Report YALEU/CSD/RR 691, Yale University, March 1989.

[MF86] Dale Miller and Amy Felty. An integration of resolution and natural deduction theorem proving. In *5th National Conference on Artificial Intelligence*, pages 198–202. American Association for Artificial Intelligence, August 1986.

[Mil79a] R. Milner. *Edinburgh LCF.* Lecture Notes in Computer Science. Springer-Verlag, 1979.

[Mil79b] R. Milner. Lcf: a way of doing proofs with a machine. In *Proceedings of the eighth symposium on the Mathematical Foundations of Computer Science.* MFCS, 1979.

[Mil87] Dale A. Miller. A compact representation of proof. Technical Report MS-CIS-87-30, Department of Computer and Information Science, University of Pennsylvania, April 1987.

[Mil90] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. Technical Report MS-CIS-90-54, Department of Computer and Information Science, University of Pennsylvania, 1990.

[MKKC86] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. In *Machine Learning*, pages 47–80. Kluwer, 1986.

[ML82] Pier Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI*, pages 153–175. North-Holland, 1982.

[MMN75] R. Milner, L. Morris, and M. Newey. A logic for computable functions with reflexive and polymorphic types. LCF Report 1, Dept. of Computable Science, University of Edinburgh, January 1975.

[Moo74] J. Strother Moore. Introducing iteration into the pure lisp theorem prover. Technical Report 3, XEROX, Palo Alto Research Center, December 1974.

[Moo75] J. Strother Moore. Computational logic: Structure sharing and proof of program properties, part ii. Technical Report 2, XEROX, Palo Alto Research Center, April 1975.

[Moo90] Raymond J. Mooney. *A General Explanation-Based Learning Mechanism and its Application to Narrative Understanding*. Research Notes in Artificial Intelligence. Pitman, Morgan Kaufmann, 1990.

[Mor69] James B. Morris. E-resolution: Extension of resolution to include the equality relation. In Donald E. Walker and Lewis M. Norton, editors, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 287–293. International Joint Conference on Artificial Intelligence, Morgan Kaufmann, May 1969.

[MS74] D. Michie and E. E. Sibert. Some binary derivation systems. *Journal of the Association for Computing Machinery*, 21(2):175–190, April 1974.

[MS82] D.P. McKay and S.C. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the International Joint Conference In Artificial Intelligence*. IJCAI, 1982.

[MSMT88] Toshiro Minami, Hajime Sawamura, Kaoru Minami, and Kyoko Tsuchiya. EIODHILOS: A general-purpose reasoning assistant system. In *Logic Programming '88*, number 383 in Lecture Notes in Computer Science, 1988.

[Mur82] Neil V. Murray. Completely non-clausal theorem proving. *Journal of Artificial Intelligence*, (18):67–85, 1982.

[Mur90] Chet Murthy. *Extracting Constructive Content from Classical Proofs: Compilation and the Foundatiosn of Mathematics.* PhD thesis, Cornell University, Department of Computer Science, 1990.

[MW85] Zohar Manna and Richard Waldinger. Special relations in automated deduction. Technical Report 1051, Department of Computer Science, Stanford University, May 1985.

[MW86] Zohar Manna and Richard Waldinger. Special relations in automated deduction. *Journal of the Association for Computing Machinery*, 33(1):1–59, January 1986.

[Nev74] Arthur J. Nevins. A human oriented logic for automatic theorem-proving. *Journal of the Association for Computing Machinery*, 21(4):606–621, October 1974.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λPROLOG. In *Fifth International Conference Symposium on Logic Programming*, August 1988.

[NS82] Anil Nerode and Richard A. Shore, editors. *Recursion Theory*, volume 42 of *Proceedings of Symposia in Pure Mathematics.* American Mathematical Association, June 1982.

[Ove74] Ross A. Overbeek. A new class of automated theorem-proving algorithms. *Journal of the Association for Computing Machinery*, 21(2):191–200, April 1974.

[Pau86] Lawrence C. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, (3):237–258, 1986.

[Pau87] Lawrence C. Paulson. *Logic And Computation. Interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Pfe84] Frank Pfenning. Analytic and non-analytic proofs. In R.E. Shostak, editor, *7th Conference on Automated Deduction*, pages 394–413. Springer-Verlag, 1984.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163, 1988.

[Pfe90] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In G. Huet and G. Plotkin, editors, *First Workshop on Logical Frameworks*, pages 391–406, 1990.

[Pie73] Tomasz Pietrzykowski. A complete mechanization of second-order type theory. *Journal of the Association for Computing Machinery*, 20(2):333–365, April 1973.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, (5):223–255, 1977.

[PM89] Christine Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions.* PhD thesis, Université Paris VII, 1989. Thèse de Troisième Cycle.

[RA84] Thomas Reps and Bowen Alpern. Interactive proof checking. In *Proceedings of POPL11*, pages 36–45, 1984.

[Rei71] Raymond Reiter. Two results on ordering for resolution with merging and linear format. *Journal of the Association for Computing Machinery*, 18(4):630–646, October 1971.

[Rit87] Brian Ritchie. *The Design and Implementation of an Interactive Proof Editor.* PhD thesis, University of Edinburgh, 1987.

[Rob65] J.A Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, (12):23–41, 1965.

[Rob69] J.A. Robinson. Mechanizing higher-order logic. In B. Meltzer, D. Michie, and M. Swann, editors, *Machine Intelligence*, volume 4, chapter 9, pages 151–173. Elsevier, 1969.

[Rob70] J.A. Robinson. A note on mechanizing higher-order logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, chapter 6, pages 121–134. Elsevier, 1970.

[Rob79] J.A. Robinson. *Logic: Form and Function, The Mechanization of Deductive Reasoning.* Artificial Intelligence Series. Elsevier North Holland Inc., 52 Vanderbilt Avenue, New York, N.Y. 10017, 1979.

[RT81] Thomas Reps and Tim Teitelbaum. The Cornell program synthesizer: A syntax-directed programming environment. *Communications of the Association for Computing Machinery*, 24(9):563–573, 1981.

[RT87] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Department of Computer Science, Cornell University, second edition, 1987.

[RT88] Brian Ritchie and Paul Taylor. The interactive proof editor, an experiment in interactive theorem. Technical Report ECS-LFCS-88-61, LFCS, Department of Computer Science, University of Edinburgh, July 1988.

[RU71] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic.* Number 3 in Library of Exact Philosophy. Springer-Verlag, 1971.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, chapter 8, pages 135–150. Elsevier, 1969.

[SB83] D.T Sanella and R.M. Burstall. Structured theories in LCF. Technical Report CSR-129-83, University of Edinburgh, February 1983.

[Sch83] David Schmidt. Natural deduction theorem proving in set theory. Technical Report CSR-142-83, University of Edinburgh, September 1983.

[Sho76] Robert E. Shostak. Refutation graphs. *Journal of Artificial Intelligence*, (7):51–64, 1976.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, January 1984.

[Sib69] E.E. Sibert. A machine-oriented logic incorporating the equality relation. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, chapter 7, pages 103–133. Elsevier, 1969.

[Sie86] Jörg H. Siekmann, editor. *8th International Conference on Automated Deduction*, number 230 in Lecture Notes in Computer Science. Springer-Verlag, July 1986.

[Sla70] James R. Slagle. Interpolation theorems for resolution in lower predicate calculus. *Journal of the Association for Computing Machinery*, 17(3):535–542, July 1970.

[Sla72] James R. Slagle. Automatic theorem proving with built-in theories including equality, partial ordering, and sets. *Journal of the Association for Computing Machinery*, 19(1):120–135, January 1972.

[Sla74] James R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the Association for Computing Machinery*, 21(4):622–642, October 1974.

[Smu68] R.M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968.

[Sok83a] Stefan Sokolowsky. An LCF proof of soundness of hoare's logic – a paper without a happy ending. Technical Report CSR-146-83, University of Edinburgh, October 1983.

[Sok83b] Stefan Sokolowsky. A note on tactics in LCF. Technical Report CSR-140-83, University of Edinburgh, August 1983.

[Ste84] Guy L. Steele Jr. *Common Lisp, The Language*. Digital Press, 1984.

[Ste90] Guy L. Steele Jr. *Common Lisp, The Language*. Digital Press, 2nd edition, 1990.

[Sti73] Rona B. Stillman. The concept of weak substitution in theorem-proving. *Journal of the Association for Computing Machinery*, 20(4):648–667, October 1973.

[Sti82] Mark E. Stickel. A nonclausal connection-graph resolution theorem-proving program. Technical Note 268, SRI International, October 1982.

[Sti83] Mark E. Stickel. Theory resolution: Building in nonequational theories. Technical Note 286, SRI International, May 1983.

[Sup60] Patrick Suppes. *Axiomatic Set Theory*. D. Van Nostrand Company, Inc., 1960.

[Tar89] Mark Tarver. DIALOG: A theorem-proving environment designed to unify functional and logic programming. Technical Report ECS-LFCS-89-80, Laboratory for Foundations of Computer Science, May 1989.

[Tay88] Paul Taylor. Using constructions as a metalanguage. Technical Report ECS-LFCS-88-70, Laboratory for Foundations of Computer Science, December 1988.

[Tay89] Paul Taylor. Playing with LEGO: Some example of developing mathematics in the calculus of construction. Technical Report ECS-LFCS-89-89, LFCS, Dept. of Computer Science, University of Edinburgh, August 1989.

[TMM88] Paul B. Thistlewaite, Michael A. McRobbie, and Robert K. Meyer. *Automated Theorem-Proving in Non-Classical Logics.* Research Notes in Theoretical Computer Science. Pitman (also: John Wiley & Sons), 1988.

[vHB88] Frank van Harmelen and Alan Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36:401–412, 1988.

[Wal88] Christoph Walther. Many-sorted unification. *Journal of the Association for Computing Machinery*, 35(1):1–17, January 1988.

[War82] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In J.E. Hayes, D. Michie, and Y-H. Pao, editors, *Machine Intelligence*, volume 10, chapter 22, pages 441–454. Ellis Horwood, 1982.

[Wat89] Richard C. Waters. XP: A common lisp pretty printing system. AI Memo 1102, MIT, AI Lab, 1989.

[Wey78] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. Technical Report STAN-CS-78-687, Computer Science Department, Stanford University, December 1978.

[Wey80] Richard W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Journal of Artificial Intelligence*, (13):133–170, 1980.

[Win82] Steve Winker. Generation and verification of finite models and counterexamples using an automated theorem prover answering two open questions. *Journal of the Association for Computing Machinery*, 29(2):273–284, April 1982.

[wL72] Donald w. Loveland. A unifying view of some linear herbrand procedures. *Journal of the Association for Computing Machinery*, 19(2):366–384, April 1972.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning. Introduction And Applications.* Prentice-Hall, 1984.

[Wos88] L. Wos. *Automated Reasoning: 33 Basic Research Problems.* Prentice Hall, 1988.

[WWS+84] L. Wos, S. Winker, B. Smith, R. Veroff, and L. Henschen. A new use of an automated reasoning assistant: Open questions in equivalential calculus and the study of infinite domains. *Journal of Artificial Intelligence*, (22):303–356, 1984.

Appendix A

The Language

In this appendix, I describe the language for writing axiomatizations. I explain the syntax of propositions and terms, review types and type declarations, and explain how to define axioms.

A.1 Terms and Propositions

LOGICALC expects terms and propositions to be written in a LISPish (i.e. prefix fully parenthesized) notation familiar to most everyone in AI. The table below summarizes the correspondance between traditional logic notation and LOGICALC expressions for connectives and quantifiers:

$\neg p$	(NOT p)
$p_1 \wedge \ldots \wedge p_n$	(AND p_1 ... p_n)
$p_1 \vee \ldots \vee p_n$	(OR p_1 ... p_n)
$p \supset q$	(IF p q)
$(\forall x_1) \ldots (\forall x_n)\ P(x_1, \ldots, x_n)$	(FORALL (x_1 ... x_n) (P x_1 ... x_n))
$(\exists x_1) \ldots (\exists x_n)\ P(x_1, \ldots, x_n)$	(EXISTS (x_1 ... x_n) (P x_1 ... x_n))
$\lambda x_1 \cdots x_n\ t(x_1, \ldots, x_n)$	(LAMBDA (x_1 ... x_n) (t x_1 ... x_n))

Because our logic is essentially first-order, free variables and lambda terms are not allowed to occur in functional position. Instead, their application must be handled indirectly using the '%' notation. For example:

```
(% (LAMBDA (X) (+ 1 X)) FOO)
```

An expression of the form `(% (lambda (`*args...*`) ...)` *parms...*`)` is called a redex and the functional application which it denotes can be effected by invoking the rule of β-reduction, provided there are as many actual parameters as formal parameters. Thus, the above example becomes `(+ 1 FOO)`.

A.2 Read Macros

Notational conventions can be greatly enhanced or simplified through the judicious use of "read macros." See e.g. [Ste84] for a discussion of read-macros in COMMON LISP.

Free Variables: Variables to the unifier are prefixed with a question mark. Thus, a free variable named `FOO` is written `?FOO`, and is internally represented by the expression `(\? FOO)`.

Skolem Terms and Abbreviations: Skolem terms are constructed and introduced by the system during skolemization (see Section 2.2). A skolem term is represented by an expression of the form:

`(sk` *fun id args...*`)`

and prints as `!.`*fun* (see Section 4.4 for further details). Similarly, the prefix '`!.`' is defined to act as a read-macro on input: `!.`*fun* turns into (`sk` *fun id args...*) (ibidem).

LOGICALC also uses skolem terms to implement a general abbreviation mechanism documented in Section 4.4.

Lists and Tuples: It would be tempting to represent lists as LISP lists. Unfortunately, this scheme would not distinguish between a list whose first element is `FOO` from an expression whose *functor* (i.e. main symbol) is `FOO`.

Obviously, we must introduce a new function to help us represent lists. The choice is essentially between CONS and LIST.[1] PROLOG chooses to go the 1st way and uses the two-argument functor '.'. DUCK chooses the other alternative, and uses the function `TUP` to represent lists, which are also called 'tuples.' The expression:

```
(TUP FOO (BAR X) BAZ)
```

denotes a 3-tuple, i.e. a list of 3 elements. LOGICALC provides the square bracket read-macro (and print method) to make tuples visually more appealing. The nicer syntax for the above example is:

```
[FOO (BAR X) BAZ]
```

Segment Variables: In PROLOG, it is possible to denote the tail of a list using the vertical bar notation. For instance, the term `[X|L]` denotes a list with `X` as its first element and `L` as its tail (i.e. remainder of the list). In LOGICALC, the corresponding

[1] Or LIST*.

idiom is [?X !?L]. The term !?L denotes a *segment variable*, which is a generalization of a 'tail' variable: it stands for an unspecified number of adjacent elements.

Segment variables are a rather convenient device when writing logic programs. For example, here is how the MEMBER predicate might be defined:

```
(DEFPRED (MEMBER ?X - obj ?L - (LST obj))
  (MEMBER ?X [!?L1 ?X !?L2]))
```

When a segment variable becomes instantiated with a tuple, that tuple is 'spliced' into the list where the variable occurred. Thus, when ?X is bound to [B C D], [A !?X E] becomes [A B C D E].

Segment variables may cause the unifier to find more than one solution; see Section 2.6.2.

Sets by Enumeration: Sets and lists are distinct concepts both representing collections of elements. For many purposes it is convenient to think of a set as the list of its elements. Lists are computationaly more attractive because they have structure; in particular, it is trivial to recursively enumerate the elements of a list. A set which is specified by enumeration (i.e. by listing its elements) can be written using curly braces. For instance:

```
{FOO (BAR X) BAZ}
```

denotes a set with three elements and is internally represented by the expression:

```
(LSET (TUP FOO (BAR X) BAZ))
```

The function LSET takes a list (a tuple) as an argument and denotes the set consisting of the elements of the list.

A.3 Syntax Macros

It is often convenient to be able to define new syntactic constructs for greater conciseness and perspicuity in the specification of axioms, without having to extend the logic with new connectives or other operators.

For example, LOGICALC defines "if and only if" as a macro which expands into the conjunction of two implications, one in each direction. It is implemented as follows:

```
(DATAFUN SYNMACRO IFF
  (DEFUN (E)
    `(AND (IF ,(CADR  E) ,(CADDR E))
          (IF ,(CADDR E) ,(CADR  E)))))
```

The reader unfamiliar with the backquote notation should consult [Ste84]. The DATAFUN construct is a NISP idiom documented in [McD88], and, here, has the effect of defining the procedure IFF-SYNMACRO as described by the defun, and storing it on property SYNMACRO of symbol IFF.

In Chapter 2, I explained that expressions are automatically syntax/type checked by LOGICALC. Syntax macros are also expanded during this process.

LOGICALC defines a few other macros: XOR for exclusive or, \\ for LAMBDA, and ELT for expressions like (ELT X L) that expands into the less perspicuous (% L X).

A.4 Types

By default,[2] DUCK makes sure that assertions (and other expressions it is given as input) are well-typed. There are two reasons for doing so.

[2]When the global variable SYNCHK* is non NIL.

Firstly, it is easy to make mistakes when editing an axiomatization; such as to mistype the name of a variable or constant, or to forget an argument to a predicate or function. DUCK's type-checking facility will catch these lapses.

Secondly, type declarations are often meant to restrict the scope of the axiom or theorem in which they occur. For example, when we say $\forall n{:}\mathbb{N}\ P(n)$, we mean that *if n is a natural number, then P holds of it.* Therefore, type declarations (e.g. $n{:}\mathbb{N}$) give rise to type constraints, of the form (IS *type object*), which must be combined with the text of the formula in which they are located.

A.4.1 Type Designators

Types are complex objects—the reasons for this complexity will not be explained here. However, they can be denoted simply by type designators. For example, `integer`, `boolean`, `string` denote the obvious types. The reader should consult the NISP manual [McD88] for an exhaustive list of primitive types. Types are arranged in a lattice: thus `integer` is a subtype of `number`. At the top of the lattice is the universal type `obj`: everything is an `obj`. At the bottom of the lattice is the empty type `bottom` which is not inhabited. `prop` is the type of propositions.

More complex types can be obtained with type constructors. Again the reader should consult the NISP manual. Here, we shall simply review a few type constructors of particular interest for writing axiomatizations.

(FCN t_0 [t_1 ... t_n])

denotes the type of functions that map n arguments of types t_1 through t_n to an expression of type t_0. For example, if `NTH` is declared to be of type `(FCN ?T [integer (LST ?T)])`, where `?T` is a type variable, and `L` is declared to be of type `(LST string)`, then `(NTH 3 L)` is syntactically acceptable and denotes a `string`.

(PRD [t_1 ... t_n])

denotes the type of predicates that take n arguments of types t_1 through t_n.

(LST t)

denotes the type of lists whose elements are of type t.

(LRCD t_1 ... t_n)

denotes the type of n-tuples whose elements are respectively of types t_1 through t_n.

(SET_OF t)

denotes the type of sets whose elements are of type t.

(EITHER t_1 ... t_n)

denotes the union of types t_1 through t_n.

The subtype relation is extended to these types, sometimes in an ad-hoc fashion to preserve decidability or for efficiency. For example, (FCN b [a]) is a subtype of (FCN b' [a']) iff a' is a subtype of a and b is a subtype of b'. (LRCD a a') is a subtype of (LST b) if both a and a' are subtypes of b. Etc...

Defining new type constructors is beyond the scope of the present exposition, but declaring new type constants is easy:

(DEFDUCKTYPE *name descriptor*)

defines the new type constant *name*, and declares it to be a subtype (in the sense of the lattice) of the type denoted by *descriptor*. Most often, the descriptor is simply `obj`.

A.5 Type Declarations

Type declarations are of two sorts: those that declare global constants, functions, and predicates, and those that declare bound variables in formulae.

A.5.1 Global Type Declarations

Global declarations are performed with the following special forms:

(DUCLARE *names-and-types...*)

Where *names-and-types* are names of new constants interspersed with type designators. A type designator follows the names to which it applies and is separated from them by a dash '-'. For instance:

```
(DUCLARE FOO BAZ - integer BAR - (FCN integer [(LST obj)]))
```

declares FOO and BAZ to be constants of type integer and BAR to be a function that maps a list of anything to an integer (e.g. it might be the length of the list).

(DEFFUN [*type*] (*name vars-and-types...*))

The first (optional) *type* is the function's "return" type; *name* is the name of the function constant. *vars-and-types* are free variables interspersed with type declarations. The free variables serve only as place holders for the corresponding arguments. The following statements are equivalent and both define BAR as a function mapping a list into an integer:

```
(DEFFUN integer (BAR ?X (LST obj)))
(DUCLARE BAR - (FCN integer [(LST obj)]))
```

(DEFPRED (*name vars-and-types*))

name is the name of the predicate constant, and *vars-and-types* is as above. The following two statements are equivalent and both define FOO as a predicate that takes an integer and a list as its two arguments:

```
(DEFPRED (FOO ?X - integer ?L - (LST obj)))
(DUCLARE FOO - (PRD [integer (LST obj)]))
```

A.5.2 Local Type Declarations

Type declaration may also appear in the binding lists of FORALL, EXISTS, and LAMBDA, and have the form of names (introducing new bound variables) interspersed with types. For example:

```
(AXIOM STOP-NEW-DOWN-NEXT
  (FORALL (S - i-situation Q - !!quantity L1 L2 - !!landmark
           LEFT RIGHT - !!(LST landmark))
    (IF (AND (DURING S (STATE Q DEC [L1 L2]))
             (DURING S (LANDMARKS Q LEFT RIGHT))
             (DURING S (STOP-NEW Q)))
        (EXISTS (L - landmark)
          (AND (DURING (NEXT S) (STATE Q STD [L]))
               (DURING (NEXT S) (LANDMARKS Q LEFT [L !?RIGHT])))))))
```

A.5.3 Controlling Type Constraints

The above example introduces another read-macro, namely '!!', which can be used as a prefix to a type designator in a binding list. The corresponding declarations play their normal role during type-checking, but do not generate type constraints to be added to the formula. Thus, the above formula is transformed (prior to skolemization) into:

```
(FORALL (S Q L1 L2 LEFT RIGHT)
  (IF (AND (IS i-situation S)
           (DURING S (STATE Q DEC [L1 L2]))
           (DURING S (LANDMARKS Q LEFT RIGHT))
           (DURING S (STOP-NEW Q)))
      (EXISTS (L)
        (AND (IS landmark L)
             (DURING (NEXT S) (STATE Q STD [L]))
             (DURING (NEXT S) (LANDMARKS Q LEFT [L !?RIGHT]))))))
```

Thanks to the '!!' type-designator prefix, the user can have the benefit of definition-time type-checking, without necessarily burdening his formulae with superfluous type constraints. For example,

```
(DURING S (STOP-NEW Q))
```

can only be derived if Q is a quantity. Therefore there is no need to add a type constraint of the form:

```
(IS quantity Q)
```

which is why Q was declared with !!quantity, rather than just quantity.

There are two other ways in which to control whether type constraints are contributed. First, setting the variable PROVE-DECLARATIONS* to NIL completely inhibits these contributions. Secondly, if a type possesses the NOCHECK feature, then it never contributes any type constraint. For example:

```
(!= (TYPE-FEATURE 'integer 'NOCHECK) '#F)
```

sets the NOCHECK property of type integer.

A.5.4 Quantifying Over Types

It is sometimes convenient to quantify over types and allow occurrences of type variables. For example:

```
(FORALL (TY - type)
  (FORALL (L - (LST ?TY))
    (IS (SET_OF TY) (LSET L))))
```

Thus, type designators may really be arbitary terms which happen to denote types.

A.5.5 Type Declarations And Lambda Expressions

We have stated earlier that the binding list of a λ-expression may consist of variables interspersed with type designators. Also, we have explained that such declarations normally contribute type constraints. Whereas it is fairly clear, in the case of quantifiers, how to combine these constraints with the formula in which they occur, it is less clear what to do with them in the case of a λ-expression.

There are two cases: either the λ-expression denotes a predicate or it is a term.

Predicate Lambda

Let $F(x, y)$ be a formula containing x and y as free variables. Then the expression $\lambda x, y \cdot F(x, y)$ denotes the set of pairs $\langle x, y\rangle$ that satisfy $F(x, y)$. Clearly, type declarations are intended to restrict the scope of this condition. Therefore, they should simply be conjoined to $F(x, y)$. Thus:

```
(LAMBDA (X - number Y - (LST number)) (F X Y))
```

must be interpreted as restricting X to be a number and Y a list of numbers, and should be transformed into:

```
(LAMBDA (X Y) (AND (IS number X) (IS (LST number) Y) (F X Y)))
```

Term Lambda

Let $T(x,y)$ be a term containing x and y as free variables. Then the expression $\lambda x, y \cdot T(x,y)$ denotes the set of terms of the form $T(a,b)$ where a and b are terms substituted for variables x and y. Clearly, type declarations are meant to restrict the admissible terms a and b. Unfortunately, our logic does not provide a means to represent such a restriction. At present, we simply drop these type constraints in the bit bucket.

A.6 Constraints In Binding Lists

Not only may binding lists containt variables interspersed with type designators, but they may also contain constraints. A constraint is an expression where the new bound variables are identified by a '??' prefix. For example:

```
(AXIOM P-TRANSITIONS-STD
  (FORALL ((REACHABLE ??S) - p-situation Q - quantity V - !!value)
    (IF (DURING S (STATE Q STD V))
        (OR (DURING S (CONTINUE   Q))
            (DURING S (START-UP   Q))
            (DURING S (START-DOWN Q))))))
```

The universal quantifier introduces the new bound variable S, with the additional constraint that S must be REACHABLE. Such a constraint is combined with the formula in the same manner as type constraints.

It is possible to declare several variables in parallel in the same constraint by following the '??' prefix with a list of names rather than a single name. The constraint is duplicated for each name. Thus the following formulae are equivalent:

```
(FORALL (X - number) (EXISTS ((< ??(A B) X) - number) (< A B)))
(FORALL (X - number) (EXISTS (A B - number) (AND (< A X) (< B X) (< A B))))
```

In a lambda binding list, it is permissible for some constraints to denote terms rather than a propositions. This extension was motivated by the desire to make set theory easier to express. For instance, the set $\{f(x) \mid x \in \mathbb{N} \wedge P(x)\}$ can be written:

```
(LAMBDA ((F ??X) - integer) (P X))
```

and is equivalent to:

```
(LAMBDA (Z) (EXISTS (X - integer) (and (= Z (F X)) (P X))))
```

A.7 Axioms And Rules

The form (`AXIOM` *name formula*) asserts a new axiom in the database. This axiom can be referred to by its *name*. You may introduce new axioms at any time; this is very convenient when, in the process of proving a theorem, you realize that your axiomatization is not sufficiently powerful or that you forgot to formalize some aspect of your application domain.

You may even redefine an existing axiom. However, you should not do so if this axiom has already been used in the proof in progress.

Back-chaining rules (i.e. axioms whose main connective is `<-`) may be used to write logic programs. Typically, these are meant to implement decision procedures to be invoked automatically on newly created goals (or rather, classes). For example:

```
(AXIOM LSET-SIMPLE-DEF
  (<- (ELT ?X (LSET ?L)) (MEMBER ?X ?L)))
```

The DUCK manual [McD85] documents the embedded logic programming language.

Related axioms can be defined in groups using the form:

(ASSERTION-GROUP *name formulae...*)

Each *formula* is given a name of the form *name-k*, where *k* is an integer (*name-1*, *name-2* ...). Thus,

```
(ASSERTION-GROUP OBVIOUSLY-REACHABLE
  (<- (REACHABLE ?S) (INITIAL ?S))
  (<- (REACHABLE (NEXT ?S))
      (AND (IS p-situation ?S)
           (REACHABLE ?S))))
```

defines assertion group `obviously-reachable` to consist of axioms `obviously-reachable-1` and `obviously-reachable-2`.

Some axioms are not especially intersting from the point of view of the reader of a proof. You may inform LOGICALC of this fact in advance with the form (`hide-from-lc` '*name*). For example:

```
(HIDE-FROM-LC 'LSET-SIMPLE-DEF)
```

has the effect that the axiom `lset-simple-def` is never mentionned is the summary of a proof where it occurs. The idea is that what it does is too trivial to require justification and would only clutter the summary.

In the case of an `assertion-group`, the same procedure must be repeated for each axiom in that group. Fortunately, there is a slightly easier way than tedious enumeration. For example, all axioms in group `obviously-reachable` can be hidden as follows:

```
(<\␣ hide-from-lc (prop 'assertion-group 'obviously-reachable))
```

The `<\␣` special form is like `mapc` in COMMON LISP, and, yes, the second character of its name is a space, which is why it must be escaped.

Appendix B

Find Mode

Certain plan generators, such as Modus Ponens, require one or more premises as arguments. Instead of such a premise, the user may supply the `-find` option. As a result, an interactive sub-mode is entered which provides help in searching the database and in selecting detachments. When this sub-mode is exited, the selected detachments are used in place of the missing argument.

Other plan generators, such as Equality, require that the user select one or more subterms in the current goal. Specifying the `-find` option has the effect of activating a sub-mode which provides help in looking for and selecting subterms. When this sub-mode is exited, the path to the selected subterms are used in place of the missing argument.

B.1 Generic Find Mode

There are currently three sub-modes offering the kind of help described above. Since they are so closely related, they are implemented as specializations of a generic utility

that provides the kernel functionality and basic commands. Here is a description of these commands:

`QUIT`

Exit from Find Mode and return all selected candidates to the caller.

`GO`

Initiate a new search for candidates. The procedure that generates the stream of candidates is called anew. This command is typically useful after the user has somehow altered the context of the search (e.g. by changing the view).

`NEXT`

From the stream of candidates, the system only display the current one. The `NEXT` command moves on to the next candidate, if any.

`TAKE`

`TAKE -ALL`

Save current candidate in the list of selected candidates to be returned to the caller upon exit. With option `-ALL`, take all remaining candidates.

`TAKE -SHOW` $i_1 \ldots i_n$

`TAKE -SHOW -ALL`

Display selected candidates i_1 through i_n (resp. all of them).

`TAKE -OUT` $i_1 \ldots i_n$

`TAKE -OUT -ALL`

Discard selected candidates i_1 through i_n (resp. all of them).

`OK`

Equivalent to `TAKE` followed by `QUIT`.

`PP`

Redisplay current candidate.

B.2 Formula Find Mode

Formula Find Mode provides an interface to the detaching procedure described in Chapter 7. It can be invoked by typing `-find` where a premise was expected. This option may also be followed by a *view* consisting of a sequence of assertion names. In that case, the system will only attempt to detach from the *view*. When the *view* is not specified, it defaults to the whole database (as if the user had typed `-find -all`).

For each candidate detachment, the system displays the detached formula, the auxiliary goals required by this detachment, and show the original assertion with the occurrence of the detached formula highlighted.

Here are the additional commands available in Formula Find Mode:

`VIEW`

Display the current view. This may be a list of assertions, or the whole database (in which case the system simply prints '`the database`').

LOGICALC also allows detachments from conjectures: the '= *formula*' specification sets up *formula* as an auxiliary goal and performs detaching from it. When such a conjecture occurs in a view, the system prints '`conclusion from auxiliary goal`'.

`VIEW -SET` *specs...*

Set the view according to the *specs* which consists of names of assertions and expressions of the form '= *formula*'. If there are no *specs*, the view is set to the whole database.

`PATTERN`

Display the pattern of the formula which we are trying to detach from the view.

`PATTERN` *formula*

If the context allows it, set the pattern to be detached to the given *formula*.

B.3 Subterm Find Mode

Subterm Find Mode provides an interface to the subterm facility described in Section 6.2.2. It can be invoked by typing `-find` where a subterm argument was expected. This option may be followed by subterm descriptors, in which case they will provide the initial stream of candidates. Here are the additional commands available in this mode:

`PATH`

Display the subterm descriptors which currently serve to generate the stream of candidates.

`PATH` *descs...*

Change these descriptors to be the given *descs*.... The user should then invoke `GO` to generate a new stream of candidates from these *descs*. Usually, the user can simply type *descs* and omit the `PATH` command altogether.

B.4 Target Find Mode

Occasionally, the context constrains the shape of admissible subterms. For example, β-reduction applies only to redexes, i.e. expressions of the form `(% (LAMDA` *args body*`)` *parms...*`)`. In such a case, Target Find Mode is invoked instead of Subterm Find Mode. There are no additional commands for this mode. The initial stream of candidates is obtained by enumerating all subterms that agree with the specified

shape. The mode of agreement may be `EQUAL`, `UNIFY`, or `SUBSUME`, and is displayed in the prompt.