THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

The Design and Implementation of an Interactive Proof Editor

Brian Ritchie

Ph. D.

University of Edinburgh

1987

Abstract

This thesis describes the design and implementation of the IPE, an interactive proof editor for first-order intuitionistic predicate calculus, developed at the University of Edinburgh during 1983–1986, by the author together with John Cartmell and Tatsuya Hagino. The IPE uses an attribute grammar to maintain the state of its proof tree as a context-sensitive structure. The interface allows free movement through the proof structure, and encourages a "proof-by-experimentation" approach, since no proof step is irrevocable.

We describe how the IPE's proof rules can be derived from natural deduction rules for first-order intuitionistic logic, how these proof rules are encoded as an attribute grammar, and how the interface is constructed on top of the grammar. Further facilities for the manipulation of the IPE's proof structures are presented, including a notion of IPE-tactic for their automatic construction.

We also describe an extension of the IPE to enable the construction and use of simply-structured collections of axioms and results, the main provision here being an interactive "theory browser" which looks for facts which match a selected problem.

At the age of fourteen my father was forced to leave school and "go down the mines" to support his family. Despite a promising scholastic performance, short-term needs outweighed the long-term academic investment. His later attempts to obtain qualifications were made difficult by having to study in addition to long hours of heavy physical labour. Though commended on his work, the strain became too great and his studies were abandoned. Nonetheless he retained an interest in scientific developments and provided a stimulating environment for his children. My mother and he determined that their children would be free to achieve their potential, at a time when the destiny of most miners' sons was to work alongside their fathers. Therefore it is only right (and I don't care how outré it is) that I should dedicate this thesis to my parents.

Table of Contents

1. Introduction 1

1.1 General Description 1

1.2 Acknowledgements 2

1.3 Overview 3

1.4 Machine-Assisted Proof Systems 3

1.5 A Demonstration 8

2. The Generation of Basic Tactics for Interactive Proof 24

2.1 Introduction 24

2.2 Inference Rules of the IPE 26

2.3 Derivation of Basic Tactics 29

2.3.1 Tactic Schemata 29

2.3.2 Basic Tactics for the IPE 31

2.4 General Principles 37

3. Attribute Grammars As A Basis For Context-Sensitive Structure Editing 40

3.1 Attribute Grammars 41

3.2 Derivation Trees 44

3.3 Semantic Trees 46

3.4 Dependency Graphs 48

3.5 Completing Productions 49

3.6 Obtaining Semantic Trees from Attribute Grammars 50

3.7 Incremental Reevaluation 51

3.7.1 Jalili's Incremental Reevaluation Algorithm 53

3.7.2 Evaluation on Demand 55

3.8 The Attribute Grammar for the IPE 57

3.8.1 The Context-Free Grammar 57

3.8.2 The Major Attribute Systems 59

3.8.3 Some Example Rules 60

3.8.4 Interface Considerations 64

3.9 Some Suggested Improvements of the Proof Grammar 67

3.9.1 Choosing Terms for All Elimination and Exists Introduction 67

3.9.2 Determining Appropriate Premises in Re-Applied Proof Structures 71

4. The User Interface **74**

4.1 Display Formats for Structured Objects 75

4.1.1 A Display Format for Sequents 77

4.1.2 Display Formats for Proof Nodes 77

4.2 The Level 1 Proof Machine 79

4.3 The Level 2 Proof Machine 82

4.4 The Level 3 Proof Machine 82

4.5 The Command Interpreter 83

4.5.1 Operation of IPE Commands 86

5. Facilities for the Manipulation of Proof Structures **90**

5.1 Multiple Buffers 90

5.2 Automatic Proof Construction 93

5.2.1 IPE-Tactics 94

5.2.2 Uses of IPE-Tactics and Extensions to Them 101

5.3 Storing Proof Structures 104

5.4 Printing Proofs 105

6. A Theory Database **107**

6.1 Introduction 107

6.2 IPE Theories 110

6.3 Using Facts in a Proof 114

6.3.1 The Facts Browser 115

6.4 Generating Lemmas 129

6.5 Remarks 131

7. Future Work and Conclusions **136**

7.1 Recent Work 136

7.1.1 Rewrite Rules 136

7.1.2 The XIPE 137

7.2 Future Work 138

7.3 Concluding Remarks 140

A. The Proof Grammar **146**

A.1 The Syntax of C-SEC 146

A.2 The C-SEC Definition of the Proof Grammar 149

B. The IPE User Manual **175**

Chapter 1

Introduction

1.1 General Description

There is a large body of research concerned with improving the power of automated reasoning systems to construct formal proofs. However, it is still the case that the most imaginative theorem prover is the human mind. The principal aim in the design of the Interactive Proof Editor was to build a proof assistant which makes it easy for people to construct and experiment with proofs, but which insists upon formality of argument, thus combining the user's intuition with the machine's rigorous proof-checking capability.

The Interactive Proof Editor (or IPE in acronym) enables the development and maintenance of machine-checked proofs of statements in an untyped first-order intuitionistic predicate calculus. The encouraged style of proof is goal-directed: after supplying an initial conjecture, proof proceeds by decomposition of a current goal into hopefully simpler subgoals The applicability of each step in the proof, and the validity of the goal at each point are incrementally maintained, providing instant feedback to user actions, such as completion of a proof, which have an effect upon distant parts of the proof. Proofs can be edited at any point; users can return to any stage of a proof and alter the decision made there.

A structured theory database can be built up to provide a library of new axioms and facts proven from them, and an interactive browser can be used to interrogate this database.

The IPE combines the use of attribute grammars (as in the Reps-Alpern work [Reps-Alpern 84]) with lemma-matching techniques inspired by the 'B' tool [Abrial 86b] and concepts from LCF [GMW 79], making consistent use of a "proof-by-pointing" interface (see §1.5); this results in an easy-to-use proof assistant with the emphasis upon navigability and ease of alteration of proofs.

The IPE is written as a hierarchy of some 100 modules in a variant of Luca Cardelli's "ML Under UNIX"[Cardelli 83], which includes low-level user interface modules written in C. The version of the IPE described here runs on Sun workstations under UNIX[1]; earlier versions will also run on VAX/UNIX. The first version of the IPE appeared in early 1985; the theory database and browsing facilities

[1]UNIX is a trademark of AT&T Technologies, Inc.

were added in 1986. In this thesis we concentrate upon "Version 5", which makes use of the SunView environment for Sun workstations[2]; however, there is a later version (which we briefly describe) that utilises the X window system[3].

The IPE has been demonstrated widely at exhibitions and to visitors to the Computer Science Department. In September 1987, the Laboratory for Foundations of Computer Science ran a three-day course on "Interactive Proof Editing" using the X windows version of the IPE ([BTJ 87]). The author has made several presentations on the IPE, including [Ritchie 87].

1.2 Acknowledgements

I am indebted to the following for their help:

Professor Rod Burstall provided able supervision, advice and encouragement. Tatsuya Hagino developed and maintained a window environment within ML Under UNIX and handled all the nitty-gritty details of window management and portability. John Cartmell provided much ground-work code in ML in the form of a vast library of reusable modules.

Claire Jones deserves mention (as does Tatsuya) for further developments to the IPE since the author's departure from Edinburgh in July 1986.

Staff and students of the Computer Science Department provided valuable feedback on the various versions of the IPE, as did many who saw or used it during visits and exhibitions.

I am also indebted to colleagues at the Rutherford Appleton Laboratory and at the University of Manchester, for giving me the time, patience and encouragement to complete this thesis.

---

[2]Sun Workstation and SunView are trademarks of Sun Microsystems Inc.

[3]The X Window System is a trademark of MIT.

Numerous friends have been instrumental in keeping me afloat whenever my spirits were low (and indeed at any other time). Keith Refson has proven particularly good at ensuring a high content of spirits for many years. Phoebe Kemp was a tremendous support, and I remain forever in her debt.

The author's work on the IPE was funded by an SERC Award.

1.3 Overview

The remainder of this chapter presents a brief overview of some proof construction systems, and a "walk-through" demonstration of a simple proof in the IPE. Chapter 2 describes the underlying logic. Chapter 3 introduces attribute grammars and shows how they can be used in the development of structure editors. The attribute grammar used in the IPE is also described here. Chapter 4 presents the layers of interface between the kernel generated from the attribute grammar and the user. In Chapter 5, we extend the description of the IPE to include multiple buffers, built-in tactics and the storage and printing of proofs. Chapter 6 concerns the design of a database of simple, structured "theory units", and tools for its use in IPE proofs. Chapter 7 describes later extensions to the IPE, and some suggestions for future work.

1.4 Machine-Assisted Proof Systems

In order to convince commercial and industrial software developers that the use of formal methods forms a sound, effective and practical framework upon which to base software engineering, it is important to develop tools which are at once easy to use and also powerful enough to be of practical help. The IPE has concentrated upon demonstrating that it is possible to develop good user interfaces to theorem provers; the initial aim was to develop a system which had a sufficiently low learning threshold to be used to teach the principles of formal proof to people who have little or no experience in the area.

Fully-automatic theorem provers (the most celebrated being the Boyer-Moore theorem prover [Boyer-Moore 79]) rely upon an optimism that their built-in strategies will work first time. The creative and intuitive abilities of the user are then relegated to determining how to recover when a proof attempt fails (or worse, to detecting that the attempt is non-terminating). In the Boyer-Moore system, this may involve determining why the prover reached such a state, i.e , re-running the prover's actions by hand. If the user still believes that the original conjecture should be true, the next step is to build and prove a lemma which the user imagines will be useful in the original proof attempt. Having done this, it is now necessary to re-run the first attempt in its entirety. Nonetheless, the heuristics employed in Boyer-Moore are powerful, and the prover prints out a wordy description of its strategy when tackling a particular problem.

From its initial version at Stanford in the early seventies, LCF (from "Logic for Computable Functions") has become the name for a family of systems for "goal-directed validated proof". Instances of LCF are described in [GMW 79],[Paulson 85b],[Petersson 82] and [Gordon 85]. The notion of *tactics* has been widely used (including in the IPE, though here they are hidden from view.) In LCF-style systems, user-constructed tactics and tacticals are the main proving tools; thus the user has greater control over the direction of a proof than in a fully-automated prover. Furthermore, even if a tactic does not completely reduce a goal (to an empty set of subgoals), it will still return a validation function, which acts as a partial proof of the goal. If the subgoals are later proven, then the validation function can be applied to the resultant theorems, to yield a statement of the original goal as a theorem. This assumes that the tactics generate the correct validation functions; a common remark about LCF is that a tactic will always prove *something* – but it might not be what was intended.

The interface to LCF is simply that of an interactive ML session, i.e. a "glass teletype" interface. Proofs (and the details of their construction) are not visible to the user.

In Larry Paulson's Isabelle [Paulson 85a], the user derives new inference rules by "composing" inference rules (matching a hypothesis of one rule against the

conclusion of another). Thus to prove a goal G, one first constructs the rule $\frac{G}{G}$, then applies matching inference rules to the hypothesis or conclusion, until a rule of the form $\frac{}{G}$ is constructed. (In fact, since the result at each point is a valid inference rule, we can stop whenever we consider the rule to be useful). Higher-order matching is used; this enables the construction and use of powerful and general inference rules.

As yet, the interface to Isabelle is of the same "ML interface" level as LCF; however, at the time of writing, Paulson is working with Brian Monahan on the design of a better user interface.

The PRL system [Bates-Constable 83] appears to be the earliest example of a theorem- proving tool which uses structure editor techniques to maintain and display a proof-in-progress. The use of a constructive logic makes it possible to "extract" functions from proofs. The system presents several windows to the user: a proof window, a library window, and a command window. The library window contains a list of defined objects (functions, definitions, theorems and functions extracted from theorems). The definitions mechanism allows the construction of parameterised templates with the same freedom of expression as available in C macros – there is no insistence that a definition should expand to a syntactic unit. Thus the visual syntax of formulae is completely controllable; the price paid is that PRL must expand the definition in order to manipulate it, and must also determine whether the application of a rule to the formula destroys the internal shape of the definition.

Each step of a proof consists of the reduction of a goal formula (under a set of hypotheses) to a list of subgoals (each with possible extra hypotheses). Such a step can be performed either by use of a refinement rule or a refinement tactic; the latter can be constructed by the user, in an LCF-like fashion.

Unfortunately, the interface to the structure editor (which controls both the proof and library windows) is awkward and ungainly. There is no mapping from points on the screen to points in the proof or library structures, and all movement commands are in terms of the internal structures.

Work on PRL continues in the NuPRL system [PRL 86], in which the library of tactics has been greatly extended. NuPRL also provides "transformation tactics" which have access not only to goals but also entire proof trees, for example allowing "proofs by analogy".

The initial inspiration for the IPE came from a simple proof editor for propositional calculus [Reps-Alpern 84], which was produced using the Cornell Synthesizer Generator [Reps-Teitelbaum 85]. The interface inherited from the CSG is very much that of a syntax-directed structure editor for programs: positioned at a Proof "placeholder", the user is presented with a list of all of the production rules for the "symbol" Proof. When a production rule is applied to a Proof, the original production is replaced by the new one, and if the rule is applicable to the goal, the goals of the subproofs are determined. As in PRL, movement was made through the shape of the syntax tree rather than the display form.

After [Reps-Alpern 84], Tim Griffin has been using the CSG to implement several prototype proof editors along the lines of NuPRL. The author was unaware of this work until Griffin visited Edinburgh in late 1986 to implement an editor for the Edinburgh Logical Framework [HHP 87]. In [Griffin 87], Tim Griffin describes an "Environment for Formal Systems" (EFS). In EFS, one can define the syntax and "refinement rules" for a wide variety of logics, building on top of either the Edinburgh Logical Framework or the Calculus of Constructions [Coquand-Huet 85]. New logical connectives and inference rules are defined as constants; definitions akin to those of PRL can be used to hide their internal representation. Refinement rules can then be defined which justify themselves in terms of the inference rules; these refinement rules can then be used as steps in larger refinements (proofs). Constants, refinement rules and refinements are all stored in *chapters*. A chapter can import the contents of other chapters, providing a simple means of structuring information.

It would appear that of all currently available proof construction tools, the EFS provides the best blend of descriptive power and usability. However, EFS lacks user-programmable tactics (other than user-defined refinement rules), and

Griffin is uncertain that EFS would be capable of supporting the scale of information that NuPRL has handled. (These criticisms also apply to the IPE.)

The "B" proof editor ([Abrial 86a], [Abrial 86b]) is interesting in that whilst being essentially interactive in nature, it searches through a database of rules to find those rules which can apply to the current goal, and presents these in turn to the user for selection. (It can also proceed by itself). A simple tactics-like language allows the user to determine the order in which B's "theories" are searched, including repetitive searching.

(The author was informed of this feature of B in 1986; this led directly to the development of the theory database of the IPE. However, upon finally seeing B in 1987, it transpired that the manner in which searching is used in B is very different from that used in the IPE. A theory in B is simply a collection of axioms and results, searched in a "last in–first out" basis; B's theories do not refer to other theories, so that the only theory structuring is that provided by the tactics defined by the user)

B's major drawback is its primitive user interface (commands are chosen by number from a menu which only appears when help is requested; the menu then replaces any other information on the screen). It is difficult to navigate through a proof, other than by undoing steps. Defining tactics in B appears to be something of a black art; however, an expert can use B impressively. The author witnessed Jean-Raymond Abrial use B to perform program transformation and "compilation" upon small programs.

Another recent development is the Muffin proof editor, built at Manchester University ([Moore 86b]). Initially specified in VDM by Richard Moore and Cliff Jones ([Moore 86a]), the editor was built over the course of a few months in early 1987 using the Smalltalk object-oriented environment ([Goldberg-Robson 83], [Goldberg 84]). Muffin serves to show that Smalltalk-80 can be used for the rapid construction of formal reasoning tools with state-of-the-art user interfaces ([Jones,K 87]). Each stage of a Muffin proof is presented as a list of knowns and a list of goals. Proof proceeds by the user selecting a known or goal and asking Muffin to list all rules which "match" the selected formula. This causes an

exhaustive search of the database; similarly to B, there is no distinction between built-in rules and user-constructed rules (which are generated from completed Muffin proofs). As in B, proof navigation is difficult; however, the presentation of each stage of a proof is pleasant – for example, it is easy to hide unwanted knowns. Muffin is restricted to propositional logic, but the developers feel that the extension to predicate logic would not present significant problems. Work on Muffin was carried out as an experiment in user interface design for the Alvey IPSE 2.5 project [Alvey 87, Simpson 87].

1.5 A Demonstration

In this section we shall run through a demonstration of the IPE. This demonstration aims to show the basic features of IPE, in particular the "proof by pointing" interface style. (Descriptions of the use of automatic proof construction, multiple buffers and retrieval of information from the theory database will be deferred until later chapters).

Once the title screen is dispensed with, we are presented with a "blank proof" (Figure 1–1). This shows the shape of our proof (trivial at the moment). The title line at the bottom of the screen gives some information that need not concern us at the moment. The "empty" box on the bottom right is an indicator window: when the IPE is busy, or when some "subtool" such as the text editor is being used, a message will appear in here.

The statement of the initial conjecture appears within angled brackets. This indicates that it is a *text-edit point* which the user may change as desired. To do this, we point the mouse anywhere between the brackets and click the middle mouse button. This invokes the text-editor upon the current statement of the formula (Figure 1–2). The new window is labelled "Formula" as an indication of the kind of object we should supply. We can type ordinary text here (and use a few simple editing commands), but when we tell the IPE to accept what we have typed, it will be parsed as a formula. If it does not parse correctly, then the IPE

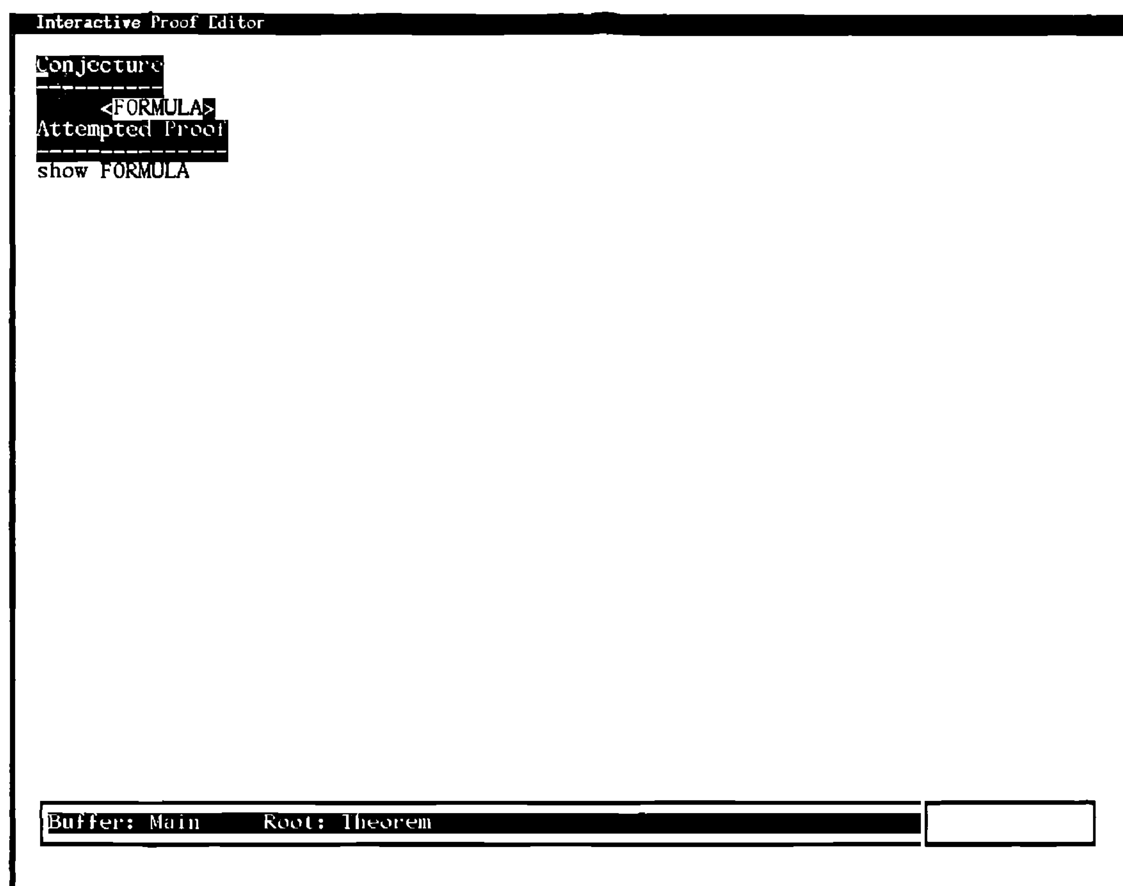


Figure 1–1: A blank proof

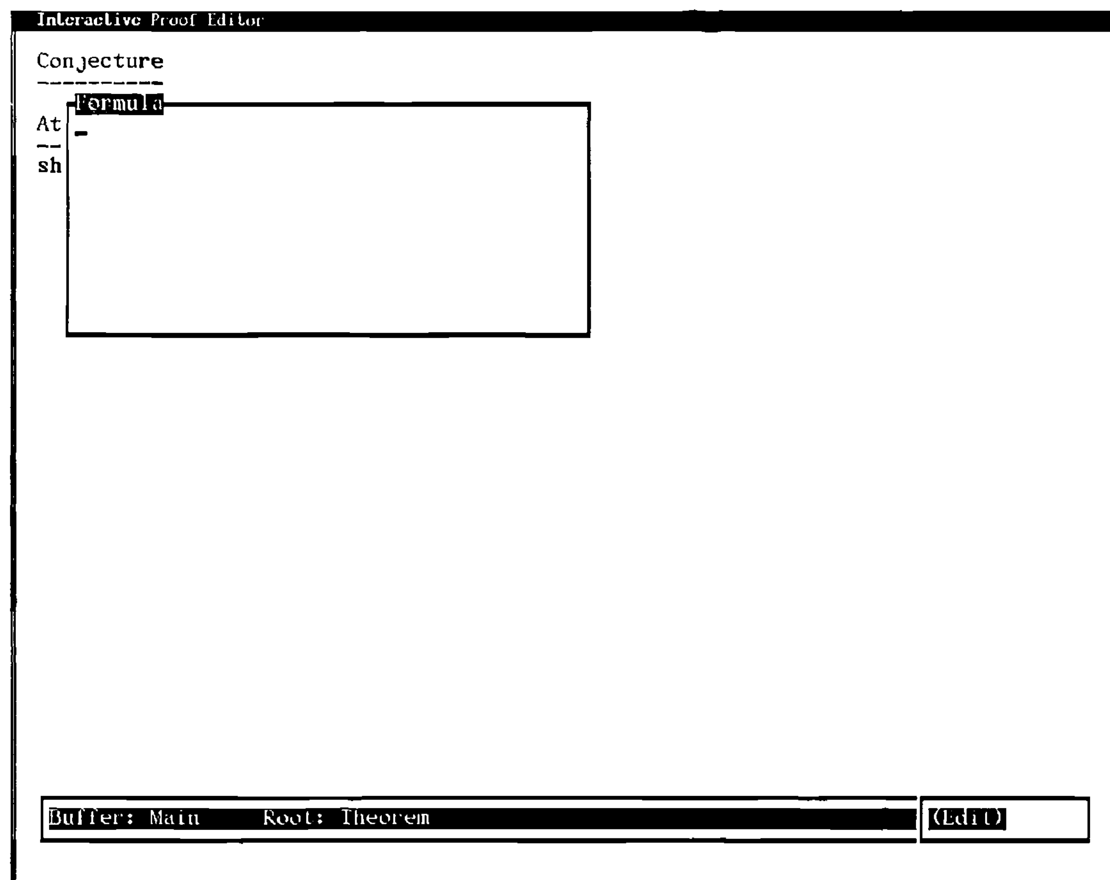


Figure 1–2: Editing the initial conjecture

will leave the cursor on the line below the text at the point where parsing failed. We must either re-edit until we have a parsable formula, or abort the edit.

Suppose that we intend to prove that existential quantification distributes over disjunction. In the IPE's notation, one statement of this result is shown in Figure 1–3 (Here, "?" represents the quantifier "$\exists$", "|" represents "or" and

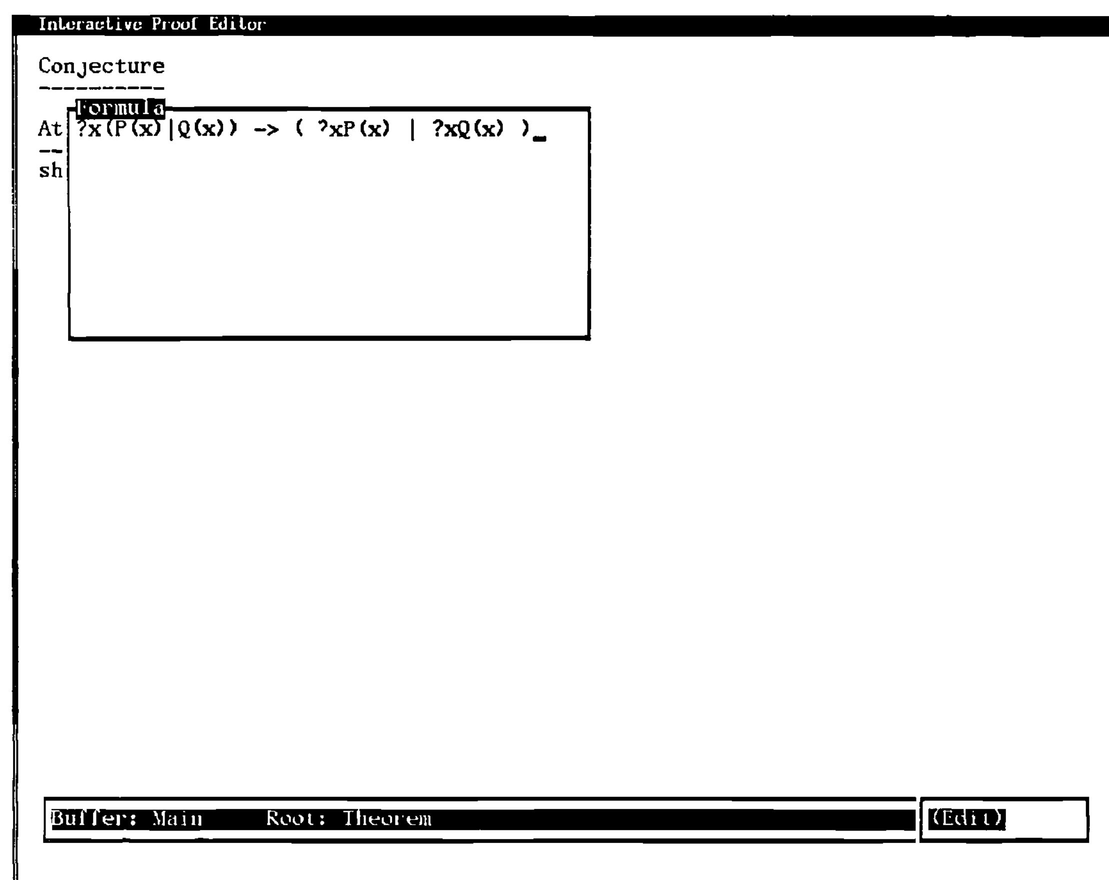


Figure 1–3: Distribution of existential quantification over disjunction

"$\rightarrow$" represents "implies").

Satisfied that this is the formula we want, we select the "exit" option from the right-button menu. IPE has no complaints about the syntax of the formula, and so the "formula slot" is updated. This information feeds through to the proof, giving Figure 1–4

Now we can begin the proof. As many steps of a proof as possible will be displayed on the screen, each step appearing as a rule name, a list of premises (possibly empty, as above) and a conclusion. "use rule-name and show . . . " indicates that the subproof of this step has not been completed yet. To alter any step in

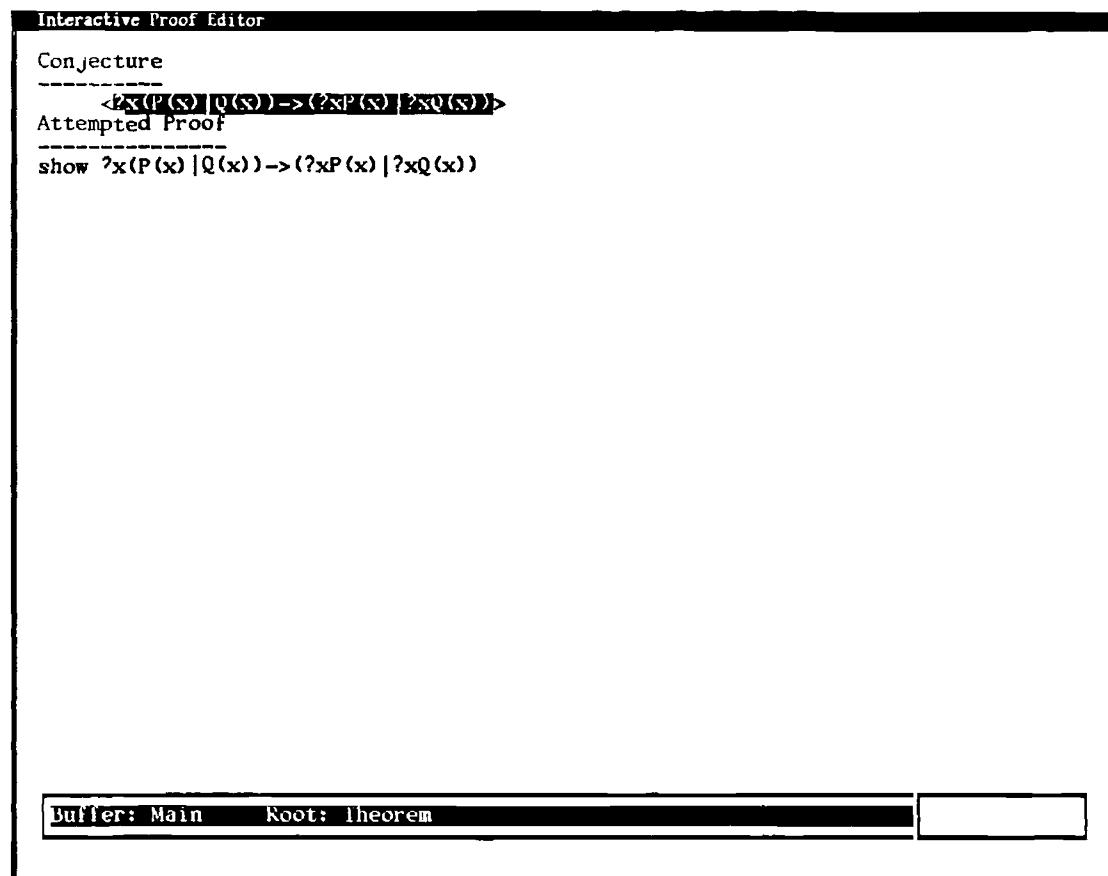


Figure 1–4: The new problem

the proof, we select one of its premises or the conclusion and click the middle mouse button over it The IPE will choose the proof rule appropriate to that premise or conclusion, and "expand" the proof at that point accordingly. (This is what we mean by "proof-by-pointing".)

Here there is only one formula – the conclusion – so to proceed we click on it[4]. Figure 1–5 shows the result. In the new subproof we have assumed the left-

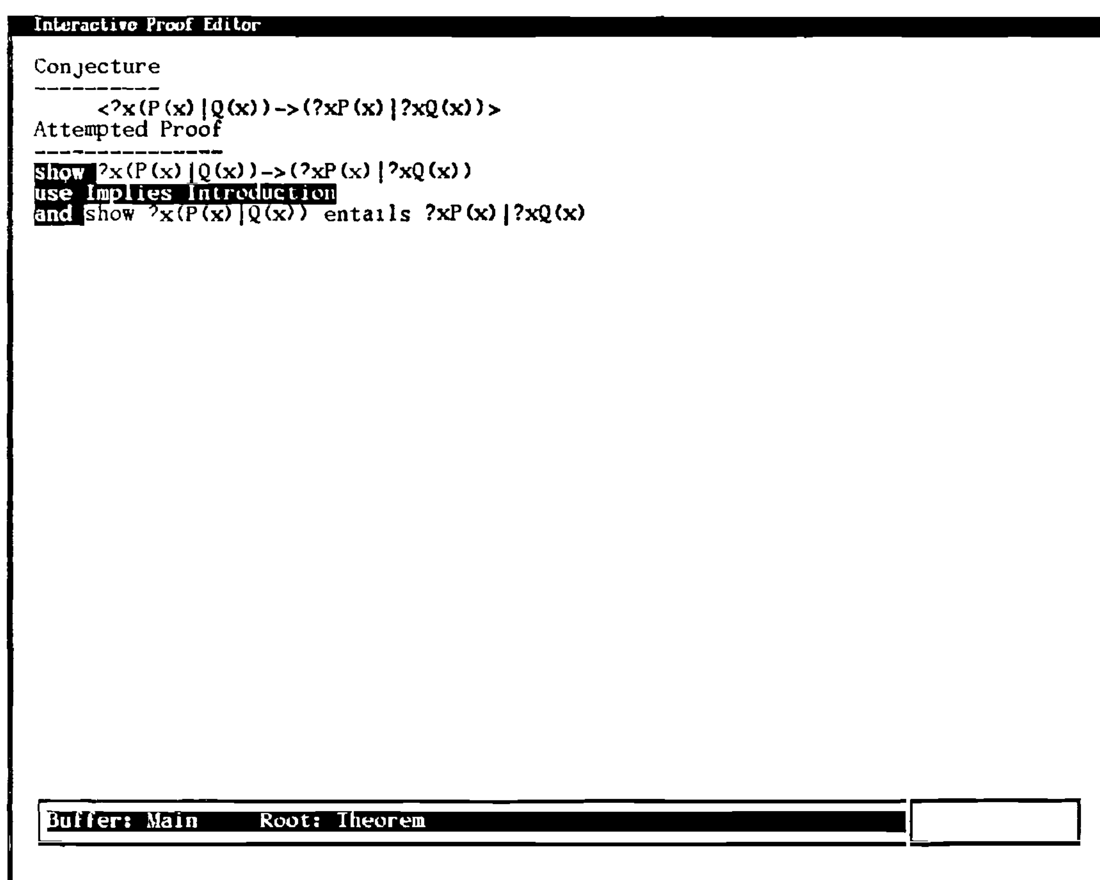


Figure 1–5: The first step of the proof

hand side of the implication, and it now remains to show that the right-hand subformula can be derived from this.

Now we have to choose which formula to attempt to simplify first. It is not too important which formula we choose, because even if the choice turns out to be incorrect, the IPE will allow us to return to the same point in the

[4]It is possible to make IPE perform such really trivial steps on its own, as we shall see in §5.2.

proof and change our minds. In this case, suppose that we choose to simplify the conclusion. Selecting it in the same manner as above leads to Figure 1–6. The resulting Or Introduction rule now presents us with two subproofs. Each

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
---------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Or Introduction
    and show ?x(P(x)|Q(x)) entails ?xP(x)
    or show ?x(P(x)|Q(x)) entails ?xQ(x)

Buffer: Main    Root: Theorem
```

Figure 1–6: Deciding to perform Or Introduction

subproof has the same premise, but their conclusions are drawn from the left- and right-hand parts of the original conclusion. Note that the initial goal of the proof step will be considered proven whenever we complete the proof of either subgoal.

For our next step, we choose to simplify the premise of the first new subgoal, giving Figure 1–7. The existential quantifier has been "stripped off" in the premise of the subgoal. We know that "P(x)" holds for *some* "x", and all that has happened here is that we have said, "Let "x" be such that "P(x)" holds." Notice that there is an "x" on the rule-name line which is enclosed in angle brackets. This indicates that we are permitted to edit it, in the same manner as for the initial formula. In this case, the "x" is an identifier, with the important

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
---------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Or Introduction
    and show ?x(P(x)|Q(x)) entails ?xP(x)
        use Exists Elimination with <x> on premise 1
        and show P(x)|Q(x) entails ?xP(x)
    or show ?x(P(x)|Q(x)) entails ?xQ(x)

Buffer: Main     Root: Theorem
```

Figure 1–7: After Exists Elimination upon the premise

restriction that it should not occur freely in any other formula in the goal. We may change its name, but the new name should also satisfy the restriction. Here, although the conclusion also mentions "x", it is bound by a quantifier, so that the IPE is happy to use "x" as the name of the "witness".

Now if we simplify the conclusion in this new goal, we get Figure 1–8. Again,

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
---------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Or Introduction
    and show ?x(P(x)|Q(x)) entails ?xP(x)
        use Exists Elimination with <x> on premise 1
        and show P(x)|Q(x) entails ?xP(x)
            use Exists Introduction with <TERM_1>
            and show P(x)|Q(x) entails P(TERM_1)
    or show ?x(P(x)|Q(x)) entails ?xQ(x)

Buffer: Main    Root: Theorem
```

Figure 1–8: Exists Introduction

the quantifier has been stripped off in the subgoal. This time however the variable has been replaced by the name "TERM_1". This indicates that we can replace occurrences of the bound variable with some term. Intuitively, we are trying to show that "P(x)" holds for some "x", and we proceed by choosing some "value" for "x" for which we believe we can show that "P(x)" is true. We have to supply this value by hand; IPE will not attempt to do this for us. However, the IPE does not force us to decide "once and for all" what this value should be before we look further into the proof. Thus, even if we make a stupid choice initially, when

this becomes obvious later in the proof, we can return to this point and supply a different value, without losing the work we have performed in the meantime.

In this case, the "new" value that we want for "x" is simply "x" itself. (Though the names are the same, what we are doing is forcing the identification of two distinct variables which were bound in different ways). Therefore we want to "undo" the IPE's choice of value. Clicking the middle button over the "TERM_1" in angled brackets calls up a text-edit window (entitled "Term" to show us that the result will be parsed as a term) with "TERM_1" in it.

Suppose that we have changed "TERM_1" to "x". Then the only action that remains in this branch of the proof is to simplify the premise (Figure 1–9). We

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
---------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Or Introduction
    and show ?x(P(x)|Q(x)) entails ?xP(x)
        use Exists Elimination with <x> on premise 1
        and show P(x)|Q(x) entails ?xP(x)
            use Exists Introduction with <x>
            and show P(x)|Q(x) entails P(x)
                use Or Elimination on premise 1
                and P(x) entails P(x)
                    is immediate
                and show Q(x) entails P(x)
    or show ?x(P(x)|Q(x)) entails ?xQ(x)

Buffer: Main    Root: Theorem
```

Figure 1–9: After Or Elimination

know that one of "P(x)" and "Q(x)" holds, but don't know which. However, if we can prove our conclusion by assuming "P(x)" alone and also assuming "Q(x)" alone, then the conclusion must follow from their disjunction.

Now, one of our subgoals is trivially true: "P(x)" occurs on both sides. (Notice how there is no "show" prefacing the goal). Unfortunately, Or Elimination requires that both subgoals be completed, and that is not the case here. Moreover, the second subgoal obviously cannot be completed. There is no relationship between "P(x)" and "Q(x)" that we can use[5]. We have to conclude that we went wrong somewhere. Either our problem is unsolvable in intuitionistic logic, or we took a wrong turning in our proof.

In this case, it is not too hard to see that we used the Or Introduction rule too soon. What we must do now is change the decision made at that point. All we have to do is click the middle button over the premise of the goal of that rule (Figure 1–10) The original proof has disappeared. In this case, there is little use that we could have made of it, but there are cases where it would be useful to be able to re-use it. (This is indeed possible, as we shall see in §5.1).

Now we select the premise, to perform Or Elimination, which gives us two subproofs, each with the same conclusion as before. In the first subproof we select this conclusion, performing Or Introduction. Performing Exists Introduction on the result gives us Figure 1–11. We now text-edit on "TERM_2", replacing it with "x". The resultant subgoal is trivial. Furthermore, we have now completed the first subproof of the Or Elimination step, and the display alters accordingly (Figure 1–12).

Now we might notice that the same sequence of steps will also prove the second subgoal. As we shall see in §5.1, we could "squirrel away" a copy of the first subproof and then re-apply it to the second, in preference to repeating the steps by hand. Whichever way we proceed, the completed proof is shown in Figure 1–13.

---

[5]In §6 we shall see how we can build up libraries of information (as axioms and derived results) that can be brought into proofs when needed.

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
----------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Exists Elimination with <x> on premise 1
    and show P(x)|Q(x) entails ?xP(x)|?xQ(x)

Buffer: Main      Root: Theorem
```

Figure 1–10: Changing direction at the second step

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
---------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Exists Elimination with <x> on premise 1
    and show P(x)|Q(x) entails ?xP(x)|?xQ(x)
        use Or Elimination on premise 1
        and show P(x) entails ?xP(x)|?xQ(x)
            use Or Introduction
            and show P(x) entails ?xP(x)
                use Exists Introduction with <TERM_2>
                and show P(x) entails P(TERM_2)
            or show P(x) entails ?xQ(x)
        and show Q(x) entails ?xP(x)|?xQ(x)

Buffer: Main    Root: Theorem
```

Figure 1–11: After several proof steps

```
Interactive Proof Editor

Conjecture
----------
     <?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Attempted Proof
---------------
show ?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
use Implies Introduction
and show ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    use Exists Elimination with <x> on premise 1
    and show P(x)|Q(x) entails ?xP(x)|?xQ(x)
        use Or Elimination on premise 1
        and P(x) entails ?xP(x)|?xQ(x)
            by Or Introduction
            and P(x) entails ?xP(x)
                by Exists Introduction with <x>
                and P(x) entails P(x)
                    is immediate
        and show Q(x) entails ?xP(x)|?xQ(x)

Buffer: Main      Root: Theorem
```

Figure 1–12: Upon completion of one subproof

```
Interactive Proof Editor

Theorem
<?x(P(x)|Q(x))->(?xP(x)|?xQ(x))>
Proof
?x(P(x)|Q(x))->(?xP(x)|?xQ(x))
by Implies Introduction
and ?x(P(x)|Q(x)) entails ?xP(x)|?xQ(x)
    by Exists Elimination with <x> on premise 1
    and P(x)|Q(x) entails ?xP(x)|?xQ(x)
        by Or Elimination on premise 1
        and P(x) entails ?xP(x)|?xQ(x)
            by Or Introduction
            and P(x) entails ?xP(x)
                by Exists Introduction with <x>
                and P(x) entails P(x)
                    is immediate
        and Q(x) entails ?xP(x)|?xQ(x)
            by Or Introduction
            and Q(x) entails ?xQ(x)
                by Exists Introduction with <x>
                and Q(x) entails Q(x)
                    is immediate
QED

Buffer: Main     Root: Theorem
```

Figure 1–13: The completed proof

The IPE intentionally represents the "opposite extreme" from fully-automated provers of the Boyer-Moore category. The user has full control over the construction of the proof, changing an earlier step in the proof is almost as easy as making the next step, and changes made to earlier stages of a proof filter through the rest of the proof.

The cost of this concentration of effort upon the user interface is in proving power. As we shall see in §5.2, IPE has only a rudimentary means of automated proof construction. Furthermore, the IPE works best upon small proofs, though in §6 we describe a means of making and storing lemmas used in the construction of "larger" results. To the best of the author's knowledge, the largest problems tackled (and completed) using the IPE were the proof of a small parser (a problem described in [Cohn-Milner 82]) and the proof of termination of a program in a simple language. Both of these proofs were performed by Claire Jones. Indeed the parser proof was Claire's first attempt at using the IPE, and took about a week to construct. Much of this time involved setting up the relevant descriptions in the IPE's theory database, rather than constructing the proof itself.

Chapter 2

The Generation of Basic Tactics for Interactive Proof

2.1 Introduction

The logical framework within which the Interactive Proof Editor operates is that of untyped intuitionistic first-order predicate calculus, excluding equivalence of predicates (which can be modelled as (A→B) & (B→A)). The basic proof steps of the IPE take the form of tactics (akin to those in the LCF system) which are used to construct proofs in a top-down fashion, by reducing an initial *goal* to a set of subgoals, where each subgoal is hopefully simpler to solve. The goals are represented as *sequents* (from Gentzen, as in [Kleene 64]), where a sequent consists of a list of *antecedent* formulae (premises) paired with a *succedent* formula (conclusion). By ensuring the validity of the proof steps with respect to the original *inference rules* of the logic, we can ensure that valid proofs of the subgoals, or of a subset of the subgoals, produced by a proof step can be used to construct a valid proof of the original goal.

The presentation in this chapter was strongly influenced by Schmidt's approach in [Schmidt 83]. We derive the IPE's basic tactics from a set of natural deduction inference rules. Such a presentation was chosen to suggest means by which we could construct a general method of deriving the basic tactics for a proof editor starting from natural deduction inference rules. In fact, the IPE's basic tactics can be drawn more directly from Gentzen's intuitionistic formal system *G3a* as presented on page 481 of [Kleene 64].

The logical connectives used in the IPE, and their intuitive meanings are as follows: (where P,Q and R denote arbitrary formulae, and where P(x) denotes a formula P possibly containing a free variable x)

$P\&Q$ — P and Q

$P|Q$ — P or Q

$P\rightarrow Q$ — P implies Q

$\sim P$ — not P

$\forall x P(x)$ — for all x, P(x)

$\exists x P(x)$ — there exists x such that P(x)

IPE Version 5 is restricted to the ASCII character set; therefore the characters ∀ and ∃ are not available[1]. Thus in the screen display of formulae, IPE version 5 uses "$!xP(x)$" for "$\forall xP(x)$", and "$?xP(x)$" for "$\exists xP(x)$". This representation is also used in this thesis.

The syntax for predicate calculus formulae in IPE is given by the following:

<formula> ::= <formula>&<formula> | <formula>|<formula>
| <formula>→<formula> | ~<formula>
| ∀<var><formula> | ∃<var><formula>
| <predicate> | (<formula>)

<predicate> ::= <ident> | <ident>(<termlist>)

<term> ::= <ident> | <var> | <ident>(<termlist>)

An <ident> is any sequence of upper- or lower-case letters and numerals beginning with a letter, and possibly ending with one or more primes ('). A <var> is similar but excluding a change from lower to upper case in the sequence (this allows us

[1]However, ∀ and ∃ are used in the display of formulae in the X windows IPE.

to juxtapose variables and identifiers, as in !xP(x)). A <termlist> consists of one or more <term>s separated by commas.

In the formula parser of IPE Version 5, there is no precedence between the binary connectives, and formula expressions are left-associative. This is sometimes counter-intuitive; for example, "$A \rightarrow B \& C$" is "$(A \rightarrow B) \& C$" and not "$A \rightarrow (B \& C)$".

2.2 Inference Rules of the IPE

A semantics for describing the construction of valid formulae using the above logical connectives is given by a set of ***natural deduction inference rules*** in the style of [Prawitz 65], where each is of the form

$$\frac{\text{<premise>} \;\ldots\; \text{<premise>}}{\text{<formula>}},$$

where <premise> is <formula> or $\begin{matrix}\text{[<formula>]} \\ \text{<formula>}\end{matrix}$. For example,

$$\frac{P \quad \begin{matrix}[Q] \\ Q'\end{matrix}}{R}$$

means "given P, and that we can infer Q' from Q, then we can infer R".

In this characterisation of intuitionistic 1st-order predicate calculus there are two kinds of inference rules for each connective. The first *introduces* the connective, by defining the conditions under which a formula can be derived with that connective as its major connective; these are called *introduction* rules and are denoted by "connectiveI" (e.g. "&I"). The second kind defines the conditions for *eliminating* the connective from a formula and exposing some of the substructure

of the formula; these are known as *elimination* rules, denoted by "connectiveE" (e.g., "$\forall E$"). Where there is more than one introduction or elimination rule for a connective (as is the case for elimination of &), the individual rules are identified by appending "r" or "l" to the rule name, as in "&Er".

The inference rules from which we shall derive the IPE's basic tactics are given below:

$$\&I\ \frac{A \quad B}{A\&B} \qquad \&Er\ \frac{A\&B}{A} \qquad \&El\ \frac{A\&B}{B}$$

$$|Ir\ \frac{A}{A|B} \qquad |Il\ \frac{B}{A|B} \qquad |E\ \frac{A|B \quad \begin{matrix}[A]\\ C\end{matrix} \quad \begin{matrix}[B]\\ C\end{matrix}}{C}$$

$$\rightarrow I\ \frac{\begin{matrix}[A]\\ B\end{matrix}}{A\rightarrow B} \qquad \rightarrow E\ \frac{A \quad A\rightarrow B}{B}$$

We introduce a special predicate **false**, and the rule

$$\frac{\textbf{false}}{A}$$

(i.e. everything can be derived from **false**) and define $\tilde{}A$ as $A \rightarrow$**false.**

$$\forall I\ \frac{P(x)}{\forall x P(x)} \qquad \forall E\ \frac{\forall x P(x)}{P(t)}$$

The $\forall I$ rule has a side-condition: it can only be used when x does not occur free in any assumptions upon which $P(x)$ depends (where x occurs free in a formula P if there is an occurrence of x which does not lie within the scope of a $\forall x$ or $\exists x$).

$$\exists I\ \frac{P(t)}{\exists x P(x)} \qquad \exists E\ \frac{\exists x P(x) \quad \begin{matrix}[P(x)]\\ Q\end{matrix}}{Q}$$

As with $\forall I$, $\exists E$ has the side condition that x should not occur free in Q, or in any assumptions upon which Q depends (other than $P(x)$).

These rules can be used to construct proofs directly: starting from a set of assumptions $A_1, \ldots, A_n$ as our initial premises, we can apply both introduction and elimination rules (when their preconditions are satisfied) to examine current premises or construct new ones. For example, the following is a proof of $A\&B\&C \rightarrow (A\&C)$:

$$
\rightarrow I \; \dfrac{\&I \; \dfrac{\&Er \; \dfrac{\&Er \; \dfrac{[A\&B\&C]}{A\&B}}{A} \qquad \&El \; \dfrac{[A\&B\&C]}{C}}{A\&C}}{A\&B\&C \rightarrow (A\&C)}
$$

However, such bottom-up methods of proof construction do not yield good mechanisms for interactive proof construction. A method such as this requires that we constantly look ahead; we need to know the overall shape and direction of the proof before we begin its construction. A better paradigm for interactive proof is that of top-down or *goal-directed* proof, where we commence by stating our final aim (or goal) and attempt to reduce it to smaller, more manageable subproblems (or subgoals), and analyse these similarly until either the subgoals become trivial or obviously unprovable. If we choose our tools for goal reduction with care, then we should be able to construct a formal proof (as in the previous section) of the initial goal by performing some composition operation upon the proofs of the subgoals. In goal-directed proof construction, we begin with a goal of the form **show** $P_1, \ldots, P_n$ **entails Q** and apply information derived from the inference rules to some of P_i , Q to produce a set of hopefully simpler goals, to which the process can be repeated until either trivial goals (i.e. goals of the form **show P,... entails P**) or obviously unprovable goals are arrived at.

As an example of information derived from an inference rule to yield a goal-directed rule, consider the &I rule. In its constructive use this says that if we already have both P and Q, then we can construct P&Q. In a goal-directed proof,

we want to turn this around so that it says that to show P&Q we must show both P and Q separately. This can then be used to reduce a goal of the form **show** $R_1, \ldots, R_n$ **entails P&Q** to the two subgoals **show** $R_1, \ldots, R_n$ **entails P** and **show** $R_1, \ldots, R_n$ **entails Q**. These subgoals are simpler, in the sense that at least one formula in each contains less structure than in the original goal (and no formula has become more complex).

The &E rules on the other hand can be used more or less directly. If we combine them, they say that from P&Q we can infer both P and Q; this rule can then be used in a top-down system to simplify premises, taking goals of the form **show P&Q,... entails R** to **show P,Q,... entails R**.

2.3 Derivation of Basic Tactics

We require a formal framework for re-phrasing the inference rules in forms more suitable for goal-directed proof. To do this, we look at David Schmidt's work on deriving tactics from inference rules ([Schmidt 83]).

2.3.1 Tactic Schemata

In the following, we use the notation $\Delta \overset{?}{\vdash} C$ to represent a *goal* with premises Δ and conclusion C. $\overset{?}{\vdash}$ is used to show that we have not yet shown that C can be derived from Δ. In the application of tactics to goals, we will allow permutation of the premises. We will also allow multiple instances of a premise. Thus the premises should be thought of as a bag of formulae rather than as a set or a list. This corresponds to Kleene's definition that two sequents are *cognate* when their premise "bags" contain the same formulae and their conclusions are also the same. The difference between Gentzen's sequents and our goals are that sequents may contain a list of conclusions; however, in the intuitionistic *G3a*, for all derivable sequents the list of conclusions consists of at most one formula. Here we have used *false* in place of an empty conclusion. We will use $\Delta \vdash C$ (and

equivalently, Δ **entails** C) to represent a proven goal. A goal $\Delta \overset{?}{\vdash} C$ is considered *immediate* or *immediately proven* if C occurs in Δ; thus we are adopting *G3a*'s axiom schema. This is encoded in the IPE's "Immediate" tactic, which succeeds if its goal is immediate and fails otherwise.

The action of a tactic upon a goal will be described in general as:

$$A, \Delta \overset{?}{\vdash} C \quad \mapsto \quad \langle A', \Delta' \overset{?}{\vdash} C'; \ldots \rangle$$

where A, A', C and C' are formulae, and Δ and Δ' are (possibly empty) bags of formulae, and that A, Δ is a bag containing at least one instance of A. This states that the tactic acts on a goal of the form $A, \Delta \overset{?}{\vdash} C$ and produces the subgoals contained in $\langle \ldots \rangle$. The tactic will fail if applied to a goal that does not correspond to the given form.

We shall distinguish between the introduction and elimination rules of inference and the Introduction and Elimination rules we are deriving for the IPE by capitalising the latter. The inference rules of *G3a* will be referred to as premise and conclusion rules (e.g. implies-conclusion) according to whether the associated connective is introduced in the premises or in the conclusion as a result of applying the inference rule.

In [Schmidt 83], David Schmidt reasons that schemes for deriving tactics from inference rules could be used to develop a set of tactics which adhere to the logic. He describes two *tactic schemes* for deriving tactics from inference rules. Given a rule

$$r \; \frac{\begin{matrix} [A_1] & \ldots & [A_k] & & & \\ B_1 & \ldots & B_k & B_{k+1} & \ldots & B_m \end{matrix}}{C}$$

(meaning, "if we can derive $B_1, \ldots, B_k$ from $A_1, \ldots, A_k$, and if we have $B_{k+1}, \ldots, B_m$, then we can infer C"), the tactic schemes are

($\vdash r$): when applied to a goal of the form $\Delta \overset{?}{\vdash} C$, this applies the rule r "backwards" to C, producing the subgoal list $\langle \Delta, A_1 \overset{?}{\vdash} B_1 \; ; \ldots \; ; \Delta, A_k \overset{?}{\vdash} B_k; \Delta \overset{?}{\vdash} B_{k+1} \; ; \ldots \; ; \Delta \overset{?}{\vdash} B_m \rangle$

($r\vdash$): when applied to a goal of the form $B_1, \ldots, B_m, \Delta \overset{?}{\vdash} D$, this applies the rule r "forwards" to $B_1, \ldots, B_m$, producing the subgoal list
$\langle B_1, \ldots, B_m, C, \Delta \overset{?}{\vdash} D\rangle$

There is also a *validation function* associated with each tactic which, given a validation of each of the subgoals constructs a validation of the original goal, using the original inference rule. (These validation functions are not used in the IPE

Schmidt proceeds to suggest ways in which these derived tactics could be used to write general "try-everything" tactics for goal-directed proof in the logic defined by a set of inference rules. However, the same schemes can be used as a first step in deriving the basic tactics used in the IPE.

2.3.2 Basic Tactics for the IPE

Schmidt's tactic schemes give two tactics per inference rule. As we shall see, not all of the tactics generated from the inference rules for a connective are useful in goal-directed proof strategy, and some require further modification (for example to take account of their applicability to certain situations).

We consider the tactics generated by Schmidt's method for each connective in turn.

For &, we obtain the tactics

($\vdash\&I$): $\Delta \overset{?}{\vdash} P\&Q \;\mapsto\; \langle \Delta \overset{?}{\vdash} P;\ \Delta \overset{?}{\vdash} Q\rangle$

($\&I\vdash$): $P, Q, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle P\&Q, P, Q, \Delta \overset{?}{\vdash} R\rangle$

($\vdash\&Er$): $\Delta \overset{?}{\vdash} P \;\mapsto\; \langle \Delta \overset{?}{\vdash} P\&Q\rangle$

($\&Er\vdash$): $P\&Q, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle P, P\&Q, \Delta \overset{?}{\vdash} R\rangle$

($\vdash\&El$): $\Delta \overset{?}{\vdash} Q \;\mapsto\; \langle \Delta \overset{?}{\vdash} P\&Q\rangle$

($\&El\vdash$): $P\&Q, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle Q, P\&Q, \Delta \overset{?}{\vdash} R\rangle$

Of the above, only $\vdash\&I$, $\&Er\vdash$ and $\&El\vdash$ perform "goal refinement" in the sense of simplifying at least one formula in the goal without introducing other connectives. $\&I\vdash$, $\vdash\&Er$ and $\vdash\&El$ make the goal more complicated. Furthermore, to perform $\vdash\&Er$ interactively, the user would have to supply a new formula Q (and then demonstrate it); to perform $\&I\vdash$, the user would have to select the two premises to be &-ed. Apart from the increased effort imposed upon the user, by increasing the complexity of the goal there is the risk of infinite chains of tactic applications (allowing the user to become stuck in a "problem loop"). The omission of the other tactics does not change the set of conjectures which can be proven.

Thus, we may adopt $\vdash\&I$ as the IPE rule And_Introduction. In order to present a single elimination rule for &, we combine $\&Er\vdash$ and $\&El\vdash$, producing a tactic which places both P and Q in the subgoal. The new tactic can be thought of as application of $\&Er\vdash$ and $\&El\vdash$ to the same premise in either order. Uses of $\vdash\&I$ can be justified by the and-conclusion of *G3a*: given justifications for the goals $\Delta \overset{?}{\vdash} P$ and $\Delta \overset{?}{\vdash} Q$ we can infer $\Delta \vdash P\&Q$.

Note that in both $\&El\vdash$ and $\&Er\vdash$, the original "argument" $P\&Q$ is left in the premises of the subgoal. In practice, this quickly leads to a large and cumbersome premise-set to be presented to the user at each stage of the proof. Normally, having "extracted" P and Q from $P\&Q$, the latter premise is no longer needed, and so we choose to omit it in the subgoal. Thus the operation of the And_Elimination rule is:

$$P\&Q, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle P, Q, \Delta \overset{?}{\vdash} R\rangle$$

We extend this behaviour (of omitting the "argument" from the subgoal) to all of the IPE's Elimination rules. However, we add a Duplication rule which enables us to add an extra copy of a premise to the premise-list, so that this removal of a premise can be undone by duplicating it prior to the application of the appropriate Elimination rule. Similarly, in the remainder of this section we omit arguments to $r\vdash$ tactics from their subgoals.

Uses of And_Elimination can be justified by the and-premise rule of *G3a*.

The tactics for disjunction $(A\,|\,B)$ are

($\vdash|Il$): $\Delta \overset{?}{\vdash} A\,|\,B \;\mapsto\; \langle \Delta \overset{?}{\vdash} B \rangle$

($\vdash|Ir$): $\Delta \overset{?}{\vdash} A\,|\,B \;\mapsto\; \langle \Delta \overset{?}{\vdash} A \rangle$

($|Il\vdash$): $Q, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle P\,|\,Q, \Delta \overset{?}{\vdash} R \rangle$

($|Ir\vdash$): $P, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle P\,|\,Q, \Delta \overset{?}{\vdash} R \rangle$

($|E\vdash$): $P\,|\,Q, R, \Delta \overset{?}{\vdash} S \;\mapsto\; \langle R, \Delta \overset{?}{\vdash} S \rangle$

($\vdash|E$): $\Delta \overset{?}{\vdash} C \;\mapsto\; \langle \Delta, A \overset{?}{\vdash} C \,;\, \Delta, B \overset{?}{\vdash} C \,;\, \Delta \overset{?}{\vdash} A\,|\,B \rangle$

Clearly, $\vdash|Il$ and $\vdash|Ir$ are useful. In order to maintain the single-Introduction-rule-per-connective pragma, we combine these two rules in the IPE, producing a tactic which generates *both* subgoals (**show A**, **show B**), but which will consider its goal to be proven when *either* subgoal is demonstrated. If we can derive either subgoal, then the or-conclusion rule of *G3a* will allow us to derive the original goal.

As with $\&I\vdash$, $|Il\vdash$ and $|Ir\vdash$ are of no real use in the formula-decomposition proof style.

Deriving Or-Elimination is slightly harder. $|E\vdash$ in its strictest form will not suffice, for it does not reveal any more information. Instead consider $\vdash|E$. Applying this to a goal of the form

$$\Delta, A\,|\,B \overset{?}{\vdash} C$$

gives the three subgoals

$$\langle \Delta, A\,|\,B, A \overset{?}{\vdash} C \,;\, \Delta, A\,|\,B, B \overset{?}{\vdash} C \,;\, \Delta, A\,|\,B \overset{?}{\vdash} A\,|\,B \rangle$$

The third goal is clearly immediate, and the result is simply a "proof-by-case-analysis" tactic. If we restrict the applicability of $\vdash|E$ to goals of this form, then we can omit the third subgoal. This is how Or-Elimination is implemented in the IPE; it is triggered by pointing at a premise which is a disjunction, and will

fail if an attempt is made to apply it to a goal which is not of the above form. Use of Or_Elimination is justified by *G3a*'s or-premise rule.

The tactics for implication $(A \rightarrow B)$ are

$$(\vdash\rightarrow I)\text{: } \Delta \overset{?}{\vdash} A \rightarrow B \quad \mapsto \quad \langle \Delta, A \overset{?}{\vdash} B \rangle$$

$$(\rightarrow I\vdash)\text{: } \Delta, B \overset{?}{\vdash} C \quad \mapsto \quad \langle \Delta, A \rightarrow B \overset{?}{\vdash} C \rangle$$

$$(\vdash\rightarrow E)\text{: } \Delta \overset{?}{\vdash} B \quad \mapsto \quad \langle \Delta \overset{?}{\vdash} A \rightarrow B \ ; \ \Delta \overset{?}{\vdash} A \rangle$$

$$(\rightarrow E\vdash)\text{: } \Delta, A, A \rightarrow B \overset{?}{\vdash} C \quad \mapsto \quad \langle \Delta, B \overset{?}{\vdash} C \rangle$$

The IPE rule Implies Introduction is simply $\vdash\rightarrow I$, whose use is justified by *G3a*'s implies-conclusion rule. Implies Elimination is almost $\rightarrow E\vdash$, except that we do not demand that A be a premise in the goal; instead we generate a second subgoal to show that A can be derived from Δ:

$$\text{Implies Elimination: } \Delta, A \rightarrow B \overset{?}{\vdash} C \quad \mapsto \quad \langle \Delta \overset{?}{\vdash} A \ ; \ \Delta, B \overset{?}{\vdash} C \rangle$$

Use of this rule is justified by the implies-premise rule of *G3a*.

In the IPE, $\tilde{}A$ is treated as $A \rightarrow$ **false**, so that $\Delta \overset{?}{\vdash} \tilde{}A$ is really $\Delta \overset{?}{\vdash} A \rightarrow$ **false** and so on. As a result, Not Introduction in the IPE is

$$\Delta \overset{?}{\vdash} \tilde{}A \ \ (\equiv \quad \Delta \overset{?}{\vdash} A \rightarrow \textbf{false}) \quad \mapsto \quad \langle \Delta, A \overset{?}{\vdash} \textbf{false} \rangle,$$

i.e., we use Implies Introduction. Intuitively, we attempt to prove $\tilde{}A$ by assuming A and trying to reach a contradiction. For Not Elimination, we have a similar situation:

$$\Delta, \tilde{}A \overset{?}{\vdash} B \ \ (\equiv \ \ \Delta, A \rightarrow \textbf{false} \overset{?}{\vdash} B) \quad \mapsto \quad \langle \Delta \overset{?}{\vdash} A \ ; \ \Delta, \textbf{false} \overset{?}{\vdash} B \rangle,$$

using Implies Elimination. In this case, the second subgoal is trivial (since the assumption of falsity renders everything provable), and can therefore be omitted. Intuitively: to prove B when we know $\tilde{}A$, we attempt to prove A from our other assumptions, thus achieving a contradiction.

If we consider **false** to represent an empty conclusion, then these two rules are justified by the not-conclusion and not-premise inference rules of *G3a* respectively.

In the inference rules for the quantifiers $\forall$ and $\exists$, we have extra side-conditions concerned with the handling of free variables.

$\forall$I insists that the variable we bind should not occur free in any assumptions upon which the inner formula depends. This carries across into the basic tactics:

($\vdash\forall I$): $\Delta \overset{?}{\vdash} \forall x P(x) \;\mapsto\; \langle \Delta \overset{?}{\vdash} P(y) \rangle$

provided that the variable y does not occur free in Δ or P

($\forall I\vdash$): $\Delta, P(y) \overset{?}{\vdash} A \;\mapsto\; \langle \Delta, \forall x P(x) \overset{?}{\vdash} A \rangle$

provided that the variable y does not occur free in Δ or P

($\vdash\forall E$): $\Delta \overset{?}{\vdash} P(t) \;\mapsto\; \langle \Delta \overset{?}{\vdash} \forall x P(x) \rangle$

($\forall E\vdash$): $\Delta, \forall x P(x) \overset{?}{\vdash} A \;\mapsto\; \langle \Delta, P(t) \overset{?}{\vdash} A \rangle$

$\vdash\forall I$ leads to the IPE's All Introduction rule; the rule is designed so that when it is used, the IPE will choose a variable name which does not occur free in the goal at that point. However, the user has the ability to change the name of the variable (for example, to a more mnemonic name); the IPE will always check that the chosen variable is indeed "new". [2]

$\forall E\vdash$ becomes the IPE's All Elimination rule. Here, since we know that $P(x)$ holds for all x, we can assume $P(t)$ for any term t [3]. In All Elimination, the IPE allows the user to supply any term to substitute for x in P.

The other tactics are rejected on the grounds that they increase the complexity of the goal, and that their omission does not effect the completeness of the IPE's rules.

---

[2] Another case where the user must change the variable name is when an instance of the All Introduction rule is passed a new goal in which the chosen variable is no longer suitable.

[3] Recall that the IPE's logic is untyped; in a typed logic, we would additionally be forced to show that the type attached to x is non-empty.

The conditions on the ∃E rule also carry across into the tactics:

($\vdash \exists I$): $\Delta \overset{?}{\vdash} \exists x P(x) \;\mapsto\; \langle \Delta \overset{?}{\vdash} P(t) \rangle$

($\exists I \vdash$): $\Delta, P(t) \overset{?}{\vdash} A \;\mapsto\; \langle \Delta, \exists x P(x) \overset{?}{\vdash} A \rangle$

($\vdash \exists E$): $\Delta \overset{?}{\vdash} Q \;\mapsto\; \langle \Delta \overset{?}{\vdash} \exists x P(x) \; ; \; \Delta, P(y) \overset{?}{\vdash} Q \rangle$
provided that the variable y does not occur free in Δ or Q

($\exists E \vdash$): $\Delta, \exists x P(x), Q \overset{?}{\vdash} R \;\mapsto\; \langle \Delta, Q \overset{?}{\vdash} R \rangle$
provided that the variable y does not occur free in Δ, Q or R

Exists Introduction is simply $\vdash \exists I$: to show that P holds for some x, we allow the user to supply any term t to substitute for x in the subgoal, hoping that this can be demonstrated later [4]. In a similar fashion to the choice of $\vdash | E$ for Or Elimination, we choose $\vdash \exists E$ for Exists Elimination Restricting its applicability to goals of the form $\exists x P(x), \Delta \overset{?}{\vdash} Q$ renders the first subgoal immediate. Just as Exists Introduction is the counterpart to All Elimination, choosing $\vdash \exists E$ for Exists Elimination makes it the counterpart of All Introduction: the IPE chooses a "new" variable y to substitute for x in the subgoal; the user is free to change this so long as the chosen variable does not occur free in the other formulae in the goal.

As before, if we succeed in deriving the subgoals resulting from the application of any of the quantifier Introduction and Elimination rules, then the corresponding quantifier-conclusion and -elimination inference rules of *G3a* can be used to derive the original goal.

Two further tactics are provided, mainly for pragmatic reasons. The first, called Remove Antecedent, removes a selected premise from a goal; in practice

---

[4]In many proofs, the choice of t will not be obvious until further work has been performed upon the subgoal; however, as we shall see later, the IPE user is free to change the choice of t at any stage in proof construction.

this is used to remove premises which are not needed in the subproof (and hence to reduce visual clutter). This relies upon the monotonicity of intuitionistic logic. Duplicate Antecedent adds a copy of a selected premise to the subgoal. The reason this is required is that the implementations of the Elimination tactics all remove the premise they acted upon, to avoid clutter in the subproofs. Unfortunately, some proofs require two rule-applications to the same premise, so Duplicate Antecedent is supplied as a means of "doubling up" such a premise prior to its removal. (It should be noted that the use of Duplicate Antecedent could allow "problem loops" of the form discussed previously; thus it is important that this rule be used judiciously in practice).

Table 2–1 summarises the basic tactics of the IPE.

The IPE's Introduction and Elimination tactics are goal-directed implementations of the corresponding rules of the intuitionistic G3a. The goal of a C-Introduction tactic for any connective C in the IPE can be constructed by an application to the subgoals of the rule in G3a which introduces the connective C into the succedent. The case is similar for a C-Elimination tactic and the G3a rule which introduces C into the antecedent, except that in the IPE the relevant antecedent is removed from the subgoals; this latter effect can be undone by a prior use of Duplicate-Antecedent. Thus the rules of G3a act as (implicit) validation functions (in the LCF sense) for the IPE's basic tactics. All of the rules of G3a are relied upon, and completeness of the IPE's tactics follows from the completeness of G3a.

2.4 General Principles

The IPE is built upon a fixed set of basic tactics derived "by hand" from a set of rules for first-order intuitionistic predicate calculus, using Schmidt's tactic schemas. Not all of the possible tactic schemas have been used; some have been discarded on the grounds that they increase the complexity of formulae in the goal, which competes with the overall aims of proof by decomposition.

Though this process of basic tactics generation has been performed on paper rather than by mechanical means, it is worthwhile to consider whether or not it could be generalised and mechanised for any logic expressed via inference rules.

It seems clear that we can generate a set of basic tactics from any set of inference rules using Schmidt's scheme. The problem lies in "thinning out" this set by removing "superfluous" tactics: how can we decide which tactics are superfluous? The essence of the problem is to obtain a set of tactics which preserve completeness and consistency with respect to the original inference rules (the set of statements provable by compositions of the basic tactics should be identical to that set provable from the original inference rules), whilst additionally en-

Table 2–1: Basic Tactics of the IPE

Name	derived from	function
And-Introduction	$\vdash\&I$	$\Delta \overset{?}{\vdash} A\&B \;\mapsto\; \langle \Delta \overset{?}{\vdash} A;\, \Delta \overset{?}{\vdash} B\rangle$
And-Elimination	$\&E\vdash$	$P\&Q, \Delta \overset{?}{\vdash} R \;\mapsto\; \langle P, Q, \Delta \overset{?}{\vdash} R\rangle$
Or-Introduction	$\vdash\mid Il$, $\vdash\mid Ir$	$\Delta \overset{?}{\vdash} A \mid B \;\mapsto\; \langle \Delta \overset{?}{\vdash} B\rangle$ OR $\langle \Delta \overset{?}{\vdash} B\rangle$
Or-Elimination	$\vdash\mid E$	$\Delta, A \mid B \overset{?}{\vdash} C \;\mapsto\; \langle \Delta, A \overset{?}{\vdash} C\,;\, \Delta, B \overset{?}{\vdash} C\rangle$
Implies-Introduction	$\vdash\!\rightarrow I$	$\Delta \overset{?}{\vdash} A \rightarrow B \;\mapsto\; \langle \Delta, A \overset{?}{\vdash} B\rangle$
Implies-Elimination	$\rightarrow E\vdash$	$\Delta, A \rightarrow B \overset{?}{\vdash} C \;\mapsto\; \langle \Delta \overset{?}{\vdash} A\,;\, \Delta, B \overset{?}{\vdash} C\rangle$
Not-Introduction	$\vdash\!\rightarrow I$	$\Delta \overset{?}{\vdash} \,\tilde{}A \;\mapsto\; \langle \Delta, A \overset{?}{\vdash} \mathbf{false}\rangle$
Not-Elimination	$\rightarrow E\vdash$	$\Delta, \tilde{}A \overset{?}{\vdash} B \;\mapsto\; \langle \Delta \overset{?}{\vdash} A\rangle$
All-Introduction	$\vdash\forall I$	$\Delta \overset{?}{\vdash} \forall x P(x) \;\mapsto\; \langle \Delta \overset{?}{\vdash} P(y)\rangle$
All-Elimination	$\forall E\vdash$	$\Delta, \forall x P(x) \overset{?}{\vdash} A \;\mapsto\; \langle \Delta, P(t) \overset{?}{\vdash} A\rangle$
Exists-Introduction	$\vdash\exists I$	$\Delta \overset{?}{\vdash} \exists x P(x) \;\mapsto\; \langle \Delta \overset{?}{\vdash} P(t)\rangle$
Exists-Elimination	$\vdash\exists E$	$\Delta, \exists x P(x) \overset{?}{\vdash} A \;\mapsto\; \langle \Delta, P(y) \overset{?}{\vdash} A\rangle$
Remove-Antecedent	monotonicity	$\Delta, A \overset{?}{\vdash} B \;\mapsto\; \langle \Delta \overset{?}{\vdash} B\rangle$
Duplicate-Antecedent		$\Delta, A \overset{?}{\vdash} B \;\mapsto\; \langle \Delta, A, A \overset{?}{\vdash} B\rangle$

A,B and C are arbitrary predicates; $P(x)$ is an arbitrary predicate possibly containing instances of a variable x; t is an arbitrary term; y is an identifier which should not occur free in the other formulae of the goal. The reader is referred to the text of this chapter for further details.

couraging top-down decomposition. At first guess, a simple criterion would be to discard those tactics which increase the complexity of goal formulae.

A second problem is that some basic tactics which have to be used do not lend themselves directly to our style of proof construction, but require some further treatment first. (For example, $\vdash\mid E$ in the previous section). In the IPE these alterations relied on insights into the realm of applicability of the tactic; it seems improbable that such insights can easily be mechanised; perhaps increased experience of translation-by-hand will reveal some guidelines.

It is probably impossible in general to maintain the "proof-by-pointing" principle of the IPE (viz , having a single decomposition rule for each connective when it occurs as the top connective in a premise or in the conclusion of a goal). In practice, although this principle is useful initially, in that it reduces the amount of effort that the user has to put into the proof, it becomes necessary as proofs of larger conjectures are sought to allow some form of compound rules which may break the "proof-by-pointing" principle. (See Section 6 for the solution adopted in the IPE). For example, adding the fact

$$\forall x \forall y P(x,y) \rightarrow \forall y \forall x P(x,y)$$

can be thought of as adding a new derived inference rule

$$\frac{\forall x \forall y P(x,y)}{\forall y \forall x P(x,y)}$$

Clearly whenever the Schmidt-tactics derived from this rule are applicable to a goal formula, then one of the rules All Introduction and All Elimination will also be applicable.

It may not be possible to preserve the principle even for the basic inference rules of a logic; for example, in cases where two inference rules refer to the same top-level connective but possibly require different substructures or different semantic constraints.

Chapter 3

Attribute Grammars As A Basis For Context-Sensitive Structure Editing

The kernel of the Interactive Proof Editor consists of a context-sensitive structure editor operating within an attribute grammar framework. The use of attribute grammars here was inspired by a paper from the Cornell Synthesizer Generator project [Reps-Alpern 84] which contained the basic idea of using an attribute grammar to define a structure editor for proofs. This paper demonstrated certain desirable properties of the resultant editor. that the validity of the proof could be maintained by the attribute grammar, that alterations to any point of the proof produced instant feedback, and that proof errors introduced by the user could be indicated at the point of occurrence. We decided to experiment further with the notion of 'editable proofs', and to try and develop a system which was exclusively tailored to proof editing (unlike the Cornell system which uses a standardised interface for all its structure editors). We had already developed a smaller version of the Synthesizer Generator, informally titled 'C-SEC' (for Context-Sensitive Editor Creator), written in the language ML (with a preprocessor in C generated by the YACC program [Johnson 78]), and we chose to use this to generate a set of kernel functions for the IPE. In this section, we look at the means by which attribute grammars can be used to form a general framework for context-sensitive structure editors, with particular reference to the C-SEC system. Firstly, the notions of ***attribute grammar***, ***derivation tree***, ***semantic tree*** and ***dependency graph*** are introduced. We then describe a method

for generating editors from attribute grammars and methods for maintaining consistency of the structures generated with respect to an attribute grammar.

3.1 Attribute Grammars

An *attribute grammar* [Knuth 68] is a context-free grammar cfG extended by

- Each symbol S_i in cfG has an associated set of ***attributes*** $A(S_i)$, partitioned into a set of ***inherited*** attributes $A_{inh}(S_i)$ and ***synthesised*** attributes $A_{syn}(S_i)$.

- Each production in the cfG of the form

$$S_1 ::= op(S_2 \dots S_n)$$

(where *op* is the *production name*) has an associated set of *semantic equations*; each equation defines an attribute of a symbol of the production $(S_1, \dots, S_n)$ in terms of a *semantic function* applied to other attributes (called the *arguments* of the equation) in the production. There is precisely one equation for each synthesised attribute of the left-side nonterminal S_1 and for each inherited attribute of each right-side symbol $S_2, \dots, S_n$.

The attributes and semantic equations extend the context-free grammar to a context-sensitive grammar. When we later describe derivation trees and semantic trees, we shall see that attributes act as storage slots for information that is passed between different points of a tree. Inherited attributes store information determined from the root of the tree, whilst synthesised attributes hold information derived from the subtrees. Both kinds of attributes may also use local information, i.e., values of other attributes of the same instance of a symbol. The grammar is said to be in *normal (canonical) form* if the arguments of each semantic equation in a production are inherited attributes of the left-side symbol or synthesised attributes of right-side symbols.

In the notation used in this chapter, each production of the grammar has a unique name, and the productions of a nonterminal are grouped together. The semantic equations of a production follow it in list brackets ('[' and ']'). Comments are enclosed in braces('{' and '}'). This notation is similar to the notation used by C-SEC; a syntax for the C-SEC attribute grammar definition language can be found in Appendix A. In C-SEC, the semantic language is ML: the attributes are typed objects in ML, and the semantic equations are ML expressions. The first major operation of C-SEC is to take an attribute grammar in the description language and 'compile' it into ML code which implements it as an ML object of type **attribute_grammar**. The choice of ML as the semantic base has proven particularly useful in the implementation of the IPE.

The following fragment of an attribute grammar for an integer expression editor gives an example. The nonterminal 'Expr' has two attributes; a synthesised 'value' and an inherited set of 'declarations'. The intuitive meaning of the value of an expression is that it is calculated from the values of any sub-expressions, and the set of declarations of an expression is that of its parent expression plus any local declarations. These intuitive meanings are enforced by the semantic equations of each production. 'Sym$n.attr' denotes the attribute named 'attr' of the nth occurrence of the symbol 'Sym' in the production (numbered from the left). '$n' defaults to '$1' when omitted.

```
Expr ::= Sum ( Expr Expr )
           [ Expr$1.value = Expr$2.value + Expr$3.value;
             Expr$2.declarations = Expr$1.declarations;
             Expr$3.declarations = Expr$1.declarations;
           ]
       | Difference ( Expr Expr )
           [ Expr$1.value = Expr$2.value - Expr$3.value;
                          ⋮
                        (etc)
           ]
       | UseVar ( Var )
           [ Expr$1.value = lookup( Var$1.value, Expr$1.declarations );
           ]
       | Bind ( Var Expr Expr )
             { 'let Var = Expr$2 in Expr$3 end' }
           [ Expr$1.value = Expr$3.value;
             Expr$2.declarations = Expr$1.declarations;
             Expr$3.declarations
                   = add-binding(Var.name,Expr$2.value,
                                   Expr$1.declarations);
           ]
       | ...(other productions of Expr)...
```

An attribute grammar can be viewed as a discipline which a context-sensitive structure editor must follow. The following sections describe structures which can be used by such an editor to create objects which remain consistent with a particular attribute grammar.

3.2 Derivation Trees

For an object **O** constructed in accordance with an attribute grammar Ag, the *derivation tree* of **O**, **DT(O)**, shows how **O** was constructed using the productions of Ag. (If the grammar Ag is ambiguous, **O** may have more than one possible derivation tree).

Each node of **DT(O)** is labelled by a symbol of Ag and the name of a production rule of that symbol. For a production

$$S_0 ::= \text{op } (S_1 \ldots S_n)$$

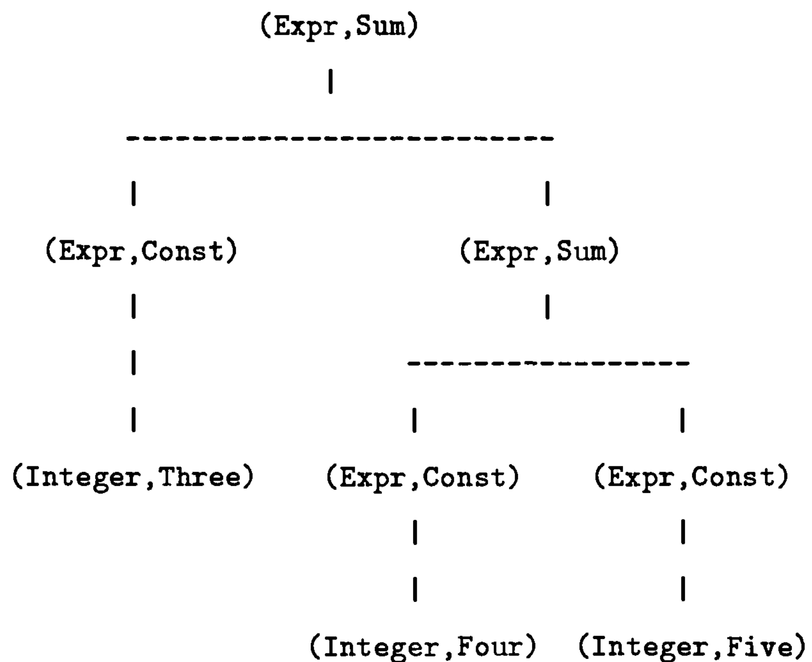
the derivation tree will consist of a node labelled (S_0,op) and child nodes labelled with the symbols $S_1, \ldots, S_n$ and the names of whatever production rules they have been expanded by.

For example, if we have

```
Expr    ::= Const ( Integer )
          | Sum ( Expr Expr )
          ;

Integer ::= One ()
          | Two ()
              ⋮
            (etc)
          ;
```

(ignoring semantic equations for the moment, and using a crude set of rules for Integer), then the following is a possible derivation tree for the expression $3 + 4 + 5$:



Usually, the *concrete syntax* of a language (how its structures are represented textually) differs from its *abstract syntax* and the derivation tree is then referred to as an *abstract syntax tree.* For our purposes, however, the term 'derivation tree' will suffice.

The derivation tree of an object can be used to represent its structure, and in fact is used for this purpose in many structure editors, although the tree structure may be hidden from the user to varying extents.

To construct a structure editor based upon derivation trees, we must extend the idea of derivation trees to incorporate a notion of 'current position', for example as follows:

A *positional derivation tree* is a derivation tree with two distinguished nodes, the *root* node and the *current* node. The current node is the node affected by most editing operations. We may expect motion operations such as 'to-parent', 'to-child-n' and 'to-root', which make the new current node the parent and nth child of the old current node, and the root of the derivation tree respectively. The most basic alteration operation would be 'expand-by-rule', which expands the current node in accordance with the chosen production rule (which must be a production of that node's symbol), replacing any existing expansion of the

current node. This combination of motion and *subtree replacement* operations gives us a simple structure editor, and, given a suitably general definition of expand-by-rule, one which is grammar-independent.

However, the derivation tree is not sufficient in itself to represent context-dependent information; although our structure editor restricts the class of constructible objects to those which satisfy the constraints of the context-free grammar (i.e., the *syntactic* restrictions of the attribute grammar), it is still possible to build objects which break the context-dependencies of the attribute grammar (for example, to construct an Expr which refers to an undeclared variable). In order to build context-dependency into our editable structures, we now introduce the notions of semantic tree and dependency graph.

3.3 Semantic Trees

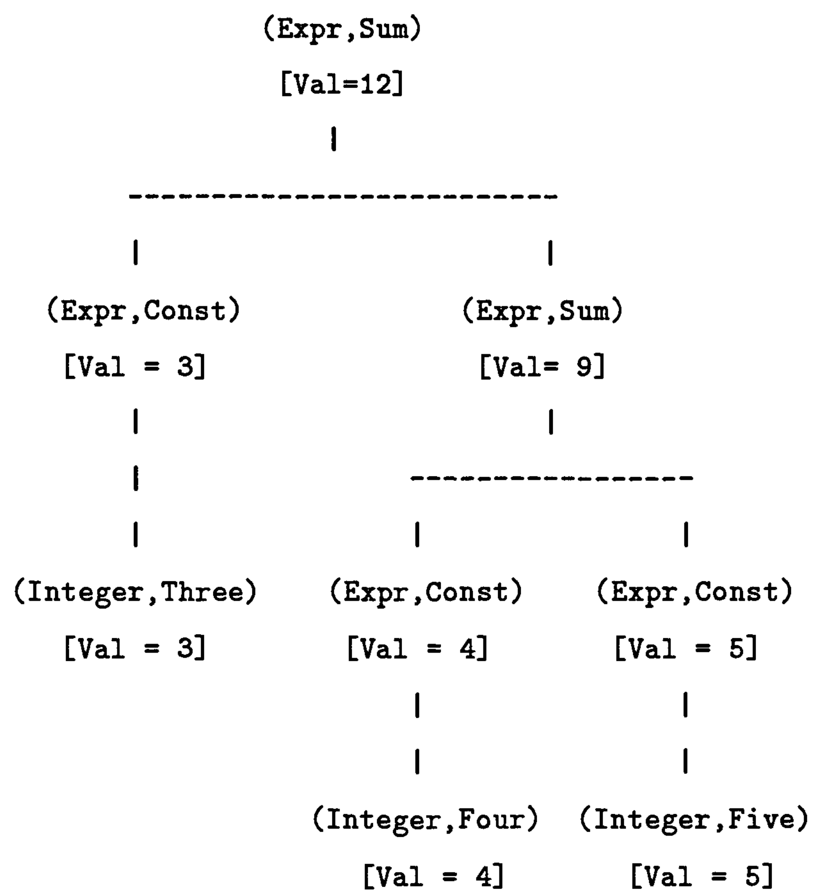
The *semantic tree* of an object combines both the structure of its derivation tree (facilitating "ordinary" structure editing operations) with context-dependent information derived from the semantic equations of an attribute grammar.

A *semantic tree* ST(O) of an object O constructed under an attribute grammar Ag is a derivation tree DT(O) with each node additionally labelled by the *attribute instances* of the symbol which labels that node, where each *attribute instance* of a symbol is the value of an attribute associated with that symbol. Attribute instances of the same symbol but different nodes of the semantic tree are distinct. The values of attribute instances of a node of the semantic tree can be determined by applying the appropriate semantic functions to their intended arguments, which can be attribute instances of the same node, its parent or its children. The semantic function for an inherited attribute of a node is determined by the production rule of the parent; for a synthesised attribute it is the production rule of the node itself. To see this more clearly, consider extending the previous example by associating a synthesised attribute 'Val' with the nonterminals Expr and Integer, and semantic equations as follows:

```
Expr    ::= Const ( Integer )
            [ Expr.Val = Integer.Val; ]
          | Sum ( Expr Expr )
            [ Expr$1.Val = Expr$2.Val + Expr$3.Val; ]
          ;

Integer ::= One () [ Integer.Val = 1; ]
          | Two () [ Integer.Val = 2; ]
            ⋮
            (etc)
          ;
```

and consider the following semantic tree for $(3+4+5)$:



The values of the attribute instances of `Val` for the `Integer` nodes come directly by evaluating the semantic functions for Integer.Val in the production rules for Integer named ‘Three’, ‘Four’ and ‘Five’, as these functions are con-

stants (with no attribute arguments). The values of `Val` in the immediate parent nodes (labelled `(Expr,Const)`) are copies of the values of the child attributes, in accordance with the semantic equation for Expr.Val in the Const production. Similarly, the attribute instance at each `Sum` node is the sum of the values of the attribute instances of the two children of the node. The final value of the overall expression has been calculated from the values of its subexpressions in a structured fashion. Such a semantic tree, where the value of every attribute instance is the same as the value calculated by applying its semantic function to its arguments, is called *consistent*. An editing operation such as "expand-by-rule" can destroy the consistency of a semantic tree, and thus it is important to be able to restore the semantic tree to a consistent state after such an operation.

3.4 Dependency Graphs

It is important to note that a semantic tree does not contain any information as to where the arguments for a semantic function may be found; in other words the semantic tree lacks the dependency relationship between attribute instances. This information is contained in the *dependency graph* of the semantic tree.

An attribute instance **a** is said to *depend directly* upon an attribute instance **b** if **b** is an argument of the semantic function for calculating the value of **a**.

The nodes of the *dependency graph* DG(St) of a semantic tree St are labelled by the attribute instances of St; for nodes labelled by instances **a** and **b** there is a directed edge (**a**,**b**) if **a** depends directly on **b**.

A dependency graph is said to be *circular* if it contains a cycle; in this case, none of the attribute instances in the cycle can be properly evaluated according to their semantic functions. An attribute grammar is similarly said to be circular whenever it is possible to construct a semantic tree under the grammar whose dependency graph is circular.

The information in the dependency graph can be used to evaluate a semantic tree in an efficient way. When a semantic tree is altered during the course

of editing, it is possible to determine and follow the altered dependencies in an efficient manner which ensures that no attribute instance is evaluated more than once after each alteration, and that no instance is needlessly re-evaluated if its argument instances do not change. This is a useful feature in context-sensitive structure editing, as a desired fast response to the user competes with the complexity of the context dependencies. The possible direct dependents of each attribute in a node are restricted to the attributes of that node, its parent and its children Therefore, when an operation such as "expand-by-rule" is performed it is easy to determine which attributes have been directly affected.

3.5 Completing Productions

During an editing session, the syntax trees built by the user are often incomplete (from the user's point of view at least), with one or more nonterminal nodes *unexpanded.* Whilst it is certainly possible to construct derivation trees, semantic trees and dependency graphs corresponding to such structures, there are problems in evaluating attribute instances in such cases. The essential question is, 'What should be the values of attribute instances of an unexpanded nonterminal?'.

The simplest solution would be to assume such instances to have a certain default value, **null**, distinct from all other values they might take. However, we follow the Cornell Synthesizer Generator in the view that such an approach is too limited and that unexpanded nodes should be capable of 'responding to' or 'bouncing back' contextual information in a similar fashion to production rules whose right-hand side is empty. Linked with this is the idea that the semantic tree should be maintained consistent by the system after each alteration.

The role of unexpanded nonterminals in editing is catered for in the notion of a *completing production.* C-SEC insists that each symbol has a completing production which represents the 'default' production for that symbol, and is grafted on to represent an unexpanded instance of that symbol. The completing

production must have no right-side symbols, and allows the definition of values for the synthesised attributes of the symbol, perhaps in terms of inherited attributes. Thus the user can receive context-dependent feedback throughout the editing session.

3.6 Obtaining Semantic Trees from Attribute Grammars

It is now possible to describe the process by which a context-sensitive editor can be generated from an attribute grammar which includes a completing production for each nonterminal. (The method of maintaining consistency of context-dependent information is not described in this section but comes under Section 3.7). The resultant structure editors will at the lowest level construct semantic trees on a rule-by-rule basis; that is, the production rules of the grammar will provide the basic building blocks for the editor, and these *elementary* semantic trees can then be grafted onto a main semantic tree at the 'current position'.

The completing rule for each symbol S yields an elementary semantic tree consisting of a single node labelled by S and a set of instances of the attributes of S. (This is called the *completing tree* of S).

A production rule of the form

$$S_0 ::= op(S_1 \ldots S_n)$$

translates to an elementary semantic tree whose node is labelled (S_0, op) (together with a set of instances of attributes of S_0) and whose children are the completing trees of $S_1, \ldots, S_n$.

In this fashion, an attribute grammar can be processed to produce a set of elementary semantic trees, each labelled by the appropriate production name, which can then be handled by a general tree manipulation package to provide the kernel of a structure editor. This is the second major operation of the C-SEC system: given an attribute grammar as an ML object, C-SEC will process

it to produce a set of ML functions for the construction of semantic trees and dependency graphs under the grammar. These can then be used as the kernel for a structure editor. (C-SEC provides no interactive user interface; this must be built from the tree-construction functions, and possibly 'hooks' inside the particular attribute grammar, such as attributes for display text generation).

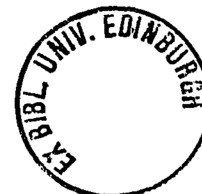
3.7 Incremental Reevaluation

There are two main problems associated with semantic trees; the attribute instances consume large amounts of storage, and the calculation of their values is often costly. C-SEC leaves problems of space-saving to the attribute grammar designer rather than attempt general solutions such as having attribute instances share space when they contain common substructures[1].
This section describes attempts to alleviate the second problem.

In any interactive editing system it is necessary to keep the users up to date with the state of the structure that they are editing. In a context-sensitive editor, this means that the semantic tree must always be consistent, at least to the extent required by the user[2]. It may be essential to reevaluate certain attribute instances after each grafting operation, upon demand for information by the user or even after every movement operation. For example, after grafting a new semantic tree onto the main tree at the current position, the resulting semantic tree is almost certainly inconsistent with the attribute grammar. Perhaps the only inconsistent attribute instances are amongst those of the 'current node'; however, in most cases the value of other instances in the semantic tree will be dependent upon the 'current attribute instances', and will also require reevaluation. (In fact, those instances which *may* require reevaluation are the

[1]Reps' thesis ([Reps 82]) describes some methods for reducing the storage problem.

[2]See Section 3.7.2 for an explanation of this qualification.


EX BIBL. UNIV. EDINBURGH

set of instances in the dependency graph of the new semantic tree which are reachable from the current instances). The process of maintaining a semantic tree by the reevaluation of attribute instances after each update of the tree is known as *incremental reevaluation.*

The essential problem of incremental reevaluation is that local changes to a semantic tree may produce non-local changes in attribute instance values, and it is not possible to predict precisely which instances will require reevaluation. Furthermore, the order in which the values of instances are re-calculated is important; if care is not taken, it is possible that some attribute instances will be evaluated more than once. As an example, suppose that attribute instance **a** depends upon instance **b**, but **a** is reevaluated first. Then **a**'s new value is calculated using **b**'s old value. If at some later point in the reevaluation process **b** is reevaluated and changes value, then **a** will have to be evaluated a second time, since it depends upon **b**. Thus the first evaluation was wasteful (and costly if **a**'s semantic function is complicated) and should have been avoided. The information contained in the dependency graph can be used to avoid such unnecessary computations; by following paths of dependencies, an algorithm can determine the order in which they should be reevaluated to ensure that when an instance is evaluated, all of its arguments are up-to-date. In addition, should none of the arguments of an instance change value during reevaluation (because either they did not require reevaluation, or their reevaluation did not change their value) then that instance need not be reevaluated; this is easily detected (provided there exists a suitable equality function for the attribute value types). Ideally, the number of reevaluations should be as small as possible; also, the number of instances accessed should be as small as possible (however this condition becomes secondary if we assume that the cost of accessing an instance is lower than the cost of instance evaluation).

The two main approaches to attribute evaluation are known as "change propagation" and "pebbling". (These terms are used in Reps' thesis). As the name suggests, change propagation assumes that the entire semantic tree was previously consistent, but now some attribute instances have changed value. Given

this set of changed instances, a change propagation algorithm works by determining whether or not their dependants' values change as a result, and then recursively propagating the changes as far as is necessary through the semantic tree. In pebbling, the value of a chosen attribute instance is determined by similarly determining the values of its arguments. This leads to a "demand-driven" approach, where we demand the value of a particular attribute instance, rather than attempt to restore consistency of the entire semantic tree.

The algorithm chosen in C-SEC uses the pebbling approach, together with the "time-stamping" of attribute instances to determine whether or not an instance *could* have changed value.

3.7.1 Jalili's Incremental Reevaluation Algorithm

The incremental reevaluator used in C-SEC is derived from an algorithm presented by Jalili in [Jalili 83] which ensures that attribute instances are evaluated in the correct sequence. This algorithm will perform correctly for any attribute grammar ., including circular grammars in the sense that it can detect and report a circularity in a dependency graph.

In its 'strict form' (as presented in [Jalili 83]) the algorithm assumes that the aim is to maintain consistency of the synthesised attributes of the root of the semantic tree, and it begins by trying to *validate* their values. To validate an instance, the algorithm first tries to recursively validate (and reevaluate if necessary) the argument instances of the semantic equation for that instance. The validation of an instance depends upon a notion of 'global time' and 'time of last update' for that instance.

Other than the semantic tree, the algorithm requires three extra sets of information:

1. The 'global time' status integer (called **increment** by Jalili). This is associated with the semantic tree. Initially zero, it is incremented after each

tree alteration This gives a notion of time against which the status of each attribute instance can be compared.

2. The local **status** of each attribute instance. **status** reflects the validity of the attribute with respect to the current time. The possible settings of status are:

   (a) **never_evaluated**. This means that the attribute instance has never been computed from its semantic equation and arguments, and is the initial setting of **status** for every instance.

   (b) **visited**. This is a temporary setting, used during reevaluation to check for circularity. When it is required to validate an instance, its **status** is set to **visited** until it is validated; if at any point in the process an instance is found whose **status** is **visited**, then this instance must depend upon itself; i.e., a circularity has been detected.

   (c) **up_to_date(time(t))**. This records that the instance has been validated up to time t; if t is the present value of **increment**, then the instance is validated.

3. The 'last update time' status integers, called **time**. Associated with each attribute instance, **time** records the last time the instance's value changed upon reevaluation, as the value of **increment** at the time of change. **time** is initially zero for all instances, indicating that they have never been evaluated.

The algorithm works by 'pebbling' the dependency graph. Given an initial attribute instance to be validated, the reevaluator must first recursively validate its arguments (ie those instances upon which the initial instance depends directly); then it determines from the values of the arguments of each instance whether or not the instance need be reevaluated. Clearly, if no argument to an instance has changed value since that instance was last validated, then the instance need not be reevaluated. The reevaluator is a short ML function which

is grammar-independent (The author is here indebted to Oliver Schoett for an efficient re-implementation of his original code). Figure 3–1 lists the code

```
val rec evaluate( time(now) )( a : 'a attr_inst )
                  { time(now) is the current time,
                    a is the instance to be validated }
  = case !(status_of a) of
       visited . escape "Circularity"
                  { a depends upon itself }
     | never_evaluated . { evaluate a for the first time }
          ( status_of a := visited;  { to check for circularity }
            it_tuple( evaluate( time(now) ))( args_of a );
                       { validate a's arguments
                         (bring them up to date) }
            value a := eval_inst a;       { evaluate a from its args
                                                and semantic function }
            time_of a := time(now);    { record time a changed at }
            status_of a := up_to_date(time(now))  { a validated }
          )
     | up_to_date(time(t))  .
          if t = now then ()                  { a already validated }
          else(    { a validated in the past, but must re-validate now }
               status_of a := visited;
               it_tuple( evaluate( time(now) ))( args_of a ); { as above }
               if some_arg_has_changed_later( a, args_of a )
               then    { must recalculate a's value }
                    let val new_value = eval_inst a
                    in
                        if not( new_value = !(value a) )
                        then (   { a has changed, so record time }
                                value a := new_value;
                                time_of a := time(now)
                             )
                        else ()   { a has not changed value - do nothing }
               else ();  { none of a's args have changed more recently
                            than a, so no need to recalculate a }
               status_of a := up_to_date(time(now))
              );
```

Figure 3–1: ML code for Jalili's incremental reevaluation algorithm

3.7.2 Evaluation on Demand

The major drawback of Jalili's algorithm is that it reevaluates only those attribute instances which are required to evaluate one or more of the initially-supplied instances, ie., those instances reachable in the dependency graph from

the initial instances. Thus, in the strict version, any attribute instances upon which the root synthesised instances do not depend will not be properly updated after alterations to the semantic tree. In other words, the strict version of Jalili's algorithm only guarantees to maintain full consistency of a semantic tree when the root synthesised instances are dependent upon every instance in the tree (excepting themselves). In editing, where the current tree is as important as the main tree, it is imaginable that there may be attribute instances whose values do not affect those of the root; in such cases, Jalili's algorithm must be used with care.

On the other hand, this feature of Jalili's algorithm is often a boon, for it is a form of *demand evaluation*; only the initially-given instance (and the instances it depends upon) will be reevaluated. This method is useful when certain attributes need only be evaluated after certain changes, or upon demand by the user. For example, in a scope-checking editor for a programming language the user may only want to check his variable declarations infrequently; by not evaluating the relevant attribute instances until requested by the user, a great deal of computation may be avoided during normal editing operations, where speed of response may be crucial[3]. Of course, it is possible to supply non-root instances to the reevaluator, but it is then important to remember that any instances depending upon these will not be reevaluated. In the IPE, we are more often interested in some subset of the synthesised attributes of the current node (of the semantic tree) than in those of the root, so demand evaluation allows us to avoid a great deal of unnecessary reevaluation.

One disadvantage of the Jalili algorithm is that although the process of validation minimises the amount of reevaluation required to validate an attribute instance, all of the instances upon which that instance depends must be at least visited. The incremental reevaluator used in the Cornell Synthesizer Generator

[3] Of course, it may happen that other instances depend upon the 'demand instance', in which case the latter will be evaluated after all. However, a well-designed attribute grammar should keep attributes used for different purposes as independent as possible.

in a sense works in the opposite direction to Jalili's algorithm, being a *change propagation* algorithm. Starting from the instances at the point of change (usually the current node), the reevaluator follows the dependency graph to find all instances dependent upon these instances, and works out an 'optimal order' in which they should be evaluated, called a *model*. The model restricts the dependency graph to the attributes of the chosen node (so that it shows any transitive dependencies amongst them). By remembering the model of each node in two parts – one for the parent of the current tree, another for the children, and by restricting to normal form attribute grammars, it is easy to determine whether or not some tree alteration has invalidated a part-model. The advantages of this method over Jalili's algorithm are that it ensures consistency of the entire semantic tree, and that it avoids the 'visitation problem' described above. However, this approach is best suited to maintaining consistency of an entire semantic tree. When demand attributes are incorporated in Reps' reevaluation algorithm, they are evaluated using the same pebbling fashion as in the Jalili algorithm. We are often more interested in attribute instances local to the current node than in full consistency, and so we chose Jalili's method.

3.8 The Attribute Grammar for the IPE

In this section we develop the attribute grammar (hereafter referred to as the Proof Grammar) from which the kernel of the Interactive Proof Editor is constructed. We describe the context-free part and the major attribute systems in general terms. Then we give some example rules, including special cases. Finally we describe how "hooks" to the user interface of the IPE are built into the attribute systems.

3.8.1 The Context-Free Grammar

In the IPE, we present a language for proving fopc ("first-order predicate calculus") formulae in a similar fashion to programming languages: just as programs in a normal programming language are constructed from a set of basic statement-forms (egs. assignments, conditionals, loop constructs), IPE proofs

are constructed from a set of basic proof steps The bulk of the grammar consists of productions of the form·

Proof ::= SomeRule(Proof Proof ...)

for the distinguished grammar symbol Proof. Typically, "SomeRule" involves the application of one of the IPE's basic tactics (see Section 2) to the goal-sequent of the left-hand instance of Proof.

Certain proof operations (such as quantifier introduction and elimination) require extra information such as a term or identifier to be substituted for the bound variable throughout a goal. In the context-free part, the symbols Term and Var are used to represent these:

Proof ::= ExistsIntro(Term Proof)

Although the IPE uses sequent-calculus notation within proofs, goals are introduced as fopc formulae. The root symbol of the Proof Grammar, called Theorem, is used to pass a Formula (represented by another grammar symbol) to a Proof as its initial goal:

Theorem ::= Theorem(Formula Proof)

When this rule is applied to an instance of the Theorem symbol, the intention is that the instance of Proof should be considered as an attempt to prove the formula associated with the instance of Formula.

In the context-free part of the Proof Grammar, formulae and terms are considered as terminal objects, their structure being contained in their attribute values. It was expected that formulae and terms would always be parsed when input, whilst proofs would always be built by structure editing. The raw result of processing an attribute grammar in C-SEC provides rudimentary tools for structure editing, but none for parsing; this guided the decision to split the grammar at this point. (It must be pointed out that the more sophisticated Cornell Synthesizer Generator supports the generation of attributed trees from parsed input, with parsing schemes presented as part of the grammar description).

3.8.2 The Major Attribute Systems

At any point in a top-down structured proof, there are two main pieces of information:

- the current goal, which will have been derived from goals of ancestor points in the proof;

- the proven/unproven status of the goal, which depends upon the status of goals of subproofs, and also upon validity checks upon the application of the current proof rule to the goal (for example, it is not valid to apply And_Introduction to show $A \rightarrow B$).

Correspondingly, in the Proof Grammar the symbol Proof has two attributes (amongst others):

- an inherited goal (called **sequent**), whose value is determined from the value of the goal of the parent Proof node and possibly other rule-specific information;

- a synthesised boolean **proven**, whose value depends upon the same attribute of Proof children, and also upon local applicability conditions.

The goal-sequents are ML objects of type "sequent". This type was defined for the IPE, together with functions corresponding to the "basic tactics" of Chapter 2. Much of the original code for this type was implemented by John Cartmell.

A third synthesised boolean attribute, **appropriate** is used to record the applicability of a proof step, primarily because this may be required in several semantic equations. Similarly an attribute **subgoals** is used to record **all** of the subgoals produced by a basic tactic.

3.8.3 Some Example Rules

A typical production in the Proof Grammar is that corresponding to the basic tactic And_Introduction:

```
Proof ::= And_Intro ( Proof Proof )
              [ Proof$1.appropriate = is_And(succedent Proof$1.sequent);
                Proof$1.proven = Proof$1.appropriate
                                 & (Proof$2.proven & Proof$3.proven);
                Proof$1.subgoals = if Proof$1.appropriate
                                   then And_intro Proof$1.sequent
                                   else [empty_sequent; empty_sequent];
                Proof$2.sequent = hd Proof$1.subgoals;
                Proof$3.sequent = hd(tl Proof$1.subgoals);
              ]
```

Thus:

- an application of And_Intro to an instance of Proof is only considered appropriate when the **sequent** of that instance has a conclusion of the form A&B (for any formulae A and B);

- the (**sequent**/goal attributing the) Proof instance expanded by And_Intro is considered proven whenever the rule is appropriate and both subproofs are considered proven;

- the subgoals of the rule are those generated by applying the basic tactic And_Intro to the sequent of the Proof instance being expanded; note that if the rule is not considered appropriate then the tactic is not applied, the subgoals being set to "empty sequents" instead (where an empty sequent is $\vdash$ False, which cannot be proven);

- each subproof inherits one of the above subgoals.

In productions corresponding to Elimination rules, another attribute **selected** is used to record to which premise the rule is intended to be applied.

selected simply records the position of the premise in the sequent as an integer index; for example:

```
Proof ::= And_Elim ( Proof )
            [ ...
              Proof$1.subgoals = And_Elim( Proof$1.sequent,
                                           Proof$1.selected );
              ...
            ]
```

As **selected** is a synthesised attribute, we have to give a semantic equation for it in each Elimination production[4]. This equation always sets **selected** to the constant **1**. In practice, this equation is ignored, because we want to have **selected** record the position of the premise which was chosen by the user. The means by which this is achieved will be shown later. *(See §3.8.4).*

Quantifier Rules

When a rule generates a subgoal from a goal by removing a quantifier from a formula, the interface (and hence the grammar) must cater for the substitution of some term for the bound variable. When All_Introduction and Exists_Elimination are applied to a goal, the substituted term must be an identifier that does not already occur free in any formula in the original goal; in the case of All_Elimination and Exists_Introduction, the choice of term is not thus restricted. The Proof Grammar has to cater for both situations.

An early design decision for IPE was that though IPE could make an initial choice of "new identifier" for the first rules, the user should be able to change

---

[4]In fact, every production for Proof in the Proof Grammar has to include a semantic equation for **selected**; this is a shortcoming of C-SEC. The CSG allows a production to have local attributes (for example to record or report errors which are specific to that production).

this. This permits the choice of a more meaningful name. A symbol Var of the Proof Grammar is used to represent identifiers; it has an attribute **self** which is a synthesised string containing an identifier name. The user can change the name of the identifier via the same mechanism which allows editing of the initial conjecture. The grammar must always check that the identifier does not occur free in the goal: this is done using an ML function, "is_unique_identifier".

```
Proof ::= All_Intro ( Var Proof )
          [ Proof$1.appropriate = is_ForAll(succedent Proof$1.sequent)
                                  & is_unique_identifier
                                      (Proof$1.sequent, Var.self);
            Proof$1.proven = Proof$1.appropriate & Proof$2.proven;
            Proof$2.sequent = if Proof$1.appropriate
                              then hd(All_intro(Proof$1.sequent, Var.self))
                              else empty_sequent;
            ...
          ]
```

The ML function "is_unique_identifier" takes a sequent and an identifier name, and returns true if the identifier does not occur free in any formula in the sequent. Whenever the identifier is changed, the reevaluation process described earlier will check the appropriateness of the production rule with the new identifier.

All_Elim and Exists_Intro are slightly simpler, in that there is no restriction upon the substitution term; thus the appropriacy check is of the usual form, ie, "is the selected formula of the form $\forall x P(x)$ (or $\exists x P(x)$)?". As with Var, a symbol Term is used to represent the substitution term. Its **self** attribute is set to a default value in the grammar (a new identifier of form "TERM_n" for some integer n), but the intention is that the user should change this to something more useful.

In 3.9.1 we discuss a possible attribute-grammar-based technique by which interactive proof editors could attempt to choose suitable terms for substitution.

The Or_Intro Rule

The Or_Intro rule is another rule of particular note. In order to prove that $A|B$ follows from a set of premises, it suffices to show that either of A, B follows from the same set. However, we need not know in advance which conclusion is the better to attempt to prove. The IPE allows the postponement of this decision by presenting both subgoals and allowing the user to continue work upon either, or indeed to expand both subproofs until the choice becomes clearer. The Or_Intro rule expands a Proof (whose goal-conclusion must have the form $A|B$) into two subproofs, one of which inherits A as goal-conclusion, the other inheriting B similarly. The rule considers the parental Proof proven when either subproof is completed.

The Duplication Rule

All of the Elimination production rules in the grammar produce a subgoal from which the chosen premise has been omitted. This has been done to reduce "clutter" in the presentation of IPE proof steps to the user. However, there are cases where the same premise may be required more than once in the same line of proof (ie as opposed to branching subproofs).

The solution presented in IPE is unfortunately rather crude: the Proof Grammar includes a Duplication rule which duplicates the chosen premise in its subgoal. Upon discovering that a deleted premise is required again, the user must look back up the proof tree until a node containing the premise is found, and then insert a Duplication rule at that point in the proof. This will generate a subgoal which contains two copies of the selected premise, so that when one is removed by the application of a suitable Elimination rule, the other will remain.

In the IOTA system [Nakajima *et al.* 83], premises are hidden from the user after they have been used, but can be recalled if required. A method such as this would be more useful: this might be performed by retaining all premises in a goal, but marking certain of them as "hidden", and modifying the displaying of

goals accordingly. Now the question becomes that of how to organise "hidden" premises and present them to a user who wishes to use some of them again.

3.8.4 Interface Considerations

The result of applying the C-SEC attribute grammar processor to the Proof Grammar is a set of ML modules which implement the Proof Grammar as an instance of the type **attribute_grammar** in ML. This polymorphic type has associated functions which can be used to build semantic trees conforming to an attribute grammar, and to perform incremental reevaluation upon them. However, no other interface is provided as part of the polymorphic package; there is no general pretty-printer for example.The interface to the Proof Grammar was "hand-built" on top of the basic semantic-tree constructors.

Interactive Attributes

As stated previously, there are points in the Proof Grammar which require information from the user: which premise to select, which term to substitute for a bound variable, and which formula to take as initial conjecture. The solution chosen was to provide "hooks" in the grammar which allow a restricted form of outside interference. These take the form of attributes whose values are supplied by some outside operator.

For example, the synthesised attribute **self** of the symbol Formula is used in the Theorem production to act as the initial conjecture of a proof. In the Proof Grammar, its semantic equation is

Formula.self = atomic("FORMULA",nil)

Since this equation contains no arguments, it defines **self** as a constant formula (the atomic predicate "FORMULA" with no arguments). However if we allow the value of (an instance of) this attribute to be changed from outside, and ensure that subsequent reevaluation of dependent attributes will use the new

value, then we have the ability to update the value of a formula supplied to a proof and see the effect it has upon the proof.

We refer to attributes whose values are intended to be supplied from outside the attribute grammar as *interactive attributes*. The semantic equation for an interactive attribute defines an initial default value for the attribute. This equation should have no argument attributes; otherwise if one of the arguments were to change, then the interactive argument could have its default value unexpectedly re-invoked. Interactive attributes in the Proof Grammar include the **self** attributes of the symbols Formula, Term and Var (for supplying the user's substitutions), the **selected** attribute used in Elim productions and attributes which are used to control the display of the proof tree and depend upon a "current position".

The next level of ML code above the Proof Grammar contains functions which can be used to "plug" new values into interactive attribute instances. These replace the semantic function given in the grammar with a new function which returns the new value, sets the **status** of the attribute instance to **never_evaluated** and its "last update time" to zero. Thus, whenever a dependent attribute instance is reevaluated, the interactive attribute instance is evaluated as if for the first time; its new value is set and used in the calculation of the dependent.

Such a system is open to abuse; a fuller implementation of C-SEC might include the declaration of interactive attributes, check that their semantic equations do not contain argument attributes, and generate functions of the above form which ensure that they are properly altered. An early version of C-SEC had "interactive symbols", which had only one attribute and one production rule (which set the single attribute to a default value). The intention was to cater for symbols corresponding to "lexical classes" (such as "integer" or "string" or, in the case of the Proof Grammar, "Formula"). This was not used in the Proof Grammar, because the "lexical" symbols Formula, Term and Var all require more than one attribute.

Attributes for Display Control

As an IPE proof structure grows in size, it becomes impossible to display all of it upon a single screen. It is possible to present the entire structure and allow the user to scroll through it, but this is both expensive to produce and cumbersome to use in large proof structures. Therefore we aim to display an area of the proof structure which lies around the user's current position in the proof. Normally, the intention is that the amount displayed should be roughly a screenfull; however, the user should have control over the amount displayed.

The display control is organised around the depth of the proof structure. At any time, the user will see no more than some fixed number of levels above and below the current position in the structure. As the current position moves down the tree, the upper levels of the structure will disappear from the display; the same applies to lower levels when the current position moves upwards.

In section 3.2 we described a positional derivation tree as a derivation tree with a root node and a current node. In the IPE's proof structures there is a third distinguished node, the *display root*. This is the node from which the display of the proof structure is generated. Initially the display root is the same as the root node; however the display root is constrained to stay within a fixed (but user-alterable) number of tree levels above the current node. Thus the display root follows the current node. At the same time, only a certain number of tree levels below the current node are to be displayed. We shall refer to the number of levels to be displayed above and below the current node as "display-above" and "display-below" respectively.

Each symbol of the Proof Grammar has an attribute **print_tree_depth** which is used to determine how far the node is below the display root. (In fact, it records to what depth its subtrees should be printed). Normally, this attribute inherits its value as one less than its parent instance. However, at the display root it must be set to the sum of display-above and display-below. This is done by using a separate interactive attribute, **set_ptd**. This has default value -1; however, when a node is made the display root, **set_ptd** is set to

display-above+display-below. The semantic equation for **print_tree_depth** is in fact

> Symbol$2.print_tree_depth =
>   if Symbol$2.set_ptd > −1
>   then Symbol$2.set_ptd
>   else Symbol$1.print_tree_depth −1

(where Symbol$1 is the left-side symbol in the production and Symbol$2 is any right-side symbol). It is now vital to ensure that the value of **set_ptd** is reset to −1 when the display root is moved elsewhere in the proof structure. This would ensure that **print_tree_depth** is correctly set for all nodes below the display root.

3.9 Some Suggested Improvements of the Proof Grammar

3.9.1 Choosing Terms for All Elimination and Exists Introduction

There is much room for improvement here concerning the choice of suitable terms for All_Elim and Exists_Intro. Rather than expecting the user to choose the term (as is the case in the IPE at present), the IPE could expend some effort in determining a "good" choice. For example, if we had a point in a proof with the goal

> show CannotSpel(Brian),... entails ?x CannotSpel(x)

then it might seem reasonable to expect that applying Exists_Intro to this would produce

> CannotSpel(Brian),... entails ?x CannotSpel(x)
> by Exists Introduction with <Brian> for x

and CannotSpel(B:ıan),... entails CannotSpel(Brian)
 is immediate

This is a very special case, which can be solved simply by matching the inner formula of the quantification against each premise in turn. In general, it need not be obvious what the substitution should be, without further expansion of the proof. For example it would be more difficult to handle the following goal:

show $P(TERM_1) \rightarrow Q(TERM_1), P(x)$ entails $?xQ(x)$

(where $TERM_1$ is a substitution for a variable bound by a universal quantifier). This could be handled by building information about implication elimination into the production for Exists_Intro; however, this would be an untidy solution as it involves encoding knowledge of other proof rules into a single rule.

As another solution we could utilise the attribute grammar mechanism and use information synthesised from subproofs to guide the choice of the substitution. A proof node could inherit the set of "su'bstitutable variables" generated by ancestral All_Elim and Exists_Intro steps, and receive (synthesise) sets of possible substitutions from subproofs. The latter sets of substitutions could then be analysed to determine whether or not a suitable substitution can be found. For example, one possible expansion in a proof including the above goal might be:

show $!x(P(x) \rightarrow Q(x)), P(x)$ entails $?xQ(x)$
use Exists_Introduction with $< TERM_1 >$
andshow $!x(P(x) \rightarrow Q(x)), P(x)$ entails $Q(TERM_1)$
  use All_Elimination with $< TERM_2 >$ on premise 1
  andshow $P(TERM_2) \rightarrow Q(TERM_2), P(x)$ entails $Q(TERM_1)$
    use Implies_Elimination on premise 1
    andshow $P(x)$ entails $P(TERM_2)$
    andshow $Q(TERM_2), P(x)$ entails $Q(TERM_1)$

The first unexpanded leaf could indicate that substituting x for $TERM_2$ would prove its goal; the second could indicate that the substitutions for

$TERM_1$ and $TERM_2$ would have to be the same. The parent node (expanded by the Implies_Elim rule) must then combine these requirements to generate the requirement that both $TERM_1$ and $TERM_2$ should be substituted by x.

In general, a proof node could generate sets of alternative substitution-sets for term variables in its goal, each set describing a set of substitutions for term variables which would complete the proof. We shall refer to the set of alternatives as the *requirements* of a node, and to each element of the same as an ***alternative***.

If the first leaf node above had an additional premise $P(y)$, then the node would produce the following set of alternative requirements:

$$\{\{TERM_2 \mapsto x\}, \{TERM_2 \mapsto y\}\}$$

The substitution-sets may record a need to unify two term variables, as in the second leaf node above. Here will use a # prefix to denote unification variables. The second leaf could produce:

$$\{\{TERM_1 \mapsto \#v, TERM_2 \mapsto \#v\}\}$$

This will be satisfied by any substitution-set in which the two term variables can have the same substitution.

Proof productions which produce a single subproof and do not introduce any term variables can simply pass the requirement of the subproof to the parent node (ie, the semantic function is the identity function).

Proof productions which are "conjunctive" (in the sense that they produce two (or more) subproofs, and where all subproofs must be proven before the proof can be completed) must unify the requirements of the subproofs to determine a single requirement. This can be done by considering all pairs of alternatives (A_1, A_2) such that A_1 is from the first requirement and A_2 is from the second. We discard all pairs which contain inconsistent substitutions for some term variable, and form the new requirement as the set containing $A_1 \cup A_2$ for each remaining pair. When an alternative-pair does not contain any unification variables, then checking for inconsistent substitutions involves checking that if a term variable

is mapped in both alternatives, then it is mapped to the same term. If one subproof's requirement is

$$\{\{TERM_1 \mapsto x, TERM_2 \mapsto y\}, \{TERM_1 \mapsto y, TERM_2 \mapsto x\}\}$$

and another has requirement

$$\{\{TERM_1 \mapsto y, TERM_3 \mapsto g(z)\}, \{TERM_1 \mapsto z\}\}$$

then the resultant requirement is

$$\{\{TERM_1 \mapsto y, TERM_2 \mapsto x, TERM_3 \mapsto g(z)\}, \\ \{TERM_1 \mapsto x, TERM_2 \mapsto y\}\}.$$

When an alternative A_1 maps a term variable $TERM_1$ to a unification variable $\#v$, then in comparing this alternative with another A_2 from the other requirement, $\#v$ must be unified with the mapping (if any) for $TERM_1$ in A_2. Then consistency can be checked as above. In the partial proof above, the Implies_Elim node would have to unify the requirements of its two subproofs, these being

$$\{\{TERM_2 \mapsto x\}\}$$

and

$$\{\{TERM_1 \mapsto \#v, TERM_2 \mapsto \#v\}\}.$$

Each requirement has only one alternative. Comparing these, we note that both define substitutions for $TERM_2$, and that these are compatible if we identify $\#v$ with x. The new requirement is constructed by forming the union of the substitutions in both alternatives under this identification, giving

$$\{\{TERM_1 \mapsto x, TERM_2 \mapsto x\}\}.$$

Proof nodes which introduce term variables (All_Elim and Exists_Intro) must pass these down to the subproofs, to distinguish them from ordinary identifiers in formulae. These nodes must also act (or attempt to act) on the requirement synthesised from the subproof, by setting the introduced term variable to a term

chosen from some alternative in the requirement. This can affect the requirement passed to the parent node.

When the above requirement is passed up to the All_Elim node, this must then set $TERM_2$ to x, and pass the new requirement

$$\{\{TERM_1 \mapsto x\}\}$$

upwards, to be handled by the node which introduces $TERM_1$.

Automatic choice of terms has not been implemented in the IPE.

3.9.2 Determining Appropriate Premises in Re-Applied Proof Structures

As explained above, an attribute **selected** is used to record the user's choice of premise to which an elimination rule is to be applied, and this simply records the position of the premise in the premise-list. As a result of this mechanism, elimination rules in the Proof Grammar are sensitive to the position of a premise. Unfortunately, this can weaken the ability of a semantic tree to react to changes to the goal supplied to its root. Suppose for example that at some point in a proof structure, And_Elimination is applied to premise 5. The position is recorded in the **selected** attribute at that point in the proof. This means that when the same structure is applied to a different goal, the new premise which plays an analogous role to the original premise should appear in position 5, otherwise the subproofs generated may not be as intended, or the rule application might be inappropriate, and fail. Clearly this is an unfortunate restriction. The crucial question here is, how might the "analogous premise" be determined?

In this section we consider one way of improving the selection of premises in the IPE. (It should be noted that this method has not been implemented).

For the purposes of the following discussion, we will say that a goal G_1 *is analogous to* a goal G_2 *with respect to* a proof structure P if when G_1 is supplied to P there is some permutation of the premises at each point in P such that the resultant goals at the leaves of P are analogous to the original leaf goals with respect to any proof structure. A special case of this occurs when the proof

structure proves the original goal (ie there are no unproven leaves): then the analogous goals are those which can be proven by the same proof structure but perhaps requiring different selections of premises for elimination rule steps. The ideal would be to make the applicability of an IPE proof structure to different goals independent of the order of the premises in these goals.

A slightly better approximation to an "analogous" premise than "occurs in the same position" would be to search through the premises for one which satisfies the applicability condition of the elimination rule (for example, And_Elim would seek out a premise of the form A&B) This would not be hard to do, but it is not a great improvement, as there remains the problem of what to do when more than one premise is appropriate.

Improving on this, during initial construction (or perhaps once initial construction is complete) we could attempt to note the ways in which premises selected by the user are used in later stages of the proof. For example, if And_Elim upon some premise produces two new premises in the subgoal, to the first of which Implies_Elim is applied at some point further down in the proof, then when the structure is reapplied, the And_Elim expansion should seek a premise of the form $(A \rightarrow B)\&C$. Thus for each application of an elimination production rule we build up an expression template which describes the shape of the desired premise at any point in the structure.

In branches of the structure which lead to completed subproofs under the original goal, it should be possible to determine relationships between the formulae in a goal. If in the example above our original proof proved that "$b|d$" is a valid conclusion from "$(a \rightarrow b)\&c$" (plus other premises), then we could tag the And_Elim expansion with the *goal-pattern*

$$(A \rightarrow B)\&C, \ldots \vdash B|D$$

which shows that a subformula of the desired premise should match a subformula of the conclusion. (Note that it would also be necessary to indicate which premise in the pattern is the one of interest).

Supporting such a method within the Proof Grammar would require

- another attribute system for generating goal-patterns as synthesised attributes; this could be generated during proof construction;

- the ability to determine how a formula in a goal was derived from the goal of the parent Proof node – either by direct copying or as the result of the parental proof step's effect on some formula;

- that the premise selection method in the grammar could be switched from its present interactive form to the pattern-matching form.

In building a proof, a user might want to make pattern-matching the default action for each proof step after the initial selection, whilst still constructing the subproof. Unfortunately, this would lead to a cycle in the attribute system: the premise selection depends upon the goal-pattern, which depends upon the leaf goals, which in turn depend upon the premise selection. When the goal changes, we want to consider the pattern as constant; but when the leaf goals change, we want the pattern to change.

Perhaps, then, we must admit that a proof structure that is being constructed interactively must differ in some respects from one that is applied *entier* to a new goal. In the first case, premise-positions supplied by the user are paramount; in the second, a "clever" automatic choice is preferred. This would mean that there would have to be some process of conversion between the two. For example, we could have two separate attribute systems for handling premise-choosing, which are mutually exclusive: the present system supporting choice by the user, and another system implementing one of the above methods. When a proof structure is re-applied (either by editing the supplied formula, or by re-grafting it onto another proof), the user-chosen premises could be used as a guide by the second attribute system in determining what the analogous premises are.

Chapter 4

The User Interface

In this section we describe those layers of the code for the IPE which lie between the dependency graph manipulator generated from the Proof Grammar and the user. These layers account for approximately half of the IPE's code.

The basic kernel provides three main operations:

- expand a node of the derivation tree in accordance with some rule in the Proof Grammar;
- graft a derivation tree (and its dependency graph) onto the current node;
- incrementally evaluate a specific attribute instance in the dependency graph.

The value of any attribute instance can be accessed (or even altered, as described in the previous section).

4.1 Display Formats for Structured Objects

In this section we show how a semantic tree generated under the Proof Grammar can be presented on a display in a fashion which permits interactive access to its components.

A general technique of building shell hierarchies for describing displays of tree-structured objects was developed in ML by John Cartmell, and was specialised by him (and later by the author) to provide a description of the displays of IPE proofs.

The ML type **shell** is defined as a **string list list**. Each string represents a line of displayed text, and the spaces between the string lists represent "holes" into which other shells may be slotted. Such subshells inherit the indentation of the parent shell: for example if the shell

```
shell( [ [ "this is shell 2";
           "it has two lines and no holes"]] )
```

is slotted into the hole in the shell

```
shell( [ [ "this is shell 1";
           "   {"];
         [ "}";
           "shell 1 has a hole which is indented";
           "and enclosed in braces"] ] )
```

then when displayed, this would look like

```
this is shell 1
   {this is shell 2
    it has two lines and no holes}
shell 1 has a hole which is indented
and enclosed in braces
```

Note that the contents of the hole fit on at the end of the last line of text of the preceding string list, and that the first line of the next list is attached to the end of the last line of the hole. This permits list-like shells such as

shell([["one: "];[" two: "];[" three: "];[""]])

which can be used to produce a line like

one: 1 two: 2 three: 3

when filled in appropriately.

A shell can be used to describe the display format for a single node of a tree structure, with holes for the displays of the subtrees.

The result of filling in the holes of a shell is an object called a **box**, which is a list of lines of text. The **dimensions** of a box are the lengths of its first and last lines, its width (ie the length of its longest line) and the number of lines. This information allows us to determine (for example) when a mouse cursor is pointing within a particular box on the display.

A **shell_hierarchy** is a tree of shells, where each node with N children ($N \geq 0$) has a shell with N holes (ie N+1 string lists). The intention is that the tree structure of the shell hierarchy should correspond to the tree structure of the object being displayed. Each node also contains the dimensions of the box that would result from filling in the shell with the boxes produced by recursively filling in the subshells.

A **path** is a list of integers describing a path down a shell hierarchy from the root. This is used to indicate a particular node in the shell hierarchy.

The function **find_shell_with_given_coordinates** takes a shell hierarchy and a point, and returns the path through the shell hierarchy to the lowest shell whose box contains the point. Hence, if we construct a shell hierarchy from an object such that the tree structures are isomorphic, then given any point on a display showing the shell hierarchy, we can derive a path which can be used to determine the corresponding substructure of the object.

In the case of the IPE, the structure of the shell hierarchy is *not* isomorphic to the derivation tree. This is because the display structure has to go into more detail in order to permit the proof-by-pointing interface. Not only must the derivation tree structure be selectable by the user (to perform grafting operations upon it, and to supply formulae and terms to the relevant nodes), but the structure of the **sequent** of each Proof node must be displayed as well, to permit the selection of a premise or the conclusion by indicating a point on the display. Fortunately, this does not present any serious problems.

4.1.1 A Display Format for Sequents

The ML function **format_sequent** takes a sequent and an integer representing the maximum number of columns available for its display, and produces a shell hierarchy which describes the structure of the sequent and will display it within that number of columns, splitting the sequent across lines if necessary. The hierarchy consists of a premise-list hierarchy and a conclusion leaf hierarchy, where the premise-list hierarchy has one child for each premise. Thus for example, given sufficient width, a sequent might be formatted as

[[[A], [B&C]] entails [(A&B)→ C]]

where the nested boxes indicate the shell hierarchy structure.

4.1.2 Display Formats for Proof Nodes

Each production rule in the Proof Grammar has a corresponding display format generating function, written in ML. The display generated depends upon both the available display width and upon the values of attributes of the node such as **sequent**,**proven** and **appropriate**.

In preference to describing each formatting function in detail, we will consider the case of a single production rule, and describe notable points in other rules.

The formatting function for an instance of the And_Intro production is typical. There are four possible "top-level" shells. (We use boxes to delineate subshells in the formats):

> show [sequent]
> use And Introduction
> and [subproof 1's format]
> and [subproof 2's format]

is used when the left-side Proof's **proven** attribute is false. If **proven** is true, the generated format is:

> [sequent]
> by And Introduction
> and [subproof 1's format]
> and [subproof 2's format].

Recall that the displays of the subproofs inherit the indentation of their starting point.

However, as we wish to view only that section of the proof around the current node, we also require formats which elide the display of the subproofs when they occur more than some fixed number of tree levels below the current node; these are:

> [sequent]
> qv

and

> show [sequent]
> qv

In quantifier rules, the format of the term or identifier is included in the line of the rule name, for example

show sequent
use All Elimination <term format>
and subproof format

The formatting of Or_Intro is interesting in that should either subproof be proven, then the other subproof will be omitted from the display. If this were to be done by constructing different top-level shells, then this could lead to problems when paths through the Or_Intro node are interpreted. What actually happens is that the top-level shell has holes for the formats of both subproofs, but when one subproof is proven the other is filled in with a "blank" box.

Shell hierarchies can be utilised in attribute grammars: each symbol could have a **format** attribute, which is appropriately defined in each production rule as a shell hierarchy describing the display format of that node in terms of the shell hierarchies of its children, as well as local information. Incremental reevaluation would ensure that the format of a node was kept up to date. In fact, such a system was used initially in the IPE. However, it was decided that the cost of keeping a shell hierarchy at each node in the semantic tree was not justified, as only a small portion of the tree was ever displayed at any time, and the display attribute system was replaced by a "tree-walking" algorithm which calculates the display anew each time.

4.2 The Level 1 Proof Machine

The first layer of code packages the general kernel facilities into functions for editing and interrogating Proof Grammar semantic trees as proof structures. A type proof_machine is defined, consisting of

- a root node – a pointer to the root of a semantic tree;
- a current node – a pointer to some node in the semantic tree. This is the position of the semantic tree to which most editing operations are applied;

- a print node – this is the display root referred to previously;
- a global time counter, used by the reevaluation mechanism.

These components are all instances of ML "ref" objects; thus, a proof_machine is a state-object describing the state of a semantic tree. Operations upon proof_machines alter the state of its tree. The major operations provided by this layer are

create_machine : unit $\rightarrow$ proof_machine. This creates a semantic tree for the production Theorem, with no subproof and the initial conjecture set to the atomic formula "FORMULA". The current and print nodes are both set to root.

accept_formula : proof_machine * formula $\rightarrow$ unit. If the current position of the proof_machine is a Formula node, then its **self** attribute is set to the supplied value. If the current position is not a Formula node then an escape is generated. (The function **will_accept_formula** can be used to check appropriacy). Similar functions are provided to handle user-supplied terms and identifiers for the quantifier productions. Note that the strong typing of ML insists that a valid formula be given at this level.

introduce : proof_machine $\rightarrow$ unit
eliminate : proof_machine * int $\rightarrow$ unit. These implement the lowest level of the "proof-by-pointing" interface. **Introduce** expands the current node of the proof_machine by the Proof Grammar rule which is appropriate to the succedent of the (current) value of the **sequent** attribute of the current node. This replaces the entire subtree of the current node. For example, when applied to a proof_machine whose current node has a sequent attribute with conclusion $A \rightarrow B$, the production applied is Implies_Intro. Subsequent alterations to the **sequent** will not change the production rule used. **Eliminate** works similarly; the int parameter being the position of the selected antecedent. However, if the current node's old production rule would still apply to the newly-selected antecedent, then the subtree is not

replaced by a new production, but only the value of the **selected** attribute is changed. This simplifies recovery in situations where the order of the antecedents changes for some reason.

duplicate_antecedent : proof_machine * int → unit. This adds a copy of the selected antecedent to the antecedents of the subproof. This should be used when a subsequent application of **eliminate** will delete an antecedent which will be required later in the proof.

remove_antecedent : proof_machine * int → unit. The selected antecedent is removed from the sequent in the subproof. The main use of this is to remove antecedents which are not needed in the subproof and thus reduce display clutter.

basic navigation operations. These are operations which permit the current node to be moved through the semantic tree; the current node pointer can be moved to its parent or a child node, or it can be resited at the root.

bring_up_to_date. This is the main reevaluation operation, which calls the upon reevaluator to reevaluate some attribute instance of the print node which will ensure that every attribute used in the display generation will be up to date. For Proof and Theorem nodes this is the **proven** attribute: reevaluating this checks the **proven** instances of all descendants, and hence in turn the **sequents**. Other reevaluation operations reevalute particular attribute instances at the root,current or print node.

display : proof_machine * int * int → shell_hierarchy.

This returns the shell_hierarchy which constitutes the display form for the proof_machine, generated from the print node. The two integers give the display width and the maximum tree depth below the print node which is to be displayed.

Other operations are provided at this level to inspect the values of various attributes of the root, current and print nodes.

4.3 The Level 2 Proof Machine

The main purpose of the "middle" proof machine level is to extend the first level machine with a **path** component. This records the path from the root node of the machine to the current node. The navigation functions of the first level are extended to maintain the path; other operations of the first level are passed through this level unaltered. The most important new function provided at this level is **position_to**, which takes a proof_machine and a path, and performs a sequence of "single-step" movements to place the current node at the position indicated by the path. In practice, the path supplied to this function is generated when the user selects a point on the display of the proof tree.

4.4 The Level 3 Proof Machine

The level 2 machine maintains and interprets paths which describe positions in the derivation tree of a proof. The level 3 machine further interprets paths to derive information about further detail indicated by a path derived from the display. We shall refer to the *detail* of a path (with respect to a proof structure) as that part of the path which describes a position within the display structure of an individual node in the proof structure. (Recall from §4.1.2 that the shell hierarchy records the structure of a Proof's sequent, as given in §4.1.1).

In this level's implementation of **position_to**, the detail of the supplied path is recorded. This is subsequently used in the operation **apply_appropriate_proof_rule**. This uses the detail to determine which part of a proof node's sequent is indicated by the path, and then invokes **introduce**

or **eliminate** as appropriate. Similar operations are provided to duplicate or remove selected antecedents.

This level provides an interface which allows navigation and manipulation of the proof tree using only path descriptions derived from the display format of the proof. In the IPE as described in the Introduction, this is the outermost level of proof machine[1]. The final level of interface is the command interpreter.

4.5 The Command Interpreter

The command interpreter forms the outermost part of the IPE's interface. A main loop awaits input from the user (in the form of individual key presses, mouse button clicks or mouse menu selections) and performs appropriate operations upon a proof machine and a "display state".

The "display state" consists of several windows and state variables. The windows are:

the main window, in which the hierarchical display of the proof structure is maintained;

the title window. This window spans the width of the display screen. It shows the name of the current proof on display, and the name of the symbol of its root node.

the indicator window, which is used to display a brief description of which task the IPE is currently performing;

[1]The addition of multiple-buffer capability (as described in §5.1) requires an additional layer which supports multiple instances of proof machines and operations between them.

the error window, which is used to display error messages. This window is normally hidden behind the main window, and is popped to the top whenever an error occurs;

the help window. This window is toggled by a "Help On/Off" mouse menu item, and displays a brief synopsis of the commands available.

Further temporary windows are created for some operations, typically to receive further input from the user. In the basic IPE, the **accept_data** operation uses the **accept_formula**, **accept_term** and **accept_identifier** operations defined in the level 1 proof machine to set interactive attributes with user-provided attributes. **accept_data** calls a text-editing function, which creates a window on the display into which the user can type ordinary text. When the user signals completion, this text is parsed to yield an object of the appropriate form (formula, term or identifier). If the parsing fails then an error message is displayed in the error window. The user must then either (attempt to) correct the text, or abandon the attempt to change that attribute instance.

The text-editing function is a general function developed by John Cartmell. It has its own command interpreter, has little access to its surrounding environment and when called assumes complete control of the keyboard and mouse. Thus it is not possible to perform other IPE operations whilst editing text. This makes it impossible to use "cut and paste" operations to pick up arbitrary pieces of text from the display and copy them into the edit window. In order to permit this, we would have to re-implement window-based tools such as the text editor in a "client–server" framework, whereby each tool maintains a local state and communicates with the user via a central server. The server reacts to input from the user and decides which tool should receive the input. Output requests from tools would also be passed through the handler. A tool such as the text editor would no longer be an ML function but an object with a state and a handler which is invoked by the server to change the state of the object and possibly generate output requests.

This would require a complete redesign of much of the interface code used in the IPE, but would probably be a worthwhile task. Indeed, Paul Taylor at Edinburgh is using "SMLX" to develop tools which incorporate their own event handlers which receive events from a central server. The handler of any kind of object can be replaced by another which (for example) extends the set of events to which the object will respond; this can be used to write tools in an "object-oriented" fashion.

The state variables of the interpreter include a repetition count for the next command, and the settings of "display-above" and "display-below" which are used to limit the amount of proof structure displayed. Indeed, the proof machine itself can be regarded as one of the state variables.

Early versions of IPE ran on VT100-type terminals. Thus the original interface was designed for a character-based terminal without a mouse or separate windows. Though many changes were made when IPE was ported to Sun workstations, the display and much of the command interface still belies its ancestry. The display is still character-based rather than bitmap-based; this extends to the presentation of windows, whose borders are still constructed from characters. Though some effort was spent upon trying to allow the use of IPE on character-based and mouseless terminals, with the development of "choosers" (see later chapters) the use of IPE without a mouse became too awkward to make it worthwhile continuing support for VT100's. The dependence of the earlier version upon the keyboard and function keys is still obvious in later versions, although many of the functions attached to the function keys are also available via the mouse buttons[2].

The lack of a mouse on VT100 terminals enforced the use of arrow keys as a pointing device; though still functional, these have been superceded by the Sun mouse.

---

[2]This does not apply to the X windows IPE, where commands tied to keys in IPE Version 5 can now be invoked using the mouse.

Excepting points where formulae, terms and identifiers have to be provided, it is possible to construct proofs largely by pointing with the mouse, or by selecting items from the mouse menu. Further commands are invoked by single keystrokes.

4.5.1 Operation of IPE Commands

Here we describe the commands relevant to the basic IPE, and how the interpreter performs them.

The left mouse button acts as a selection pointer. When the button is clicked over a point on the screen, the corresponding path through the proof display is determined, and the proof machine repositioned accordingly. This enables selection of a point in the proof structure (or a formula in some goal) for future application of some other operation (egs printing that subproof into a file, or deleting the selected premise).

The middle mouse button also repositions the proof machine when clicked. However, it also performs some operation upon the resultant proof machine, depending upon the symbol of the selected node in the proof structure:

- If we are now positioned at a Proof node, then **apply_appropriate_proof_rule** is applied to the proof machine;

- If we are positioned at a "text-edit point" (ie a Formula, Term or Var node), then **accept_data** is invoked to change the value of the interactive attribute instance of that node;

- If the current node is none of the above, then the default action is to "zoom in" to the selected point: it is made the new centre of the display and the display is regenerated around it. (This does not happen when a selection is made using the left button).

When pressed (rather than clicked), the right mouse button presents a menu containing further commands. Those commands which apply to the IPE as described so far are:

Help On/Off This toggles the display of the help window which contains a brief description of the IPE's commands; this display itself can be toggled between displaying the functions of the mouse buttons and the keyed commands;

Zoom In (to current) This makes the currently-selected node of the proof machine the focus of the display; selecting with the left button then choosing this option is equivalent to clicking the middle mouse button on the same spot (unless the selected spot is appropriate for entering data or applying a proof step).

Zoom Out(n) This moves the display focus to the nth ancestor of the selected node, where n is a repeat-count. For example to refocus the display upon a node which is 5 tree levels above the current node, the user would type '5' at the keyboard and then select this option. This gives a simple method of moving upwards through a proof tree to levels which are no longer visible on the display;

Zoom to Root This moves the current position (and hence the display focus) to the root of the proof machine;

Scroll Up and Scroll Down These options can be used when the display text is longer than the available screen height. Scroll Down moves n lines down the display text, and Scroll Up similarly, where n is a repeat-count (defaulting to 10). This affects only the display; the proof machine's current position etc are not affected;

Weaken (remove premise) To use this, the user should first select a premise of a Proof node (using the left button). This replaces the expansion (and all subtrees) of the current node with a Remove_Antecedent production. The intention is to remove premises which will not be needed in the subproof. The name "weaken" is somewhat misleading, as it applies to the corresponding sequent-calculus rule which adds a premise to a sequent: thus the IPE operation performs the "backwards tactic". If the selected

object is not a premise, then the display flashes and an appropriate message appears in the error window;

Duplicate Premise This is used similarly to the Weaken option, this time creating a second copy of the selected premise in the subgoal;

Exit IPE This creates a "confirmer" window which asks if the user really intends to quit the proof session. If the user affirms this, then IPE terminates, and all work since the last save or print is lost.

(The remaining menu options will be discussed as further features of the IPE are introduced in later chapters). Many of the above commands can also be activated by function keys; this is a remnant of the IPE's mouseless beginnings.

The single-letter commands relevant to the basic IPE are:

H This switches the help display between a description of the mouse button operations and a description of the keyed commands;

d,W These perform the Duplicate and Weaken operations which are also available on the mouse menu;

< sets the value of the display variable "display-below" and regenerates the display accordingly. If preceded by an argument count, display-below is set to that value, otherwise the present value is incremented;

> sets the value of "display-above";

Control-R redraws the display, should it be affected by outside interference;

Control-P appends the text of the current proof display to the file IPE.proofrecord. This provides a crude means of printing proofs; a better method will be presented later.

As stated earlier, the arrow keys and function keys can also be used, but their functions are duplicated by the mouse functions. However, this does allow the use of the IPE on ordinary terminals.

This completes the description of the basic IPE, which first appeared on VT100's in early 1985. Subsequent chapters describe the ways in which the IPE was developed over the following $1\frac{1}{2}$ years.

Chapter 5

Facilities for the Manipulation of Proof Structures

In this chapter we cover a variety of operations which act upon the IPE's proof structures themselves, ranging from tree-grafting operations to the automatic generation of proof structures to satisfy (or at least reduce) goals.

5.1 Multiple Buffers

The representation of an IPE proof as a tree of basic rules attributed by goals in an attribute grammar framework gives the IPE's proof structures a high degree of goal-independence As we have already seen, changing the goal supplied to an IPE proof structure does not alter the structure itself, although its proven/unproven status may change, and some points of the proof may fail if their rule applications are inappropriate to their new goals.

Thus far, the only means of altering an IPE proof structure is by expansion of a node by the production rule appropriate to the decomposition of some formula in its (present) goal. (Recall from §3 that this is effectively a subtree replacement operation, where the new subtree is one of the 'basic templates' of the Proof Grammar). It is in fact perfectly feasible to perform subtree replacement with a compound proof structure. Previously, we have talked of supplying proof

structures with a new goal; now we can turn this around and talk of ***applying*** a proof structure to a goal.

An example of where we might take advantage of this re-usability of proof structures would be a symmetrical proof; having constructed one half of the proof, we could then apply this structure to the other subgoal, instead of expanding the proof step-by-step. Even if the proof were not fully symmetrical, this could still reduce repetitive work on behalf of the user. If the structure is not fully applicable to the goal, or if any of its leaves are unproven, the points of failure can be edited by the user as normal.[1]

The IPE takes advantage of the goal-independence of proof structures via its multiple-buffer facility. Each buffer in the IPE is a distinct proof structure, with its own root and current position. A buffer can be rooted on any symbol of the Proof Grammar, so that we can have Theorem, Formula and Term buffers as well as Proof buffers. When the IPE is initialised there is only one buffer, called **Main**, which is rooted on Theorem. The user can create new buffers, either as blank Theorem buffers or by copying the current subtree of the current buffer to a new buffer. Proof structures can be re-used by applying the current subtree of one buffer to the current position in another.

The buffer operations available in the IPE are:

change_to_buffer Given the name a buffer, makes it the current buffer. If it already exists then its current position is restored, otherwise it is created as a buffer rooted on Theorem and positioned at the root

copy_to_buffer Grafts a copy of the current subtree of the current buffer onto the current position of the named buffer, or, if the named buffer does

[1]Although the above discussion concentrates upon proof structures, attribute-independence is a property of semantic trees in general: there is no reason for not being able to graft Theorem, Formula, Term, (etc) structures onto others of the same kind; however the uses in such cases are limited.

not exist, creates it rooted on the current subtree and positioned at root. This is the means by which a useful proof structure can be saved for re-application.

apply_buffer Grafts the current subtree of the named buffer onto the current position of the current buffer.

(For a fuller description of these commands and their user interfaces, see Appendix 2).

Thus for example to perform the 'symmetrical-proof' re-application, the user would move to the root of the structure to be re-applied, copy it to another buffer, move to the other (symmetrical) subgoal and apply that same buffer.

Related to the buffer operations is the notion of *yanking*. Since it is possible that the user may wish to undo a subtree replacement (if for example they have replaced the wrong tree), the last (non-trivial) subtree deleted by any replacement operation is saved in a special **Yank** buffer. The **yank** command will graft this tree onto the current position (which need not be the same place from which it was saved). The tree deleted by yanking is not saved in the **Yank** buffer, so that the same tree can be yanked more than once.

Multiple buffers can also prove useful in the production of sub-lemmas (see Section 6.4): if some subgoal of the current problem would be best handled as a lemma, then work on the main problem can be left pending in that buffer whilst the lemma is worked on in a new buffer. It is possible to work upon multiple attempts to prove the same conjecture (or of the same subproof) by copying the proof or subproof to one or more buffers.

5.2 Automatic Proof Construction

We can think of the application of compound proof structures (as described in Section 5.1) as a form of automatic proof expansion, where the proof strategy is completely inflexible. In fact, IPE proof structures are akin to LCF tactics constructed without using the tacticals ORELSE or REPEAT; the major distinctions being that IPE structures are interactively editable, stand as their own validation, and have a notion of "positional partial success" (ie., failures are pinpointed visually). However, this is a rather limited notion of 'automated proof', since these structures must first be constructed by hand by the user in response to some particular problem before they can be applied to other problems. A better method of automatic proof would generate proof structures with a minimum of user intervention.

Although one of the initial aims of the IPE project was to concentrate upon interactive, user-directed proof, there is no escaping the fact that with the possible exception of novice users, many people find the 'trivial' details of proofs tedious to perform by hand. It was decided that the IPE should contain some form of proof automation, if only to elide the donkey-work. This eventually evolved into the notion of *IPE-tactic*.

In order to remove at least some of the aforementioned 'donkey-work', we chose to extend the IPE's command-set by an 'autoprove' mode switch, the idea being that when autoprove was turned on, the IPE would perform some rather simple autoproving techniques to the proof structure after each alteration by the user. It was considered important that the interactive aspects of the IPE should not be buried under a mountain of automated-proof technology. For this reason, two major restrictions were imposed upon the IPE's 'autoproof' mechanism:

Firstly, it should not make decisions which could later turn out to be wrong, in the sense that a proof is directed into a dead end when other successful directions exist. This would force the user to backtrack through the automatically-generated proof to search for the bad decision, or even worse, over-zealous faith

in computers might lead the user to believe that his problem is intrinsically insoluble. The autoprove mechanism should ideally make all of the *unimportant* decisions (or 'non-decisions') in the proof, but leave all of the *important* decisions to the user.

Secondly, autoprove should perform no hidden, 'magical' steps, but should express its performance in terms of ordinary IPE proof structures, so that the user can see the strategy chosen, and alter it if desired. In other words, autoprove should be able to justify itself to the user.

The first autoprove strategy used in the IPE was nick-named 'prove-by-boredom' on account of its sheer simplicity. It satisfied both requirements, but interpreted the phrase 'non-decisions' above extremely literally. A proof node was expanded only when there was only one possible expansion, ie when only one formula in the goal was compound. (As an example of its limitations, it would not proceed with **show A,B&C entails A&C** because the goal has two &-formulae). Not surprisingly, this proved to be excessively limited in its application. Something more powerful was required.

5.2.1 IPE-Tactics

To allow some freedom in the choice of our new autoprove algorithm, first we implemented a means of describing proof strategies, which we named IPE-tactics. IPE-tactics are defined in ML, and have to be incorporated into the ML code of the IPE. There is no facility which allows users to define IPE-tactics within an IPE session.

An *IPE-tactic* is a function which takes a goal as its argument and returns both a list of (unproven) subgoals and a description of an IPE proof structure which when applied to the initial goal will have the returned subgoals at its leaves.

The proof structure description returned by an IPE-tactic is a tree whose nodes consist of the names of basic tactics and possibly a "selected premise" number. A special node `nilTree` is used to indicate an unexpanded proof node.

By forcing an IPE-tactic to return a (description of a) proof structure we ensure that the 'visible justification' criterion can be satisfied. When a (top-level) call of an IPE-tactic returns, the IPE builds the proof structure corresponding to the returned description and grafts it onto the current position in the main structure.

The proof description also plays a similar part to the validation functions returned by tactics in LCF. An IPE-tactic can use any form of heuristics to generate its subgoals, but it must be capable of providing an IPE proof structure which achieves the same effect. In other words, the proof structure returned need not reflect the actual process involved in determining the subgoals.

In IPE at present, there is only one IPE-tactic available to the user: the improved version of "autoprove". When "autoprove" mode is selected, the autoprove tactic is applied to the goal of each unproven leaf or failure point in the proof structure. The resultant proof structure description is used to build a proof structure which is then grafted onto the appropriate node.

Thus IPE-tactics represent a 'middle ground' between the tactics of LCF (which operate solely upon goals) and the transformation tactics of NuPRL which operate upon proof trees.

Clearly, the application of an IPE-tactic to a goal can have one of three possible outcomes:

1. The IPE-tactic can **succeed**, returning no subgoals and a (descriptor for) a proof structure which completely proves the goal;

2. It can **fail**, by making no advance upon the goal, in which case the same goal is returned together with a null descriptor; or

3. It can **partially succeed**, returning a non-empty set of (hopefully simpler) subgoals and a non-null structure descriptor.

The implementation of *IPE-tacticals*, analogous to LCF's tacticals, is fairly straightforward. The subgoals from one tactic are supplied as arguments to other

tactics as in LCF, whilst the composition of validation functions is replaced by tree-appending the structure descriptors returned.

(Given a proof structure descriptor P and a list of proof structure descriptors L, then *tree-appending L to P* involves scanning P in a depth-first manner and replacing occurrences of `nilTree` in P with successive structure descriptors in L.)

When "THEN(tac1,tac2)" is applied to some goal, tac1 is applied to the goal, giving a list of subgoals SG and a structure descriptor SD; then tac2 is applied to each goal in SG. Each application of tac2 produces a new subgoal list SG' and descriptor SD'; THEN tree-appends SD' to the appropriate point of SD (i.e., it expands the appropriate leaf of SD by SD'), and returns the entire (new) descriptor and all of the subgoal sets SG'.

For example, if we imagine that the tactics AndIntroTac and ImpIntroTac simply (attempt to) perform the IPE rules And_Intro and Implies_Intro upon their goals, then the tactic application:

THEN(AndIntroTac, ImpIntroTac)(**show (A→ B)&(C|D)**)

will produce the subgoals:

show A entails B
show C|D

and produce a descriptor for the IPE proof structure:

show (A→ B)&(C|D)
use And Introduction
and show A→ B
  use Implies Introduction
  and show A entails **B**
and show C|D

Note that ImpIntroTac failed when applied to the subgoal **show C|D**; however the whole application partially succeeded, for it did manage to produce new subgoals.

"THENL(tac,tac-list)(goal)" is similar to THEN, except that each IPE-tactic in tac-list is applied to the corresponding goal in the subgoals produced by (tac goal). It is an error for the number of subgoals and the length of tac-list to differ, so it is safest only to use THENL when tac is guaranteed always to produce the same number of subgoals.

As an example,

THENL(AndIntroTac, [OrIntroTac; NotIntroTac])
(**show (A|B)&(~C)**)

produces the subgoals

show A
show B
show C entails contradiction

and the IPE proof structure

```
show (A|B)&(~C)
use And Introduction
and show A|B
     use Or Introduction
     and show A
      or  show B
and show C entails contradiction
```

In "REPEAT(tac)(goal)", tac is repeatedly applied to the results of its earlier application, until an application either succeeds or fails. Thus the safest use of REPEAT is with tactics which are guaranteed to 'bottom-out' at some point in the repetition process (for example, any tactic which performs Duplication

would be suspect, for by increasing the number of premises upon which successive application can work, infinite chains of applications may be possible).

Example: if AndElimTac performs a single And_Elimination upon the first suitable premise it finds in its goal, then

REPEAT(AndElimTac)(**show A&B&C entails A&C**)

will produce the subgoal

show **A,B,C entails A&C**

and the IPE proof structure

show A&B&C entails A&C
use And Elimination on premise 1
and show A&B,C entails A&C
    use And Elimination on premise 1
    and show A,B,C entails A&C

The idea of ORELSE(tac1,tac2) (goal) is that it should return the 'best' proof structure and set of subgoals for the goal. Choosing the best proof is easy if either tac1 or tac2 succeeds or fails, but the notion of partial success complicates the issue. If both tac1 and tac2 partially succeed, then some decision must be made as to which is better. In the present implementation, the choice is crude: the results produced by tac1 are returned. This is dangerous, since it is possible for tac1's subgoals to be unprovable whilst those of tac2 *are* provable; which is to say that we are at risk of breaking our first criterion, that the automatic prover should not leave the user with a dead-end proof so long as a valid proof is possible. However, with judicious construction, tactics can still be made to satisfy this constraint, the solution being to restrict application of ORELSE to pairs of tactics (tac1,tac2) where given any goal only one of tac1 or tac2 can partially succeed. (In fact, we do this by ensuring that tac1 and tac2 are mutually exclusive, so that in any situation at least one will fail).

Examples:

ORELSE(AndIntroTac,ImpIntroTac)(**show A→ B**)

will perform as for ImpIntroTac,

ORELSE(AndIntroTac,AndElimTac)(**show A&B entails A&D**)

will perform as for AndIntroTac.

At present, IPE-tactics are solely used to implement the fixed strategy 'autoprove'; no further IPE-tactics are available to the user. Further applications of IPE-tactics and extensions of their idea are given in the following subsection.

We are now ready to present the version of autoprove currently available in the IPE. We assume that certain sequences of proof steps commute, in that the order in which they are performed is irrelevant to the final result of the proof, and that certain proof steps can always be performed without prejudicing the outcome of a proof. A set of guide rules were drawn up, along the lines of , "it is always safe to perform And_Introduction", or , "performing Implies_Elimination too soon can lead to a dead end". Those proof steps which are always safe were encoded as tactics "AndIntroTac", "AndORExistsElimTac"[2] , etc, and used as the basic building blocks of the first stage of autoprove (together with ImmedTac, which simply checks for immediacy in the same fashion as the IPE rule):

```
val AutoIntroTac = ORELSE( ImmedTac,
                   ORELSE( AndIntroTac,
                   ORELSE( ImpIntroTac,
                   ORELSE( NotIntroTac,
                           AllIntroTac))));
```

(The use of ORELSE here does not run the risk of breaking the "no dead ends" restriction, as the –IntroTacs are all mutually exclusive (only one can

[2]"AndORExistsElimTac" looks through the premises of a goal until a conjunction or existential premise is found; it then performs the appropriate elimination rule. The tactic fails if no such premises are found.

succeed or partially succeed), whilst ImmedTac can never partially succeed). AutoElimTac is similar although shorter:

```
val AutoElimTac = ORELSE( ImmedTac,
                          AndORExistsElimTac);
```

The intention for AutoproveTac is that it should alternate between attempting introductions and eliminations, and only halt when both AutoIntroTac and AutoElimTac fail (or when some combination succeeds). We define a new tactical AlternateTac as

```
val AlternateTac( t1,t2 )
    = REPEAT( ORELSE( THEN( t1,t2 ), t2 ) );
```

(This will repeatedly attempt to perform t1 and t1 alternately, even if one or the other should fail). We can now define AutoproveTac as

```
val AutoproveTac
    = AlternateTac( AutoIntroTac, AutoElimTac );
```

Now we build in the old autoprove, encoded as a tactic BoredomTac (ie., the latter will generate a new subgoal when the goal contains precisely one formula that can be further decomposed):

```
val FullAutoproveTac
    = AlternateTac( BoredomTac, AutoproveTac );
```

The resultant version of autoprove is considerably more powerful than the previous version. In practice it is still somewhat restricted in its applicability, often stopping at points where the user feels that the next step is too obvious: for example, it will not alter **show !xP(x) entails ?xP(x)**; this is because to do so might result in a plethora of machine-chosen instantiation terms, all with names of the form TERM_n, which could be unsightly and confusing (furthermore, another decision taken about autoprove was that it should not travel too far past an All_Introduction or Exists_Elimination node, so that the user would not have to backtrack far to change the instantiation).

As a final footnote to autoprove, it must be pointed out that in fact we have not completely satisfied the first criterion (of avoiding leading the user into dead ends), due to the need for the duplication (or non-removal) of premises in certain proofs (see §3.8.3). In such cases, it is possible that autoprove will lead to a dead end, and the user must backtrack to the point where a duplication was required and perform it by hand.

5.2.2 Uses of IPE-Tactics and Extensions to Them

Let us return to the problem of "difficult" decisions in the ORELSE IPE-tactical. One possible solution would be to alter the definition of IPE-tactical (and by extrapolation, that of IPE-tactic) to return lists of *alternative* solutions. Faced with two partially-successful strategies, ORELSE could simply return both partial solutions, leaving the decision to some higher authority. Each IPE-tactical could then perform its usual operations on each alternative: if any succeeds, then it alone can be returned, otherwise all alternative partial solutions could be returned.

Whilst being thorough, this technique could lead to an exponential explosion of alternative partial solutions, unless some "smart" heuristics were incorporated within the IPE-tacticals Furthermore, the top-level call upon an IPE-tactic may return a set of alternatives, which must then be chosen between by some other process, eg user intervention. It could prove too confusing to ask the user to choose from a bewildering array of alternative subgoals and proof structures.

However there is a method that, while not avoiding the combinatorial explosion, will permit the user to keep his option open as regards the choices between alternative proofs. The method is simply to add an "alternative-proof" construct to the basic productions of the Proof Grammar:

Proof ::= ALT (Proof Proof)
[Proof$2.sequent = Proof$1.sequent;
Proof$3.sequent = Proof$1.sequent;
Proof$1.proven = Proof$2.proven or Proof$3.proven;
⋮
(etc)

Each subproof inherits the same goal as the parent, and the parent is considered proven when either subproof is proven. This allows two different proof structures to be applied to the same goal at the same position, so that the user can work on either subproof (or both) until a choice can be made between them. ALT could be formatted in a similar fashion to Or_Intro, so that when one subproof is proven, the other is hidden from view. The definitions of IPE-tactic and IPE-tacticals would be the same as at present, except that upon finding that both substrategies are partially successful, ORELSE would wrap up the alternatives in an ALT node, leaving the final decision for the user to make at leisure.

As mentioned above, IPE-tactics at present are very under-utilised; their primary use thus far has been to experiment with different 'flavours' of essentially the same autoprove IPE-tactic, before settling upon a final choice.

The first obvious step would be to build up an internal library of useful IPE-tactics which could be made available to the user (who may either set one up as the default 'autoprove', or call one specifically to solve a single goal). This would not be too hard to do, as the work for autoprove has involved producing basic IPE-tactics corresponding to many of the IPE's proof steps. We could relax the first restriction for some of these tactics and allow greater proving power at the cost of "dead-end risk". Useful tactics to add might include 'semi-smart' tactics capable for example of choosing suitable terms for All_Elimination/Exists_Introduction (although this may mean some reworking of the definition of IPE-tactics, allowing them to return substitution sets as well).

Next, we might give users the ability to define their own IPE-tactics, using a definition language consisting of certain basic IPE-tactics and the IPE-tacticals (thus preventing the user from writing any "dirty" IPE-tactics which cheat). This again would not be too hard to implement.

If we were prepared to forego our second criterion (that the tactic should justify itself visibly to the user), and implement an "apply-tactic" rule in the Proof Grammar, we could then use LCF-style tactics to solve (or partially solve) goals without a visible justification. The advantages here would be that little storage would be required to represent each tactic application (a single Proof node instead of many), and that these tactics need not be restricted to the rules of the IPE in their operation (dangerous as this suggestion may sound, there are situations where we may want this; for example, when we chose to work within a particular theory (see §6), we could load up a library of theory-specific tactics which perform operations which are not available as basic IPE proof steps). Another advantage would be that the tactic would be re-invoked should the goal change. (With IPE-tactics at present, the fact that a tactic was invoked at a certain point is not recorded in the proof tree. The proof tree might fail whereas the tactic may have chosen a new, more successful, proof tree if it were re-invoked). The disadvantages would be that the tactic would operate "as if by magic", with little to show how its subgoals (if any) were arrived at, and that the inner workings would not be editable by the user (the tactic application would be acting as a hard-wired function rather than as an editable structure).

It is debatable as to whether or not tactics should be lucid, and the answer probably depends upon the application: for teaching purposes, it would seem better that all of the available built-in tactics should have a simple and easily-visible relationship between their goals and subgoals, whilst in a specific practical application the power of tactics may be more important than clarity of operation.

5.3 Storing Proof Structures

Though it might be claimed that the major aim of a proof editor is to create new lemmas and extend the knowledge of some theory (see Section 5), the proofs themselves are important objects. This is especially true in the IPE, where a single proof structure may be applied to different goals. It is desirable to have the ability to save proofs in a re-usable form between proving sessions, thus allowing the construction of a library of "proof fragments" as well as permitting users to save partial proofs and thus complete them over several sessions.

The IPE permits the saving of proof structures on file in a concise, goal-independent format. Proof structures are written to files using a very simple description language. Each production in the proof grammar is assigned a one- or two-letter identifier. A proof node is written as its code followed by any special attribute settings, such as the value of the selected-premise indicator. Proof nodes involving substitutions (i.e., those with Term or Var sons) have the unparsed text of the substitution appended; the initial conjecture below a Theorem is similarly handled. The entire description is preceded by a single letter denoting the root symbol of the tree. Thus for example, the proof structure

```
A&B&C entails A&C
by And Elimination on premise 1
and A&B, C entails A&C
    by And Introduction on conclusion
    and A&B, C entails A
        by And Elimination on premise 1
        and A,B,C entails A
            is immediate
    and A&B, C entails C
        is immediate
```

is saved as

P
&E1
&I
&E1

This allows large proof structures to be saved using very little storage. The price is that it is now a costly process to re-build a tree from its description.

In practice, the main use of the proof-saving facility has been to save incomplete proofs between sessions, or to record solutions for tutorial purposes, rather than to build up a library of reusable structures. This is because of the fact that one major disadvantage of the IPE's proof structures is that they are very large objects; it is not unknown to use several megabytes of run-time store upon a moderately-sized proof.This makes it inadvisable to build proofs solely by applying the proofs of earlier-proven goals *in situ*; then even a simple proof would rapidly explode in size. As new facts are proven, ideally we would like to be able to use these facts without regard to their proof. In some sense, we would like to be able to regard lemmas and theorems as extensions to our basic set of proof rules. The IPE's solution to this is given in §6. However, the ability to save reusable proofs on-file is still useful in those cases where we might want to take an existing proof which partially proves a goal, and then edit it to complete the proof.

5.4 Printing Proofs

The on-screen display of proofs in the IPE is geared towards navigability and editability. To facilitate the "proof-by-pointing" style, proofs are goal-directed, and the current premises are all displayed at each point in the proof display. When we want a more permanent record of a proof however, the considerations are different. We no longer need redundant repetition of premises, nor need we

print the proof in a top-down fashion. The IPE's printed proofs are radically different from those displayed on-screen. (It is worth noting at this point that the "proofs" displayed by the IPE are not really proofs but displays of "goal achievement state"; the real proofs lie in the composition of the justifications for the tactics which comprise the IPE's basic rules).

Proofs are printed in a bottom-up fashion, with attempts to reduce the redundancy of premises. Several screenfuls of IPE display can collapse to a single page of proof in this style. For example, consider our proof of A&B&C→A&C; this prints as

Theorem: A&B&C→A&C

Proof:

1 assume A&B&C	
1.1 A&B	&E(1)
1.2 C	&E(1)
1.3 A	&E(1.1)
1.4 B	&E(1.1)
1.5 A&C	&I(1.3,1.2)
2 A&B&C→A&C	→I(1,1.5)

QED

Incomplete proofs can also be printed, although they are rarely useful as a guide in completing the corresponding IPE proof. Printed proofs are generated from IPE proof structures in a similar manner to the generation of saved proof structures. Each rule in the proof grammar has an associated printout style. Rules which extend the set of premises print the new premise (unless it is already visible as the result of some previously printed step) and then print the subproofs, whilst those Introduction rules which decompose the conclusion print the subproofs first and then use the original inference rule to generate the conclusion. The printing function makes a single top-down pass of the proof structure; a multi-pass algorithm could additionally flag which premises actually contribute to the proof and omit those that don't from the final printout.

Chapter 6

A Theory Database

6.1 Introduction

An important facility in any proof maintenance system is the ability to use the results of one proof in another. We have already seen (in §5.1) that the IPE's multiple buffers can be used to apply one proof structure to another, providing one possible method. However, such a method becomes infeasible as we build up a "hierarchy" of proofs. With each proof being expanded in full, the physical size of proofs becomes very large indeed (it is not unknown for IPE sessions to require several megabytes of run-time store), resulting in a very slow response time or even system overflow errors. What is really needed is the ability to encode the result of a proof in a "shorthand" form which requires little space; for example, a form which allows one proof to refer to the result of another in a single step. This should produce a similar result to that obtained by simply applying the original proof, but consume much less space. The cost of this function would be the loss of the ability to edit the proof should the original turn out to be not quite what was wanted. The ability to use the result of one proof as a **lemma** in another in the above fashion corresponds to normal practice in the construction of proofs, in that it breaks a large problem into smaller and easier-to-manage parts, and also in that previous results are not normally considered mutable (excepting perhaps "by analogy with ..." or using the vague transformation "similarly,..." in which

cases the IPE's proof editing facilities can be used). Naturally, we would like to be able to keep these lemmas on-file between IPE sessions, if possible in a structured fashion which represents their dependencies.

In the following, we work towards a description of the result of an IPE proof that encapsulates some of the proof structure's generality.

Defn. A *formula schema* is a formula F paired with a set of terms and a set of formulae which occur in F. These subterms and subformulae are designated as *generic*. A formula schema thus represents the set of all formulae obtained by replacing the generic formulae and terms in the original by some other formulae and terms. A formula thus constructed is known as an *instance* of the formula schema, and the subformulae or terms used to replace the generics are known as the *generic substitutes*.

For example, the formula schema

(P&Q→R(x,y), {x},{P,Q})

represents the set of all formulae of the form P&Q→R(x,y), where P and Q are any formulae, and x is any term. (Note that the predicate R and the term y are fixed). The formula (A|B)&C(z)→R(f(y),y) is an instance of this, in which P has been replaced by A|B, Q by C(z) and x by f(y).

The generic formulae and terms may be higher-order, in that they can contain *parameter* terms. For example, we might define a formula schema for induction over the natural numbers:

(phi(0) & !x (phi(x)→phi(S(x))) → !x phi(x), {},{phi(x)})

(where 0 and S(x) are the constant zero and successor functions respectively). The meaning of the formula part is, "if we can show that phi holds for 0, and that if for any x phi holding for x implies that it also holds for S(x), then we have that phi holds for any x". By making phi(x) generic, we are then able to construct induction formulae for any phi in x. For example, choosing x+x=S(S(0)).x for phi(x), we obtain the instance

$$0+0=S(S(0)).0 \ \& \ !x(x+x=S(S(0)).x \rightarrow S(x)+S(x)=S(S(0)).S(x)) \\ \rightarrow !x \ x+x=S(S(0)).x$$

Each instance of the generic formula phi(x) has been replaced by the generic substitute, and each instance of the generic parameter x as it appears in the substitute has been replaced by the parameter of phi as it appears in the original formula at that point.

The motivation behind the above definition is to arrive at a means of describing the result of an IPE proof as a descriptor of those formulae which the proof structure would prove without alteration. For example, a proof structure which proves A&B&C→(A&C) will also prove P&Q&R→(P&R), so we describe the set of formulae this structure can prove by

(A&B&C→(A&C), {},{A,B,C}).

A lemma in the IPE system is a formula schema constructed from the root formula of an IPE proof. The generic formulae and terms are automatically determined from the root formula as those predicates and terms which are wholly uninterpreted in the proof, which is to say that they have no special properties which were used in the proof. (For example, if a proof of x+y=y+x uses any properties of + not shared by every term expression (as seems likely), then the function + should not be genericised; on the other hand, the terms x and y will be genericised, to yield a commutativity lemma on + for any x and y).

As described so far, all predicates and terms in an IPE proof are uninterpreted, so that when we write x+y, the + symbol has no special meaning but is just another (infix) function. In order to extend the IPE from a logic of uninterpreted predicates and terms to one where we can reason about special symbols and prove properties thereof, we need some means of defining these symbols and giving them meanings which distinguish them from other symbols.

6.2 IPE Theories

In the previous section it was stated that all symbols in the IPE are uninterpreted, so that symbols such as + and S(_) have no special significance. However, consider the formula schema

$$(x+y=y+x,\ \{x,y\},\ \{\})$$

Here, x and y are generic, but + and = are not. This means that although we may substitute any term for x and y, + and = cannot be substituted. Thus, this formula schema gives a property of addition (and equality) that cannot be extended to any other symbols. When an instance of this formula schema is used, the act of its use distinguishes the symbols + and = from any other symbols.

This use of formula schemata forms the basis of the IPE's mechanism for extending the "realm of discourse": IPE-theories.

Defn. An *IPE-theory* consists of

- A set P of predicate symbols, each with an arity (≥ 0) denoting the number of its arguments
- A set F of function symbols, each with an arity (≥ 0) as for P
- A set C of constant symbols
- A set GF of name-labelled formula schemata, partitioned into
  - A set A of *axioms*
  - A set L of *lemmas*

The intention is that P,F and C *declare* symbols which are special within the IPE-theory, the set A *defines* the special properties of these symbols, and the set L contains IPE-generated lemmas of new properties proven using axioms from

A. (We shall see in the next section how axioms and lemmas can be used in IPE proofs). The set GF is known as the *facts* of the theory.

Before giving examples of IPE-theories, we introduce some "pretty-printing" notation to make the examples more readable.

Notation. An axiom

$$(\text{Name},(F,\{gt_1,\ldots,gt_m\},\{gf_1,\ldots,gf_n\}))$$

(where "Name" is the label of the axiom) is written as

```
axiom Name is
   F
generic terms gt_1
          and ...
          and gt_m
generic formulae gf_1
             and ...
             and gf_n
```

We format lemmas similarly by replacing the word "axiom" by "lemma".

Notation We shall also intermingle the sets P,F and C of an IPE-theory as a linear list of items of the forms

predicate Q(x,y,z)	for a 3-place predicate symbol Q
function f(x)	for a 1-place function symbol f
constant c	for a constant c

For example, to construct a theory of natural numbers, presupposing rules for equality, the sets P,F and C would be set up by the declarations

constant 0	for zero
function S(x)	the successor function
function x+y	infix addition
function x.y	infix multiplication

and we would add axioms such as

```
axiom Plus0 is
  x+0 = x
generic terms x

axiom PlusS is
  x+S(y) = S(x+y)
generic terms x
            and y
```

We would also want to add the induction formula schemá of the previous section as an axiom. Let us call this IPE-theory "Peano".

It must be pointed out that since the logic of the IPE is untyped, we have not defined the naturals as a type, but merely provided new axioms from which we can derive properties about expressions of particular forms. If we later define a theory of lists, including a new constant "nil" (intended to represent the empty list), then when we combine this with the Peano theory above, there is nothing to prevent us from proving (for example) that x+S(nil)=S(x+nil). Strictly speaking, we should have added another predicate, "IsNat", and the axioms

```
axiom IsNatZero is
   IsNat(0)

axiom IsNatS is
   IsNat(x) →IsNat(S(x))

axiom IsNatPlus is
   IsNat(x) &IsNat(y) →IsNat(x+y)
generic terms x
            and y

axiom PlusS is
```

```
    IsNat(x) &IsNat(y) →x+S(y)=S(x+y)
generic terms x
        and y
```

and so forth. Though this would increase the security of results, it was felt that it would make proofs extremely tedious; thus it was decided to omit such type-guarding, and rely upon the proof-builder to perform type-checking. (Of course, it is still possible to redefine the present set of IPE theories using guards).

The purpose in declaring symbols is to inform the IPE of symbols which cannot be made generic in lemmas constructed in an IPE-theory. However, at present it is the theory-designer's responsibility to ensure that axioms are correctly stated; in particular, care must be taken to ensure that a declared symbol is not stated as a generic in an axiom. (For example, the effect of making "0" generic in Plus0 would enable the collapsing of every expression of the form "x+y" to "y") As we shall see later, it is not possible to make a declared symbol into a generic in an IPE-generated lemma.

Pragmatically, IPE-theories are stored on file as UNIX directories; the symbol declarations reside in a file called ".environment", and each fact (axioms and lemmas alike) occupies a file of the same name as the label of the fact. Some organisation is imposed via an optional "includes" header at the top of the environment file. This allows IPE-theories to be built up from other IPE-theories; IPE-theory inclusion involves the unions of all of the corresponding subcomponents of the theories.

An IPE-theory is used in the IPE by *loading* it: this activates the symbol declarations (so that all instances of those symbols are recognised as "special" in future), and renders all of the facts in the theory visible for use in proofs.

6.3 Using Facts in a Proof

The simplest means by which facts can be used in the IPE is to view them as schemata for "invisible" premises. A special rule can then be used to invoke a single instance of a fact as a "new" premise. This was the method used in the first theory-extended version of the IPE. The instantiation of the fact (by substituting for each generic term or formula) had to be performed entirely by the user.

For example, suppose that the following goal arose in a proof:

show x+0 = x, x = 0+x entails x+0 = 0+x

Obviously, the transitivity of equality would be of use here. In the IPE, this property of equality is not assumed, but can be found as a lemma in the theory of equality:

```
lemma EqualTransitive is
  x=y&(y=z)→x=z
generic terms y
          and x
          and z
```

To use this, we select the Proof node with the above goal and choose a "Recall Fact" option on the right mouse button menu. This expands the Proof node with a general "recall fact" template:

```
show x+0 = x, x = 0+x entails x+0 = 0+x
use <FACT-NAME>
and show x+0 = x, x = 0+x entails x+0 = 0+x
```

Now we edit the text at "FACT-NAME", replacing it with "EqualTransitive". This gives us:

```
show x+0 = x, x = (0+x) entails x+0 = (0+x)
use lemma <EqualTransitive>
    with <y> for y
    and  <x> for x
    and  <z> for z
and show x=y&(y=z)→x=z, x+0 = x, x = (0+x)
         entails x+0 = (0+x)
```

The lemma has been instantiated to variables of the same name as its generic terms, and added as a new premise in the subgoal. We can replace any or all of the substitutions by a series of text-edits, so that after editing each of the substitutions for x,y and z we get:

```
show x+0 = x, x = (0+x) entails x+0 = (0+x)
use lemma <EqualTransitive>
    with <x> for y
    and  <x+0> for x
    and  <0+x> for z
and show x+0=x&(x=0+x)→x+0=(0+x), x+0 = x, x = (0+x)
         entails x+0 = (0+x)
```

Finally, we perform Implies Elimination and And Introduction to prove the original goal. (Note that these latter steps could have been applied before the final substitutions for the generic terms were chosen; this ability can be useful when the lemma is a large complicated formula).

Needless to say, this style of interaction made any proof involving more than several fact instantiations tedious.

6.3.1 The Facts Browser

To reduce the tedium of using facts, it was decided to use one-way matching to perform partial instantiation of facts, by matching a formula schema against

some formula in a goal. The initial inspiration for this derives from the matching facilities used in the B tool [Abrial 86a]. To facilitate this, although facts are still filed as formula schemata, they are used as sequent schemata.

Facts as Sequents

Defn. A *sequent schema* is the sequent analogy of formula schema, consisting of a sequent paired with sets of generic formulae and terms. For example:

$$(x = y, y = z \vdash x = z, \{\}, \{x, y, z\})$$

Conversion of the formula part of a formula schema into a sequent is performed by a simple tactic called "Factic". This converts any formula into a single sequent by repeated application of the IPE rules Implies_Intro and And_Elim (The application of any other rules would result in more than one sequent (eg by And_Intro) or loss of information (eg All_Elim)).

The aim of this conversion is to reduce the formula to a form which might be matched more usefully in a goal-directed proof. For example, consider the Equal-Transitive lemma. A common case where it is required is when the conclusion is of the form A=C and some B is required such that the proofs of A=B and B=C are more obvious to the user. Then ideally we wish to replace the conclusion A=C with the two conclusions A=B and B=C and an appeal to EqualTransitive. If the matcher sought a goal-conclusion which matched the EqualTransitive formula-schema, then it would not consider the lemma applicable in this case. However, if the lemma is converted into a sequent, then the conclusions would match, and the hypotheses of the partially-instantiated lemma would become A=<y> and <y>=C. Note that we can also match a premise of the form A=B against either hypothesis of EqualTransitive.

However, such an approach is not without its disadvantages. By only matching against the sequent form of a fact, we render the matcher incapable of matching EqualTransitive against a conclusion of the same shape as the original statement of the lemma. This does not occur often in practice for this lemma, but

for some other facts the conversion goes too far. The result is that sometimes the goal has to be decomposed further (ie beyond the "natural" matching point) before the fact will be matched. A simple solution would be to match against the original formula and the sequent produced at each stage of the conversion, and present any matches resulting. Whilst being more thorough, this has the disadvantage of being more costly, and of producing large numbers of irrelevant or redundant matches.

The New Recall_Fact Production

The form of the new Recall_Fact rule is different, taking advantage of the presentation of facts as sequents. Given a goal:

show Premise-1,. . . ,Premise-n entails Conclusion

and a fact-sequent which has been matched against it:

Hypothesis-1,. . . ,Hypothesis-m entails Result

Recall_Fact generates the two subgoals:

show Premise-1,. . . ,Premise-n
    entails Hypothesis-1&. . . &Hypothesis-m

and

show Result, Premise-1,. . . ,Premise-n entails Conclusion

That is, in order to use an instantiation of a fact-sequent, we must prove that its hypotheses are derivable from the current premises, and that the addition of the result to the premises can lead to a proof of the current conclusion.

Since the instantiations of generic variables in a fact-sequent are arrived at by matching a single formula of the sequent against a formula of a goal, it is possible that some generic variables (those not mentioned in the matched formula) will remain uninstantiated. Thus the Recall_Fact rule must still permit the user to instantiate these.

The Fact-Matching Algorithm

The matching algorithm used is a simple one-way matcher. Given a formula (chosen from a goal by the user) and a formula containing generic term or formula variables (from the sequent form of a fact), the matcher determines whether or not some instantiation of the generic variables could identify the fact-formula with the goal-formula, and what the instantiation should be. It does this by comparing the two formulae structurally, moving through the tree-form of the formula in a depth-first fashion. The the topmost operators are compared initially, then their corresponding arguments if the operators match Whenever the matching process arrives at a generic variable in the fact-formula, then the variable is set locally to the corresponding formula or term in the goal-formula; this is considered to be a locally successful match. Matching fails if either formula differs in structure (other than at generic points on the fact-formula), or if two locally-successful matches give different substitutions for the same generic term. This latter check is performed at each branch in the syntax trees of the formulae.

For example, suppose that "x" were a generic variable in the formula "x+0=x". Then to matching against the goal-formula "S(0)+0=S(0)", the matcher would compare the "=" symbols, then the "+" symbols. Next it would match the leftmost instance of "x" against the leftmost instance of "S(0)"; since "x" is a generic variable, it would be locally set to "S(0)". The "0"'s would be matched next, and since there is only one substitution for "x", the term "x+0" would be deemed to locally match "S(0)+0". Then the rightmost "x" would also be locally set to "S(0)", and since this does not conflict with the setting for the left-hand side of the equation, the entire fact-formula would be considered to have matched the goal-formula under the instantiation of "x" to "S(0)".

However, "S(0)+0=0+S(0)" would not be matched, since the two occurrences of "x" would require different substitutions.

One important point that should be made about the matching algorithm is that it cannot deal with second-order generic variables. This means that goal-formulae will not be matched against fact-formulae which contain a second-order

generic. This has the consequence that the naturals induction rule will not be matched against any problem expressed as a formula, for example, if we have the problem "!m !n m+n=n+m", and the induction rule is expressed as a sequent schema:

(phi(0), !x (phi(x) →!x phi(S(x))) ⊢ phi(x), {}, {phi(x)}),

then the "!x phi(x)" succedent will not be matched against the problem, because "phi(x)" is a second-order generic formula.

It is still possible to match against facts which contain second-order generics, so long as the formula matched within the fact-sequent does not contain a second-order generic. In the case of the Substitution lemma:

```
lemma Substitution is
x=y →(f(x)=f(y))
generic terms x
          and y
          and f(x),
```

the lemma can still be matched against a goal-premise which is an equality, for example:

```
show m+0=m entails n+(m+0)=(n+m)
use lemma <Substitution> on premise 1
    with m+0 for x
    and  m for y
    and ⟨f(x)⟩ for f(x)
andshow m+0=m
andshow m+0=m, f(m+0)=f(m) entails n+(m+0)=(n+m)
```

The user now has to replace "f(x)" in the edit-place with "n+x"; this will complete the second subproof.

The decision to avoid handling second-order matching was made out of expediency. As we are only performing "one-way" matching (in the sense that when

matching, only one formula contains generic variables), and since the IPE's expression syntax does not include lambda-expressions, then there should be a finite number of matches.

The Theory Database Browser

The third component of our new means of applying facts to goals is a browser for the IPE's theory database. This extracts facts from the database one at a time and presents them to the matcher. The most-recently-loaded IPE-theory is searched first, followed by the IPE-theories it includes, and so on. (Thus the visible theory structure is searched in a breadth-first fashion). The hope is that those facts most specific to the problem will be found early in the search. Unfortunately this places the onus upon the user to conduct a proof of a problem in the relevant theory, though what often happens in practice is that the contents of irrelevant theories are "glossed over" by the matcher.

Due to a limitation in the ML interface to Unix, the only way in which the IPE can learn which facts are present in a theory is by reading a file in that theory (called .facts) which lists them. This file defines the order in which facts are extracted from the theory, typically, the axioms are extracted first, followed by the lemmas in order of generation.

As a simple heuristic, new lemmas created during an IPE session are presented to the matcher before the theory structure is searched. If a user suspends one proof in an IPE session whilst proving a new lemma to be used in the main proof, then the new lemma will be matched first when it is required (provided that it does match the problem in the original proof).

We are now ready to describe how these tools are combined in the "facts chooser".

Using the Facts Chooser

The user selects a formula from a goal, choosing the formula that looks most likely to yield a good match. (Knowing what makes a good choice comes with

experience; however the IPE's navigability and flexibility to change encourages experimentation). If this formula is the conclusion of the goal then the matcher will match the conclusion of each fact (produced by the store searcher) against it, otherwise each premise of each fact is matched against it.

Suppose we are working within Peano number theory, and have the goal:

show $\boxed{S(0)+S(0)=S(S(0))}$

and chose to search for facts which match the boxed formula.

A "facts chooser" window appears on the screen, containing the following subwindows:

- A window displaying the current goal, with the selected formula highlighted;
- A window in which matched facts are displayed;
- A panel of buttons, presenting options available to the user.

The browser receives facts from the matcher, one at a time. Each fact is converted into a sequent, and the relevant match is performed. If any match succeeds, then the result of the match is shown to the user.

Any match instances on the display can be selected by pointing with the mouse and pressing the left or middle mouse buttons. Selecting a match highlights it, and de-selects any previous selection.

The button panel comprises:

- an "Accept" button, which exits the facts chooser and uses the selected match in the proof;
- "Prev" and "Next" buttons; these are used to scroll the facts-display when there are too many matches to fit onto the window;

- a "More" button, which directs the browser to find another matching fact from the theory database and add the match(es) derived from it to the facts-display;
- a "Cancel" button, to exit the chooser and leave the proof unchanged.

Each button is only visible when it can be applied; for example, the "Accept" button will only appear once some match has been selected, "Prev" only appears when some matches disappear off the top of the window, and "More" disappears when there are no more facts in the database which match with the selected formula of the goal.

(The idea of "hiding" inapplicable buttons was inspired by the Apple Macintosh style of "greying-out" options which do not apply, and rendering them unselectable. The easiest way of mimicking this in our ML window interface was to simply hide the button windows underneath the background).

The right button menu presents the same options which are available on the buttons (but without any form of hiding); this is solely for consistency with the IPE's main interface and the interface of its other subtools.

In our example, the first matching fact is the axiom S2; this is shown in Figure 6–1 In other words, if we can show that "S(lhs)=S(rhs)", then the axiom S2 will give us "lhs=rhs". However, intuition suggests that this new subproblem is no easier than the old one; so we tell the browser to continue searching for more matches.

After several more inappropriate matches, the display appears as in Figure 6–2. Note in particular the matching of EqualTransitive. This is the result of matching the generic terms "x" and "z" in the original statement of the lemma to "S(0)+S(0)" and "S(S(0))" respectively. (Note that "y" has been left unmatched: unfortunately, this is not made completely obvious in the display).

Suppose that we decide to accept this match. We click the left button over the lines of "EqualTransitive"; this results in the match being highlighted, and the "Accept" button appears (Figure 6–3). . When we click in the "Accept"

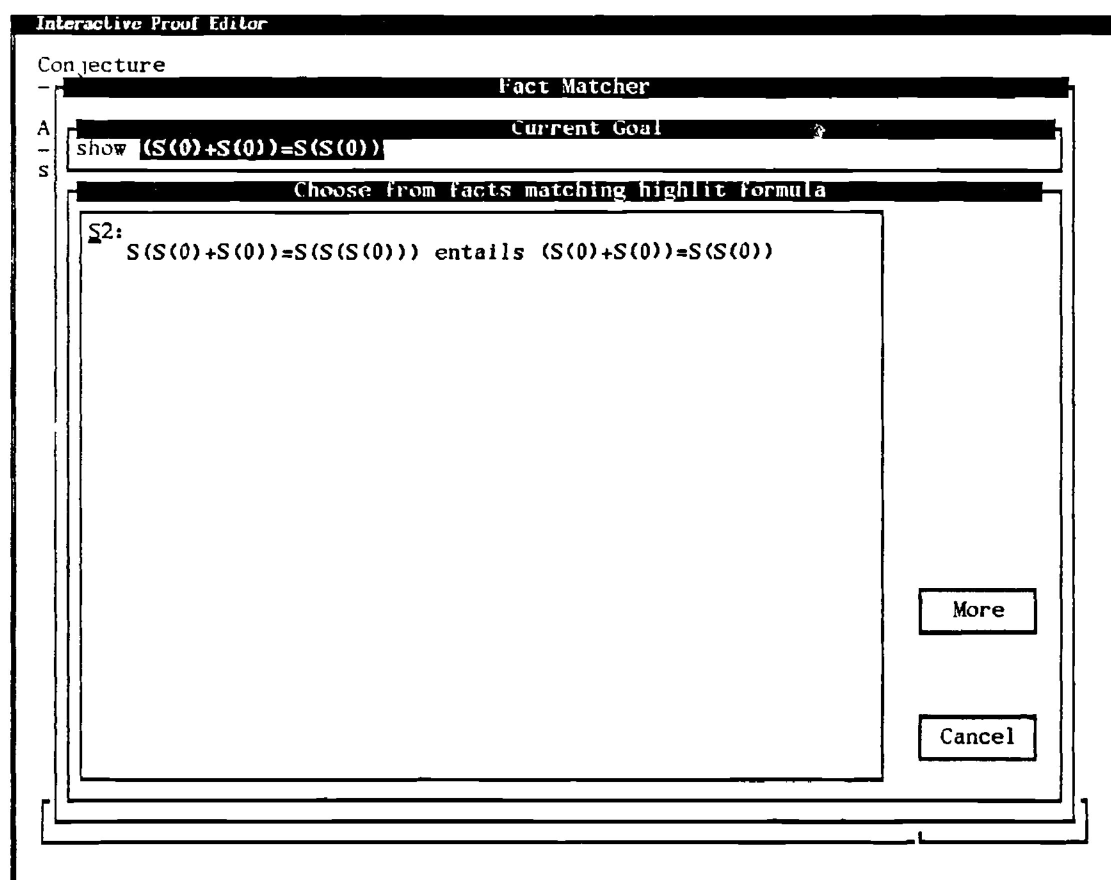


Figure 6–1: Upon entering the Facts-Chooser

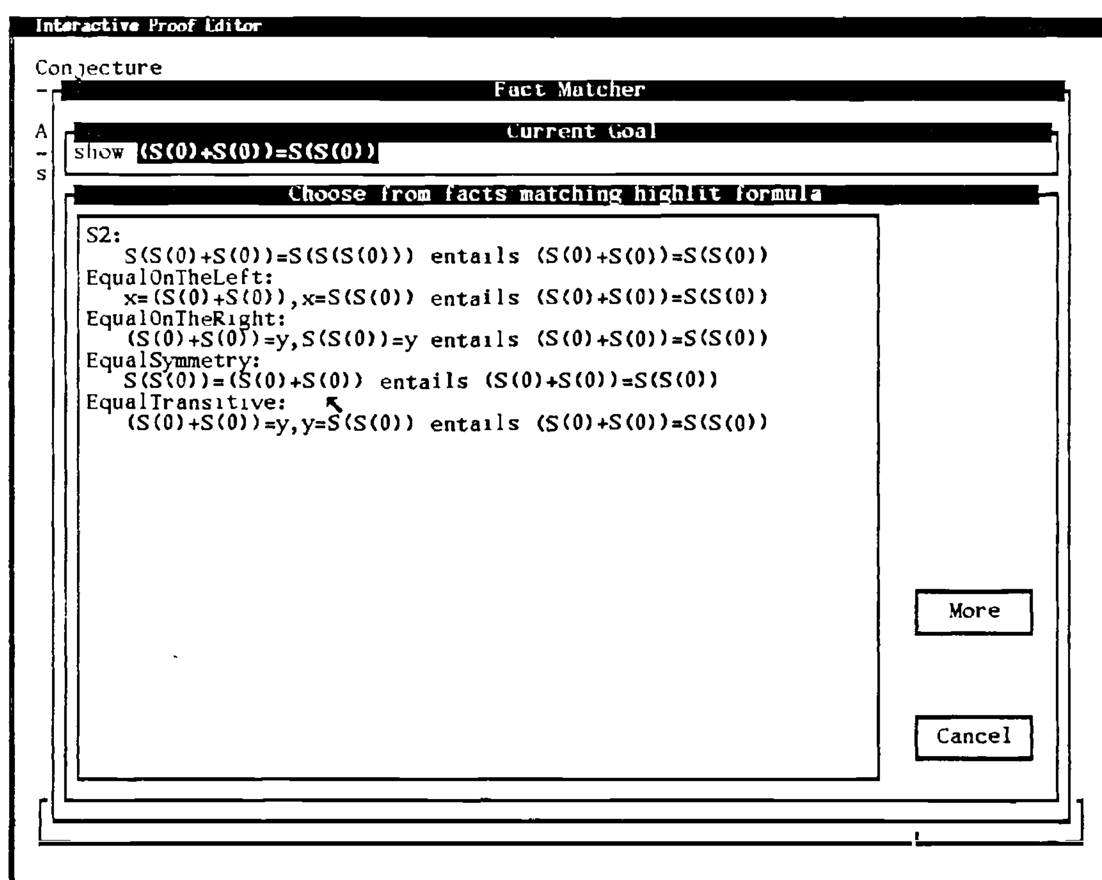


Figure 6–2: Display of several matched facts

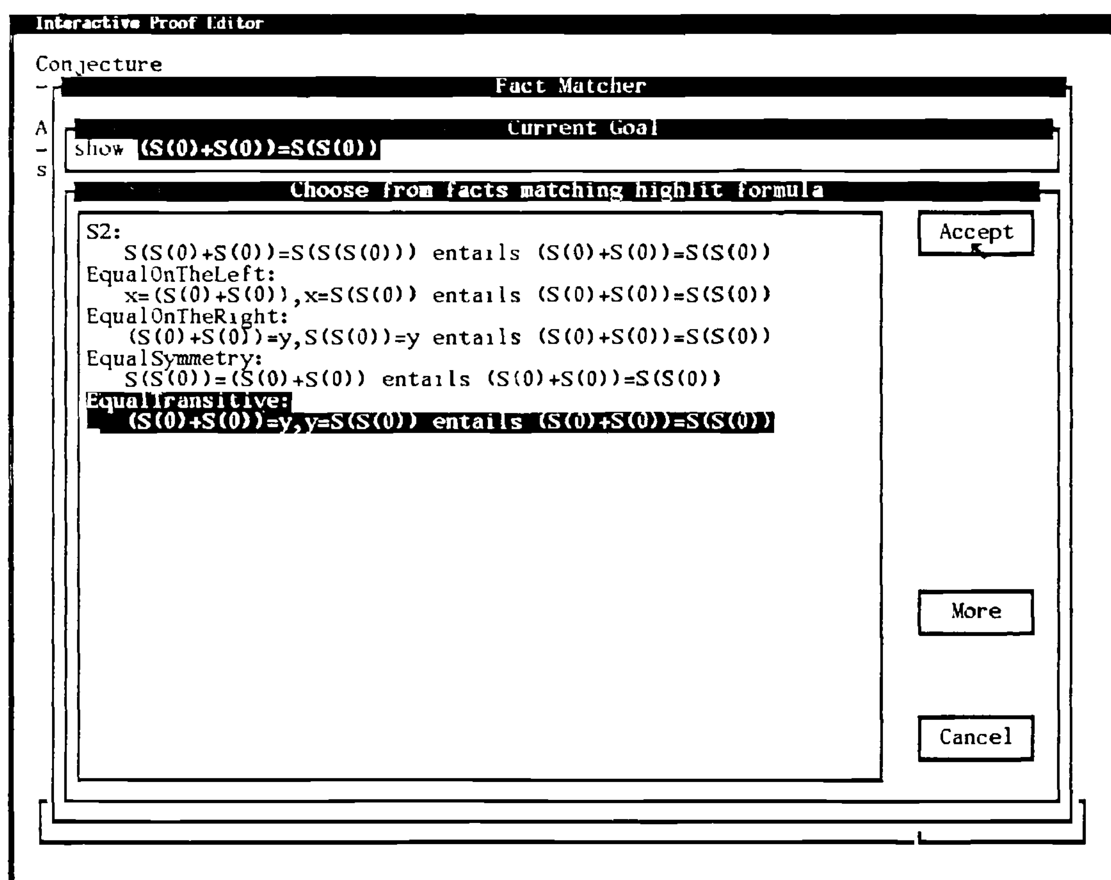


Figure 6–3: Selecting the EqualTransitive match

button, the chooser exits, and the proof at the point where we invoked the facts chooser is updated as shown in Figure 6–4.

```
Interactive Proof Editor

Conjecture
----------
     <(S(0)+S(0))=S(S(0))>
Attempted Proof
---------------
show (S(0)+S(0))=S(S(0))
use lemma <EqualTransitive> on conclusion
    with S(S(0)) for z
    and  S(0)+S(0) for x
    and  <y> for y
and show (S(0)+S(0))=y&(y=S(S(0)))
and (S(0)+S(0))=S(S(0)) entails (S(0)+S(0))=S(S(0))
    is immediate

Buffer: Main      Root: Theorem
```

Figure 6–4: After accepting the EqualTransitive match

The format of the Recall_Fact rule shows us that the generic terms "x" and "z" have been instantiated by the match, and are fixed, but that we are free to choose some term for "y". Since the result of the fact was matched against the conclusion of the goal, the second subproof is trivial; thus we have used EqualTransitive in a goal-directed fashion.

Now we must look at our new subproblems, and use our intuition to guide our choice for "y"; there are no heuristics in the IPE to do this for us. A little thought (and perhaps a look at the Peano axioms) will suggest that "S(S(0)+0)" would be a good choice for "y" (Figure 6–5. With this choice, the two subgoals (following And_Introduction) are:

show S(0)+S(0)=S(S(0)+0)

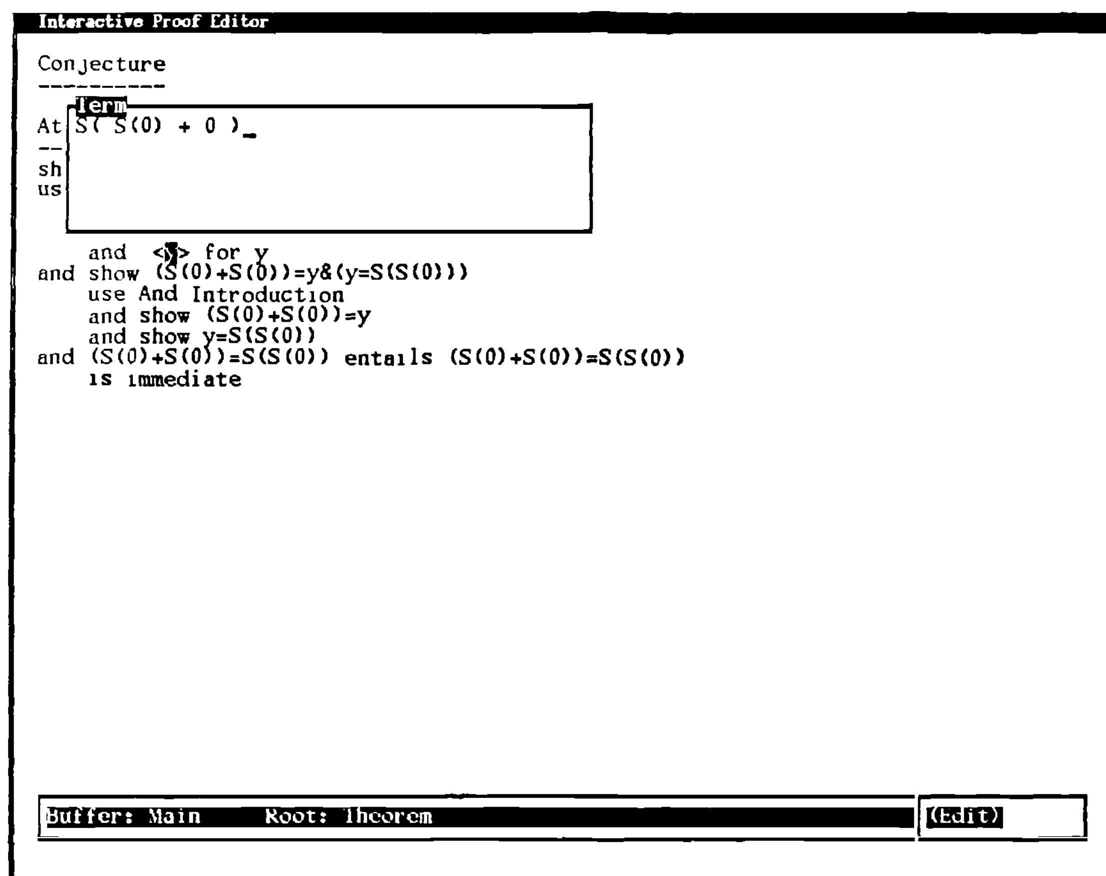


Figure 6–5: Making the substitution for "y"

and

show S(S(0)+0)=S(S(0))

Applying the facts chooser to the first subgoal shows that it is an instance of the Peano axiom "x+S(y)=S(x+y)". The second problem can be proven using the lemma "(x=y)→(S(x)=S(y))" and the Peano axiom "x+0=0". Figure 6–6 shows the completed proof.

```
Interactive Proof Editor

Theorem
-------
<(S(0)+S(0))=S(S(0))>
Proof
-----
(S(0)+S(0))=S(S(0))
by lemma <EqualTransitive> on conclusion
    with S(S(0)) for z
    and  S(0)+S(0) for x
    and  <S(S(0)+0)> for y
and (S(0)+S(0))=S(S(0)+0)&(S(S(0)+0)=S(S(0)))
    by And Introduction
    and (S(0)+S(0))=S(S(0)+0)
        by axiom <PlusS> on conclusion
            with 0 for y
            and  S(0) for x
        and (S(0)+S(0))=S(S(0)+0) entails (S(0)+S(0))=S(S(0)+0)
            is immediate
    and S(S(0)+0)=S(S(0))
        by lemma <S2rev> on conclusion
            with S(0) for y
            and  S(0)+0 for x
        and (S(0)+0)=S(0)
            by axiom <PlusZero> on conclusion
                with S(0) for x
            and (S(0)+0)=S(0) entails (S(0)+0)=S(0)
                is immediate
        and S(S(0)+0)=S(S(0)) entails S(S(0)+0)=S(S(0))
            is immediate
and (S(0)+S(0))=S(S(0)) entails (S(0)+S(0))=S(S(0))
    is immediate
QED

Buffer: Main      Root: Theorem
```

Figure 6–6: The completed proof

Other Browsers For IPE-theories

The above implementation of the facts chooser has greatly reduced the effort in constructing proofs in user-defined theories. However, there are occasions when other forms of browsing are required. The other forms of obtaining facts from the database which are available in the IPE are:

- from a selection of named facts. This is a different "front end" to the chooser, which presents a scrollable list of the names of every fact visible from the currently loaded IPE-theories. The user can select any number of these names; upon acceptance, the chooser is entered, matching only the chosen facts against the selected goal-formula. Facts are displayed even if they fail to match the goal; this allows us to browse facts which fail to match because they contain second-order generic variables.

- from all facts. In this version, every fact found is shown, whether or not a match was obtained against the selected goal. This acts as a full browser for the database.

- by name. This is similar to the earlier form of using facts, except that the list of fact names is presented. When one fact is chosen and accepted, the Recall_Fact rule is applied as if no generic variables had been matched.

In practice, the first variant is the most useful, once the user has acquired some knowledge of the structure and content of the theory database.

It would also be useful to be able to see and make use of the relationships between the various IPE-theories in the database. At present, when (say) Peano is loaded, it is not obvious that it includes Equality, until the user starts to browse for facts. Even then, the only distinction between Peano facts and Equality facts is that the Peano facts are matched (or are displayed) first.

6.4 Generating Lemmas

Generating a lemma from an IPE proof is a simple task for the user. The "make-lemma" command (invoked by the "L" key) checks that the current proof is complete; if so, it then asks the user for the name of the theory in which the new lemma will be placed, and a name for the new lemma. If the theory exists, and if it doesn't already contain a lemma with that name, then the lemma is added to the contents of the theory (together with the printed version of its proof).

It would be more secure to insist that lemmas be stored within the topmost currently-loaded IPE-theory. However, once a theory is loaded in IPE, it cannot be unloaded[1]; furthermore, theory-loading is global rather than buffer-specific. Choosing the theory by name makes it possible to store a lemma in its most general IPE-theory. For example, if whilst working on a proof in Peano we discovered that we required a result from Equality which has not yet been proven, we can start a proof of it in a new buffer; once complete, we can add it to Equality as a lemma, then return to our original proof.

When converting the theorem of an IPE proof into a lemma, the IPE uses information from the currently-loaded theories to determine the generic variables of the lemma. Any term or predicate symbol used in the theorem which has not been declared in the loaded IPE-theories is recorded as a generic variable. This ensures the validity of the lemma with respect to the topmost loaded theory, as it is impossible to "genericise" a predicate or term in a theorem whose proof relies on properties of that symbol defined by the axioms of some loaded theory. Unfortunately, there is also the risk that some symbol used by the user will coincide unintentionally with a symbol declared in the theory, and will not be made generic even if no special properties of the declared symbol were used in the proof.

The generic formula produced from a theorem is stored in the database in the same format as axioms, viz:

```
lemma Lemma_Name is
  formula
generic terms ...
generic formulae ...
```

As an example, suppose that working in Peano we had proven the formula:

S(x)+S(y)=S(S(x+y)).

[1]The ability to unload theories was added by Claire Jones in 1986.

Now if we decide to add it to Peano as a lemma called "DoubleS", we select the "make-lemma" command and supply the theory and lemma names appropriately. The symbols "S" and "+" are declared in Peano, and "=" is declared in Equality (which Peano includes), therefore these symbols are not made generic in the resultant lemma. However, "x" and "y" are not declared, so IPE makes them generic. The resultant lemma is:

```
lemma DoubleS is
S(x)+S(y)=S(S(x+y))
generic terms x
          and y
```

6.5 Remarks

It must be admitted that the IPE's theory database is not secure; the intention in its design and implementation was to investigate how the IPE could be used in conjunction with a database of results, rather than to construct a full system for building theories. This is an area where much improvement could be made.

There are no tools to help with the construction of IPE-theories. The user must create the theory as an ordinary UNIX directory under the "theories" directory, and add its name to the .theories file in that directory. (The latter file is used by the load-theory command to obtain the names of all available theories). The .environment and .facts files, and even the axiom files, must be created using an ordinary text editor. This makes it all too possible to introduce syntactic errors in constructing a theory. (In fact, some effort was made to build a version of a UNIX directory browser (written in ML by John Cartmell) which could be used to build theories. In this, users would construct the .environment file and axiom files textually, but as for text-edit points in the IPE, these would be parse-checked before acceptance. The .facts file would be automatically maintained. However, this work was superceded by the construction of the facts chooser, and was never completed.)

The "inclusion" method of putting IPE-theories together in the simplest possible way is too limited: often one would like to be able to use other theory constructions, for example, using ideas from the LARCH Shared Language ([GHW 85]):

renaming The ability to construct a new IPE-theory by renaming some or all of the declared predicates, functions or constants would allow greater reuse of IPE-theories. For example, we could build an IPE-theory "Group" which states the basic group axioms in terms of a constant "0" and functions "x.y" and "inv(x)", then rename these to obtain particular instances of groups;

assumption An IPE-theory "T" might make assumptions about its defined symbols, for example that a particular binary function is commutative. This could be done by defining an IPE-theory "Commutative" and then assuming this theory with the function suitably renamed. Now any theory "T2" which refers to "T" must discharge the assumption, by showing that the axioms of the (renamed version of) Commutative can be proven in T2;

implication Saying that one IPE-theory implies another would allow users of the first IPE-theory to access results in the second. An IPE-theory for integer arithmetic might imply "Group" (with a suitable renaming). Note however that we would have to show that the implication was valid; this could be done by proving that the axioms of the implied theory hold in the implier.

Catering for such constructions would require a major redesign of the theory database and the browser. Any tool for constructing theories would have to maintain the proof obligations arising from implication and assumption. Renaming causes special problems when browsing: when browsing a theory A which includes a renaming of a theory B, then facts in B must be renamed before being presented either to the facts-matcher or to the user. This could be an expensive process where a large number of facts are involved. In special cases where the

renaming is invertible, it might prove more efficient to apply the inverse renaming to a problem, perform matching in B's "own language", and then rename the successful matches.

The method of "genericising" theorems to produce lemmas is over-strict. For example, suppose that whilst working in Peano, we proved that

```
(x+0=x) &(x=0+x) →(x+0=0+x)
```

using only results from equality, and realised that we had just proven transitivity of equality. When we try to save this as a lemma in Equality, the result would be

```
lemma EqualTransitive is
  (x+0=x) &(x=0+x) →(x+0=0+x)
generic terms x
```

which is not what was intended. A similar situation occurs when subformulae are not decomposed in a proof: it is possible to prove that:

```
(A|B) &C →C
```

without using Or_Elimination on A |B; however, the resulting lemma would be

```
lemma AndElimLeft is
  (A|B) &C →C
generic formulae A
          and B
          and C
```

when we would prefer

```
lemma AndElimLeft is
  A &C →C
generic formulae A
          and C
```

Ideally, the process of generating a lemma from a theorem should genericise:

- all subformulae whose structure was not used in the proof;
- all subterms t where no axiomatically-defined property of any symbol in t was used in the proof;
- all predicates P such that no defined properties of P were used in the proof.

At present, the lemma generator does not use any information from the proof of the lemma, only of the currently-loaded IPE-theories. To proceed as above, the generator would have to able to extract from a proof the set of axioms it depends upon. It would be easy to extract the set of axioms and lemmas used directly in a proof; to determine which axioms have been used to prove the lemmas would require storing this information with the lemma (or analysing the proof of each lemma similarly). It would also be necessary to determine the set of symbols whose properties are defined by a particular axiom. A simpler but overstrict solution (though less so than the present method) would be to assume that whenever a lemma from a particular IPE-theory is used in a proof, then no predicates or terms declared in that IPE-theory can be genericised. This assumes that every lemma has been stored in the appropriate IPE-theory, but would not be difficult to implement.

It would also be desirable to be able to determine the most general IPE-theory in which a lemma could be placed. Determining this *given a particular proof of the lemma* would involve finding the most general theory containing all of the facts and declared symbols used in the proof. Were lemmas to be automatically placed in this manner, there would be the risk that users might be surprised when lemmas disappear from the theory chosen by the user. In the current presentation of facts in the IPE, this would simply mean that the lemma might not appear in the list of matched or viewed facts until later than expected; however, if theory-structured browsing were realised, this will become a more serious issue.

The user-directed storage of lemmas is certainly dangerous at present. No checks are made to ensure that the lemma could genuinely be proven within the theory chosen by the user. The ability to analyse proofs as above could be used to restrict the user's choice. Such proof analysis could even be used to decide the most general IPE-theory automatically. However, it could be confusing when a result is generalised in ways unexpected by the user, producing an unrecognisable lemma stored in a different theory from the expected position.

As stated earlier, it would be preferable if different buffers could be "opened upon" different IPE-theories. This could be used to improve the storage of lemmas. Suppose that a user working upon a Peano proof, discovers the need for a new Equality lemma; it would be possible to create a buffer upon Equality in which the lemma could be stated and proved. "Store-lemma" might insist that the lemma be stored in the theory associated with the buffer. If such a system were adopted, then copying of information between buffers would be more complicated: it should only be possible to copy from buffer A into buffer B when the theory associated with buffer A is already loaded in buffer B.

Chapter 7

Future Work and Conclusions

7.1 Recent Work

As stated in the Introduction, this thesis has concentrated upon describing the IPE as far as Version 5, thus covering the main part of the author's contribution to the work. However, the IPE has been developed further since the author's involvement. This section briefly describes some of the ways in which the IPE has been extended.

7.1.1 Rewrite Rules

In mid-1986, Claire Jones extended the IPE to include rewrite rules. Each IPE-theory can have a list of rewrite rules (stored as a list of facts in a file ".rules"). When an IPE-theory is loaded, its rewrite rules are added to the set of loaded rules. Any proven formula of the form $x = y$ (where x and y are terms) can be used as a rewrite rule for rewriting instances of x to the corresponding instances of y (recalling that the rule may have generic subterms). Once the proof of such a formula is completed, it can be added to the set of rewrite rules in a (user-selected) IPE-theory. Claire has extended the Proof Grammar with a new Proof production "Rewrite", which applies the loaded rewrite rules to the selected premise or conclusion. Rewriting is performed repeatedly, rewriting terms "from the outside in", until none of the rewriting rules apply.

The addition of rewrite rules has greatly improved the usability of the IPE; a small number of rewrite rules can achieve a great deal in a single step. (A crude

example is that the proof in §6.3.1 could now be performed in a single step, with only the axioms PlusS and PlusZero as rewrite rules). However, the rewrite rules must be chosen carefully. Should the user inadvertently add "$t_2 = t_1$" as a rule when "$t_1 = t_2$" is already present, then later attempts to perform rewriting will fail to terminate whenever one of these rules applies. Furthermore, changes to the set of rewrite rules may cause earlier rewrites to produce different results upon reevaluation.

7.1.2 The XIPE

In 1986, the Laboratory for Foundations of Computer Science decided to adopt the X windows system in preference to SunView. Furthermore, the differences between the new and old versions of SunView were such that Tatsuya Hagino's ML window system would require extensive redesign to work under the new SunView environment (indeed, Tatsuya tried and failed). To continue development of the IPE, Tatsuya built an ML window system which used X but provided the same interface as the old system, thus enabling the same IPE code to run under X. Tatsuya has since proceeded to extend and improve IPE's user interface in a variety of ways, including the "proper" display of the quantifiers "$\forall$" and "$\exists$" and redesign of the formula and term parser to accept a more standard syntax. More recently, the use of the keyboard for numerous "single-stroke" commands has been superceded by mouse menus.

7.2 Future Work

In this section we consider ways in which the Interactive Proof Editor could be extended. Many of these involve incorporating features from other theorem proving systems into the IPE to increase its practical applicability, rather than areas for novel research.

An obvious extension would be to provide a typed predicate calculus, which would increase confidence in our theorems, preventing us from proving results such as

nil+0 =nil.

One could go further and provide an implementation of a "logical frame" (as done in the EFS). At present, the syntax of formulae and terms in the IPE is extremely rigid. The ability to give a syntax (or just a presentation, as in PRL) would enhance the readability of formulae. Similarly, the ability to define new logical connectives (even if only defined in terms of the existing set of connectives) would be useful.

The uses of IPE-tactics have not been fully developed in IPE. An immediate extension of IPE would be to build up an internal library of IPE-tactics and allow the user to choose from these. Slightly more long-term would be the provision of a language in which users could construct IPE-tactics from a set of basic tactics plus "IPE-tacticals".

Much could be done to improve the theory database. In particular, a secure method of constructing theories is required. It would be very useful to be able to change an IPE-theory and have the effects of the change propagate through the rest of the database. Following from her experiences in proving a simple parser using IPE, Claire Jones added an "Unload theory" command to the IPE as one method of allowing the declarations of symbols and definitions of axioms in an IPE-theory to change during a session. Unfortunately, proofs built using the

old version of the theory are not automatically forgotten! A proper dependency structure of lemmas upon the facts used in their proofs would be the first step (after a secure database editor) towards a change-sensitive IPE.

Claire Jones' work in adding rewrite rules to the IPE could be developed more fully, for example taking advantage of existing work in organising sets of rewrite rules, for example using the Knuth-Bendix algorithm to derive a confluent and terminating set of rewrite rules (see [Dick 84]). Another possible approach would be "user-directed" rewriting, where the user could select a subterm and ask the store searcher to find possible (single-step) rewrites for the term. Though slower, this would have the advantage that the user would be aware of each step made, and that no checks need be made upon the set of rewrite rules other than that they be proven lemmas or intended axioms.

Further experimentation upon improvements to the reusability of IPE structures is required, for example by further developing some of the ideas in §3.9.2.

Later extensions to the IPE's user interface (in particular the "chooser" interface style used in the facts-matcher and for most buffer operations) suffered from restrictions imposed by low-level details of the user interface. (For example, unless a menu is bound to a mouse button, then no distinction is made between pressing a button (holding it down) and clicking it. It would be interesting to reconstruct the IPE using a more versatile set of I/O primitives; work in Edinburgh upon providing an interface to the X window system in Standard ML is of interest here.

For a system which is intended to be easy to learn, the IPE's help system is severely deficient. Some form of introductory help is required, even if only in the form of a step-by-step guide through the development of several proofs.

7.3 Concluding Remarks

Many people have used the Interactive Proof Editor, covering a wide range from novices to experienced practitioners of formal proof. User response to the IPE has generally been favourable. The IPE demonstrates that proof navigability and ease of alteration are valuable properties for proof assistants. This is particularly true when – as is often the case – the full statement of a problem is only determined during the process of attempting its proof.

The proof-by-pointing paradigm makes it easy to use the IPE's basic rules. It would be worthwhile to consider how this could be extended beyond the IPE's single-rule-per-connective restriction. For example, mouse-clicks could be used to invoke theory-specific transformations, depending upon the shape of the selected formula. Ideally, where the user frequently performs some operation upon formulae of a particular pattern, it should be possible to have this operation invoked by mouse-clicks upon formulae matching the pattern. Proof-by-pointing should be capable of adaptation to the circumstances. That it is not so in the IPE is one of the IPE's main shortcomings.

The present need to display all of the premises and the conclusion at each step of the proof leads to a somewhat cluttered display. It would be better to allow the possibility of hiding this information, at the cost of an extra "expose" operation. There is much scope for experimentation with alternative proof displays which are more succinct without sacrificing navigability.

When work began upon the Interactive Proof Editor, in order to perform machine-assisted proof, one had to be an expert in the use of a particular theorem-proving tool. Today, it seems to be the case that the learning threshold for such systems is falling. The ideal interactive theorem proving system would be perfectly transparent, in the sense that users could concentrate upon the essence of constructing proofs, rather than upon learning, or fighting with, a poor or intransigent user interface. It was not intended that the Interactive Proof Editor should be all things to all men, and it cannot be claimed that it has achieved the ideal. For example it seems unlikely that the IPE in its present form will be useful in tackling problems of the scale that arise in "real" formal software development. On the other hand, systems which have been used to tackle "large but dull" theorems (as produced by verification condition generators) have been criticised for the crudity of their man-machine interfaces, and for the incomprehensibility of their machinations. There still remains a gap between the two. Nonetheless, the IPE has provided an interface to a theorem prover which is simple to learn, and generally "forgiving" in operation.

The IPE has also demonstrated that ML can be used to build large systems with "proper" user interfaces, though the standard I/O mechanism requires supplementary window-management primitives.

Bibliography

[Abrial 86a] Abrial, J.R., *An Informal Introduction to B*, Oxford University Program Research Group internal report, 1986.

[Abrial 86b] Abrial, J.R., *B User Manual*, Oxford University Program Research Group internal report, 1986.

[Alvey 87] *Alvey Programme: Annual Report 1987 Poster Supplement.*

[Bates-Constable 83] Bates, J.L. and Constable, R.L., *Proof As Programs*, Cornell University Department of Computer Science report TR 82-530, 1983.

[Boyer-Moore 79] Boyer, R.S. and Moore, J.S., *A Computational Logic*, Academic Press, New York, 1979.

[BTJ 87] Burstall, R.M., Taylor, P. and Jones, C., *Interactive Proof Editing using the Edinburgh Interactive Proof Editor*, course organised by the Laboratory for Foundations of Computer Science, University of Edinburgh, 14–16 September 1987.

[Cardelli 83] Cardelli, L., ML Under UNIX, in *Polymorphism*, **1.3**, 1983.

[Cohn-Milner 82] Cohn, A. and Milner, R., *On Using Edinburgh LCF to Prove the Correctness of a Parsing Algorithm*, University

of Edinburgh Computer Science Department Internal Report CSR-113-82.

[Coquand-Huet 85] Coquand, T. and Huet, G., *Constructions: A Higher-Order Proof System for Mechanising Mathematics*, Lecture Notes in Computer Science **203** pps. 151-184, 1985.

[Dick 84] Dick, A.J.J., *Equational Reasoning and the Knuth-Bendix Algorithm - an Informal Introduction*, Imperial College Technical Report DOC 84/21, 1984.

[Goldberg-Robson 83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[Goldberg 84] Goldberg, A., *Smalltalk-80· The Interactive Programming Environment*, Addison-Wesley, 1984.

[Gordon 85] Gordon, M.J., *HOL: a machine oriented formulation of higher order logic*, University of Cambridge Technical Report 68, 1985.

[GMW 79] Gordon, M.J., Milner, A.J. and Wadsworth, C.P. *Edinburgh LCF*, Lecture Notes in Computer Science **78**, 1979.

[Griffin 87] Griffin, T.G., *An Environment for Formal Systems*, Cornell University Computer Science Department, technical report TR 87-846, 1987.

[GHW 85] Guttag, J.V., Horning, J.J. and Wing, J.M., *LARCH in Five Easy Pieces*, DEC SRI, 1985.

[HHP 87] Harper, R., Honsell, F. and Plotkin, G., A Framework for Defining Logics, *Proceedings of the Second Symposium on Logic in Computer Science*, 1987.

[Jalili 83] Jalili, F., A General Linear-Time Evaluator for Attribute Grammars, *SIGPLAN Notices*, Vol. **18**, No. 9, 1983.

[Johnson 78] Johnson, S.C., *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, N.J. 07974, 1978.

[Jones,K 87] Jones, K.D., *The Muffin Prototype: Experiences with Smalltalk-80*, Alvey Software Engineering Report 060/00065, 1987.

[Kleene 64] Kleene, S.C., *Introduction to Metamathematics*, North-Holland, 1964.

[Knuth 68] Knuth, D.E., Semantics of Context-Free Languages, *Math. Syst. Theory 2*, **2**, pps. 127-145, 1968.

[Moore 86a] Moore, R., *The Muffin Database*, Alvey Software Engineering Report 060/00060, 1987.

[Moore 86b] Moore, R., *The Muffin Prototype*, Alvey Software Engineering Report 060/00066, 1987.

[Nakajima *et. al.* 83] Nakajima, R., Yuasa, T., Hagino, T., Honda, M., Koga, A., Kojima, K. and Shibayama, E., *The IOTA Programming System*, Lecture Notes in Computer Science **160**, Springer-Verlag, 1983.

[Paulson 85a] Paulson, L.C., *Natural Deduction Theorem Proving via Higher-Order Resolution*, University of Cambridge Computer Laboratory, Technical Report No. 67, 1985.

[Paulson 85b] Paulson, L.C., *Interactive theorem proving with Cambridge LCF: a user's manual*, University of Cambridge Computer Laboratory technical report no. 80, 1985.

[Paulson 86] Paulson, L.C., *A proposal for theories in Isabelle*, draft report, 1986.

[Petersson 82] Petersson, K.P., *A programming system for type theory*, University of Göteborg Laboratory for Programming Methodology memo no. 21, 1982.

[Prawitz 65] Prawitz, D., *Natural Deduction*, Almqvist and Wiskell 1965.

[PRL 86] The PRL Group (Constable et.al.), *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, New Jersey, 1986.

[Proofrock 83] Proofrock, J.A., *PRL Programmer's Manual*, Cornell University Department of Computer Science, 1983.

[Reps-Alpern 84] Reps, T. and Alpern, B., Interactive Proof Checking, *ACM Symposium on Principles of Programming Languages*, Salt Lake City, 1984.

[Reps 82] Reps, T., *Generating Language-Based Environments*, Ph.D. thesis, Cornell University Computer Science Department, 1982.

[Reps et.al. 83] Reps, T., Teitelbaum, T. and Demers, A., Incremental Context-Dependent Analysis for Language-Based Editors, *ACM TOPLAS* **5**, No. 3, 1983.

[Reps-Teitelbaum 85] Reps, T. and Teitelbaum, T., *The Synthesiser Generator Reference Manual*, Cornell University Department of Computer Science, 1985.

[Ritchie 87] Ritchie, B., Interactive Proof Construction, in *IEE Colloquium on "Theorem Provers in Theory and in Practice,*

Institution of Electrical Engineers Digest No. 1987/39, 1987.

[Schmidt 83] Schmidt, D., A Programming Notation for Tactical Reasoning, in *Proceedings of the Seventh International Conference on Automated Deduction*, Lecture Notes in Computer Science **170**, pp. 445–459, Springer-Verlag, 1984.

[Simpson 87] Simpson, D., Some Formal Methods Work, in *Alvey News*, Issue No. 23, 1987.

Appendix A

The Proof Grammar

A.1 The Syntax of C-SEC

This section gives a brief description of the syntax used in C-SEC to describe attribute grammars.

Keywords are placed in bold font; words in normal font are nonterminals.

An attribute grammar in C-SEC is described in several sections:

- a title line of the form, "**attribute grammar** Name";
- an optional "**include**" keyword followed by a space-separated list of names of ML modules. These modules are then made visible throughout the attribute grammar;
- a types section, which defines the types of all the attributes used;
- the declaration of the root symbol of the grammar;
- a list of definitions of all symbols in the grammar;
- a list of the productions of the grammar.

The type definitions are separated by semicolons and enclosed in "**types end**" delimiters. Each type definition has the following form:

type-name = ML-type-expression;;
(**equality** ML-code;;)

where "type-name" must be a single word, ML-type-expression any type expression in ML which makes sense with respect to any modules included earlier, and ML-code defines an equality function over the given ML type (e.g. if the given type is "`int`", then the function should have type "`int * int -> bool`"). (The double semicolon is used as a terminator for all sections of ML code.) The "= ML-type-expression" can be omitted when "type-name" coincides with the name of the intended ML type.

The symbol definitions are enclosed in "**symbols** ... **end**" delimiters and separated by semicolons. Each symbol definition has the following form:

symbol-name (attribute-def1; attribute-def2; ...)

where symbol-name is a single word, and each attribute-def is of the form "**synthesised** type-name attribute-name" or "**inherited** type-name attribute-name", thus declaring the named attribute as a synthesised or inherited attribute of the symbol, which will be used to hold ML values of the corresponding type.

Grammar productions for the same symbol are grouped together, each group being separated by a semicolon. The first production of each group is considered to be the completing production of the symbol (and should therefore have no right-side symbols). Each production-group is of the form:

```
symbol-name1 ::= rule-name ( symbol-name2 symbol-name3 ... )
                     [ semantic-equation1
                       semantic-equation2
                       ...
                     ]
                 | rule-name2 ...
```

(There is also an optional "**inML**... **end**" section prior to the semantic equations section, which can be used to load any ML code required by a particular production).

Each semantic equation has the form:

symbol-name$n.attribute-name = semantic-function;;

which indicates that it is defining the semantic function for the named attribute of the nth symbol of that name in the production (numbering symbols from the left-side of the production, and starting with 1). The semantic function can be any ML expression whose type matches the type of the selected attribute. The semantic-function can also have embedded references to other attributes in the production, of the form:

%symbol-name$n.attribute-name

(the leading "%" is required to distinguish it from the ML code). If the "$n" is omitted from an attribute reference, it defaults to the first symbol of that name in the production.

As in ML Under Unix, comments in C-SEC are delimited with braces.

A.2 The C-SEC Definition of the Proof Grammar

```
attribute grammar Proof_Grammar
{ An attribute grammar for the core of the Interactive Proof Editor.
}

include Proof_Formatting
        Facts_Matching
        Recall_Prelims ;

types sequent (equality fun(m,n).m=n;;);
               { sequent ==
                       list of premise formulae * conclusion formula }
      seqlist = sequent list;; (equality fun(m,n).m=n;;);
      term (equality fun(m,n).m=n;;);
               { term = variable(string)
                      | expression(op:string,term list) }
      formula (equality fun(m,n).m=n;;);
               { see ~/jwc/Formulae for the definition }
      int (equality fun(m,n).m=n;;);
      bool (equality fun(m,n).m=n;;);
      string (equality fun(s1,s2).s1=s2;;);
      factinfo (equality eqfactinfo;;);
               { the tuple returned by recall_fact
                 -- see ~/gforms/Recall_Prelims }
      formlist = formula list;; (equality fun(f,g).f=g;;);
      termlist = term list;; (equality fun(t1,t2).t1=t2;;);
      subst_set = subst list option;; (equality fun(a,b).a=b;;);
      subst_list = subst list;; (equality fun(a,b).a=b;;)
               { subst == term_subst(term1,term2)
                        | form_subst(form1,form2) }
end
```

```
root Theorem

symbols Theorem  { The root symbol of the grammar, with only one
                   effective production linking a conjecture formula to
                   its attempted proof
                 }
                (synthesised bool proven;
                    { true if the underlying proof is proven }
                 synthesised int print_tree_depth;
                     { the number of tree levels below that are to be
                       displayed }
                 synthesised int set_ptd;
                     { The user-changeable version of print_tree_depth;
                       This will always have no arguments in semantic
                       equations
                     }
                 synthesised int no_of_columns;
                     { Width of display left to the current node, after
                       indentation }
                 synthesised int set_noc
                     { Is to no_of_columns what set_ptd is to
                       print_tree_depth }
                );
        Proof   { The main symbol in this grammar, representing each
                  step of a proof }
                (synthesised bool proven;
                    { true if the given rule is appropriate and if some
                      function of the validity of its subproofs is also
                      true }
                 synthesised bool appropriate;
                     { true if local conditions for the given proof rule
                       are met }
                 synthesised int selected;
```

```
            { The index of the premise to which the rule
              applies (not relevent to introduction rules) }
        inherited sequent sequent;
            { The goal supplied to the Proof }
        synthesised seqlist subgoals;
            { The subgoals generated from applying the
              operations of the proof rule to the given goal }
        inherited int print_tree_depth;
        synthesised int set_ptd;
        inherited int no_of_columns;
        synthesised int set_noc);

Term    { A single term; this allows user editing of terms }
       (synthesised term self;
           { The value of the term associated with this node }
        inherited int no_of_columns;
        synthesised int set_noc);

Formula { Similar to Term }
       (synthesised formula self;
        inherited int no_of_columns;
        synthesised int set_noc);

Var     { similar to Term }
       (synthesised string self;
        inherited int no_of_columns;
        synthesised int set_noc);

Fact    { Represents a fact from the theory database used in a
          Recall proof step; unlike other user-editable points,
          this has more information associated - the name of the
          fact, whether it is an axiom or a lemma, its generic
          parameters, and IPE-chosen substitutions for them.
        }
       (synthesised string name;
```

```
    { The name-label of a fact (eg EqualTransitive) }
  synthesised factinfo recall;
    { The bundle of info that comes from
      "recall_fact (name)" }
  synthesised bool valid;
    { True if the named fact actually exists }
  synthesised formula fact;
    { The fact as a single formula }
  inherited formlist genfs;
    { The generic formulae of a fact, renamed to avoid
      coincidence of generic parameter names with
      variables in the goal, ie generic phi(x) becomes
      generic phi(x') if x occurs at all in the goal
    }
  inherited termlist gents;
    { Similar to genfs }
  inherited subst_set autosubsts;
    { Those substitutions for generics which are
      automatically decided by the IPE }
  synthesised sequent sequent;
    { The fact expressed as a single sequent, eg
       x=y&y=z->x=z |==> x=y,y=z entails x=z }
  synthesised int selected
    { If formula-matching is carried out between a
      premise of the goal and a premise of the
      fact-sequent, this holds the index of the chosen
      fact-premise }
  );

GenFormList { This node allows the user to provide substitutions
              for those generic formulae not substituted for by
              the IPE }
            (synthesised subst_list substs;
                { The substitutions provided by the user, plus
                  any others neither automatically nor user
```

```
                     chosen. This is necessary because the set of
                     system-chosen substitutions is likely to
                     change when the goal changes.
                   }
                 synthesised subst_list user_substs;
                     { The set of substitutions provided
                       by the user }
                 inherited formlist still_genfs;
                     { Those generic formulae not chosen
                       by the system }
                 inherited int print_depth;
                 inherited int no_of_columns;
                 synthesised int set_noc);

      GenTermList { Similar to GenFormList }
                (synthesised subst_list substs;
                 synthesised subst_list user_substs;
                 inherited termlist still_gents;
                 inherited int print_depth;
                 inherited int no_of_columns;
                 synthesised int set_noc)
end


productions
   Theorem ::= Carte_Blanche ()      { Completing production for Theorem
                                       -- never used!! }
                [ Theorem.proven = false;;
                  Theorem.print_tree_depth = 100;; { These values are }
                  Theorem.no_of_columns = 79;;     { quite arbitrary! }
                  Theorem.set_ptd = ~1;;
                  Theorem.set_noc = ~1;;
                ]
            |  Theorem ( Formula Proof )
               [ Theorem.proven = %Proof.proven;;
```

```
Theorem.print_tree_depth = if %Theorem.set_ptd > 0
                           then %Theorem.set_ptd
                           else 10;;
Theorem.set_ptd = ~1;;
        { set_ptd is altered by a higher level in the
          IPE: whenever this node is the current node,
          set_ptd is set to the current print tree
          depth value (eg 5).  This value is then
          passed to Theorem.print_tree_depth, and thus
          to the chain of print_tree_depth attributes
          below the current node.  When we move away
          from this node, set_ptd is made negative so
          that print_tree_depth inherits its value
          from above (or as in this case, is set to
          a default value, since there is no "above")
        }
Theorem.no_of_columns = if %Theorem.set_noc > 0
                        then %Theorem.set_noc
                        else 0;;
Theorem.set_noc = ~1;;
Proof.sequent = make_sequent([],%Formula.self);;
        { The initial goal is "show Formula" }
Proof.print_tree_depth =
                if %Proof.set_ptd > ~1
                then %Proof.set_ptd
                else %Theorem.print_tree_depth - 1;;
       { See the note under Theorem.print_tree_depth }
Proof.no_of_columns =
          if %Proof.set_noc > ~1
          then %Proof.set_noc
          else subproof_width %Theorem.no_of_columns;;
Formula.no_of_columns =
                if %Formula.set_noc > ~1
                then %Formula.set_noc
                else %Theorem.no_of_columns - 5;;
```

```
          ];


Proof  ::=  Still_To_Prove ()
                   { The completing (or default) production for
                     Proof, this is similar to the immediate rule,
                     except that it won't complain if the goal is
                     not immediate
                   }
             [ Proof.appropriate = true;;
                      { Always applicable }
               Proof.proven = is_immediate %Proof.sequent;;
               Proof.selected = 1;;
               Proof.subgoals = nil;;
               Proof.set_ptd = ~1;;
               Proof.set_noc = ~1;;
             ]
         |  Immediate ()
            [ Proof.appropriate = is_immediate %Proof.sequent;;
              Proof.proven = %Proof.appropriate;;
              Proof.selected = 1;;
              Proof.subgoals = nil;;
                 Proof.set_ptd = ~1;;
                 Proof.set_noc = ~1;;
            ]
         |  And_Intro ( Proof Proof )
                  { show A&B |==> [show A;show B] }
            [ Proof$1.appropriate =
                              is_And( succedent %Proof$1.sequent );;
              Proof$1.proven = %Proof$1.appropriate &
                               %Proof$2.proven & %Proof$3.proven;;
              Proof$1.subgoals =
                             if %Proof$1.appropriate
                             then And_intro %Proof$1.sequent
                             else [empty_sequent;empty_sequent];;
```

```
        Proof$2.sequent = hd %Proof$1.subgoals;;
        Proof$3.sequent = hd (tl %Proof$1.subgoals);;
        Proof$2.print_tree_depth =
                     if %Proof$2.set_ptd > ~1
                     then %Proof$2.set_ptd
                     else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                 if %Proof$2.set_noc > ~1
                 then %Proof$2.set_noc
                 else subproof_width %Proof$1.no_of_columns;;
        Proof$3.print_tree_depth =
                 if %Proof$3.set_ptd > ~1
                 then %Proof$3.set_ptd
                 else %Proof$1.print_tree_depth-1;;
        Proof$3.no_of_columns =
                 if %Proof$2.set_noc > ~1
                 then %Proof$2.set_noc
                 else subproof_width %Proof$1.no_of_columns;;
        Proof$1.selected = 1;;
        Proof.set_ptd = ~1;;
        Proof.set_noc = ~1;;
       ]


|  Or_Intro ( Proof Proof )
            { show A|B ==> either show A or show B }
     [ Proof$1.appropriate =
                 is_Or(succedent %Proof$1.sequent);;
       Proof$1.proven = %Proof$1.appropriate &
                     (%Proof$2.proven or %Proof$3.proven);;
       Proof$1.subgoals =
                    if %Proof$1.appropriate
                    then Or_intro %Proof$1.sequent
                    else [empty_sequent;empty_sequent];;
       Proof$2.sequent = hd(%Proof$1.subgoals);;
       Proof$3.sequent = hd(tl(%Proof$1.subgoals));;
```

```
          Proof$2.print_tree_depth =
                          if %Proof$2.set_ptd > ~1
                          then %Proof$2.set_ptd
                          else %Proof$1.print_tree_depth-1;;
          Proof$3.print_tree_depth =
                          if %Proof$3.set_ptd > ~1
                          then %Proof$3.set_ptd
                          else %Proof$1.print_tree_depth-1;;
          Proof$2.no_of_columns =
                     if %Proof$2.set_noc > ~1
                     then %Proof$2.set_noc
                     else subproof_width %Proof$1.no_of_columns;;
          Proof$3.no_of_columns =
                     if %Proof$3.set_noc > ~1
                     then %Proof$3.set_noc
                     else subproof_width %Proof$1.no_of_columns;;
          Proof$1.selected = 1;;
          Proof.set_ptd = ~1;;
          Proof.set_noc = ~1;;
        ]


  |  Imp_Intro ( Proof )
                 { show A->B ==> show A entails B }
        [ Proof$1.appropriate =
                       is_Implies(succedent %Proof$1.sequent);;
          Proof$1.proven =
                       %Proof$1.appropriate & %Proof$2.proven;;
          Proof$2.sequent =
                       if %Proof$1.appropriate
                       then hd(Implies_intro %Proof$1.sequent)
                       else empty_sequent;;
          Proof$2.print_tree_depth =
                       if %Proof$2.set_ptd > ~1
                       then %Proof$2.set_ptd
                       else %Proof$1.print_tree_depth-1;;
```

```
          Proof$2.no_of_columns =
                    if %Proof$2.set_noc > ~1
                    then %Proof$2.set_noc
                    else subproof_width %Proof$1.no_of_columns;;
          Proof$1.selected = 1;;
           Proof.subgoals = nil;;
           Proof.set_ptd = ~1;;
           Proof.set_noc = ~1;;
        ]

|  Not_Intro ( Proof )
              { show ~A ==> show A entails contradiction }
        [ Proof$1.appropriate =
                       is_Not(succedent %Proof$1.sequent);;
          Proof$1.proven =
                       %Proof$1.appropriate & %Proof$2.proven;;
          Proof$2.sequent = if %Proof$1.appropriate
                            then hd(Not_intro %Proof$1.sequent)
                            else empty_sequent;;
          Proof$2.print_tree_depth =
                       if %Proof$2.set_ptd > ~1
                       then %Proof$2.set_ptd
                       else %Proof$1.print_tree_depth-1;;
          Proof$2.no_of_columns =
                    if %Proof$2.set_noc > ~1
                    then %Proof$2.set_noc
                    else subproof_width %Proof$1.no_of_columns;;
          Proof$1.selected = 1;;
           Proof.subgoals = nil;;
           Proof.set_ptd = ~1;;
           Proof.set_noc = ~1;;
        ]

|  All_Intro ( Var Proof )
              { show !xP(x) ==> show P(Var)
```

```
                                + Var not free in goal }
inML
    import Symbol_Table;;
end
[ Proof$1.appropriate =
        is_ForAll(succedent %Proof$1.sequent)
      & is_unique_identifier(%Proof$1.sequent,%Var.self)
      & not(is_global_constant %Var.self);;
  Proof$1.proven =
              %Proof$1.appropriate & %Proof$2.proven;;
  Proof$2.sequent =
        if %Proof$1.appropriate
          & is_unique_identifier( %Proof$1.sequent,
                                  %Var.self )
        then hd(All_intro (%Proof$1.sequent, %Var.self))
                    else empty_sequent;;
  Proof$2.print_tree_depth =
                  if %Proof$2.set_ptd > ~1
                  then %Proof$2.set_ptd
                  else %Proof$1.print_tree_depth-1;;
  Proof$2.no_of_columns =
            if %Proof$2.set_noc > ~1
            then %Proof$2.set_noc
            else subproof_width %Proof$1.no_of_columns;;
  Proof$1.selected = 1;;
  Proof.subgoals = nil;;
  Proof.set_ptd = ~1;;
  Proof.set_noc = ~1;;
  Var.no_of_columns = if %Var.set_noc > ~1
                      then %Var.set_noc
                      else %Proof$1.no_of_columns - 5
                           - (size "All Introduction");;
]

| Exists_Intro ( Term Proof )
```

```
                { show ?xP(x) ==> show P(Term) for any Term }
      [ Proof$1.appropriate =
                is_ThereExists(succedent %Proof$1.sequent);;
        Proof$1.proven =
                %Proof$1.appropriate & %Proof$2.proven;;
        Proof$2.sequent =
                if %Proof$1.appropriate
                then hd(Exists_intro (%Proof$1.sequent,
                                      %Term.self))
                else empty_sequent;;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                if %Proof$2.set_noc > ~1
                then %Proof$2.set_noc
                else subproof_width %Proof$1.no_of_columns;;
        Term.no_of_columns =
                if %Term.set_noc > ~1
                then %Term.set_noc
                else %Proof$1.no_of_columns - 5
                          - (size "Exists Introduction");;
        Proof$1.selected = 1;;
         Proof.subgoals = nil;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
      ]


 |  And_Elim ( Proof )
                { show A&B entails C ==> show A,B entails C }
      [ Proof$1.selected = 1;;
        Proof$1.appropriate =
              is_And(antecedent (%Proof$1.sequent,
                                 %Proof$1.selected))
```

```
                              ?? ["antecedent"] false;;
        Proof$1.proven =
                %Proof$1.appropriate & %Proof$2.proven;;
        Proof$2.sequent = if %Proof$1.appropriate
                          then hd(And_elim( %Proof$1.sequent,
                                         %Proof$1.selected ))
                          else empty_sequent;;
        Proof$2.print_tree_depth =
                           if %Proof$2.set_ptd > ~1
                           then %Proof$2.set_ptd
                           else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                  if %Proof$2.set_noc > ~1
                  then %Proof$2.set_noc
                  else subproof_width %Proof$1.no_of_columns;;
         Proof.subgoals = nil;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
       ]


 |  Or_Elim ( Proof Proof )
           { show A|B entails C
               ==> [ show A entails C; show B entails C ] }
       [ Proof$1.selected = 1;;
         Proof$1.appropriate =
                 is_Or(antecedent (%Proof$1.sequent,
                                  %Proof$1.selected))
                             ?? ["antecedent"] false;;
         Proof$1.proven = %Proof$1.appropriate
                        & %Proof$2.proven & %Proof$3.proven;;
         Proof$1.subgoals =
                 if %Proof$1.appropriate
                 then Or_elim (%Proof$1.sequent,
                              %Proof$1.selected)
                 else [empty_sequent;empty_sequent];;
```

```
        Proof$2.sequent = hd(%Proof$1.subgoals);;
        Proof$3.sequent = hd(tl %Proof$1.subgoals);;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$3.print_tree_depth =
                if %Proof$3.set_ptd > ~1
                then %Proof$3.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                if %Proof$2.set_noc > ~1
                then %Proof$2.set_noc
                else subproof_width %Proof$1.no_of_columns;;
        Proof$3.no_of_columns =
                if %Proof$3.set_noc > ~1
                then %Proof$3.set_noc
                else subproof_width %Proof$1.no_of_columns;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
      ]


| Imp_Elim ( Proof Proof )
              { show A->B entails C
                  ==> [ show A; show B entails C ] }
    [ Proof$1.selected = 1;;
      Proof$1.appropriate =
              is_Implies(antecedent (%Proof$1.sequent,
                                    %Proof$1.selected))
                         ?? ["antecedent"] false;;
      Proof$1.proven = %Proof$1.appropriate
                       & %Proof$2.proven & %Proof$3.proven;;
      Proof$1.subgoals =
              if %Proof$1.appropriate
              then Implies_elim (%Proof$1.sequent,
```

```
                                        %Proof$1.selected)
                    else [empty_sequent;empty_sequent];;
                Proof$2.sequent = hd(%Proof$1.subgoals);;
                Proof$3.sequent = hd(tl %Proof$1.subgoals);;
                Proof$2.print_tree_depth =
                        if %Proof$2.set_ptd > ~1
                        then %Proof$2.set_ptd
                        else %Proof$1.print_tree_depth-1;;
                Proof$3.print_tree_depth =
                        if %Proof$3.set_ptd > ~1
                        then %Proof$3.set_ptd
                        else %Proof$1.print_tree_depth-1;;
                Proof$2.no_of_columns =
                        if %Proof$2.set_noc > ~1
                        then %Proof$2.set_noc
                        else subproof_width %Proof$1.no_of_columns;;
                Proof$3.no_of_columns =
                        if %Proof$3.set_noc > ~1
                        then %Proof$3.set_noc
                        else subproof_width %Proof$1.no_of_columns;;
                 Proof.set_ptd = ~1;;
                 Proof.set_noc = ~1;;
              ]


        | Not_Elim ( Proof )
                        { show ~A entails B ==> show A }
              [ Proof$1.selected = 1;;
                Proof$1.appropriate =
                        is_Not(antecedent (%Proof$1.sequent,
                                          %Proof$1.selected))
                              ?? ["antecedent"] false;;
                Proof$1.proven =
                        %Proof$1.appropriate & %Proof$2.proven;;
                Proof$2.sequent =
                        if %Proof$1.appropriate
```

```
                then hd(Not_elim (%Proof$1.sequent,
                                  %Proof$1.selected))
                else empty_sequent;;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                if %Proof$2.set_noc > ~1
                then %Proof$2.set_noc
                else subproof_width %Proof$1.no_of_columns;;
         Proof.subgoals = nil;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
      ]


| All_Elim ( Term Proof )
              { show !xP(x) entails C
                  ==> show P(Term) entails C for any Term }
      [ Proof$1.selected = 1;;
        Proof$1.appropriate =
                is_ForAll(antecedent (%Proof$1.sequent,
                                      %Proof$1.selected))
                         ?? ["antecedent"] false;;
        Proof$1.proven =
                %Proof$1.appropriate & %Proof$2.proven;;
        Proof$2.sequent =
                if %Proof$1.appropriate
                then hd(All_elim (%Proof$1.sequent,
                                  %Proof$1.selected,
                                  %Term.self))
                else empty_sequent;;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
```

```
            else %Proof$1.print_tree_depth-1;;
    Proof$2.no_of_columns =
            if %Proof$2.set_noc > ~1
            then %Proof$2.set_noc
            else subproof_width %Proof$1.no_of_columns;;
    Term.no_of_columns =
            if %Term.set_noc > ~1
            then %Term.set_noc
            else %Proof$1.no_of_columns - 5
                    - (size "All Elimination");;
     Proof.subgoals = nil;;
     Proof.set_ptd = ~1;;
     Proof.set_noc = ~1;;
   ]


| Exists_Elim ( Var Proof )
              { show ?xP(x) entails A
                ==> show P(Var) entails A
                    + Var not free in goal }
   inML
       import Symbol_Table;;
   end
   [ Proof$1.selected = 1;;
     Proof$1.appropriate =
             is_unique_identifier(%Proof$1.sequent,
                                  %Var.self)
           & (not (is_global_constant %Var.self))
           & (is_ThereExists(antecedent(%Proof$1.sequent,
                                        %Proof$1.selected))
                          ?? ["antecedent"] false);;
     Proof$1.proven =
             %Proof$1.appropriate & %Proof$2.proven;;
     Proof$2.sequent =
             if %Proof$1.appropriate
              & is_unique_identifier(%Proof$1.sequent,
```

```
                                             %Var.self)
                then hd(Exists_elim (%Proof$1.sequent,
                                     %Proof$1.selected,
                                     %Var.self))
                else empty_sequent;;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                if %Proof$2.set_noc > ~1
                then %Proof$2.set_noc
                else subproof_width %Proof$1.no_of_columns;;
        Var.no_of_columns =
                if %Var.set_noc > ~1
                then %Var.set_noc
                else %Proof$1.no_of_columns - 5
                         - (size "Exists Elimination");;
         Proof.subgoals = nil;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
      ]


  |  Remove_Antecedent ( Proof )
                { show A entails B ==>
                  show B (for tidying up!) }
       [ Proof$1.selected = 1;;
         Proof$1.appropriate =
                let val test=antecedent(%Proof$1.sequent,
                                        %Proof$1.selected)
                in
                  true
                end
                ?? ["antecedent"] false;;
         Proof$1.proven = %Proof$2.proven;;
```

```
        Proof$2.sequent =
                if %Proof$1.appropriate
                then remove_antecedent (%Proof$1.sequent,
                                        %Proof$1.selected)
                else empty_sequent;;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                if %Proof$2.set_noc > ~1
                then %Proof$2.set_noc
                else subproof_width %Proof$1.no_of_columns;;
         Proof.subgoals = nil;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
      ]


|  Recall (Fact GenTermList GenFormList Proof Proof)
             { Match the named fact against the current
               goal. See ~/gforms/Facts_Matching.ml for
               details }
   inML
     import Generic_Formulae;
         { Now we need to import Recall_Prelims again, in
           order to make the desired definition of snd
           visible
         }
     import Recall_Prelims;;
   end
  [ Proof$1.subgoals
      = let val substs
              = give_priority_to( %Fact.autosubsts,
                {over} (some(%GenTermList.substs
                             @ %GenFormList.substs)));
```

```
            val (GF,GT) = unzip_substs substs
                    { give_priority_to S1 S2 will remove
                      those substitutions in S2 which occur
                      in S1
                    }
        in
           fact_subgoals(Instantiate_Generics(%Fact.sequent,
                                              GF,GT),
                        %Proof$1.sequent)
        end {of let..in};;
        { the subgoals will be
            show premises,fact-conclusion entails conclusion
            show premises entails AND(Fact-premises)
          The latter goal comes second so that we can easily
          omit it from the display if the fact has no
          premises.
        }
Proof$2.sequent = hd %Proof$1.subgoals;;
Proof$3.sequent = hd(tl %Proof$1.subgoals);;
GenTermList.still_gents =
    undetermined_terms %Fact.gents %Fact.autosubsts ;;
GenFormList.still_genfs =
    undetermined_formulae %Fact.genfs %Fact.autosubsts ;;
Fact.autosubsts =
        if %Proof$1.selected < ~1
        then none
        else if %Proof$1.selected = ~1
             then match_formulae(%Fact.genfs,%Fact.gents)
                       (succedent(%Fact.sequent),
                        succedent(%Proof$1.sequent))
             else match_formulae(%Fact.genfs,%Fact.gents)
                       (antecedent(%Fact.sequent,
                                   %Fact.selected),
                        antecedent(%Proof$1.sequent,
                                   %Proof$1.selected))
```

```
                    ??["antecedent"] none;;
          { Thus we match the conclusion of the fact
            against the goal conclusion, unless otherwise
            informed, in which case we match a particular
            premise in the goal to a particular fact-
            premise: both of these are determined outside
            the grammar.
              NOTE: We should still allow old-fashioned
            facts matching, where we don't want to perform
            any auto-substitution.  We will probably
            represent this by setting Fact.selected to ~2.
              NOTE 2: autosubsts is set to none should
            either call of antecedent (goal or fact) fail.
          }
Fact.genfs = map (avoid_parameter_clash_in_formula
                    (all_vars_of_sequent %Proof$1.sequent))
                   (snd %Fact.recall);;
          { Thus for example, a generic phi(x) becomes
            phi(x') if the variable x occurs in the goal }
Fact.gents = map (avoid_parameter_clash_in_term
                    (all_vars_of_sequent %Proof$1.sequent))
                   (third %Fact.recall);;
Proof$1.selected = ~2;;
         { This setting ensures that Autosubsts is none
           by default.
         }
Proof$1.appropriate = %Fact.valid;;
         { I suppose we should also check that if we are
           matching premises, then both selected premises
           should actually exist.
         }
Proof$1.proven = %Proof$1.appropriate
                   & %Proof$2.proven
                   & Proof$3.proven;;
 GenTermList.print_depth =
```

```
                %Proof$1.print_tree_depth - 1;;
        GenFormList.print_depth =
                %Proof$1.print_tree_depth - 1;;
        GenTermList.no_of_columns =
                if %GenTermList.set_noc > ~1
                then %GenTermList.set_noc
                else %Proof$1.no_of_columns;;
        GenFormList.no_of_columns =
                if %GenFormList.set_noc > ~1
                then %GenFormList.set_noc
                else %Proof$1.no_of_columns;;
      Proof$1.set_ptd = ~1;;
      Proof$1.set_noc = ~1;;
      Proof$2.print_tree_depth =
              if %Proof$2.set_ptd > ~1
              then %Proof$2.set_ptd
              else %Proof$1.print_tree_depth-1;;
      Proof$2.no_of_columns =
              if %Proof$2.set_noc > ~1
              then %Proof$2.set_noc
              else subproof_width %Proof$1.no_of_columns;;
      Proof$3.print_tree_depth =
              if %Proof$3.set_ptd > ~1
              then %Proof$3.set_ptd
              else %Proof$1.print_tree_depth-1;;
      Proof$3.no_of_columns =
              if %Proof$3.set_noc > ~1
              then %Proof$3.set_noc
              else subproof_width %Proof$1.no_of_columns;;
     ]

  | Duplicate_Antecedent ( Proof )
                { show A entails B ==> show A,A entails B }
       [ Proof$1.selected = 1;;
         Proof$1.appropriate =
```

```
                let val test=antecedent(%Proof$1.sequent,
                                        %Proof$1.selected)
                in true end
                ?? ["antecedent"] false;;
        Proof$1.proven = %Proof$2.proven;;
        Proof$2.sequent =
                if %Proof$1.appropriate
                then duplicate_antecedent (%Proof$1.sequent,
                                           %Proof$1.selected)
                else empty_sequent;;
        Proof$2.print_tree_depth =
                if %Proof$2.set_ptd > ~1
                then %Proof$2.set_ptd
                else %Proof$1.print_tree_depth-1;;
        Proof$2.no_of_columns =
                if %Proof$2.set_noc > ~1
                then %Proof$2.set_noc
                else subproof_width %Proof$1.no_of_columns;;
         Proof.subgoals = nil;;
         Proof.set_ptd = ~1;;
         Proof.set_noc = ~1;;
       ];


Term ::= Term()
         [ Term.self = variable(NewTermName());;
                { Initially "TERM_n", this can be
                  altered by the user }
           Term.set_noc = ~1;;
         ];

Formula ::= Formula()
         [ Formula.self =  atomic("FORMULA",[]);;
           Formula.set_noc = ~1;;
         ];
```

```
Var ::= Var()
         [ Var.self = "VAR";;
           Var.set_noc = ~1;;
         ];



Fact ::= A_Fact ()
           inML
             import Factic;
               { A version of tactics especially implemented to
                 allow the function Factic without dependency
                 upon the Proof_Grammar!
               }
             import Recall_Prelims;;
           end
           [ Fact.name = "FACT-NAME";;
                   { Altered by the user, either directly, or by
                     choosing from a menu of facts }
             Fact.recall =
                     if %Fact.name = "FACT-NAME"
                     then dummy_recall_option
                     else let val (f,fl,tl,aorl)
                                  = recall %Fact.name
                                    ?? ["unknown fact"] dummy_recall
                          in
                             case f of
                               atomic(fname,nil) .
                                    if fname = "dummy-fact"
                                    then factinfo(f,nil,nil,none,
                                                  "dummy-fact")
                                                 { no such fact }
                                    else factinfo(f,fl,tl,some(aorl),
                                                  %Fact.name)
                             | _ . factinfo(f,fl,tl,some(aorl),
```

```
                                    %Fact.name)
                        end {of let..in};;
    Fact.sequent = (Factic (fst(%Fact.recall)));;
            { The sequent corresponding to the fact,
              eg x=y&y=z->x=z ==> x=y,y=z entails x=z }
    Fact.selected = ~1;;
            { Which premise of the fact is to be used in
              the matching (~1 === conclusion)}
    Fact.valid = (fifth %Fact.recall) <> "dummy-fact";;
    Fact.fact = fst %Fact.recall;;
  ];



GenFormList ::= A_GenFormList ()
                { Those generic formulae which the user
                  has to supply }
           [ GenFormList.substs
                  = let val new_subst f
                          = form_subst(f,f);
                                { turns a term into a subst }
                        val new_substs
                              { Those generic formulae for which the
                                user has not supplied a
                                substitution }
                          = map new_subst
                                (undetermined_formulae
                                    %GenFormList.still_genfs
                                    (some %GenFormList.user_substs))
                    in
                         new_substs @ (%GenFormList.user_substs)
                    end {of let..in};;
             GenFormList.user_substs = nil;;
                 { user_substs will change as the user supllies new
                   formula substitutions.
                 }
```

```
                GenFormList.set_noc = ~1;;
              ];

   GenTermList ::= A_GenTermList ()
              [ GenTermList.substs
                      = let val new_subst f
                              = term_subst(f,f);
                                  { turns a term into a subst }
                            val new_substs
                                  { Those generic terms for which the
                                    user has not supplied a
                                    substitution }
                              = map new_subst
                                    (undetermined_terms
                                        %GenTermList.still_gents
                                        (some %GenTermList.user_substs))
                        in
                           new_substs @ (%GenTermList.user_substs)
                        end {of let..in};;
                GenTermList.user_substs = nil;;
                    { user_substs will change as the user supllies new
                      term substitutions.
                    }
                GenTermList.set_noc = ~1;;
              ];;

end
```

Appendix B

The IPE User Manual

This appendix consists of a copy of the user manual assembled by the author for distribution with IPE Version 5. The only constraint used in formatting the manual was to fit it into 80 character columns, enabling it to be scanned through on a standard tty. Thus the contents page expresses the positions of sections as a percentage of the whole text. (There is a LaTeXversion, created at Edinburgh by Claire Jones, describing the XIPE, which draws upon material included here).

```
              Interactive Proof Editor -- Brief User Guide
              --------------------------------------------
```

Contents

```
--------                                                       (approx.%)
```


```
    Starting Off.................................................0
    Some Notation................................................7
    The Sun Mouse...............................................18
    The Keypad..................................................21
    Keyboard Commands...........................................29
    Mouse Menu Commands.........................................50
    The Text Editor.............................................55
    Using the Choosers..........................................60
    Using Facts.................................................70
    The Theory Database.........................................80
    Syntax of Formulae..........................................95
    Bugs.......................................................100
```


Starting Off (on the Sun console)

```
------------
    Read the file IPE.README for info on termcaps, etc.

    IF using the Sun console:
        Within the "suntools" window system, start a new shell window.
        Stretch it so that it covers almost the entire screen (this MUST
            be done before entering the IPE, as it cannot cope with
            changes in window size within a session).
    END_IF

    Ensure that the environment variable IPE_THEORIES is set to the
        path of the theories directory (or the desired
        theories directory, if there are more than one).
    Make this directory the current directory.
```

```
IF using Sun console
    Type "BIGipe5"
ELSE
    Type "ipe5"
ENDIF
```

```
Some Notation
-------------

    This document is quite free with its notation.  Some terms that will
remain more or less fixed are:

    "goal/premise/conclusion/conjecture":
            A goal is the immediate objective of a proof, consisting
            of a list of premises and a conclusion (all of which
            are 1st-order predicate calculus formulae in the IPE).
            The idea of top-down goal-directed proof is to take
            a goal and do something to it to break it into a set
            of (hopefully simpler) subgoals.  Rather than look
            deep into each formula in a goal and work magic, the
            IPE relies on good-old-fashioned structural decomposition;
            ie the formulae in the subgoals differ from those of the
            parent goal only by the loss of their outermost logical
            connective. An IPE goal looks like this:
                "show premises entails conclusion"
            (the show being omitted once the goal has been proven).
            On the other hand, by a conjecture here we will mean
            the initial formula that we are trying to prove. The
            idea is that, being the nouns in the language of proof,
            goals should not be available outside that realm.
            (Besides which, it makes some things easier).  Hence,
            the ultimate purpose of any IPE proof is to justify a
            single formula, which we can then turn into a lemma
            (meaning an IPE-lemma rather than a proof in itself).
```

"proof structure":

The IPE's internal representation of a proof. This is actually a dependency-labelled attributed derivation tree for an attribute grammar (just thought you might like to know!), and as goals are attributes the tree shape is independent of the goals, which is why it is often called a proof structure rather than just a proof. (But it does get called a proof sometimes too).

"pointed region/area/selection":

the node in the proof structure corresponding to the cursor position on the display. Also pointed premise/proof/node. Note that this is distinct from the notion of current selection.

"selected region/current selection":

The position of the IPE's internal "tree cursor"; ie the current node in the proof structure. In the active buffer this is normally highlit. Note that when a formula in a goal is highlit the current selection is actually the proof node whose goal holds the premise, but the IPE remembers that an individual premise is selected as well. Thus any operation that applies to a proof node will still apply whenever a formula of its goal is selected.

"active buffer":

The IE is a multi-buffer system, but only one buffer can be displayed and worked on at one time; this is called the active buffer. Most of the IPE's commands operate solely on the current buffer, some operate between the active buffer and one other, and only one operates on all the buffers at once (it lists them) - not counting Exit, of course.

```
The SUN Mouse (only available on Sun workstation console)
-------------

    Left button:   PICKs the region being pointed at by the mouse (PICK
                   simply changes the currently selected region).

    Middle button: IF pointing at a premise or conclusion in a goal,
                      performs the appropriate PROOF STEP;
                   IF pointing at a text-edit position (eg initial
                      conjecture, term/var in quantifier rule), performs
                      an ENTER DATA (text edit of that object)
                   Otherwise, Zooms In on the pointed area (this will
                      make it the "centre" for display generation)

    Right button:  Presents a menu of further commands.

The Keypad
----------

    If your terminal has a numeric keypad, then hopefully within the IPE
it will be bound to the following functions:


                   IPE       Keypad
                   ---       ------
                ----- ----- ----- ----
               |     |     |     |    |  ie Home        is Home
               |Home |     |     |    |     (aka Zoom to Root)
                ----- ----- ----- ----      Scroll Up   is keypad 9
               |Zoom |Zoom |Scrll|    |     Zoom In     is keypad 8
               |Out  | In  | Up  |    |     Zoom Out    is keypad 7
                ----- ----- ----- ----      Print       is keypad 6
               |Enter|Pick |Print|    |     Pick        is keypad 5
               |Data |     |     |    |     Enter Data  is keypad 4
                ----- ----- ----- ----      Scroll Down is keypad 3
               |     |Proof|Scrll|    |     Proof Step  is keypad 2
```

```
|Help |Step |Down |    |     Help       is keypad 1
 ----- ----- -----     |
|  Exit      |    |    |     Exit       is keypad 0
 ----------- ----- ----
```

If this does not seem to be the case, try any other function keys; if that doesn't work, then typing ESC then n instead of keypad n should work!

On the Sun console, four of the keypad keys are used as arrow keys, there is no Home key and no keypad zero. Furthermore, the keypad is upside-down with respect to the above! As a result, some of the keypad commands are not available as such on the console; however they are all implemented either in the mouse menu or as an individual mouse button, so this is no great loss.

Proof Step is the main operation of the IPE. By pointing to a premise or conjecture in a goal and hitting the Proof Step button (or just by pointing with the mouse and clicking the middle button), the user can expand the proof at that point by a rule appropriate to the "active" operator of the formula. (For example, Proof Stepping on A&C in "show A|B,C entails A&C" will expand the proof at that point by an And Introduction rule, producing

```
show (A|B),C entails A&C
use And Introduction
and show (A|B),C entails A
and A|B,C entails C
    is immediate
```

(ie, generating two subproofs, one of which follows immediately). If Proof Step is applied by pointing to the "entails" part of the goal display, then an Immediate rule is applied. This differs from the (default) rule RTP? in that it is an error for the goal not to be immediate.

Enter Data is used when the user wishes to change a text-edit

point, that is, an area on the screen enclosed in angle brackets ("<...>"). Hitting this button whilst pointing within such an area (or pointing with the mouse and pressing the middle button) will place the user in a text editor (described below). Once editing is complete, an appropriate parser for the class of object (Formula, Term, Identifier) is applied to test the text; the editor is not exited until the user supplies a parseable expression or aborts the edit.

The Pick command is used to make the pointed region the current selection; this corresponds to the left mouse button. Although commands such as Proof Step and Enter Data automatically perform a Pick before acting, others do not (the buffer application/copying commands, read from file, weaken/duplicate) and require an explicit Pick beforehand. (On the Sun console, this problem vanishes because of the mouse).

Scroll Up and Scroll Down move the screen-sized window over the entire proof display; when preceded by a number they scroll by that many lines, defaulting to 10 lines. Zoom In makes the pointed area the current region and forces a regeneration of the display (this can be useful in some buggy situations), whilst Zoom Out makes the nth parent of the pointed node the current region, where n is typed beforehand and defaults to 1. This command is useful for moving back up the tree to parts no longer on display.

Print appends a printout of the current proof to the file IPE.out. The style of printed output is very different; the proof is presented in bottom-up fashion using a compact notation whereby the original introduction and elimination derivation rules are used to construct premises from assumptions, axioms, lemmas and premises derived earlier. Unlike the interactive display style this method attempts to minimise the repetition of premises within the same scope in a proof. Unfortunately, the proof printing is still incorrect, in that correct proofs are sometimes printed wrongly and marked as unproven.

Keyboard commands

These commands are simply typed on the keyboard; all are single-letter commands (no RETURN required), although some may make use of a prefixed argument count.

The actions of the following will only be described briefly here; a better idea of their operation might be had from the demonstration scripts.

A - Apply Buffer. Asks for buffer name and applies current selection of that buffer to the current selection of the active buffer. As in all proof expansion operations, the original subtree in the active buffer is lost, although it can normally be yanked back.

B - Change to Buffer. Asks for the name of a buffer and makes it the active buffer, creating it as a buffer rooted on Theorem if it does not exist. The current position of the newly-active buffer is restored, although autoprove and automove may change this if in effect.

C - Copy to Buffer. Asks for a buffer name and copies the the current selection of the active buffer to that of the named buffer. If the named buffer's current selection is its root and the types of the two selections do not match (eg Proof vs Theorem) then the contents of the buffer are completely overwritten (normally a mismatch causes an error). If the named buffer does not exist then it is created with the active selection as its root.

d - Duplicate a premise. Makes a second copy of the pointed premise in a goal. This is necessary for some proofs as elimination rules always remove their arguments from the premise list. This is a proof expansion operation, so that any current subproof is lost. Normally, it can be restored using Yank, but it is safer to copy the subproof to another buffer before Duplicating.

D - Delete Buffer. The Chooser lists all buffers other than the current buffer. More than one may be selected for deletion;

upon acceptance all selected buffers are deleted. As buffers take up quite a lot of storage this is a useful operation.

H - Show other help. Toggles between the two IPE help windows. This can sometimes get out of step, particularly after text-edit operations, so that it occasionally has to be hit twice before it works.

L - Save Lemma. Zooms to the top of the proof (the active buffer must be rooted on Theorem), checks it and if it is proven attempts to construct a lemma from it. The name of the theory in which it is to be stored, and a name for the lemma are asked for. The theory must exist (as a subdirectory of the current theories-directory), but must not contain a file with the same name as the lemma. BEWARE: lemma names must be valid IPE identifiers, but this is not checked by Save Lemma, which will accept any valid file name! The lemma is written to the named theory directory; however due to a discrepancy the proof is printed to lemma-name.proof not in the theory directory, but in the current directory.

M - Toggle automove mode. In this mode, after each proof-altering operation the IPE moves the current selection to the nearest proof node requiring work, where "nearest" is in a depth-first sense and nodes "requiring work" are either unexpanded or inappropriate rule applications. Setting automove mode resets autoprove mode.

P - Toggle autoprove mode. In this mode the IPE will repeatedly expand proofs after each alteration, so long as the goal of a proof has only one possible operator-expansion (ie so long as only one formula in the goal is non-atomic). Autoprove uses automove to repeatedly find nodes requiring work (see above). If the node is amenable, it is auto-expanded, otherwise the current selection is left there for the user to expand.

R - Read a proof structure from a file and replace the current subtree with it. A one-line edit window appears to allow the user to enter a file name, which is interpreted relative to the current directory. (For example, select the Conjecture/Theorem part of a buffer then type R followed by Example1 (note the case) and Return. This will load the first example into the buffer. Solution1 can be similarly read).

S - Save the proof structure of the current buffer in a file. Together with Read from File, this can be used to store partial proofs between IPE sessions.

T - Load a Theory. Asks for the name of a theory, and loads it. This means that the theory's .environment file is processed (included theories are recursively loaded, symbol declarations are instantiated), and that all the facts (axioms and lemmas) in the theory are now visible. The named theory must exist as a subdirectory of the current theories-directory. More than one theory may be loaded at top-level, but duplicate symbol declarations will cause loading to fail and corrupt the loaded declarations. Theories are searched for facts in a depth-first fashion through the tree of recursive loads; where more than one theory has been loaded at top-level the most recently loaded has precedence.

W - Weaken (remove a premise). Removes the pointed premise of a goal. The main use of this is simply to tidy up goals by removing premises not needed in the subproof. As with Duplicate, this is a proof expansion.

Y - Yank. Whenever a structure other than a blank Theorem or unexpanded proof is overwritten (by Proof Step, Apply Buffer, Recall, Duplicate or Weaken), it is stored in the Yank tree, and can be applied to the pointed selection. The yank tree is not buffer-specific: there is only one. It is not recommended that Yank be relied on, as the yank

tree is liable to change with great frequency; with forethought, saving trees in buffers should be used. The tree replaced by Yank is lost for good.

^C Break out of the IPE by generating an interrupt it cannot ignore, though actually it has been known to sometimes! Be careful: this doesn't ask for permission before terminating the entire session.

^L List Buffers. Presents a list of all the buffers in the present session, showing the types of their roots and current selections. Input following ^L deletes the buffer list window and returns to command level.

^P Dump the display text of the current selection to the file IPE.out in the current directory.

^R Redraw the screen. Useful for blotting out system messages and the like

^Z Suspend the IPE and return to UNIX. This is not possible on the Sun console-with-mouse version, which is why the latter should be run inside suntools.

0-9 Build up an argument count for a command. Relevant to Zoom Out, Scroll Up/Down, >,< and the arrow keys. The argument should precede the command, eg 25 then left arrow moves left 25 characters.

> - Display-above controller. On its own, increments the current value of display-above (the maximum number of nodes to be displayed above the current selection); preceded by an argument, sets display-above to that value. For example 0> indicates that none of the structure above the current selection is to be displayed. The change is effected immediately (ie display recalculation is forced).

< - Display-below controller. Similar to >, but there is a lower limit of 2 on its value, so that the sons of the current selection will always be visible.

Mouse Menu Commands (on Sun)

On the Sun workstation, the right mouse button presents a list of further commands. Most of these are duplicates of commands listed above; however the fact-using comamnds are not duplicated elsewhere. (Thus it is not possible to use facts on a mouseless terminal). The facts commands presented on the menu are:

```
Use Fact
Choose Matching Facts
Choose from Named Facts
Choose from All Facts
Choose Fact by Name
```

These are described in the "Using Facts" section.

A Demonstration

Let's suppose that we approach IPE wanting to prove the formula

```
if for all x, P(x) implies Q(x),
   and for some y P(y) holds
then for some z Q(z) holds
```

An intuitive proof of this is not too hard: let y be such that P(y) holds (as allowed by "for some y, P(y) holds"), then by the first statement we have that Q(y) holds, and so therefore Q(z) holds for some z (namely z=y). However, we want to use IPE to construct a machine-checked formal proof of this. Re-expressed in the IPE's syntax for logical expressions, the above becomes

```
!x (P(x) -> Q(x)) & (?y P(y)) -> (?z Q(z))
```

where "!x" is used for "for all x", "?y" is used for "for some y", "&" is "and" and "->" is "implies". (See the "Syntax of Formulae" section

towards the end). Once the IPE's title page has been dispensed with, the screen looks something like this:

```
Conjecture
----------
    <FORMULA>
Attempted Proof
---------------
show FORMULA
```

plus some other information at the bottom of the screen which we shall ignore for the moment. What we are looking at is the top of an IPE proof tree, which states our initial goal, and the (attempted) proof constructed thus far. This is the display of a tree structure; we can point at and select areas on the screen which correspond to nodes in the tree, so that when we select an area on the display and perform some action upon it, it is really the underlying tree structure that is being affected. Selecting a part of the tree is easy; using the mouse, simply point at its corresponding display (ie move the mouse until the mouse arrow lies over that region of the screen), and click (press then release) the LEFT button. The area on the screen representing that node of the tree will be highlit. For example, pressing the left button when the mouse arrow is anywhere on the word "Attempted" will cause the phrases, "Conjecture" and "Attempted Proof" to be highlit, together with their underlines. This shows us that we have selected the root node of the structure.

At the moment, we've not built up any proof at all, so the proof is just a single "unexpanded" leaf, "show FORMULA". What we have to do first of all is to replace FORMULA with our own formula, as given above. We position the mouse cursor over the copy of FORMULA which appears within the angle brackets (points on the display between angle brackets are known as "text-edit" points -- places where the user can alter information fed into the IPE) and this time press the MIDDLE mouse button (the "action" button). This causes the appearance of a

window with the text "FORMULA" in it. This is a simple text editor which we can use to manipulate pieces of text. (A full list of the operations available is given below). However, the header "Formula" informs us that whatever we type will be parsed as a formula when we quit the editor; if the parsing fails then we will have to re-edit the text until it succeeds or we abort the edit. For the moment, we will simply type Control-K (Control together with K) to delete the word "FORMULA" and type in our own formula as shown above. We then quit the editor by pressing (NOT clicking) the RIGHT mouse button (the "menu" button) and selecting the appropriate entry by dragging the cursor to it then releasing the button. Assuming we've typed the formula correctly, the display updates to

```
Conjecture
----------
   <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
```

Our new formula has been accepted, and passed down to the proof. Now we can start constructing the proof...

In the IPE, we build proofs in a goal-directed fashion: we take a problem and attempt to reduce it to one or more simpler problems by the application of some appropriate rule. A "goal" in the IPE has the general form

```
show premise-formula,... entails conclusion-formula
```

In the case above, there are no premises, so a shorter form is displayed.

There are a small set of built-in rules for "simplifying" goals, with two rules for each logical connective. One rule applies when the connective is the "topmost" connective in the conclusion, and the other

when it occurs similarly in a premise. To invoke a rule, we simply have to point at the formula we wish to "make use of" in simplifying our goal, and click the action button. (Note that it is important which instance of the formula we select on the display, as different instances will (usually) "belong to" different goals). Here we only have one goal, and only one formula instance which we can use to simplify the goal, so we point at it and press the Action button. The IPE notes that the topmost connective of the formula is an implication, and since it is the conclusion, IPE applies its "Implies Introduction" rule, updating the proof and the display to

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
```

(with the Implies Introduction text highlit). Thus we now have a simpler subgoal whereby we've assumed the LHS of the implication and have to demonstrate the RHS. Here we have a choice of actions: we could simplify the conclusion or the premise; we shall choose the premise. "Actioning" on this gives us

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
```

```
        use And Elimination on premise 1
        and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
```

This gives us two "smaller" premises, bringing further connectives "to the surface" for application of IPE rules. Suppose that we decide to work upon the "!x" premise: this gives us

```
    Conjecture
    ----------
        <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
    Attempted Proof
    ---------------
    show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
    use Implies Introduction
    and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
        use And Elimination on premise 1
        and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
            use All Elimination on premise 1 with <TERM_1>
            and show P(TERM_1)->Q(TERM_1),?yP(y) entails ?zQ(z)
```

The All Elimination rule chooses TERM_1 as a single instance of x, so that we can now assume P(TERM_1)->Q(TERM_1). Since TERM_1 appears in angle brackets (like the initial conjecture), we can change it to something else (after all, if we have "for all x P(x)" then we should be able to assume P holding for any term we like in place of x). So we can point at the TERM_1 in angle brackets and press the action button to get a text-edit window which we can use to supply a new term. (This time the window header informs us that a Term is expected). Here, if we simply replace TERM_1 by "a", the display resumes as

```
    Conjecture
    ----------
        <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
    Attempted Proof
```

```
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use All Elimination on premise 1 with <a>
        and show P(a)->Q(a),?yP(y) entails ?zQ(z)
```

with our new instance of the universally quantified premise in place. If we now choose to simplify this goal using ?yP(y), we get

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use All Elimination on premise 1 with <a>
        and show P(a)->Q(a),?yP(y) entails ?zQ(z)
            use Exists Elimination on premise 2 with <y>
             and show P(a)->Q(a),P(y) entails ?zQ(z)
```

Again, the IPE has chosen a name for us; here it simply used the name that was already there when it stripped of the existential quantifier. However, "y" is not what we wanted, so we edit it to "a":

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
```

```
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use All Elimination on premise 1 with <a>
        and show P(a)->Q(a),?yP(y) entails ?zQ(z)
            use Exists Elimination on premise 2 with <a>
                                              ^--- non-unique identifier!
```

We have made a mistake! When we know that P(y) holds for some y, we do not know for which y it does hold; we cannot assume that P holds for any of the variables or terms already occurring in the goal. The IPE "eliminates" the existential quantifier by first choosing some variable name which doesn't occur free (ie ignoring variables which are "bound" by some quantification) in the goal. The user is free to change the name to something that looks more meaningful or prettier, but the IPE will check that no variable of that name appears freely in the goal. Referring back to our earlier informal proof, we see that our mistake was to eliminate the universal quantifier too soon; the existential quantifier should have been dealt with first.

This is not difficult to remedy in IPE, because we are not committed to a proof step when we make it. We can go back to any point in the proof and perform any alternative (applicable) rule. Here, we need to replace the All Elimination step with an Exists Elimination: we do this by pointing at the instance of ?yP(y) in that goal and pressing the action button, giving

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
```

```
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use Exists Elimination on premise 2 with <y>
        and show !x(P(x)->Q(x)),P(y) entails ?zQ(z)
```

The two steps below the And Elimination have been replaced by this single step. (Note: the original two steps have not been lost forever (yet); they are squirreled away but can be brought back and applied to any point in the proof, or saved in another buffer, or applied to a different proof in another buffer. However, that's getting a little ahead of things...)

This time we can safely replace "y" by "a", as "a" doesn't occur freely (or even at all) in the goal. We don't have to do this, since "y" will do quite well, but maybe we believe that "a" is a better name. Again we Action on the "y" in <y> and text-edit it to an "a"...

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use Exists Elimination on premise 2 with <a>
        and show !x(P(x)->Q(x)),P(a) entails ?zQ(z)
```

Now we can instantiate the universal formula as previously; Action

on !x(P(x)->Q(x)), then editing <TERM_2> to <a> yields

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use Exists Elimination on premise 2 with <a>
        and show !x(P(x)->Q(x)),P(a) entails ?zQ(z)
            use All Elimination on premise 1 with <a>
            and show P(a)->Q(a),P(a) entails ?zQ(z)
```

Let's work on the implication; Action on it gives

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use Exists Elimination on premise 2 with <a>
        and show !x(P(x)->Q(x)),P(a) entails ?zQ(z)
            use All Elimination on premise 1 with <a>
            and show P(a)->Q(a),P(a) entails ?zQ(z)
                use Implies Elimination on premise 1
```

```
                  and P(a) entails P(a)
                      is immediate
                  and show Q(a),P(a) entails ?zQ(z)
```

If the Implies Elimination rule looks a little confusing, then it might help to see it used on an unprovable goal, "show A->B,C entails D":

```
show A->B,C entails D
use Implies Elimination on premise 1
and show C entails A
and show B,C entails D
```

The two subgoals of Implies Elimination are:

1) Show that the left-hand side of the implication (A) can be derived from the other premises (C);
2) Assume the right-hand side (B) as a new premise,

and we have to prove both before Implies Elimination is satisfied; in other words, we can only assume B if we can derive A from the other premises.

In our example, the left-hand side, P(a), already occurs as a premise; as a result, the first subgoal is immediate, since the conclusion also occurs as a premise. Thus we have completed a subproof, for the first time. Note that the word "show" vanishes, since the goal has now been shown.

It is still required of us to demonstrate the second goal. Action on the conclusion gives

```
Conjecture
----------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Attempted Proof
```

```
---------------
show !x(P(x)->Q(x))&?yP(y)->?zQ(z)
use Implies Introduction
and show !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    use And Elimination on premise 1
    and show !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        use Exists Elimination on premise 2 with <a>
        and show !x(P(x)->Q(x)),P(a) entails ?zQ(z)
            use All Elimination on premise 1 with <a>
            and show P(a)->Q(a),P(a) entails ?zQ(z)
                use Implies Elimination on premise 1
                and P(a) entails P(a)
                    is immediate
                and show Q(a),P(a) entails ?zQ(z)
                    use Exists Introduction with <TERM_3>
                        and show Q(a),P(a) entails Q(TERM_3)
```

Exists Introduction is very similar to All Elimination: in the latter we choose any term as an instance of the quantified variable; in the former we can choose any term "t" for which we believe that we can demonstrate Q(t). In this case, the obvious choice is "a". Editing TERM_3 to "a" gives

```
Theorem
-------
    <!x(P(x)->Q(x))&?yP(y)->?zQ(z)>
Proof
-----
!x(P(x)->Q(x))&?yP(y)->?zQ(z)
by Implies Introduction
and !x(P(x)->Q(x))&?yP(y) entails ?zQ(z)
    by And Elimination on premise 1
    and !x(P(x)->Q(x)),?yP(y) entails ?zQ(z)
        by Exists Elimination on premise 2 with <a>
```

```
          and !x(P(x)->Q(x)),P(a) entails ?zQ(z)
              by All Elimination on premise 1 with <a>
              and P(a)->Q(a),P(a) entails ?zQ(z)
                  by Implies Elimination on premise 1
                  and P(a) entails P(a)
                      is immediate
                  and Q(a),P(a) entails ?zQ(z)
                      by Exists Introduction with <a>
                          and Q(a),P(a) entails Q(a)
  QED
  ---
```

Since Q(a) appears on both sides, we have produced a trivial (and hence proven) goal. The Exists Elimination rule recognises that this means that its goal (Q(a),P(a) entails ?zQ(z)) has also been proven; the Implies Elimination sees that both its subgoals are now proven,... and so on upwards until it transpires that our original goal and thus our original conjecture has been proven. When each rule realises that its goal has been demonstrated, its display alters so that it now says, "goal by me and subgoals" instead of, "show goal using me and subgoals"; and the root of the proof now displays the initial formula as a "Theorem" and the proof as a "Proof" rather than as a "Conjecture" and an "Attempted Proof"...plus a throwaway "QED".

So we have completed our first proof using the IPE. This demonstrates most of the basic operations IPE provides for proving theorems of "bare" first-order intuitionistic predicate calculus.

The Text Editor

This is used to edit formulae, terms, identifiers and some user responses. Text can be entered as expected, and is used by the IPE only when the editor is exited (using the appropriate selection from

the mouse menu/keypad).

In this mode, the left & middle mouse buttons move the text cursor (as do the arrow keys), whilst the right button again presents a menu of commands (including EXIT). As in the main IPE, keypad 1 toggles a help display showing what else is available on the keypad.

Further text-edit commands are:

```
Ctrl-A : beginning of line
Ctrl-E : end of line
Ctrl-P : previous line
Ctrl-N : next line
Ctrl-F : forwards 1 character
Ctrl-B : backwards "" ""
Ctrl-D : delete next character
Ctrl-K : delete to end of line
Ctrl-C : abort text edit (asks for confirmation)
```

A single-line version of this editor is used to read user input in some commands: in this case, hitting the RETURN key is equivalent to EXIT.

Using the Choosers

Certain commands of the IPE involve the use of an interaction package called a Chooser. In the IPE, three similar sorts are used: the plain Chooser, the MiniChooser and the EditOrChooser. Essentially each of these presents a window with a list of options; the user can then point at these options to select them, usually by clicking a mouse button when pointing at an item. What happens next depends upon the sort of Chooser.

The plain Chooser presents an options window plus several buttons. The left and middle mouse buttons can be used to select an item in the options window, by pointing and clicking (which highlights the

item). If only one item can be chosen at a time, then selecting an item deselects any other selection; otherwise the item is additionally selected (and can be unselected by clicking it again).

The right mouse button presents a menu of options as follows:

Accept	Accept the current selection(s) and exit the Chooser This is only applicable when there is a selection.
Prev	Show the previous page of items. This only applies when there is a previous page
Next	Show the next page of items (similar to Prev)
More	Add a new item to the list of items. This is mainly used whenever it would be very expensive to generate the entire list of items prior to the user making a choice (eg in Facts-Matching). It only applies when more items can be generated.
Cancel	Exit the chooser without accepting any selection

To the right of the options window is a stack of "button windows" corresponding to the menu options; however whenever a command is not applicable, its button is invisible.

A command may be invoked either by selecting the appropriate entry from the mouse menu or by clicking with the left or middle buttons on the relevant button.

The MiniChooser presents an options window. The only possible operations are the acceptance of a single item (exiting the MiniChooser) or cancellation. Clicking any mouse button on an item selects that item and leaves the MiniChooser; clicking outside the options window cancels the MiniChooser. This is used when the number of items is small and fixed, eg in command menus or when an operation involves a buffer that must exist.

The EditOrChooser is similar to the plain Chooser except that it also offers an edit window within which the user can enter an option of their own. To enter the edit window, click the left or middle button on it; this turns off any current selection in the options window, and any text subsequently typed goes into the edit window and is considered the current selection. Leaving the edit window (by clicking the mouse elsewhere) delselects it. The EditOrChooser is mainly used in buffer operations to allow the user to create a new buffer for the operation or to simply use an old one.

If there are no current objects to choose from, then a one-line edit window is presented instead.

Using Facts

The IPE is designed to work with a database of axiomatic theories. Each theory contains a number of facts (axioms and lemmas) which can be used in IPE proofs. An IPE proof can be converted into a lemma which is then stored in a particular theory for later use. A theory may also declare certain symbols as 'special', so that they have some particular meaning within that theory and all of its dependent theories.

Commands:

Load Theory

This presents a list of possible theories to load, using the MiniChooser. When a theory is selected, its axioms and lemmas are rendered visible to the IPE, and its special declarations are activated. Any theories upon which the named theory depends are loaded first.

Recall Template

Expands the current tree position (which must be a Proof node or a goal therein) by the blank "Recall fact" template. The user can then Text-Edit the "FACT-NAME" component to the name of some

existing fact. (In practice, the command Choose Fact By Name is probably better).

Choose From Matching Facts

Expects the current position to be a formula within a goal. Given this formula, it is matched against facts in the theory database. Each fact which matches with a valid substitution of its free variables is displayed in a Chooser window. (The search is only performed when requested by the user via More to generate new items). When a paritcular match is chosen, the current proof node is expanded by a Recall Fact rule with the appropriate substitutions (if any) automatically performed by the IPE.

Note that as the matcher is only operating on a single formula it may not necessarily perform all the subsitutions needed to instantiate the axiom; the remainder will then have to be supplied by the user, by Enter Data (or middle mouse button) on the text-edit points corresponding to the unsubstituted terms/formulae. Also, as the matching algorithm only handles first-order cases, higher order generics will not be matched.

Choose From All Facts

is very similar to the above, except that it also shows those facts for which no successful match with the selected formula was obtained. This allows the use of higher-order facts.

Choose from Named Facts

first presents a Chooser list of all the visible facts. The user can select one or more of these; matching is then performed only upon the chosen facts. This is useful when the user knows roughly which facts will be useful for the current goal, and avoids trudging through lots of irrelevant facts and their matchings.

Choose Fact By Name

Presents a Chooser list of all the visible fact names; upon

selection of a fact, no automatic substitution is performed.

The Theory Database

This section describes the innards of the database and how to make your own theories. (Please note the addenda at the end of this section).

The IPE theory database allows the extension of what is essentially an editor aimed at purely propositional proofs with uninterpreted symbols into a system allowing the construction of hierarchies of axiomatic theories. The theories present are Equality, Peano, List and ListOps, each containing a small number of axioms and a growing number of IPE-generated lemmas.

An IPE theory is a UNIX directory containing a .environment file plus axiom and lemma files (possibly with proof printouts for the lemma files). A single collection of theories consists of a directory containing theories. No theory in a collection may refer to a theory elsewhere (although setting a soft link of the same name in the theories-directory would work).

The .environment file tells the IPE which other theories this theory depends upon, and contains declarations of special predicate, function and constant symbols. As an example, the environment of the theory List is

```
includes Equality
predicate null(x)
constant nil
function cons(a,l)
```

This shows that List depends upon Equality, and declares the symbol null to be a unary predicate, nil to be a constant term and cons to be a 2-place function (term expression).

Any included theories in an environment must precede any symbol declarations; the latter must each appear on a seperate line, although

more than one theory may be mentioned in a single includes (which may spread over several lines). The .environment may be empty, but it must exist.

When the IPE loads a theory, it begins by processing the .environment. Any included theories are recursively loaded first (it is possible for a theory to include itself, but the IPE avoids loading the same theory twice and marks a theory as loaded before processing the environment). Then each symbol declaration is processed. The motivation behind symbol declarations follows.

In making a lemma from a theorem such as A&B&C->(A&C), the IPE tries to generalise it as much as possible, attempting to capture some of the "reusability" of the original proof structure. Here for example, it is clear that the proof structure would also prove E&F&G->(E&G) or even (!xP(x)) & (a|b) & (c->d) -> ((!xP(x)) & (c->d)): in short, we could replace A,B and C by any formula and the proof structure would still work. So when the IPE saves such a lemma, it saves it as a formula schema with substitutable parameters (called generics). When the lemma is used, the user is allowed to substitute any formula for these generics (generic terms are also possible) to create a lemma instance.

If the lemma created from the above theorem were called Example, then it would look like this:

```
lemma Example is
    A&B&C->(A&C)
generic formulae A
             and B
             and C
```

and would exist as a file to itself in some IPE theory directory. When the FACT_NAME in a Recall Fact rule is changed, the IPE searches through its loaded theories (in depth first order of loading, includer before the included) until it finds a file of that name. Upon finding it, it reads it in and converts the information above into an "editable premise" in the proof.

IPE axioms are simply lemmas that are taken for granted, and have not been proven using the IPE. The sole difference in appearance is in the word "axiom" replacing the word "lemma". The IPE does not care whether or not a fact is an axiom or lemma, and the user needn't know either.

As an example of the use of axioms, consider the "definition" of the length of a list (from ListOps):

```
axiom Length1 is
    length(nil)=0

axiom Length2 is
    length(cons(a,L))=(S(0) + length(L))
generic terms a
        and L
```

Induction is also defined axiomatically:

```
axiom ListInduction is
    phi(nil) & (!L(phi(L)->!a phi(cons(a,L)))) -> !L phi(L)
generic formulae phi(L)
```

The main problem with axioms is that they are user-created. Not only is it possible to create inconsistent theories, it is also possible to construct axioms with bad syntax. At present, the IPE only remembers that a symbol has been declared; it does not check the arity of further occurences. So for example, the "cons" in "cons(nil)" will be considered valid and special even although in reality it should be on or the other but not both. Once the full power of declared-symbol checking is put into operation, such cases will be checked and sent one way or the other.

Suppose we construct a proof of "length(cons(a,nil))=S(0)". When we come to make a lemma from it, it is important that the IPE should know which symbols can be substituted for without destroying the essential

meaning. Having used axioms which define properties for the special symbols "length", "cons","nil","S" and "0", it would not be correct to allow these symbols to be replaced by other terms to whom these properties do not apply! By informing the IPE that these symbols are special, we avoid this problem altogether. However, we now introduce the problem of accidentally giving some arbitrary symbol in a conjecture the same name as a declared symbol, proving it without using any properties of the symbol and then saving it only to discover that the saved lemma is too restrictive. Far worse would be the case when an axiom is similarly over-restricted. Enforcing the naming of the generics in axioms solves the latter problem; to solve the former requires keeping a tally of all the facts used in a proof and what symbols they define properties for, which the IPE does not do at present.

To sum up: to create your own theory you need:

1) a file .environment, whose most general form is

```
includes Theory1 Theory2
         Theory3
 constant blah
 function f(place1,place2,...,place_n)
 predicate s(place1,...,place_n)
 function g(x)
 predicate C
```

Notes: the file may be empty, but must exist; blah,f,s,place_i,g,x and C can be any valid IPE identifier, but blah,f,s,g and C must not be already declared when this theory is loaded (or occur more than once in the evironement, including as place markers).

2) A set of axiom files; each file must have the same name as the

axiom and have the following general structure:

```
axiom AxiomName is
    FORMULA
generic terms t1
          and t2(x,y,z)
generic formulae phi(x)
             and A
```

Notes:
AxiomName must be a valid IPE identifier;
FORMULA must be a valid IPE formula;
no more than one generic to a line; use repetition with and as above; the generic terms (if any) must precede the generic formulae (if any); the generics must make sense with respect to the formula! (ie, if G is a generic then it must have the same arity throughout the formula, and should ideally not occur as anything else (eg as a predicate instead of/as well as a term).

To create your own theories directory as a subdirectory of some_dir:

```
(in CShell)
cd some_dir                        # move to desired parent dir.
mkdir my_theories                  # make a new subdirectory
cd my_theories                     # move to new directory
mkdir My-Very-Own-Theory           # Creating new theories
  ...(etc)...
ln -s $IPE_THEORIES/Peano Peano    # Links to existing theories
  ...(etc)...
setenv IPE_THEORIES some_dir/my_theories
                                   # so the IPE looks here
```

Now fill My-Very-Own-Theory with .environment and axioms.

ADDENDA: The above was written prior to the construction of IPE5 with facts-matching. Now each theory requires an additional file, ".facts", which will contain the names of all of the facts in that theory. Normally, the IPE will add new lemmas to this file, but the axioms must be put there by the creator of the theory. As with .environment, this file must exist and contain no blank lines.

Secondly, the theories directory itself requires a file ".theories", which lists the names of all the theories therein.

Syntax of Formulae - some examples

The syntax for IPE formulae is a little strange, in that there is no precedence of infix operators; instead, all expressions associate to the left. Thus "A&B&C->A&C" means "(A&B&C->A)&C", and not "A&B&C->(A&C)" as one might expect.

```
"A & B" means A and B
"A | B" means A or B
"A -> B" means A implies B
"~A" means not A
"!x p(x)" means for all x p(x)
"?x p(x)" means there exists x such that p(x)
```

Examples:

```
(A->B)&(B->C)->(A->C)

!x(P(x)->Q(x)) & ?x P(x) -> ?x Q(x)

length(cons(a,nil))=S(0)
```

BUGGINESS

There are several bugs; some have semi-known causes and will be fixed "soon"; others occur in ill-determined circumstances or refuse to occur twice under (seemingly) the same conditions. All ML-generated escapes not handled within the IPE will be trapped at the very top and generate an "Oops" message. Some of these are benign, in that they can be ignored and will not destroy anything. Others are more serious - errors during attribute reevaluation can leave the proof structure in a half-evaluated state, so that further operations generate "Oops: circularity" messages. (This means that the dependency tree reevaluator believes that it has detected a semantic loop). In this case nothing more can be done than to abandon the proof and start again in a new buffer. If the text editor crashes for some reason it can leave its window on-screen permanently, in which case the entire session has to be abandoned through illegibility.

Quite a serious (and recently noticed) bug is that when the screen is scrolled, the positioning by mouse still acts as though the screen was at the top of the display.

Generally, the best thing to do if an error message comes up is to try and carry on, perhaps by doing something else (this is where Zoom In can be useful), and only give up when a message repeats itself or things look completely crazy.