User interfaces for theorem provers : informal proceedings of the workshop, Eindhoven University of Technology, 13-15 July 1998

Eindhoven University of Technology
Department of Mathematics and Computing Science

Informal proceedings of the Workshop on

User Interfaces
for
Theorem Provers

Eindhoven University of Technology
13 - 15 July 1998

edited by R.C. Backhouse

98/08

ISSN 0926-4515

editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 98/08
Eindhoven, July 1998

Informal proceedings of the Workshop on

User Interfaces
for
Theorem Provers

Eindhoven University of Technology
13-15 July 1998

edited by R.C. Backhouse

98--08

Workshop on

User Interfaces
for
Theorem Provers

Eindhoven University of Technology
13 – 15 July 1998

Informal Proceedings

Foreword

Thirty years ago, at the time that N.G. de Bruijn began the Automath project here at Eindhoven University of Technology, computer technology was far less advanced than it is now. It was still not unknown, for example, for computer programs to be prepared on paper tape, punch cards being the preserve of the more affluent establishments. It is not surprising, therefore, that the early computer systems were for experts only, and systems offering support for De Bruijn's dream of automating mathematics paid no attention to the interface with the human user.

The technological advances that have been made since then are mind-boggling. Most recently, the technology of human-computer interaction (HCI, for short) has progressed in leaps and bounds, bringing everyday computer usage at long last to the man on the street. Systems for automating mathematics have also made substantial progress, but the interfaces with the user have not kept pace and are most often still based on teletype technology. Thanks to the Internet, the world may be at your feet, but ergonomic interaction with automated mathematics is still way up in the clouds!

The workshop on User Interfaces and Theorem Provers was begun in 1995 in recognition of the fact that the difficulty in using powerful theorem proving software frequently lies with a poor user interface. There are gaps between the knowledge required by designers of such interfaces and present state of the art in general interface design technology. Effective solutions require the collaboration of HCI practitioners and the authors and users of existing theorem proving software. The increased level of interest, judged by the number of submissions, in this, the fourth in the series, is evidence that more and more implementors of theorem provers are becoming aware of the importance of good interface design, and the possibilities that modern technology offers.

In keeping with the nature of a workshop, this volume contains a number of working papers describing ongoing research at various stages of completion. The immediate goal of the workshop is to stimulate discussion based on actual experimentation with real-life systems and to feed that discussion back into further development. The long-term goal is to make the workshop defunct as a result of the improvements that have been effected. I look forward to a lively, enjoyable and memorable workshop.

Roland Backhouse
2nd June, 1998.

Contents

Invited lectures 1

1 **J.-R. Abrial** – Overview and Rationale of an Industrial Prover 3

2 **Harold Thimbleby** – The detection and elimination of spurious complexity 15

Submitted papers 23

1 **Stuart F. Allen** – From dy/dx to []P: a Matter of Notation 25

2 **James H. Andrews** – On the Spreadsheet Presentation of Proof Obligations 34

3 **Myla Archer, Constance Heitmeyer, and Steve Sims** – TAME: A PVS Interface to Simplify Proofs for Automata Models 42

4 **Roland Backhouse and Richard Verhoeven** – Extracting proofs from documents 50

5 **Richard Bornat and Bernard Sufrin** – Using gestures to disambiguate unification 59

6 **Paul Callaghan and Zhaohui Luo** – Mathematical Vernacular in Type Theory-based Proof Assistants 67

7 **Ingo Dahn** – Using ILF as a User Interface for Many Theorem Provers 75

8 **Hans van Ditmarsch** – User interfaces in natural deduction programs 87

9 **Katherine Eastaughffe** – Support for Interactive Theorem Proving: Some Design Principles and Their Application 96

10 **Mike Jackson, David Benyon and Helen Lowe** – Using ERMIA for the Evaluation of a Theorem Prover Interface 104

11 **Nicholas Merriam and Michael Harrison** – Making Design Decisions to Support Diversity in Interactive Theorem Proving 112

12 **Richard Moot** – Grail: An automated proof assistant for categorial grammar logics 120

13 **Olivier Pons, Yves Bertot and Laurence Rideau** – Notions of dependency in proof assistants **130**

14 **Joerg Siekmann, Stephan Hess, Christoph Benzmueller, Lassaad Chheikhrouhou, Detlef Fehrer, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Volker Sorge** – A Distributed Graphical User Interface for the Interactive Proof System OMEGA **139**

15 **Bernard Sufrin and Richard Bornat** – User Interfaces for Generic Proof Assistants. Part II: Displaying Proofs **147**

16 **Koichi Takahashi and Masami Hagiya** – Proving as Editing HOL Tactics **157**

17 **Patrick Viry** – A user-interface for Knuth-Bendix completion **165**

18 **Jan Zwanenburg** – Aspects of the Proof-assistant Yarrow **173**

Programme commitee

Stuart Aitken,
Roland Backhouse (chairman),
David Benyon,
Yves Bertot,
Richard Bornat,
Herman Geuvers,
Joseph Goguen,
Masami Hagiya,
Gilles Kahn,
Helen Lowe,
Tom Melham,
Nick Merriam,
T. Nakagawa,
Steve Reeves,
Bernard Sufrin,
Laurent Théry

Local organisation

Roland Backhouse,
Olga Caprotti,
Arjeh Cohen,
Herman Geuvers,
Marianne Jonker,
Martijn Oostdijk,
Elize Russell,
Richard Verhoeven

Part I

Invited lectures

Overview and Rationale of an Industrial Prover

J.-R. Abrial

Consultant*
26, rue des Plantes 75 014 Paris
abrial@steria.fr

In this document, we briefly present a program called the Predicate Prover (for short PP). This program essentially offers four functionalities, which are the following:

- A decision procedure for Propositional Calculus.
- A partial semi-decision procedure for First-Order Predicate Calculus.
- A systematic translation of Set-Theoretic Predicates.
- A coherent treatment of Linear Arithmetic statements.

In what follows, we shall quickly present these features in turn. We then show how PP is integrated within the B-Technology[1] [1] [2], as implemented by Atelier B[2] [3]. In the last section, we comment on a number of rationales and concepts that have been used in the design of PP. Finally, an appendix contains some problems solved by PP and shown in a demo.

A Propositional Calculus Decision Procedure.

PP essentially first contains an implementation of the decision procedure of Propositional Calculus, which is presented in the B-Book [1]. This procedure is very close to what is elsewhere proposed under the technical name of Semantic Tableaux [4]. It is a Sequent Calculus. Next is a sample of a classical proposition proved by PP:

$$\vdash \ ((a \Leftrightarrow b) \Leftrightarrow c) \ \Leftrightarrow \ (a \Leftrightarrow (b \Leftrightarrow c))$$

The proof procedure gradually transforms an original sequent with no hypotheses into some sequents with atomic hypotheses only (either positive or negative). Such a sequent is discharged as soon as its collection of hypotheses contain a certain atomic formula together with its negation.

A demo is available showing the step by step behaviour of PP. The following items are presented at each step: (1) the sequent at hand, (2) the inference rule that is applied to it, (3) the newly generated sequent (if any), and (4) the tree structure of the proof (this is done by means of proper indentations). A less verbose trace only presents the successive sequents (still with the indentation but without the rules). A completely silent execution is also proposed that makes the prover a genuine little "pocket prover".

* Supported by STERIA, SNCF, RATP and INRETS.

[1] B is a model oriented method used in industry to develop safety critical (and other) software systems.

[2] Atelier B is the set of industrial tools associated with the B Method.

A Partial Semi-Decision Procedure for First-Order Predicate Calculus.

The just presented simplified form of PP has then been extended to handle First-Order Predicate Calculus statements. Within this new framework, the collection of hypotheses associated with a sequent contains not only, as usual, some atomic propositions, but also some universally quantified predicates that are "normalized" in a certain systematic fashion. The prover transforms the original sequent (as above in the Propositional Calculus case) until the right-hand part of the remaining sequent is reduced to $\perp$ (unless, of course, it has been discharged in the meantime).

At this point, it then remains for us to prove that the hypotheses of the sequent are contradictory. This is attempted by means of some instantiations of the universally quantified hypotheses. Some "interesting" instantiations are discovered in a systematic fashion by means of the atomic hypotheses. This technique is a special case of the, so-called, "set of support" technique, where the set in question is represented by the atomic hypotheses. The conjunct of the retained instantiations is then down loaded into the right-hand part of the sequent, where it implies $\perp$, thus forming a new sequent. After the usual (Propositional Calculus) treatment, either this new sequent is discharged (showing that our instantiations were well chosen), or alternatively, the right-hand part of the remaining sequent is reduced to $\perp$ again. Some new instantiations are then exhibited as above, and so on.

This straightforward "ping-pong" technique proves to be extremely efficient, and, above all, it is *entirely automatic.* It has no pretension to be complete, of course. Here is a sample of what can be proved very easily using this technique:

$$\begin{array}{l}
\forall (x,y) \, . \, (\, P(x,y) \;\Rightarrow P(y,x)\,) \;\;\wedge \\
\forall (x,y,z) \, . \, (\, P(x,y) \;\wedge\; P(y,z) \;\Rightarrow\; P(x,z)\,) \;\;\wedge \\
\forall x \, . \, \exists y \, . \, P(x,y) \\
\Rightarrow \\
\forall x \, . \, P(x,x)
\end{array}$$

A demo is available presenting some "macro-steps" of the proof only. The "micro-steps" (not shown in this demo) are those already presented in the Propositional Calculus demo. Such macro-steps occur when the right-hand side of the sequent at hand reduces to $\perp$. The hypotheses (whose contradiction is then to be proved) are listed. They are followed by the instantiations that are automatically proposed by the program, together with the contradiction that is possibly emerging. In case the contradiction does not appear, the execution resumes, new hypotheses are then listed at the next macro-step, and finally some new instantiations are exhibited, and so on. A completely silent version of the execution is also available.

The Translation of Set-Theoretic Statements.

The next part of PP is the Set Translator. It is built very much in accordance with the spirit of the set-theoretic construction presented in the B-Book, where Set Theory just appears as a mere extension of First-Order Logic. The goal of this extension is essentially to formalize the abstract concept of set membership.

Statements involving the membership operator are reduced as much as possible by the translator by means of a number of rewriting rules. It results in predicate calculus statements, where *complex* set memberships have disappeared, the remaining set membership operators being left uninterpreted. For instance, a set-theoretic predicate such as $s \in \mathbb{P}(t)$ is transformed into $\forall x.(x \in s \Rightarrow x \in t)$. The translator then just performs the translation of the various instances of set membership. They correspond to the classical set operators ($\cup$, $\cap$, etc), to the generalization of such operators, to the binary relation operators, to the functional operators (including functional abstraction and functional application), and so on.

A demo is available that presents set-theoretic lemmas together with the corresponding translations. It then shows the corresponding proofs using the above macro-step traces. Next is such a lemma with its translation (where $r[a]$ and $r[b]$ respectively denote the images of the sets a and b under the relation r):

LEMMA

$$\begin{array}{l} r \subseteq s \times t \ \wedge \\ a \subseteq b \ \wedge \\ b \subseteq s \\ \Rightarrow \\ r[a] \subseteq r[b] \end{array}$$

TRANSLATION

$$\begin{array}{l} \forall (x,y) \,.\, ((x,y) \in r \ \Rightarrow \ x \in s) \ \wedge \\ \forall (x,y) \,.\, ((x,y) \in r \ \Rightarrow \ y \in t) \ \wedge \\ \forall x \,.\, (x \in a \ \Rightarrow \ x \in b) \ \wedge \\ \forall x \,.\, (x \in b \ \Rightarrow \ x \in s) \\ \Rightarrow \\ \forall y \,.\, (\exists x \,.\, (x \in a \wedge (x,y) \in r) \ \Rightarrow \ \exists x \,.\, (x \in b \wedge (x,y) \in r)) \end{array}$$

The Treatment of Linear Arithmetic.

The Predicate Prover is then once again extended in order to handle Linear Arithmetic. For this, we introduce new predicates involving the classical order operators between integers ($<$, $\leq$, etc). Such predicates are treated when it is time to discover of a contradiction in the collection of hypotheses (see above). Such arithmetic hypotheses are first normalized, then a straightforward linear technique is used in order to search for a possible contradiction between them. As a very simple example, the prover is able to prove statements like this:

$$a \leq b \ \wedge \ c \leq d \ \Rightarrow \ a + c \leq b + d$$

The Set Translator is also extended accordingly so as to treat sets that are related to arithmetic as well, namely intervals and sequences. It also translates predicates involving the minima and maxima of non-empty finite sets of integers. Next is an example of a simple lemma that is proven very easily after such a translation:

$$c \in (a \,..\, b) \ \wedge \ b \in (c \,..\, d) \ \Rightarrow \ (c \,..\, b) = (a \,..\, b) \cap (c \,..\, d)$$

The Integration of the Predicate Prover within the B-Technology.

In this section, we present the genesis of PP, we show how it has become one of the pieces of the B-Technology, and we explain how it is integrated within Atelier B. The Prover of Atelier B (for short PB), which constitutes a distinct project from that of PP presented above, works according to two modes: automatic and interactive (the interactive mode being just, in first approximation, a way of manually pulling the various strings offered by PB).

We remind the reader that a typical B development resulting in n lines of code, demands the proof of approximatively $n/2$ lemmas. At the moment, the behavior of PB corresponds to the following typical figure, which is valid for an entirely proved industrial project (say, 50,000 lines of ADA code): 80% of the proofs is discharged automatically by PB, versus 20% interactively. In this case then, approximately 5,000 lemmas have been proved interactively (less, in fact, because the user of PB can take advantage of the systematic discovery of certain proof sequences, which can then be incorporated into some tactics able to be called automatically). This figure has oriented the way PB has been designed. Automatization is indeed indispensable but, as the interactive part of the proof effort is also not negligible, both aspects of the proof technology must be implemented with great care.

The main part of PB is based on a number of rules (more than 2,000) that have been introduced gradually during the multi-year construction of this prover. It also contains certain proof mechanisms that may be handled by the user in an automatic or interactive way. Finally, the user might himself introduce some new rules and new tactics that may also be handled automatically or interactively. As can be seen, the process by which PB has been constructed is essentially a *pragmatic* one.

As time passes, we were confronted (under the pressure of some industrial users) with the problem of the correctness of the rules of PB. This is indeed a very serious problem that cannot be treated by means of some reassuring (hand-waving kind of) discourses. This is how PP has started, essentially as an extraneous project to be used in order to validate PB. The result has been more or less what we feared: *a number of rules of* PB *were slightly erroneous* (less than 5% however, but still not 0%).

In order to keep the construction of PP under control, we choose an incremental design that followed the incremental construction of Mathematics that is presented in the B-Book. This allowed us to use PP to validate PB in an incremental fashion. In other words, as soon as some stage of PP were finished, we used it to validate the corresponding rules of PP. The incremental design of PP resulted in an incremental validation of PB.

More serious even than the possibility of erroneous rules in PB is the possibility of the user introducing some erroneous rules during the proof of a B design. In order to cope with this problem we had no choice but to *integrate* PP *within* PB. Thanks to this integration, a user-defined rule can thus be validated (proved) before being used.

The practice of proving user-defined rules within PB pretty soon induced the idea of sometimes using PP *directly* on the problem at hand rather than first proving a necessarily ad-hoc rule and then instructing PB to use it. This results in a deeper integration of PP within PB. This process is still under way.

Of course, this direct usage of PP has its limitations. It is essentially due to the fact that a typical B lemma may have *many hypotheses* (more than one hundred is a not an exception). Clearly, among these hypotheses, a few of them only are relevant to prove the lemma at hand. As PP is very sensitive to noisy (useless) hypotheses, it may sometimes fail (or run for too long a period of time) on problems on which it is normally due to succeed very easily.

In order to cope with this problem, we introduced the possibility to choose the hypotheses to be kept before launching PP on a certain lemma. As this choice, however, differs from one lemma to the other, it is not easily generalizable. In order to circumvent this difficulty, we introduced some heuristics, whose intended effect is to automatically remove some apparently useless hypotheses. This has given some interesting results. Consequently, the possibility was given to incorporate these heuristics in some automatic tactics expanding the standard one. Note that the problem is complicated by the fact that, sometimes, the validity of the lemma is simply due to the presence of some *contradictory hypotheses* that have thus nothing to do with the main part of the problem. In such circumstances, as one can imagine, the hypotheses removing heuristics might fail.

This integration of PP within PB has significantly modified the user practice. At present, a typical interactive proof session with Atelier B first starts by invoking some classical features of PB: adding an hypothesis, transforming the goal by means of some assumed equality, proposing some existential witness, and so on. When the goal and the hypotheses seem to be *ripe enough*, a simple invocation of PP (with an automatically reduced set of hypotheses) then quite often discharges the goal without any further intervention. In case such a protocol seems to repeat itself on other proofs, it can then be proposed as a new automatic tactics.

Some Rationale Behind the Construction of PP.

In this section, we present some ideas and concepts that have driven us in the construction of PP. We have already explained above how PP has been developed *incrementally* on the basis of a hierarchy of provers. Although important, this strategy is, after all, nothing else but a good design practice.

The most important idea, we think, behind the construction of PP, lies in the fact that it has been designed around a fixed *wired-in* logic, which is the most classical of all, namely First-Order Predicate Calculus with Equality (used as the *internal engine*), and Set Theory (used as the *external vehicle*).

In no way is PP constructed from a meta-prover able to be parameterized by a variety of distinct logics. This contrasts with what can be seen in academic circles where extremely powerful general purpose *Proof Systems* are usually offered. Our approach is quite different, it is rather similar to that used in the development of some "industrial" programs handling symbolic data. For instance, a good C compiler is not a meta-compiler specialized to C; likewise, a good chess-playing program is not a general purpose game-playing program specialized by the rules and strategies of the chess game.

In our case, we have internalized classical logic because it is clearly that very logic that is to be used in order to handle the usually (mathematically) simple lemmas that are to be proved in order to validate software developments. This is *not* to say, however, that classical logic is the logic of software development. Clearly, it is not. Our view is that the logic of software development is whatever logic one wants (Hoare-logic, wp-logic, temporal logic, etc). Such logics, we think, are not the ones concerned by the "how-to-prove", they are the ones used to generate the "what-to-prove". We think that it is important to completely separate these two functions in two distinct tools: this is what is done in Atelier B where you have the, so-called, Proof Obligation Generator based on a weakest pre-condition analysis of formal texts (specifications, refinements, or programs), and, as a distinct tool, the Prover alluded above (PB extended with PP). Our view is that, whatever the logic of program development you decide to use, it will generate some "final" lemmas that are, inevitably, to be proved within classical logic.

Another important concept that we gradually re-discovered while developing PP is the concept of *normalization*. For instance, our decision procedure for Propositional Calculus is nothing else but a systematic transformation of the proposition to be proved into an equivalent normalized proposition of the following shape:

$$P \Rightarrow (Q \Rightarrow \ldots (R \Rightarrow A \wedge B \wedge \ldots) \ldots)$$

The interesting aspect of this transformation is that the proposition at hand can be discharged while its normalization is not completed. In other words, the normalization process is fully intermixed with the proof process.

A second normalization takes place in the Predicate Calculus part of PP. In fact, the universally quantified hypotheses are systematically transformed into equivalent predicates of the following shape:

$$\forall (x, y, \ldots) \cdot \neg (P \wedge Q \wedge \ldots)$$

where P and Q are either atomic formulae or other universally quantified formulae that are themselves normalized. This shape greatly facilitates the discovery of instantiations.

Clearly, the systematic translation of Set-Theoretic statements into Predicate Calculus statements constitutes a third normalization process. The advantage of this approach is the drastic simplification of the prover. The set-theoretic axioms and numerous set-theoretic definitions are all concentrated in the translator, not in the logic of the prover that remains unchanged.

Finally, our fourth normalization is the one undertaken in the arithmetic sub-prover of PP, where the relevant predicates are systematically transformed as follows (this form facilitates the discovery of contradictions between various arithmetic predicates):

$$a_1 x_1 + a_2 x_2 + \ldots \leq 0$$

Another concept, which we begin to integrate within PP, is that of *indexing*. This idea, presented in the referenced work [5], might have, when implemented, some spectacular effects on the behaviour of a prover, very much in the same way as a good hash-coding scheme has some spectacular effects on a compiler. A prover is full of "searching" structures (pattern matching, unification, sub-formulae searching, etc), whose influence on the speed of the prover is certain. The systematic replacement of searches performed in a (more or less) linear way by some more direct indexing schemes is, without any doubt, highly beneficial.

A concept that is clearly missing in the ones listed above is that of *induction/recursion*. Such a negative aspect is, in general, not very interesting to mention. But, in this case, because of its massive presence in other similar work and, more generally, in computing science, its absence, as a founding concept, obviously deserves some explanations. This is not to say, of course, that proofs by induction should be excluded from PP: although it is not implemented at the moment, it shortly will, since we clearly deal with inductive structures, namely numbers and sequences.

Our view is that inductive structures have been overemphasized in *computing*, perhaps because such structures are immediately *computable*. One should look a little more at computing from a non-computable world. Mathematics is full of examples where a point of view taken from the "complement" of a certain field helps studying it: infinity is used to study finiteness, the complex numbers to study the reals, more recently non-standard analysis provides a very interesting view point on classical analysis, etc. We think that the art of program development is precisely that of extracting the computable from richer not (necessarily) computable worlds. To do this, one should build *abstract* mathematical models of such worlds, models that are thus certainly not computable (at least in their more abstract versions). In order to validate our reasoning on such models, we might need some mechanical aids which are thus perhaps not necessarily tailored to work on computable models.

Some Concluding Remarks.

A prover technology, like the compiler technology more than three decades ago, is starting to emerge. The question of the automation and power of such provers becomes central. Hence old techniques should be applied and new techniques discovered in order to optimize them.

Such provers will certainly be integrated into some tools associated with certain methods of software development. But they should also constitute, in my opinion, some *independant tools* at the disposal of the designers, not only the software designers, but also, perhaps, the system designers.

At the moment, the initial analysis and architectural design of complex systems is done in a rather manual way. People perform some simulations to convince themselves that a certain architecture that they have in mind is viable. With a powerful prover it is, I think, possible to transform such simulations into genuine proofs. In very much the same way as the civil engineer is using its pocket calculator to quickly compute some order of magnitude, we could think of a future where the system designer will use also very often its "pocket prover" to validate some sketchy architecture he has in mind.

Acknowledgments.

I like to thank Nicolas Carré very much. He is the person in charge of integrating my numerous versions of PP within PB. He makes a number of very useful remarks and comments. As usual, discussions with and comments from Louis Mussat are very welcome and pertinent. Many thanks to him.

APPENDIX: Sample Problems Solved by PP.

The formulae presented below are written with a certain classical mathematical setting through LATEX. Of course, they are not entered as such in PP. However, the general structure of the formulae given to PP is almost exactly the same, the operators being conventionally represented in ASCII by means of one or more symbols.

Propositional Calculus.

$$(P \lor Q \lor R) \Leftrightarrow (P \lor (Q \lor R))$$

$$(P \land (Q \lor R)) \Leftrightarrow ((P \land Q) \lor (P \land R))$$

$$((P \Leftrightarrow Q) \Leftrightarrow R) \Leftrightarrow (P \Leftrightarrow (Q \Leftrightarrow R))$$

First-Order Predicate Calculus.

$$\begin{array}{l}
\quad \forall\,(x,y) \cdot (\,P(x) \land Q(y) \Rightarrow R(x)\,) \quad \land \\
\quad \exists z \cdot (\,\neg\, R(z) \land P(z)\,) \quad \land \\
\Rightarrow \\
\quad \forall t \cdot \neg, Q(t)
\end{array}$$

$$\begin{array}{l}
\quad \exists x \cdot A(x) \quad \land \\
\quad \exists y \cdot B(y) \quad \land \\
\quad \forall z \cdot (\,A(z) \Rightarrow \forall t \cdot (\,B(t) \Rightarrow C(z,t)\,)\,) \\
\Rightarrow \\
\quad \exists\,(u,v) \cdot C(u,v)
\end{array}$$

$$\begin{array}{l}
\quad \forall\,(x,y,z) \cdot (\,R(x,y) \land R(y,z) \Rightarrow R(x,z)\,) \quad \land \\
\quad \forall\,(u,v) \cdot (\,R(u,v) \Rightarrow R(v,u)\,) \quad \land \\
\quad \forall a \cdot \exists b \cdot R(a,b) \\
\Rightarrow \\
\quad \forall t \cdot R(t,t)
\end{array}$$

$$\begin{array}{l}
\quad \forall x \cdot (\,P(x) \Rightarrow x = a \lor x = b\,) \quad \land \\
\quad \forall x \cdot (\,R(x) \Rightarrow P(x)\,) \quad \land \\
\quad \neg\, R(a) \quad \land \\
\Rightarrow \\
\quad \forall x \cdot (\,R(x) \Rightarrow x = b\,)
\end{array}$$

$$\begin{array}{l}
\exists x \cdot A(x) \quad \wedge \\
\forall (y,z) \cdot (\, A(y) \;\wedge\; A(z) \;\Rightarrow\; y = z \,) \\
\Leftrightarrow \\
\exists\, u \cdot (\, A(u) \;\wedge\; \forall v \cdot (\, A(v) \;\Rightarrow\; u = v \,)\,)
\end{array}$$

$$\begin{array}{l}
T(a) \quad \wedge \\
\forall (x,y) \cdot (\, T(x) \;\wedge\; R(y) \;\wedge\; x \neq a \;\Rightarrow\; F(a,x,y)\,) \quad \wedge \\
\exists\, (u,v) \cdot (\, T(u) \;\wedge\; R(v) \;\wedge\; W(u,v)\,) \quad \wedge \\
\forall (b,c,d) \cdot (\, R(d) \;\wedge\; F(b,c,d) \;\Rightarrow\; \neg\, W(c,d)\,) \\
\Rightarrow \\
\exists\, t \cdot (\, R(t) \;\wedge\; W(a,t)\,)
\end{array}$$

Elementary Set Theory.

$$a \subseteq b \;\wedge\; b \subseteq c \;\Rightarrow\; a \subseteq c$$

$$a \subseteq b \;\Rightarrow\; \mathbb{P}(a) \subseteq \mathbb{P}(b)$$

$$a \times b \subseteq c \times d \;\wedge\; a \neq \emptyset \;\wedge\; b \neq \emptyset \;\Rightarrow\; a \subseteq c \;\wedge\; b \subseteq d$$

Generalized Set Operations.

$$s \in \mathbb{P}(\mathbb{P}(S)) \;\wedge\; t \in \mathbb{P}(\mathbb{P}(T)) \;\Rightarrow\; \bigcup_{x \in s} x \;\times\; \bigcup_{x \in t} x \;=\; \bigcup_{(x,y) \in s \times t} x \times y$$

$$s \in \mathbb{P}(\mathbb{P}(S)) \;\wedge\; t \in \mathbb{P}(\mathbb{P}(T)) \;\Rightarrow\; \bigcup_{x \in s} x \;\cap\; \bigcup_{x \in t} x \;=\; \bigcup_{(x,y) \in s \times t} x \cap y$$

$$s \in \mathbb{P}(\mathbb{P}(S)) \;\wedge\; s \neq \emptyset \;\Rightarrow\; \overline{\bigcup_{x \in s} x} \;=\; \bigcap_{x \in s} \overline{x}$$

Operations on Relations.

$$r \in s \leftrightarrow t \;\wedge\; a \subseteq s \;\wedge\; b \subseteq s \;\Rightarrow\; r[a \cup b] \;=\; r[a] \cup r[b]$$

$$p \in s \leftrightarrow t \;\wedge\; q \in t \leftrightarrow u \;\wedge\; r \in u \leftrightarrow v \;\Rightarrow\; ((p\,;q)\,;r) \;=\; (p\,;(q\,;r))$$

$$r \in s \leftrightarrow t \;\wedge\; a \subseteq s \;\wedge\; b \subseteq t \;\Rightarrow\; r[a] \subseteq b \;\Leftrightarrow\; a \subseteq \overline{r^{-1}[\bar{b}]}$$

Operations on Functions.

$$f \in s \mathrel{⇸} t \;\wedge\; a \subseteq t \;\wedge\; b \subseteq t \;\Rightarrow\; f^{-1}[a \cap b] \;=\; f^{-1}[a] \cap f^{-1}[b]$$

$$f \in s \rightarrow t \;\wedge\; b \subseteq t \;\Rightarrow\; f^{-1}[\bar{b}] \;=\; \overline{f^{-1}[b]}$$

$$f \in s \mathrel{⇸} t \;\wedge\; b \subseteq t \;\Rightarrow\; f[f^{-1}[b]] \;\subseteq\; b$$

Injections and Surjections.

$$f \in s \mathrel{⤔} t \;\wedge\; g \in t \mathrel{⤔} u \;\Rightarrow\; (f\,;g) \in s \mathrel{⤔} u$$

$$f \in a \rightarrowtail b \;\wedge\; r \in a \leftrightarrow b \;\wedge\; s \in a \leftrightarrow b \;\wedge\; (r\,;f) = (s\,;f) \;\Rightarrow\; r = s$$

$$f \in a \rightarrow b \;\wedge\; \mathsf{ran}\,(f) = b \;\wedge\; r \in b \leftrightarrow c \;\wedge\; s \in b \leftrightarrow c \;\wedge\; (f\,;r) = (f\,;s) \;\Rightarrow\; r = s$$

Equivalence relations.

$$\begin{array}{l}
\quad r \in s \leftrightarrow s \quad \wedge \\
\quad \mathsf{id}\,(s) \subseteq r \quad \wedge \\
\quad r = r^{-1} \quad \wedge \\
\quad (r\,;r) \subseteq r \quad \wedge \\
\quad x \in s \quad \wedge \\
\quad y \in s \\
\Rightarrow \\
\quad (x,y) \in r \;\Leftrightarrow\; r[\{x\}] \;=\; r[\{y\}]
\end{array}$$

$$\begin{array}{l}
\quad f \in s \rightarrow t \quad \wedge \\
\quad r \;=\; (f\,;f^{-1}) \\
\Rightarrow \\
\quad \mathsf{id}\,(s) \subseteq r \quad \wedge \\
\quad r = r^{-1} \quad \wedge \\
\quad (r\,;r) \subseteq r
\end{array}$$

Arithmetic.

$$a < b \;\wedge\; b < c \;\Rightarrow\; a < c$$

$$a \le b \;\wedge\; c < d \;\Rightarrow\; a + c < b + d$$

$$a \le b \;\wedge\; n < 0 \;\Rightarrow\; n \times a \ge n \times b$$

Intervals.

$$c \in (a\,..\,b) \;\wedge\; b \in (c\,..\,d) \;\Rightarrow\; (c\,..\,b) \;=\; (a\,..\,b) \cap (c\,..\,d)$$

$$a < b \;\wedge\; c \in (a\,..\,b) \;\Rightarrow\; (a\,..\,b) - (c\,..\,b) \;=\; (a\,..\,c-1)$$

$$\begin{array}{l} \quad r \in (a\,..\,b) \qquad \wedge \\ \quad k \in (a\,..\,b) \qquad \wedge \\ \quad l \in (r\,..\,k) \qquad \wedge \\ \quad m \in (r+1\,..\,k) \\ \Rightarrow \\ \quad m \le l \;\Rightarrow\; m \in (a\,..\,b) \;\wedge\; l \in (m\,..\,k) \;\wedge\; k - m < k - r \quad \wedge \\ \quad l < m \;\Rightarrow\; m - 1 \in (a\,..\,b) \;\wedge\; l \in (r\,..\,m-1) \;\wedge\; m - 1 - r < k - r \end{array}$$

Minimum and Maximum.

$$a \le b \;\Rightarrow\; \mathsf{min}\,(\{a, b\}) = a$$

$$s \subseteq t \;\Rightarrow\; \mathsf{min}\,(t) \le \mathsf{min}\,(s)$$

$$s \ne \emptyset \;\Rightarrow\; \mathsf{max}\,(s \cup t) = \mathsf{max}\,(\{\mathsf{max}\,(s)\} \cup t)$$

Operations on Sequences.

$$s \in \mathsf{seq}\,(S) \;\wedge\; x \in S \;\Rightarrow\; s \leftarrow x \in \mathsf{seq}\,(S)$$

$$s \in \mathsf{seq}\,(S) \;\wedge\; t \in (1\,..\,\mathsf{size}\,(s)) \mathrel{+\!\!\!\!\rightarrow} S \;\wedge\; x \in S \;\Rightarrow\; (s \mathbin{<\!\!\!+} t) \leftarrow x \;=\; (s \leftarrow x) \mathbin{<\!\!\!+} t$$

$$s \in \mathsf{seq}\,(S) \;\wedge\; t \in \mathsf{seq}\,(S) \;\wedge\; \mathsf{size}\,(t) > 0 \;\Rightarrow\; (s \leftarrow \mathsf{first}\,(t)) \frown \mathsf{tail}\,(t) \;=\; s \frown t$$

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press (1996)
2. D. Bert (Ed.). *B'98: Recent Advances in the Development and Use of the B Method.* LNCS 1393 Springer (1998)
3. Steria. *Atelier B Version 3.3.* (1997)
4. M. Fitting *First-Order Logic and Automated Theorem Proving.* Springer (1996)
5. P. Graf *Term Indexing.* LNAI 1053 Springer (1996)

The detection and elimination of spurious complexity

Harold Thimbleby
Computing Science
Middlesex University
London, N11 2NQ, GB

Abstract

Computer science develops complex systems that demand all our attention to just begin to understand. Critical thinking is overwhelmed, that might otherwise have been directed at rhetoric blocking and hubris detection. This paper shows that there is much unchecked hyperbole in computing, which affects our own standards and ability to design well. The paper explains why such bullshit comes about, how people collude in its propagation, and proposes ways of reducing the problem. Furthermore, we show that detecting and eliminating it is a high calling, and must be seen as engaging in justice and fighting hypocrisy (even in ourselves), and is an extremely worthwhile, if daunting, task.

> "Learning how to not fool ourselves is, I'm sorry to say, something that we haven't specifically included in any particular course that I know of. We just hope you've caught it by osmosis." Richard Feynman

Introduction

When computers work well, they work very well. Handheld calculators would have been miracles a few years ago; fly by wire aircraft are very impressive ... there are many other examples. But when things go wrong, as they do from time to time, they can go brain dead in ways we would rather quickly forget than think about.

I was told recently of a frustrated user who jumped up and down on his electronic personal organiser, until there was broken plastic and glass around on the floor. I am sure it was a satisfying experience! But can you imagine someone jumping up and down on their paper diary? You'd have to be mad to get much satisfaction from destroying one.

There is something special about computer systems, which personal organisers in the story represent. They are complex, unreliable — and yet we depend on them, and buy upgrades to go even faster.

So what uniquely identifies computing? We could start with an approach like Turing's, but this defines an object of study, not what characterises it. What is unique is the impact of spurious, man-made complexity. Most computing is not based on elegant programs, or even ones that work, but consists of hugely complex systems like Windows 98, the World Wide Web (and all its browser software), aerospace systems, financial systems, nuclear control systems, video recorders, and a host of consumer gadgets, from toasters to tamagotchis. Indeed, tamagotchis represent computing rather well:

- They are promoted as fashion accessories. Tamagotchis are available in a wide variety of distinctive packaging.
- They appear to be very simple (so simple that they are children's toys, and children can do better with them than adults).
- They have a life of their own. (Tamagotchis run animal simulations, such as dogs, and they require virtual feeding, training, virtual cleaning, exercise, and so on.)
- Their purpose in life is to be difficult to use. They have to be cared for. Cults of experts gain esteem from becoming knowledgeable about them.
- They are badly engineered. They have reset buttons, which indicates that their designers anticipated that the internal firmware could fail.
- Despite these problems, people consume them eagerly.
- Even though they fail, this does not put off hope in a new one working even better.
- They are mass produced, and have very little intrinsic value.
- They are not difficult to design.

I know a 12 year old who has written a Visual Basic program to behave like her (now defunct) tamagotchi. The complexity of tamagotchis is reasonable for a 12 year old to construct, yet what they are is complex enough to challenge the skills of someone like myself, with postgraduate qualifications in computing!

We could make many more observations — e.g., computing systems can fail but not stop working (a broken bridge doesn't bridge a river, but a financial program that fails is still a financial program). Without over-philosophising, computing concerns objects that

- have enormous potential for autonomy, creativity, control, and performing chores.
- are easy to construct, and to replicate once constructed.
- but whose behaviour is hard to identify or comprehend.

and, in consequence, they:

- are unlikely to achieve all intended goals (but they achieve 'enough' intended goals most of the time).
- support and are promoted by a social structure that rationalises their continued production and consumption.

The conspicuous feature is the difference of construction complexity versus comprehension complexity. We can view this difference from 'inside,' examining the programming process, or from 'outside' examining the assessment process.

From inside, the programming effort is effectively linear. If a program is a string of bits, programs grow sub-linearly with the typing the programmer does (some typing may be deletions). The number of things a program *can* do, however, grows exponentially with the number of interactions it performs while executing — the programmer does not know what the outcome of interactions will be, so each interaction bit doubles the space. So, a program has a complexity of behaviour that grows faster than its complexity of construction. If humans have a bounded rationality, it follows that there are programs people can write whose model they cannot understand. Indeed, *routinely* people who are sufficiently skilled to build objects achieve behaviour that is incomprehensible to them — though they may have techniques to deny it. (As an aside, this is why formal methods are necessary: to compress the behaviour into something manageable.)

From outside, interesting things happen. A person watches the execution of a program mediated through its peripherals, such as a window on a screen. Any observation records a trace, which the person generalises into a model of what the program should be able to do in principle. Unfortunately, there are no guarantees to this generalisation, yet evolution has endowed us with over-powerful mechanisms to generalise. The so-called "media equation" (Reeves & Nass, 1996) says we take media as reality — evolutionarily speaking, media are so recent that we tend to treat everything our senses perceive as real. A *real* program behaving like one demonstrated would work everywhere else in its domain; yet the demonstration has only shown us a single trace, and in a demonstration one cannot distinguish between a simulation (which need be no more than a "film") and the real thing.

We regularly exploit the media equation for enjoyment — for the willing suspension of our critical faculties. Theatre is the projection of a story through the window of a stage, and typically the audience gets immersed in the story *as if* it was real. This is deliberate. We willingly suspend asking questions about the story that is not projected, such as we don't worry about unrepresented details of King Lear's life. However, if the theatre represented a real model, such questions would have answers. In computing, the power and technique of the theatre is recruited to demonstrations — there is a literature urging the exploitation of dramatic technique to enhance interactive systems (Laurel, 1991). It is very hard to watch a demonstration and to enquire about the off-stage issues: it is as if one is breaking the cultural taboos of interacting with actors. It is therefore tempting to come away from a demonstration believing (or not knowing otherwise) that the trace was typical of the general behaviour of the program.*

There would be no problem except we require systems to meet certain *prior* requirements, and for most systems (apart from games) these requirements are hard to meet. The people who design and build computing systems need certain skills.

* *Theory* and *theatre* have similar Greek roots, derived from θεα: theory is about objects of study, and theatre presents objects to view or study (Knuth, 1996).

The issue is how to eliminate spurious complexity (that is the consequence of inadequate skill applied to the task of constructing objects of particular behaviour) when we are not disposed to see it, whether we are users or designers.

Brief examples of problems

Casio calculators

Calculators are an example of a mature technology. Basic calculators have well-defined requirements, of accuracy and performance and so on. There have been many generations of calculator designs, and the manufacturers have had many opportunities to 'step' their production to fix known problems. The only limitations on calculators are the manufacturers' imagination and skill. I want to devote some space to this example because so few people see any problem at all.

Casio is the leading manufacturer of handheld calculators. Two of their basic models are the SL-300LC and the MC-100.

- Pressing the buttons AC 1 + 5 % leaves the MC-100 displaying 1.0526315 and the SL-300LC displaying 1.05. Yet these calculators look very similar.
- Both calculators have memories, which (so far as I can tell) are identical. The button MRC recalls the stored number and displays it, but pressed twice in succession sets it to zero. The button M+ adds the displayed number to memory, and M− subtracts from the memory. Given that the calculators have a memory, how can a number displayed be stored in memory? (Pressing M+ presupposes the memory contains zero. And to make the memory zero, you have to press MRC twice, but doing that sets the display to the memory — losing the number we wanted to store!)

Thus a market leader, Casio, makes two similar calculators that work in subtly different ways, and both proclaim features that are ironic. Memory should save paper and help the users do sums more reliably. Yet most users (especially those that *need* calculators) would need a scrap of paper to work out how to avoid using paper to write down the number!

Casio has been making calculators for a long time, and the two calculators are not "new" in any way. It is not obvious how Casio can justify either the differences or the curious features shared by both calculators. Neither comes with user manuals or other information that reveal any problems.

Any calculator, and the Casio ones in particular, can be demonstrated. They are impressive, especially if a salesman shows you them going through some typical (but unsophisticated) calculations. It is possible to demonstrate the memory in action, and only some critical thought would determine that it is a very weak feature.

Canon cameras

The Canon EOS500 is one of the most popular automatic SLR (single lens reflex) cameras, and is a more complex device, with more complex requirements, than a calculator.

In the Casio calculator examples, despite Casio's undisputed ability to *make* calculators, we might query their ability to *design* them. In the Canon camera example, we have more evidence. The EOS500 camera manual warns users that leaving the camera switched on is a problem. Canon evidently *know* that the lack of an automatic switch-off is a problem! There is an explicit warning in the manual on page 10:

> "When the camera is not in use, please set the command dial to [L]. When the camera is placed in a bag, this prevents the possibility of objects hitting the shutter button, continually activating the shutter and draining the battery."

So Canon knows about the problem, and they ask the user to set the camera off — rather than designing it so that it switches itself off. A cynic might suppose that Canon make money selling batteries or film; the next example is another case of Canon apparently trying to sell more film:

> "If you remove a film part-way, the next film loaded will continue to rewind. To prevent this press the shutter button before loading a new film."

There are many other admissions of flaws. Thus Canon is aware of design problems, but somehow fail to improve (the EOS500N is a new version of the EOS500, with similar problems).

Java

Java is promoted as a programming language with a buzzword list of virtues. We will look at one problem: it's very easy to confuse the different behaviour of fields and methods. This is a point made in the book *The Java Programming Language* (Arnold & Gosling, 1998), written by some of Java's designers:

> "You've already seen that method overriding enables you to extend existing code by reusing it with objects of expanded, specialized functionality not forseen by the inventor of the original code. But where fields are concerned, one is hard pressed to think of cases where hiding them is a useful feature."
>
> "Hiding fields is allowed in Java because implementors of existing super-classes must be free to add new `public` or `protected` fields without breaking subclasses."
>
> "Purists might well argue that classes should only have `private` data, but Java lets you decide on your style."

Purists may define all fields to be `private`, and will provide accessor functions if the field values are needed outside a class body. Unfortunately, this safer programming has efficiency implications, which is probably the reason Java is designed the way it is.

Like the Canon camera, we see the English description of a system admitting avoidable problems with the system.

Collusion

We've shown that commonplace systems are badly designed, and we argued that bad design is a consequence of unmanageable complexity. Ideally, systems should be better engineered, but they aren't.

There are many reasons why we collude with bad system design, whether as consumers of attractive gadgets that promise to do wonderful things; whether as programmers who make a living from developing systems; or as academics who can make a living solving the problems. The reasons are deep and varied psycho-social mechanisms (e.g., Baudrillard, 1998; Postman, 1992).

Lottery effect: computers seem to be more successful than they are

Lottery winners are reported in the media, and we become familiar with success. But success is infrequent — just sampled with bias by the media! In technologies that depend on media (e.g., the Web) it isn't possible to sample failures anyway. Companies that experience computer failures and hence go out of business don't exist.

Realism–reality gap: designers are under pressure to deliver because it is "so easy"

Realism is easy: a look at any arcade game will show the sophisticated realism that is possible. The media equation implies we tend to treat realism *as* reality — good design is easy to fake, especially when you can't assess the mechanism.

Most people therefore think programming *is* trivial. (Even if it is hard, the scale of production means the marginal cost of design is trivial.) So, designers are put under pressure from marketing, management, and everyone else, to deliver complex products faster than is possible consistent with doing a good design.

Oracle effect: experts under-estimate complexity

Experts (particularly programmers) know how complex systems should be used ("press the twiddle key when you do that!"), and often the reason why a user cannot operate a system is because they do not know some apparently trivial fact. The expert tells the user, and the user is impressed with the skill. The expert thinks the user is stupid, because the fact is trivial.

One way to use computers

Because oracles are so successful, there "must be" a right way to use computers. It is useful to have a word for deliberately avoiding their narrow-mindedness. A system is *permissive* if it permits itself to be successfully used in more than one way. One that is not permissive is *restrictive*. For example, my to get my VCR from record-pause mode to record mode, I must press Play: yet both Pause and Record do nothing — this is both odd and restrictive. (It probably comes about because programmers write straight-line imperative programs, rather than declarative programs.)

Even human factors experts may assume there is *one* right design, and that users must know it. Nielsen (1993) describes a permissive system, yet users were classified as "erroneous" if they knew only one of the alternatives!

Confusing automation for computation: mindless efficiency

Computers can automate bureaucracy, and they can do it faster than by hand. This results in a mindless application of computers to 'solve' problems by making inefficient activities faster, rather than more efficient. A specific example is the way calculators merely do what mechanical calculators do, rather than something new (Thimbleby, 1996); and, worse, on-screen calculators mimic handheld ones!

Inertia

Lawyers have ensured that there is no liability attached to shoddy design. Consider the warranty that comes with any piece of software:

> "**Disclaimer of Warranty.** ... makes no warranties express or implied, including without limitation the implied warranties of merchantability and fitness for a particular purpose, regarding the software or its use and operation alone ... In no event shall ... [there be] ... product liability or otherwise. "

There is no reason to improve (this is the tragedy of the commons: because we all benefit by not improving), which leads to the 'state of the art' defence in law.

Superficial usability

Because there is one way to use computers, and because programmers "don't need" to improve, and because of the media equation ..., a huge emphasis has come to be placed on appearances and post-design methods. New computers look attractive, but they still run the same old software. The disciplines involved in assessing usability of systems have developed various non-technical approaches, and because of their effectiveness (in the face of apathy) they have gained ascendancy (cf. Landauer, 1995). User interface designers have seen their job as understanding the psychologically-interesting human responses to bad technology, rather than avoiding the problems in the first place. See Rettig's (1992) paper "Interface Design When You Don't Know How," summarising the wisdom of conventional HCI.

If we want to improve computing beyond placebos and palliatives, we cannot look to usability experts for serious help — see (Thimbleby, 1998) for a constructive diatribe arguing we need ways to help designers.

Usability is the user's problem

Ralph Nader's classic book *Unsafe at Any Speed* (1965) shocked the 1960s car industry. He strongly criticised the industry for making intrinsically unsafe cars and for blaming drivers for accidents. "The driver has the accident and the driver is responsible," the manufacturers argued. Pedestrians "gently knocked" were killed, cut open by sharp body styling. Car manufacturers responded that in any collection of accident statistics one would be bound to get some gruesome cases: they denied that the inherent dangers of sharp fins were their fault, and anyway drivers wanted such grotesque styling!

In the sixties people were told to drive more safely (to be 'car literate' just as today users read *X for Dummies* to be computer literate), but the manufacturers said this to deny their responsibility for designing safer cars. Today people have problems with computers. Today people are told to read the computer manual and make themselves computer literate. If a user does that, the problem for the manufacturer goes away. This approach feeds an industry in training and consultancy.

Nader showed that many designs were intrinsically unsound and could not be driven well even by highly skilled drivers. The onus — not admitted in the 60s — was on the designers to make cars that were easy and safe to drive. It required force to make this change in perception, led by consumer pressure, as well as legislative and professional standards.

Good design as engaged explanation

Somehow, there is a gap, and it needs bridging. Good user manuals seem to be conscious of usability problems, but the manuals are somehow not engaged in the design process — rather, they are commentaries on it, written by powerless authors too late in the product design cycle. How can *effective* consciousness be brought about? One way would be to make a method of taking warnings in manuals as indicators of improvements.

User manuals are often scapegoats for bad things. They are indeed often unintelligible, and thereby contribute to the confusion and difficulties users have. But it isn't possible to write good manuals for bad systems. However, manual writers do "stand back" from mere manual-writing and provide users with useful advice about how to cope with problems with a thing's design. A user manual is a partial program for the user to 'execute' to run their side of the user interface;

we ought to use all the tools of computing to make user manuals better (e.g., declarative, if we think declarative programming is good).

It is self-evident, and borne out by experiment (Carroll, 1990), that short manuals are better than long manuals. Combining this idea with the previous gives a design approach to make better things:

❿ Construct the initial user manual. This step should be automated.

❡ Find problems. Clearly, good technical authors are able to do this. It is likely that the act of explaining clearly how to use a system helps uncover problems with it. Some aspect of a design that cannot be explained briefly and clearly is likely to be hard to understand.

① Fix the design: the user manual, along with its warnings, lengthy explanations and invocations of oracles, is a direct indicator of the design areas that need attention.

④ Fix the manual, having fixed the specification. (This step should be automatic if step 1 is automatic.)

And repeat, while each step improves the design and the product. Many manufacturers have the luxury of producing a range of products, and of updating them regularly. In such cases, one might manufacture a design before the improvement cycle is complete, leaving further improvements for future products. Thus, the method not only improves design, but gives marketing a method for continually enticing consumers. It ought to be easy to justify!

To the extent that this *is* a good method, then systems should be designed so that user manuals can more easily be generated from them or their specifications (cf. literate programming: Thimbleby, 1990). While at it, we can also generate other sorts of "manual" (paper, interactive, diagnostic, and so on) with little additional effort.

If the user manual is written (or partly written: see Thimbleby & Ladkin, 1995) by automatic tools, there is little delay in this cycle; it could be fully concurrent. If manuals have to be written by people without help from the formal specifications (help!), then at least in manufacturing, last year's manuals can be fed into this year's products.

It is easy to write manuals that are vague, inexact and misleading. To be effective, manuals need to be complete and sound. Perhaps there could be internal documents that are used in the design process, and actual user manuals that are derived from the internal manuals, made more readable for users.

More generally, for "manual" substitute any view. The formal specification of a design (whether as a logical formula or a circuit diagram or computer program) is "just" another way of explaining the design — but to a different sort of user (a mathematician, an electronic engineer, a programmer). These "manuals" can give the "technical author" opportunities to explain and help the "user." Different sorts of design problems will be brought to consciousness, and fixes will be suggested. Thimbleby and Ladkin (1997) use a logic specification of an Airbus subsystem to show that quite complex system manuals can be improved (and that minimisation algorithms can be used to reduce their size).

Justice

What do we mean by designing *better* things? What is good anyway, what is this goal of getting better? These are questions of ethics (or moral philosophy), the study of what is right. Ethics has a long history, going back to Aristotle (384–322BC) and earlier.

Aristotle defines justice as the act of giving a person good. This is what designers who strive to design better things do. They design "good" which is embedded in the things they design. This good is then passed on to the users of the things. To do good design, then, is to be engaged in acts of justice.

There are different sorts of justice. A user of a gadget is typically unable to negotiate over details of the design: in a sense, the designer has authority over the user, at least in so far as the product constrains the user. Justice as an act of authority is the maintenance of rights: the user of gadgets have rights, and just design is to maintain those rights. And there is contributive justice, which is the obligation to enable individuals to achieve good. In contributive justice, the designer contributes to the users' ability to make good use of the gadgets. Clearly, good manual writers contribute to a just world.

If design is justice, can we make use of this fact? A few thousand years of philosophising on justice has had little effect on the world. John Rawls wrote the classic book (1971) *A Theory of Justice*, where he promoted the idea of justice as fairness. Rawls defined justice as a system of rules that would be designed by people under a "veil of ignorance" of whether and to what extent those rules apply to themselves. By this he meant the designers do not know how they might be affected, so they will build a world that treats *them* fairly. For example, one might imagine that the planners of a just system are as-yet unborn. They might be brought into the world at any age; they do not know whether they will be rich or poor, black or white, handicapped or athletic, male or female, blue-eyed or green. Under this veil of ignorance they would be foolish to behave other than as fairly as they possibly could. They might be brought into the world too old to operate a video recorder!* The scope of the fairness applies to the designers themselves as well as to the users.

Do designers of things act justly by Rawls' definition? Mostly not. They design things they know they will not use, and even if they did use, they would have oracular knowledge. Designers are *never* in a veil of ignorance. Many programmers build systems that they have no intention of using. If, instead, they worked under the Rawls veil of ignorance, they might try harder — in case they ended up being a user of their system. If they were programming a tax program, they might end up "born as" accountants, tax-payers, civil servants designing tax law, tax evaders, auditors, managers, as their own colleagues having to maintain their system at a later date, or even as the manual writers ... they would have to design their tax program carefully and well from all points of view, including making manual writing easy (which gains the advantages described above). They might prefer to contain complexity rather than risk it being unmanageable.

This idea is anyway enshrined in conventional good practice: "know the user" (cf. Thimbleby, 1990; Landauer, 1995). Rather than merely "know" one's way into all the other possible roles, one might more easily, and more reliably, do some experiments and surveys with other people (though to do this requires the product, or perhaps an earlier version of it, to exist). It is pleasing that accepted design practice is also just (who wants to be called unjust?)

To summarise: good design is engagement with justice, and we have seen two ways to do this. First, to stand back and be conscious of the ways in which others (users) will operate the product — use concurrent engineering with user manuals; secondly, to put oneself into the many different roles of usage. A consequence is that designing systems to support easier manual generation becomes a higher priority, and this in turn helps improve the systems themselves.

Design by accident?

Aristotle claims justice is the only virtue that can be achieved by accident. You can't have integrity (another virtue) by accident: integrity has to be intentional. Someone who claims to have integrity but does not is faking, and has no integrity. But acts of justice do not depend on the judge, they are outcomes and are just or unjust to the extent that they fairly affect others. The point for designing better things is that some designs will be good by accident.

The market helps ensure (but unfortunately does not guarantee) that good design thrives, and conversely, poor design gets less market share in the face of better competition. The market is a force of "natural selection." Designers are the evolutionary equivalent of mutagens — they create mutations: they produce new designs and new variations. By Aristotle's argument, in design we can have a successful blind watch maker. That some blind watch makers may be successful by chance is no reason to copy them. If we want to design deliberately, we need a commitment to justice in design. This cannot be done by accident.

Conclusions

The argument of this paper is that computing systems are so complex and unreliable that they are really a different kind of thing that requires a different kind of thinking. In particular, they are so complex that we are no longer able to assess them for quality, and so we take them as objects for uncritical consumption. Our entire culture is taken up in this game: it suits almost

* There are difficulties with taking Rawls too seriously. There are duties of just action to non-contracting parties, such as to the environment. How we design things to take their 'responsible' place in a larger ecosystem beyond other users, say to be recyclable, is beyond the scope of this paper — but that is not to imply such issues are optional; see Borenstein (1998).

everyone in different ways — manufacturers make lots of money (selling systems to fix problems that should not have been there), book publishers sell "dummies" books, marketing people have lots to advertise, and we all seem to swallow it whole. Indeed, it *is* fun to have a fancy device!

If we are designing systems, we are caught up in the culture, and design over-complex systems that we are over-proud of. This paper suggested an approach to help design better; moreover, a method that can be used to help direct the design so that automatic user manual generation is easier. To try to escape from the cultural forces is not easy, but it may help to see that the effort is engagement in justice, and therefore a noble cause.

References

Aristotle, *Nicomachean Ethics*, in *Great Books of the Western World*, **8**, Encycopædia Britannica, 2nd ed., 1990.

K. Arnold & J. Gosling, *The Java Programming Language*, Addison-Wesley, 2nd. ed., 1998.

J. Baudrillard, *The Consumer Society*, SAGE Publications, 1998.

N. S. Borenstein, "Whose Net is it Anyway?" *Communications of the ACM*, **41**(4), p19, 1998.

Canon Inc., *EOS500/500QD Instructions*, part no. CT1-1102-006, 1993.

J. M. Carroll (1990), *The Nurberg Funnel*, MIT Press.

D. E. Knuth, *Selected Papers on Computer Science*, p143 Cambridge University Press, 1996.

T. Landauer, *The Trouble with Computers*, MIT Press, 1995.

B. Laurel, *Computers as Theatre*, Addison-Wesley, 1991.

R. Nader, *Unsafe at Any Speed* , Pocket Books, 1965.

J. Nielsen, *Usability Engineering*, Academic Press, 1993, p61.

N. Postman, *Technopoly*, Vintage, 1992.

B. Reeves & C. Nass, 1996, *The Media Equation*, Cambridge University Press.

M. Rettig, "Interface Design When You Don't Know How," *Communications of the ACM*, **35**(1), pp29–34, 1992.

J. Rawls, *A Theory of Justice*, Oxford University Press, 1972.

H. W. Thimbleby, *User Interface Design*, Addison-Wesley, 1990.

H. W. Thimbleby, "A New Calculator and Why it is Necessary," *Computer Journal*, **38**(6), pp418–433, 1996.

H. W. Thimbleby, "Design Aloud: A Designer-Centred Design (DCD) Method," *HCI Letters*, **1**(1), pp45–50, 1998.

H. W. Thimbleby & P. B. Ladkin, "A Proper Explanation When You Need One," in M. A. R. Kirby, A. J. Dix & J. E. Finlay (eds), BCS Conference HCI'95, *People and Computers*, **X**, pp107–118, Cambridge University Press, 1995.

H. W. Thimbleby & P. B. Ladkin, "From Logic to Manuals Again," *IEE Proceedings Software Engineering*, *144*(3), pp185–192, 1997.

> "Engineering is the art of moulding materials we do not wholly understand ... in such a way that the community at large has no reason to suspect the extent of our ignorance." A. R. Dykes.

Part II

Submitted papers

From dy/dx to [ ]P : a matter of notation

Stuart F. Allen
Cornell University

Abstract

An analysis is given of the conventional $\frac{dy}{dx}$ notation for derivatives that explains it as a notational abbreviation for expressions using the simpler binding structure standard in modern formalizations. The Nuprl display system was used to implement examples of such notation.

It turns out that the same methods can be used to explain conventional modal logic notations. We construe necessity as a first-order quantifier, in a well known way, then explain standard modal notation as a way simply to display these formulas of a non-modal logic.

We contrast the method with the interpretation of necessity as a sentential operator, and also with higher-order interpretations that have been used to interpret temporal logic in HOL. The methods are then applied to a simple first-order temporal logic. The intention is that the user can work in this notation interactively, not just produce it for printing.

The methods to be discussed here for formalizing a few mathematical and logical concepts are *already* well known, or are small variations on well known methods, and are *not* the true subject of this paper. This paper is about notational enhancements for exploiting those methods, and may also serve as an explanation for some notations that are conventional, but do not obviously conform to the simpler syntax and semantics of current-day computerized formal mathematics.

We apply a particular combination of notational devices to a few examples, revealing their notational similarity. We start with Leibniz's notation for derivatives, $\frac{dy}{dx}$, and end with first-order temporal logic for programs. These notational methods have been made precise, and implemented in the Nuprl proof development system,[1] where they are meant for use as working notation. These examples were developed within it, although almost none of the mathematics for which these notations were implemented has been carried out in Nuprl.

The basic idea: How $\frac{dy}{dx}$ works.

Suppose `Deriv(x. e(x) ; a)` is a binding operator used to stand for the derivative, at a, of the function denoted by $e(x)$ in variable x.[2] So, for example,

[1] The author was the principal designer of the current (since 1991) Nuprl display system and editing primitives; Richard Eaton implemented them and provided supplementary design. The Nuprl project[6, 8] is based at Cornell, and is directed by Robert Constable. See www.cs.cornell.edu/Info/Projects/NuPrl .

[2] Dummett presents such a notation as the form adhering to Frege's notational demand that the sign for a function occur only in application to its arguments.[5]
Of course, another useful way of denoting the derivative is with a non-binding one-place operator

```
    Deriv(f) == λa.Deriv(x. f(x) ; a)
```

or if we take `Deriv(f)` as more basic we may define

```
    Deriv(x. e(x) ; a) == Deriv(λx.e(x))(a)
```

Each is a composition of the other with function application and lambda. These are perfectly compatible ways of denoting the derivative, each circumstance determining which is simpler to use. See the section below (p6) on "lifting".

```
∀b:ℝ. Deriv(x. x·x+b·x ; 3) = 6+b
```

or more generally,

```
∀b,a:ℝ. Deriv(x. x·x+b·x ; a) = 2·a+b
```

or equivalently, and this is the *key move*, by changing bound variable "a" to "x",

```
∀b,x:ℝ. Deriv(x. x·x+b·x ; x) = 2·x+b
```

At this point, we notice that the standard Leibniz style notation for derivatives acts a lot like the last notation; familiarly,

```
∀b,x:ℝ. d(x·x+b·x)/dx = 2·x+b
```

Construing `d(e(x))/dx` as a mere notational abbreviation for `Deriv(x. e(x) ; x)` makes plain both the dependency of the whole expression on "x", and the use of "x" simultaneously to indicate the argument place in the expression for the function being differentiated.

Although there are other more complicated usages of the "d/dx" notation, nevertheless, this one is fairly common, and explanation along these lines may alleviate confusion about how the notation works semantically. We take the key characteristic of this usage to be simply that the binding variable is also used as the other (non-function) argument.

Another familiar kind of construction from mathematical vernacular, which one might explain similarly, is exemplified by:

x is the unique integer such that $P(x)$

which we could take as simply a way of displaying a term:

u is the unique integer x such that $P(x)$

in the special case that u is x, i.e., the binding variable is also the free variable.

The Nuprl display system allows such display methods to be stipulated for terms of a general purpose syntax of (possibly binding) operators. Below we shall apply this notational method more elaborately to modal and first-order temporal logic, but first let's digress a bit and give some background on the system in which the notation is implemented.

Nuprl Term Structure

All the expressions in this paper using tt font were produced by Nuprl's display system. This is the notation that the Nuprl user sees while editing the underlying terms with a structure editor which does essentially no parsing; the forms of display are not inherent in these terms and the display forms may be changed at any time.

The terms of Nuprl are iterated operators of various arities; one specifies for each subterm which variables become bound. The concepts of free and bound variable, and capture-avoiding substitution are then the usual ones. A Term is essentially a 5-tuple $< op, n, t, k, x >$ where, treating the class Op of "operator names" and the class Var of variables abstractly,

- $op \in$ Op
- $n \in \mathbf{N}$, indicating the number of places for immediate subterms
- $t \in \{1..n\} \rightarrow$ Term, indicating the immediate subterms
- $k \in \{1..n\} \rightarrow \mathbf{N}$, with k_i indicating the number of binding variables that may become bound in the i-th subterm.
- $x \in \Pi i : \{1..n\}.\ \{1..k_i\} \rightarrow$ Var, with $x_{i,j}$ indicating the j-th binding variable for the i-th subterm.

For our purposes, we can assume that Op is a sequence of one or more strings, numbers, etc. Usually the Op of a term is just a single identifier.

There are no further restrictions on term structure. How terms are displayed in Nuprl is not inherent in either the structure of terms or the definitions of constants and operators; display is specified separately. Operations for term editing act directly on these structures[3], modulated by the display forms in force at the time.

When we can't think of a better way to display a term, we usually just write the Op followed by the subterms, if any, and prefix each subterm by the binding variables for that place, if any. So, $op(t_1; u.t_2)$ would be the Term $< op, 2, t, k, x >$, where $k_1 = 0$, $k_2 = 1$, and $x_{2,1} = u$.

For example, `all(A; x.B(x))` is used in the standard Nuprl libraries to represent instances of the universal quantifier, where `A` is the domain of quantification, and `B(x)` is the formula being quantified; the standard display is `∀x:A. B(x)` .

Sometimes other values are included in the operator as a way of getting those values into the term structure as literals. When we don't have a better way of displaying them, these extra values are usually just written directly after the first identifier, in braces. For example, the basic numeric literals used in Nuprl are exemplified by "`natural_number{2}` ", which has no immediate subterms, and is normally displayed simply as 2.

Normally, such a discussion of term structure would lead to a description of operator definitions, and indeed we'll see some examples below, but our concern here is really with how such terms are displayed.

Displaying Terms

At any point during a Nuprl session, there is a set of named objects of various kinds, mostly loaded from library files. In addition to proofs, operator definitions, inference rules, program code, and documentation objects, Nuprl libraries contain objects that specify how to display terms.

A specification includes a term such as `deriv(<x>.<e>; <a>)` , called the "display model," which may contain schematic variables such as `<e>`, in place of various parts. The display spec is applied by matching for these schematic variables, then instantiating into a "formatting command" that is also part of the specification; formatting commands specify what characters to display, as well as break/margin control similar to Oppen's pretty printer methods[14].

Here are the display specifications used above to display the derivative. The main things to observe are the display models; the reader need not really understand the rest of the specification, but we show it simply to demonstrate that it is a fairly simple schematic method. Here is the display specification that generates the non-d/dx display form used above:

```
Model: deriv(<x>.<e>; <a>)
  Deriv(<x:var>. <e:real>
                ←MARGIN ;[ ]
        <a:real>)
        ←MARGIN←SOFT
```

Given a particular term, there may be several ways to display it – there may be several specifications having display models which match the term. In addition to this display spec, we have added another one for our special case:

[3] Structure editing of terms, via grammars for those terms, was pioneered by Teitelbaum[16, 15]

```
Model: deriv(<x>.<e> ; <x>)
Attrs: *Open form*; =apply.standard
  d<e:real:(<self), AddIparms(<x>)>
   ←MARGIN[]
  /d<x:var>
  ←MARGIN←SOFT
```

Notice how the display model indicates the special circumstance of applicability, namely, that the same variable name must be used both as the binding variable, and as the second argument to the operator. (During substitution, Nuprl usually attempts to retain variable names, as well as identity and difference between variables bound by the same operator occurrence.) The "AddIparms" element will become significant for our discussion, and will be addressed below.

When a term is displayed, it runs through the display forms in a specific order trying to find one that may be applied to the term in question. As a result, the term d(x·x+b·x)/dx is normally displayed as such rather than as Deriv(x. x·x+b·x ; x) , although the user may temporarily disqualify the d/dx form for whatever motive, such as finding the notation mysterious or ambiguous. The term Deriv(x. x·x+b·x ; a) , however, is simply ineligible for the d/dx form, and so the long form is used. Indeed, if "a" is substituted for free "x" in d(x·x+b·x)/dx , say in the course of a proof, then the d/dx form will be automatically abandoned in favor of Deriv(x. x·x+b·x ; a) . Or, utilizing a simple substitution operator defined by e(x)|x=a == e(a) ,

d(x·x+b·x)/dx|x=a rewrites by definition to Deriv(x. x·x+b·x ; a) .

Let us return to the "AddIparm" element in the display spec above, which has been attached to the formatting command for a subterm. It is rather common in informal practice to elide certain variables from expressions which nevertheless depend upon them, such as when the same variable is used repeatedly throughout a long argument or other discourse. Nuprl terms must include any variables they depend on, so these *implicit parameters* must be elided merely as as matter of display.

The recursive descent display algorithm has as one argument a set of variables considered to be implicit parameters. The display form in question stipulates that whatever variable of the instance matches the schematic variable <x> will be added to the implicit parameter set when the subterm is displayed.

It *is possible* to stipulate that a given display form is usable only if certain variables are in the implicit parameter set. To continue with our d/dx example, suppose we wish to work with functions that will normally depend on the variable x. We may define a special function-application form that is intended for use mainly with x as its argument, and whose display elides x when it's an implicit parameter, but shows it otherwise. Here's an example of using this apply form with function y.

∀b:ℝ,y:ℝ→ℝ. (∀x:ℝ. y(x) = x·x+b·x) ⇒ ∀x:ℝ. dy/dx = 2·x+b

(Note the abbreviation of the iterated quantifier. The display system has several iteration-related capabilities.) If we alpha-convert ∀x:ℝ. dy/dx = 2·x+b , changing all binding x's to v, say, we get ∀v:ℝ. dy(v)/dv = 2·v+b , in which the argument to y now stands revealed because it is not x. There is further discussion of calculus notation in [13].

To summarize, Nuprl has been used to explain certain d/dx notations, *not* by extending the basic term structure and altering concepts of binding, but rather by construing them simply as notational abbreviations for certain forms of notation having the more commonly understood binding conventions, and explicitly containing the variables upon which they depend. In search of other applications for this device, let us turn our attention to the notations of Modal Logic.

Modal Logic: first-order necessity

Rather than proceeding directly to first-order temporal logic for programs, we begin more simply with modal logic under a possible-worlds semantics.

Nonstandard Nomenclature Warning: Our consideration of modal language will compare three ways that modality may be expressed, depending on whether necessity is taken as a propositional operator, a first-order quantifier, or an operation on proposition-valued functions. We assume that any modal logics we are interested in might have ordinary quantifiers, and perhaps higher-order functions; so we shall appropriate the adjectives *propositional*, *first-order*, and *second-order*, when applied to modal concepts, to indicate which of these three treatments of the modal operators is pertinent. We shall use the adverb "modally," to modify these three adjectives.

Let us use the term *modally propositional formula* in reference to the standard syntax of modal logic, in which the modal operators are sentential operators, and with the semantics defined with respect to possible worlds in the manner of Kripke[9]. Below, we assume the type `PW` is the type of all possible worlds, and (`W Pwrt W'`) means `W` is possible with respect to `W'`.

It is well known that (what we call here) modally propositional formulas, can be translated into a non-modal language.[3, 2, 1] The modal sentential operators are eliminated in favor of explicit quantification over possible worlds, and various properties and relations are extended to take a possible world as an extra parameter. The same goes for temporal logics and logics of tense. Thus, the definition

```
Nec(W. P(W) ; W') == ∀W:PW. (W Pwrt W') ⇒ P(W)
```

having as its definiens the result of the standard translation, may be regarded as a definition of the necessity operator. Let's call this operator *first-order* necessity, since it is works like a first-order quantifier.

Now, applying the method used above to explain d/dx, we stipulate that terms of the form `Nec(theWorld. <P> ; theWorld)` are to be displayed as `□<P>` . Note that we are further restricting this method of display so that it applies *only* when the variable is exactly `theWorld`. Again, we shall make `theWorld` an implicit parameter in the display of the subterm `<P>`, in order to allow its suppression.

We can now characterize the *modally first-order formulas* of our language as those in which `theWorld` is the only variable, free or bound, that ranges over `PW` , and further, that `theWorld` is bound only by modal quantifiers, such as necessity. These formulas are very nearly those that result from the standard translations of modally propositional formulas into non-modal formulas. The differences are two: we use the first-order necessity operator instead of what it expands to by definition, and we completely limit the choice of possible world variable. So the actual standard translations are all the alpha-variants of our modally first-order formulas after eliminating the first-order necessity operator by expanding it according to its definition.

Next, let us say an operator is *first-order world-relative* when it has an argument place for expressions of type `PW` . When adding first-order modal operators such as necessity, it is not necessary to make all other operators world-relative as well; we just need to ascertain whenever a new operator's meaning depends on possible worlds, and make *it* world-relative.

For example, let `Person(W)` comprise the persons existing in world `W`, and when `theWorld` is implicit, display `Person(theWorld)` simply as `Person` . Further, for `W` ∈ `PW` and for all *possible* persons `x`, let "`x invented bifocals in W` " mean what it looks like. Then

```
¬□(∃ x:Person. x invented bifocals)
```

is simply

```
¬Nec(theWorld. ∃ x:Person(theWorld).
                x invented bifocals in theWorld ;
    theWorld)
```

but is more effectively displayed. More generally, if for every world-relative operator, the user has specified a display form which elides `theWorld` when it is implicit, then modally first-order formulas will be displayed with ordinary modal notation.

It should be emphasized here, that *this* is not a translation from modally propositional to modally first-order formulas – there is only the one non-modal language here, and we are simply providing an alternative explanation of the standard modal *notation* as a combination of notational devices for the non-modal language. When one constructs proofs in Nuprl, the display forms are not visible to the inference engine, so the usual tactics apply to these formulas.

If in the course of using modally first-order formulas, the user should generate a formula that is not, this becomes evident from the disappearance of the standard modal notation. For example, even rewriting a first-order necessity operator by its definition results in a term that is not modally first-order, since avoiding capture of `theWorld` forces a change of bound variable.

```
[][](∃ x:Person. x invented bifocals)
```

becomes

```
∀W:PW. (W Pwrt theWorld) ⇒
Nec(theWorld. ∃ x:Person(theWorld). x invented bifocals in theWorld ; W)
```

Even though the second occurrence of the necessity operator remains, it is no longer a modally first-order formula because it is no longer necessity with respect to `theWorld`; and observe that no alpha conversion of the whole can restore it. Roughly put, denoting a relation between two possible worlds, gets you kicked out of the modal notation. Of course, you can still proceed with the non-modal formulas, and maybe you'll recover modal formulas down the road. Either way, it will be easy to discern in the display of the terms.

Before moving on to temporal logic, we shall review how to construe modal notation in non-modal higher-order logic.

Lifting: second-order necessity

Here we examine an already-existing method for embedding modality in a non-modal higher-order language. The HOL system[7] has been host to embeddings of Lamport's Temporal Logic of Actions (TLA)[12]. Exposition may be found in [11, 17, 4]. Let us examine how their methods apply to the simpler modal logic we have been using.

As it was with the derivative,[2] we may also define

```
NEC(F) == λW'.Nec(W. F(W) ; W')
```

which may even be used in concert with the first-order necessity operator we have been discussing.[4] These two operators denote the same function, in a sense, by different methods. Their use differs by how one applies them in order to form propositions. Let us call this operator *second-order* necessity, since it is most likely to be useful in a higher-order language.

In the previous section, modal notation was explained by adopting notational conventions for eliding `theWorld` from the display of world-relative operators. Any other constants and operators were simply left intact with the usual meanings and notations. Observe, for example, that in
`¬[](1 = 1 ∨ (∃ x:Person.  x invented bifocals))` , the operators for negation, equality, existence, disjunction, and 1 are the ordinary operators, with no unusual interpretation.

In contrast, interpreting necessity as the second-order necessity operator `NEC(F)` entails interpreting the propositional formulas generally as *functions* on possible worlds. Adding an operator for bifocal inventing would go like this. Who there is, and bifocal inventing, are world dependent, so `PERSON ∈ PW⟶Type` is a simple function-valued constant, having no built-in argument places. The operator "`x INVENTED BIFOCALS` " has only the one built-in argument

[4] Again, we could have defined these operators in the other order: `Nec(W. Q(W) ; W') == NEC(λW.Q(W))(W')`

place, and for any *possible* person x, (x INVENTED BIFOCALS) ∈ PW→Prop so the possible world must be supplied through function application. Call such operators *second-order* world-relative.

Let us say a *modally second-order* formula is an expression of type PW→Prop , having *no* variables ranging over PW. So, just as expanding the definition of first-order necessity within a modally first-order formula destroys its status as such by forcing the use of two different PW variables, expanding the definition of second-order necessity within a modally second-order formula destroys is status as such by forcing the use of a binding PW variable.

Now we come to what may be the main drawback of this perfectly legitimate higher-order method of interpreting modality. In order to build complex formulas to which we shall apply modal necessity, we must *lift* all the non-world-relative operators of the language, which means to define variations of them that operate on functions from PW. For example, we must define a lifted version of negation, perhaps by,

```
NOT(F) == λW.¬F(W)
```

Then, by stipulating that the second-order modal operators, the lifted operators, and the remaining second-order world-relative operators are displayed the same as their modally propositional counterparts, we achieve yet another alternative explanation of standard modal notation. Then, NEC(NOT(x INVENTED BIFOCALS)) would display as []¬(x invented bifocals) . Again, this is essentially the method of [11, 17, 4].

Of course, in a higher-order language, both first-order and second-order necessity can coexist. If all the first-order world-relative operators have been supplied with second-order relative counterparts, and all the non-world-relative operators have lifted counterparts, then one can translate between the modally first- and second-order formulas. To translate a modally second-order formula, form its application to theWorld, then do the beta-conversions. Reverse the process to translate back.

So, if you want to use a first-order logic, or otherwise avoid lifting all your operators, then use first-order necessity.

Temporal Logic: addresses

Now we move to temporal logic, which is a modal logic where the possible worlds are times. The interesting additional feature is a class of "variables" whose values are taken relative to a state that varies with time. Let us use the terms *temporal* and *temporally* instead of *modal* and *modally*. Our purpose here will be to give an alternative explanation of temporal *notation* as a display form for *temporally first-order formulas*.

Let's take the first-order temporal logic given by Kröger[10] as our standard. Its temporal operators such as *nexttime*, *henceforth*, and *atnext* are applied to propositions, so we may say this is a *temporally propositional* logic. Time is taken to be the natural numbers, with successor being the next time.

Each "variable" is classified as "local" or "global." The globals are ordinary logical variables, but the locals cannot become bound, and their values are time-dependent. This is effected by a parameter to the semantics which is a function from time and locals to values. We shall avoid this nomenclature, reserving the term "variable" for logical variables susceptible of quantification-as-usual, and instead of "local variable" we shall say "address" in order to emphasize the connection to program state.

Let's begin. The type of addresses will be Addr, which we may assume here to be some class of identifiers. Address literals will be distinguished by some characteristic form such as addr{XYZ}, which we'll simply display as XYZ in the same way as done above for numeric literals. Here are our first-order temporal operators:

```
NextTime(t. P(t) ; r) == P(r+1)
HenceForth(t. P(t) ; r) == ∀n:{r...}. P(n)
AtNext(t. Q(t), P(t) ; r)
```

```
== ∀n:{r+1...}. (∀j:{r+1...}. Q(j) ⇒ n≤j) ⇒ Q(n) ⇒ P(n)
```

We define `{i...} == {j:ℤ| i≤j}` . The AtNext operator says that the next time, if any, that `Q` holds, then so does `P`. Also note that when the same variable binds in two subterms, we don't need to display the binding occurrence twice. Now we apply the display device for first-order modal operators to these first-order temporal operators, using **`theTime`** as the distinguished variable, getting these three standard temporal notations:

```
o<P> ; []<P> ; <P> atnext <Q>
```

Now for the time-dependent values at addresses. The type of state sequences, or *behaviors*, is `ℕ→Addr→T` for some type `T` of values. We define an operator for the value of an address `a`, at a time `t`, for a behavior `s` as `a{s@t} == s(t,a)` and we stipulate that the display of `<a>{theBeh@theTime}` when **`theBeh`** and **`theTime`** are implicit, shall just be `<a>`. (Of course, the user will need help from the editor to effectively use such "invisible" operators, to make their detection and manipulation easy.) Thus, the formula

```
∀x:T. x = a ⇒ o(x = a)
```

where `a` is an address literal, is just an informative display of

```
∀x:T. x = a{theBeh@theTime} ⇒
NextTime(theTime. x = a{theBeh@theTime} ;
         theTime)
```

This temporal formula is used by Kröger as part of a demonstration that you are not free to perform all-elimination on arbitrary expressions. In particular you must make sure that no address expressions are introduced into the scope of a temporal operator. One must not, say for address literal `b`, infer `b = a ⇒ o(b = a)` from `∀x:T. x = a ⇒ o(x = a)` .

The corresponding phenomenon in the temporally first-order interpretation, where one may of course do the substitution without such a restriction, is this:
we are actually substituting `b{theBeh@theTime}` for free `x`, which forces a change of bound variable in the first-order NextTime operator. So, the substitution works, but we have been kicked out of the temporal notation for trying to relate two Times, `t` and **`theTime`**, in the scope of the temporal operator, ending up with

```
b = a ⇒ NextTime(t. b = a{theBeh@t} ; theTime)
```

(Remember, **`theTime`** is implicitly attached to `b` here.)

Let us conclude with a remark about Lamport's TLA[12]. He separates it into the *simple* TLA and the *full* TLA. In [17] there is a straightforward treatment of the simple TLA in HOL using the higher-order method; there is quite a lot of operator lifting, and several kinds of it. With the first-order methods we have been describing, the simple TLA is quite tractable, and there is no lifting (although there is one coercion from "actions" to "formulas" of TLA). There is no space to give it here, but with a copy of Lamport's Figure 4 of [12] in hand, you may well find that our methods apply fairly directly, though non-trivially. Suggestions if you want to try it: Use three special variables, two for states and one for behaviors, say theState, theNextState, and theBeh. Use the TLA "prime" notation f' simply as a display form for "let theState = theNextState in f", where f may contain theState free. Let the notation be your guide.

The difficulty with extending our temporally first-order explanation of simple TLA to full TLA is that we treat state-dependent "variables" as addresses, and this doesn't help us to explain Lamport's use of these "variables" (which he calls *flexible variables*) for existential quantification, which is key to the full TLA.

Thanks

Bob Constable and Jim Caldwell discussed these ideas with me, and commented on drafts. I appreciate their generous efforts and the resulting improvements in my presentation. My thanks also to the referees, who provided useful criticisms.

References

[1] Martín Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. *Journal of the ACM*, 37(2):279–317, 1990.

[2] J. F. A. K. van Benthem. *The logic of time : a model-theoretic investigation into the varieties of temporal ontology and temporal discourse*, volume 156 of *Synthese library*. D. Reidel Pub. Co., Dordrecht, Holland, 1983.

[3] Brian F. Chellas. *Modal logic : an introduction*. Cambridge University Press, 1980.

[4] Ching-Tsun Chou. Predicates, temporal logic, and simulations. In *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, 1994.

[5] Michael Dummett. *Frege: philosophy of language, Second Edition*. Harvard University Press, 1981.

[6] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[7] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL*. University Press, Cambridge, 1993.

[8] Paul Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, January 1996.

[9] Saul Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[10] Fred Kröger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[11] Thomas Långbacka. A HOL formalisation of the temporal logic of actions. In *Higher Order Logic Theorem Proving and its Applications. 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[12] Leslie Lamport. The temporal logic of actions. Systems Research Center 79, Digital Equipment Corp., Palo Alto, CA, December 1991.

[13] Conal Mannion and Stuart Allen. A notation for computer aided mathematics. Technical report, Cornell University, Ithaca, NY, June 1994.

[14] Derek C. Oppen. Prettyprinting. *ACM Trans. on Prog. Lang. and Systems*, 2(4):465–483, October 1980.

[15] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, New York, third edition, 1988.

[16] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: a syntax-directed programming env ironment. *Comm. Assoc. Comput. Mach.*, 24(9):563–73, 1981.

[17] Joakim von Wright. Mechanising the temporal logic of actions in HOL. In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 155–159. IEEE Computer Society Press, 1992, 1992.

On the Spreadsheet Presentation of Proof Obligations

James H. Andrews

Dept. of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract. A compact and structured format for presenting proof obligations is described. The format places the formulas and proof obligations in the form of a spreadsheet, where rows are formulas, columns are obligations, and cells record whether and how a formula appears in an obligation. This spreadsheet presentation frees the proof system from some interface-related restrictions, and allows users to follow a more natural style of problem solving. It can be applied to either sequent or tableau logics, and can be used by most theorem proving systems. An initial implementation is discussed, some recommendations are made for future effort, and a graphical user interface design is proposed based on the spreadsheet model.

1 Introduction

When we want to prove a theorem using a proof assistant, we the users are faced with the task of processing a large amount of complex information. The single, small theorem we wanted to prove may expand, during the course of the proof, into many "proof obligations" – sequents we need to prove or tableau branches we need to close – each of which contains many formulas. Most proof assistants handle this complexity by displaying only the "current obligation", and allowing us to change to another current obligation if we want. However, we still often need to answer questions like:

- "How is this obligation different from the other one I just proved?"
- "Is this formula significant in this obligation?"
- "Does this formula appear in some other obligation?"
- "Where is that easy obligation that I wanted to prove first?"

These questions are difficult to answer with conventional presentations of the set of active proof obligations, when that set becomes large. Displaying all obligations one by one takes up too much of the screen or window, and swamps the user in details; but displaying only some obligations hides information which may be important.

The problem of displaying a collection of objects, each containing many subobjects, also comes up in the design of computer-based accounting packages (for instance, when financial data for several months contains data for several kinds of expenses). This problem was largely solved by Bricklin and Frankston's revolutionary 1978 program Visicalc [Nor83]. Since then, accounting packages have consistently displayed information in an electronic form of spreadsheet, in which information is presented in related rows and columns. The computer spreadsheet's strength is that it provides a "good division of labor between text and graphics [and also] a careful *integration* of text and graphics" [Nar93].

Why should a spreadsheet format be appropriate for proof obligations? The key observation here is that although we may have many obligations, each containing many formulas, the number of *distinct* formulas is often small. Consider an initial formula of the form $A \vee ((B \vee (D \& F)) \& (C \vee (E \& G)))$. The first steps of a proof of this formula in a sequent-based system can result in four sequents, each with three formulas each. These sequents would take at least $4 \cdot 3 = 12$ lines to display in a conventional proof assistant interface, and the underlying structure of the sequents would not be apparent. It is arguably more compact and better organized to present the sequents as:

	Sequent a	Sequent b	Sequent c	Sequent d
Formula A	•	•	•	•
Formula B	•		•	
Formula C		•		•
Formula D	•			
Formula E		•		
Formula F			•	
Formula G				•

Here, a bullet in a cell indicates that the formula is in the sequent, and a blank space indicates that the formula is not in the sequent. This simplified tabular format contains the kernel of the idea of the spreadsheet presentation.

The spreadsheet presentation condenses and correlates all the information relevant to a proof, with associated benefits. However, it can be less clear than conventional presentations in some respects, unless certain problems are avoided.

In this paper, I discuss the spreadsheet presentation in detail. First, I report on the implementation of it in the experimental proof assistant SVP, and describe the merits and deficiencies of spreadsheets in general and that implementation in particular. Next, I give some suggestions for spreadsheet presentation implementation when conventional scrolling user interfaces are used. I then describe a proposed design for a graphical user interface (GUI) based on the spreadsheet presentation, which could significantly increase the usability of proof assistants.

2 An Initial Implementation: SVP

It is perhaps easiest to introduce the spreadsheet presentation by describing an actual implementation. The proof assistant SVP is a system for proving properties of Prolog programs, based on the author's semantics for depth-first, left-to-right Prolog with ground negation [And91, And93, And97]. It uses a scrolling, ASCII command-loop interface. At every command which changes the state of

```
Stage 5, proof of "\s list(X) => \s append(X, [], X)"
-----+---------------------------------------------+---------------
     |                                             | Sequent
Fml# | Formula                                     | a  b  c  d
-----+---------------------------------------------+---------------
  1. | \s append([X_#1|Xs#2], [], [X_#1|Xs#2])     |    /  /  /
  2. | \s append([], [], [])                       |TC
  3. | \s append(‘Ind_X#3, [], ‘Ind_X#3)           |          \
  4. | \s list(‘Ind_X#3)                           |       /
  5. | \s list(Xs#2)                               |    \  \  \
  6. | \s tail([X_#1|Xs#2], ‘Ind_X#3)              |    /
  7. | \all Ind_X (\s tail([], Ind_X) => \s        | \
     | list(Ind_X) => \s append(Ind_X, [], Ind_X)) |
=====+=============================================+===============
 Sequent provable?...                              | Y  -  -  -
```

Fig. 1. A spreadsheet representing an example proof state.

the proof, the system automatically displays the spreadsheet which describes the current state. SVP is sequent-based, so every proof obligation is a sequent.

Stärk [Stä97] has written a proof system with similar capabilities, though a different interface style. Here I focus only on the aspects of SVP that are most relevant to discussion of spreadsheets.

2.1 The Layout of the SVP Spreadsheet

Figure 1 shows a spreadsheet from one stage of an actual inductive proof using SVP. The property being proved is that appending any list to the empty list results in the original list.

There are really two parts to an SVP spreadsheet: the spreadsheet proper (above the line of = characters) and the summary line (below the line of =). The spreadsheet proper has two large columns on the left and one more small column for each sequent. The second column contains the text of the formula; the first column contains a number associated with the formula, for reference purposes. If a formula does not fit within the formula column (formula 7 is an example), SVP displays it split over several lines at blank spaces.

Each sequent is given a unique, one- or two-letter name for reference purposes. These names appear at the top of the sequent columns. The spreadsheet cells appear at the intersection of every formula row and sequent column. The cell at the intersection of the row of formula ϕ and sequent σ contains a code signifying whether and how ϕ appears in σ. The most common cell codes are the following:

- Blank space: ϕ does not appear at all in σ.
- \\: ϕ appears on the left (as an assumption) in σ.

```
Stage 5, proof of "\s list(X) => \s append(X, [], X)"
-----+---------------------------------------------+--------------
     |                                             | Branch
Fml# | Formula                                     | a  b  c  d
-----+---------------------------------------------+--------------
  1. | \s append([X_#1|Xs#2], [], [X_#1|Xs#2])     |    -  -  -
  2. | \s append([], [], [])                       |T-
  3. | \s append(‘Ind_X#3, [], ‘Ind_X#3)           |          +
  4. | \s list(‘Ind_X#3)                           |       -
  5. | \s list(Xs#2)                               |    +  +  +
  6. | \s tail([X_#1|Xs#2], ‘Ind_X#3)              |    -
  7. | \all Ind_X (\s tail([], Ind_X) => \s        | +
     | list(Ind_X) => \s append(Ind_X, [], Ind_X)) |
=====+=============================================+==============
 Branch closable?...                               | Y  ?  ?  ?
```

Fig. 2. A tableau version of the spreadsheet in the previous example.

- /: ϕ appears on the right (as a conclusion) in σ.
- X: ϕ appears on both the left and the right of σ.

If a formula appears on both sides of a sequent, the sequent is considered proven. A sequent is also considered proven when a true formula, for instance $a = a$, appears on the conclusion side or a false formula appears on the assumption side. For these cases there are two other cell codes:

- TC: ϕ is a "true conclusion"; it appears on the right in σ and is true.
- FA: ϕ is a "false assumption"; it appears on the left in σ and is false.

The summary line ("Sequent provable?") comes below the spreadsheet proper. It uses the columns corresponding to the sequents, but ignores the two leftmost columns. This "bottom line" shows what the current provability status of each sequent is: whether it is provable (Y) or unprovable (N), or whether its provability is currently unknown by the system (-). The user's goal, of course, is to end up with a line of Ys on the summary line.

SVP is sequent-based, but we could use the spreadsheet presentation in tableau-based systems as well; all we would really need is a change of symbols. Figure 2 shows how the example spreadsheet from Figure 1 might look in a tableau-based system, where obligations are branches.

2.2 Types of Rules in SVP

The spreadsheet presentation informs the kinds of operations we can perform with SVP, and particularly the view of the sequent calculus proof rules. The logic of SVP is classical first order logic with equality, augmented by the unary connectives `\s` and `\f` for success and failure of Prolog goals. The proof rules of SVP are divided into three kinds: automatic, default and special.

The *automatic* rules include the rules for the propositional connectives, ∀ on the right, and ∃ on the left. These are the rules which can be applied without fear that they may lead away from a proof. The user can issue the command `auto` to ask the system to apply automatic rules until no more are applicable, or set a switch to ask the system to do an `auto` operation after every command. Note that with the spreadsheet presentation, the system can apply even the sequent-splitting rules for conjunction on the right or disjunction on the left automatically. This is typically not done in conventional proof assistants, because of the risk of producing a set of obligations too big for the user to handle. With the spreadsheet presentation, each sequent split simply adds a column to the sheet.

Some given types of formulas on a given side of a sequent have *default* rules associated with them – for instance, ∃ formulas on the right and ∀ formulas on the left. These are rules which may lead away from a proof, but are so commonly used that it is easier to have a single command for them. The user can ask the system to apply these default rules by issuing the command `def` n, where n is the spreadsheet number of the formula.

The *special* rules include the thinning, cut and induction rules. These are rules which may or may not apply to a single formula, but are used less frequently than the default rules or need extra arguments. The user can ask the system to apply these rules by special-purpose commands.

2.3 Working with the Spreadsheet

SVP displays the spreadsheet after every command, but it is always possible to get a conventional display of a sequent's contents in SVP by issuing the command `sequent(`*label*`)`, where *label* is the sequent's label. SVP keeps track of which sequents are provable, marking `Y` in the summary line when appropriate. The user can ask the system to hide any sequents which have already been proven. Formulas which appear only in hidden sequents are also hidden.

To prove a formula, we typically first indicate how to prove the formula by induction, a subject outside the scope of this paper. Our "only" task then is to enter commands, in the sequence that will lead to a proof, which apply default and special rules. I place "only" in quotation marks because the task is by no means trivial; however, it is still easier than having to select the sequence of applications of both the sequent-splitting rules and the default and special rules.

3 Strengths and Weaknesses

The spreadsheet presentation has some advantages over conventional presentations, but is not uniformly superior. In addition, the SVP implementation of spreadsheets can be criticized in some regards.

3.1 Strengths of Spreadsheets and SVP

The most obvious strength of the spreadsheet presentation is that the display of the current proof state is more compact and structured than in conventional presentations. This in turn means that it is easy to compare sequents directly, easy to find a formula of interest and the sequents it is in, and easy to find sequents with predicted properties (such as the base case of an inductive proof). This facilitates a natural, "opportunistic" style of problem solving in which the subproblem which the user believes to be easiest is solved first.

With spreadsheets, more rules are able to be applied automatically, as noted in Section 2.2. Also, rules which apply to a formula which appears in many sequents (such as a Prolog predicate unfolding, in SVP) need be applied only once. This cuts down on repetitive steps.

A more subtle advantage is that proofs are made linear by the spreadsheet presentation. Even steps that apply to only one sequent do not require a conceptual shift in direction; the spreadsheet still summarizes the state of the entire proof. For a typical size of proof, the previous sheet appears in its entirety immediately above the current sheet in the window in which the system is being run, so the user can compare the two states and see the effect of the last command. The user can back up to any previous state in SVP with the `backup` command.

3.2 Weaknesses of Spreadsheets and SVP

One inherent disadvantage of the spreadsheet presentation is that it is unfamiliar and not immediately obvious. A user must follow a certain learning curve to be able to read the spreadsheet easily.

In the SVP implementation, the rows and columns have a fixed ordering, so it is sometimes difficult to focus on a given sequent and pick out which formulas are its assumptions and conclusions. Also, the number of a formula in the sheet may change from stage to stage of the proof, forcing the user to look up a number for each command. These problems will be addressed in the next section.

4 Recommendations for Scrolling Interfaces

Most theorem provers are implemented at present using a scrolling, ASCII command-based interface. It seems clear that the spreadsheet presentation can be implemented, at least as a user alternative, in most such systems. Here I give some recommendations for implementing spreadsheets on such theorem provers, based on my experiences with SVP. I group these into recommendations for managing formulas, and those for managing obligations.

4.1 Managing Formulas

There are various ways in which the rows (formulas) of a proof obligation spreadsheet could be ordered. For instance:

- In the SVP order (by lexicographic order on internal representation).
- By order of first appearance of the formula. This would make a tableau appear very close to its traditional format.
- By lexicographic order on output format.
- By listing the assumptions of a given sequent first, then the conclusions, and sorting the rest by any of the above methods. This would allow the given sequent to be read easily.

We should probably have the ability to choose between these orderings.

Formulas should be numbered in such a way as to keep the numbers low, but to retain a unique number for each formula during the entire course of a proof. I suggest that the numbers reflect the order in which the formulas were first displayed, ignoring any formulas which have been created temporarily within a given step but never displayed.

Finally, in SVP, formulas are displayed within the formula column, split across lines at the most convenient blank. Some form of prettyprinting within the column seems to be more desirable.

4.2 Managing Obligations

SVP orders obligations by order of creation. This has not proven particularly useful. The only really useful ordering seems to be by order of position (left to right) in a traditional sequent proof or tableau. However, it is not clear that *labelling* the columns to reflect this ordering is useful. It may be more useful to label them according to their order of creation, as with formulas, so as to keep the labels small and still attach a permanent label to each obligation.

5 A Graphical User Interface

Here I give some recommendations on how a graphical user interface could be constructed to aid a spreadsheet proof system.

The window containing the spreadsheet should be laid out as in the scrolling interface, but with all formulas, column labels and cells responding to mouse clicks. In a drawing program, users are able to select from a set of "tools", such as a line-drawing tool or a box-drawing tool. The tools are displayed as icons which can be selected by the user. Similarly, in a spreadsheet-based proof assistant, the user should be able to select from a set of "formula tools" or "obligation tools". When a particular tool is selected, the user should be able to click on a particular formula, obligation or cell to apply the tool. Some possible formula tools are:

- Default: apply the default proof rule to the formula. This should be the default tool, as well.
- Thin: eliminate the formula.

Choosing a formula tool and then clicking on a cell should apply the appropriate operation to just that formula in that sequent. Clicking on a formula should apply the operation to all copies of the formula in all sequents.

Some possible obligation tools are:

- Sort: sort the formulas to make this obligation's assumptions and conclusions come first. This should be the default obligation tool.
- Hide: hide the obligation.
- Induction: apply induction to the obligation. This should cause a dialogue box to pop up to allow the user to enter further information.

It may be preferable for the user to do column operations by selecting the column first and then issuing a keystroke command or selecting from a menu.

6 Acknowledgments

I developed SVP as a result of discussions with Robert Stärk about his own proof assistant. My ideas on the spreadsheet presentation were further developed while working for Paul Gilmore, funded in part by his NSERC grant and in part by the FormalWare project, a joint industry-university research project funded by the BC Advanced Systems Institute, Hughes Aircraft Canada Ltd., and MacDonald Dettwiler, and headed by Jeff Joyce of Hughes Aircraft.

My thanks for valuable discussions and suggestions to Hassan Aït-kaci, Michael Donat, Paul Gilmore, and Alan Smaill.

References

[And91] James H. Andrews. *Logic Programming: Operational Semantics and Proof Theory.* Distinguished Dissertation Series. Cambridge University Press, March 1991.

[And93] James H. Andrews. A logical semantics for depth-first Prolog with ground negation. In *Proceedings of the International Logic Programming Symposium,* Vancouver, October 1993. MIT Press.

[And97] James H. Andrews. A logical semantics for depth-first Prolog with ground negation. *Theoretical Computer Science,* 184(1-2):105–143, September 1997.

[Nar93] Bonnie A. Nardi. *A Small Matter of Programming.* MIT Press, Cambridge, Mass., 1993.

[Nor83] Donald A. Norman. Cognitive engineering. In Donald A. Norman and Stephen W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction,* pages 31–61, Hillsdale, N.J., 1983. L. Erlbaum Associates.

[Stä97] R. F. Stärk. Formal verification of logic programs: foundations and implementation. In S. Adian and A. Nerode, editors, *Logical Foundations of Computer Science LFCS '97 — Logic at Yaroslavl,* pages 354–368. Springer-Verlag, Lecture Notes in Computer Science 1234, 1997.

This article was processed using the LaTeX macro package with LLNCS style

TAME: A PVS Interface to Simplify Proofs for Automata Models*

Myla Archer, Constance Heitmeyer, and Steve Sims
Code 5546, Naval Research Laboratory, Washington, DC 20375
{archer, heitmeyer, sims}@itd.nrl.navy.mil

Abstract

Although a number of mechanical provers have been introduced and applied widely by academic researchers, these provers are rarely used in the practical development of software. For mechanical provers to be used more widely in practice, two major barriers must be overcome. First, the languages provided by the mechanical provers for expressing the required system behavior must be more natural for software developers. Second, the reasoning steps supported by mechanical provers are usually at too low and detailed a level and therefore discourage use of the prover. To help remove these barriers, we are developing a system called TAME, a high-level user interface to PVS for specifying and proving properties of automata models. TAME provides both a standard specification format for automata models and numerous high-level proof steps appropriate for reasoning about automata models. In previous work, we have shown how TAME can be useful in proving properties about systems described as Lynch-Vaandrager Timed Automata models. TAME has the potential to be used as a PVS interface for other specification methods that are specialized to define automata models. This paper first describes recent improvements to TAME, and then presents our initial results in using TAME to provide theorem proving support for the SCR (Software Cost Reduction) requirements method, a method with a wide range of other mechanized support.

1 Introduction

A good theorem prover interface has several goals. At the first, most immediate level, an interface should be designed to make the customary interactions with the theorem prover as convenient as possible. For example, the interface should make it easy and convenient to display known facts (such as theories and lemmas) [19] and should provide automated support for selecting applicable proof rules [21, 20]. It should also facilitate the selection of proof rules, lemmas to apply, expressions inside a proof goal to be used as instantiations, and so on. At a second, somewhat deeper, level, an interface should provide supplementary services in the theorem prover itself. For example, a user should be able to add annotations to proofs [13] or to obtain human-understandable proof scripts [5]. At a third, even deeper, level, an interface should provide derived proof rules that allow the user to reason in familiar ways, e.g., using his favorite logic and syntax. Isabelle [15, 14] is designed particularly to support such interfaces, but any "programmable" prover can support them. For example, a proof assistant for the duration calculus has been built on top of PVS [18].

We are developing a tool called TAME (Timed Automata Modeling Environment) [1, 3, 2] that provides interface features at both the second and third levels for PVS (Prototype Verification System) [17]. In particular, TAME supports reasoning about automata models by providing specialized PVS proof steps that are appropriate for proving properties of such models and that automatically annotate both proof goals and saved proofs with meaningful labels and comments. Currently, TAME is comprised of these specialized proof steps together with a set of standard theories and automata templates upon which the steps rely, and in itself has no interface features at the first level. However, we have recently investigated the integration of TAME into a set of tools called the SCR toolset [9, 6, 7]. The SCR tools are designed to support editing and performing various kinds of analysis on requirements specifications of control system software. Once TAME has been fully integrated into the SCR toolset, the user who wishes to apply TAME to an SCR specification will have first-level interface support, including the extensive interface support already provided by the SCR tools. Ultimately, we also plan to provide direct first-level interface support for TAME.

Our goals in developing TAME are somewhat different from the goals of the developers of the duration calculus assistant. Rather than supporting a particular logic, TAME supports proof steps "natural" to humans reasoning about automata models [2]. To the extent feasible, TAME hides the the raw PVS logic from the user who is proving properties of automata, since interacting with PVS in its "raw" form has its difficulties. For example:

- Formula numbers may be needed, e.g., in skolemization or instantiation; this can lead to non-portability of proofs or difficulty in defining generic strategies.
- Rather obscure proof steps, e.g., the APPLY-EXTENSIONALITY rule, are occasionally needed to perform reasoning that is "obvious" to a human.

*This work is funded by the Office of Naval Research. URLs for the authors are
http://www.itd.nrl.navy.mil/ITD/5540/personnel/{archer,heitmeyer}.html

- Quantified formulae may need as instantiations complex, sometimes multi-line, expressions that are cumbersome to supply using the standard PVS interface.
- Proofs are often structured more for the needs of the prover than for the needs of the human constructing the proof. For example, splitting an hypothesis can produce several proof goals whose significance is hard to understand, and require the user to provide obscure proof steps. This is particularly a problem when, as often happens, the split hypothesis is an implication and the user's arrival at the later proof goal or goals is delayed in time.
- Many proof steps provided by raw PVS are either too small or too large when compared with reasoning steps natural to a human. Proofs that require many small steps become quite tedious. When a large step involving the PVS decision procedures is used in a proof, the significance of resulting subgoals (if the step does not complete the proof) is obscure.

The problems listed above are not for the most part peculiar to PVS. They or their analogues are likely to arise with any mechanical theorem prover. However, these difficulties demonstrate why following a more natural style of reasoning in PVS requires improvements to the PVS interface.

As indicated in [2], we have successfully defined and applied TAME proof steps which are more "natural" for humans than those provided directly by PVS. An important reason for our success is that TAME is an interface specialized for particular proof styles and particular models. However, certain barriers prevented us from doing more. Recent enhancements to PVS, soon to be incorporated in the general PVS release [16], remove some of these barriers allow more sophisticated proof steps that avoid the difficulties listed above. The enhancements allow us to provide more information to the user about the proof goals during the course of a proof and to automatically provide better documentation in PVS proof scripts.

We believe that specialized interfaces such as TAME will encourage the more widespread use of theorem provers in practical software development. Our experience in developing TAME suggests that adding certain capabilities to existing general theorem proving systems can encourage the development of specialized interfaces for these provers.

The remainder of this paper is organized as follows. Section 2 reviews PVS, timed and non-timed automata, TAME, and SCR. Section 3 describes in detail the improvements to TAME made possible by the PVS enhancements. Section 4 provides examples that illustrate the new features of TAME. Section 5 discusses how TAME has been customized to support SCR automata models, and the progress we have made in integrating TAME into the SCR toolset. Finally, Section 6 discusses some issues that arise in developing specialized interfaces for PVS and other provers and our future plans for TAME.

2 Background

2.1 PVS

PVS [17] is a higher order logic specification and verification environment developed by SRI. Proof steps in PVS are either *primitive* steps or *strategies* defined using primitive proof steps, applicative Lisp code, and other strategies. Strategies may be built-in or user-defined. PVS's support for user-defined strategies makes it possible to implement specialized prover interfaces such as TAME on top of PVS. Recently, several enhancements to PVS have been developed at SRI. These include new features better supporting specialized interfaces. The new features include support for labeling formulae appearing in proof goals, support for documenting proof structure and proof steps (both interactively and in proof scripts) through comments, and the availability of certain fine-grained proof steps. The addition of some new access functions and documentation allows computations based on the internal data structures maintained by PVS. These enhancements, which will ultimately become standard features of PVS, have led to major improvements in TAME.

2.2 LV Timed Automata and IO Automata

An LV timed automaton is a very general automaton, i.e., a labeled transition system that incorporates the notions of current time and timed transitions. An automaton need not be finite-state: for example, the state can contain real-valued information, such as the current time, the water level in a boiler, velocity and acceleration of a train, and so on. LV timed automata can have nondeterministic transitions; this is particularly useful for describing how real-world quantities change as time passes, given upper bounds on their rate of change.

The following definition of timed automaton, based on the definitions in [8], was used in our case study of a deterministic timed automaton [1]. A *timed automaton* A consists of five components: (1) *states*(A), a (finite or infinite) set of states, (2) *start*(A) $\subseteq$ *states*(A), a nonempty (finite or infinite) set of initial states, (3) a mapping *now* from *states*(A) to $R^{\geq 0}$, the non-negative real numbers, (4) *acts*(A), a set of actions (or events), which include special *time-passage* actions $\nu(\Delta t)$, where Δt is a positive real number, and *non-time-passage* actions, and (5) *steps*(A) : *states*(A) $\times$ *acts*(A) $\rightarrow$ *states*(A), a partial function that defines the possible steps (i.e., transitions). This definition describes a special case of LV timed automata that requires the next-state relation, *steps*(A), to be a function. Careful use of the Hilbert choice operator ϵ, allows us to use the same basic definition in the nondeterministic case as well [3]. The definition of IO automata is similar, except that it has no references to time.

Actions (or events) may at any point be enabled or disabled. The typical specification of an LV timed automaton or an IO automaton describes the actions in terms of preconditions under which they are enabled, and effects on the state. Below, we refer to these preconditions as *specific* preconditions; other, uniformly applied components of the full precondition may also exist, such as general timing constraints in a timed automaton. The *transitions* (or steps) of an automaton correspond to the state changes induced by enabled actions. The *reachable states* of an automaton are those states that can be reached from an initial state via a sequence of zero or more transitions.

The properties of automata that one wants to prove fall into three classes: (1) state invariants, i.e., properties of all reachable states, which are typically proved by induction; (2) simulation relations; and (3) ad hoc properties of certain execution sequences. Proofs in both (1) and (2) have a standard structure with a base case involving initial states and a case for each possible action. They are thus especially good targets for mechanization. The proof examples in this paper all fall into class (1).

2.3 TAME

TAME provides a standard template for specifying automata, a set of standard theories, and a set of standard PVS strategies. The TAME template, originally intended for specifying LV timed automata, provides a standard organization for defining an automaton. To define either a timed or non-timed automaton, the user supplies the following six components: (1) declarations of the non-time actions, (2) a type for the "basic state" (usually a record type) representing the state variables, (3) any arbitrary state predicate that restricts the set of states (the default is **true**), (4) the preconditions for all transitions, (5) the effects of all transitions, and (6) the set of initial states. The user may optionally supply declarations of important constants, an axiom listing any relations assumed among the constants, and any additional declarations or axioms desired.

To support mechanical reasoning about automata using proof steps that mimic human proof steps, we have constructed a set of standard strategies using PVS, and included these as part of TAME. These strategies are based on the set of standard theories and certain template conventions. For example, the induction strategy, which is used to prove state invariants, is based on a standard automaton theory called **machine**. To reason about the arithmetic of time, we have developed a special theory called **time_thy** and an associated simplification strategy called **TIME_ETC_SIMP** for time values that are either non-negative real values or ∞. The important template conventions include a standard naming scheme and a standard format for lemmas of certain classes, such as state invariant lemmas. A more detailed description of TAME in its original form is available in [1]. Some enhancements to TAME in support of the verification of hybrid automata are described in [3].

Using the recent enhancements to PVS described in Section 2.1, we have simplified the original set of TAME strategies and provided several new strategies. In addition, progress has been made towards making both proof scripts and interactive proof goals "literate". Section 3 describes the details of these improvements to TAME, and Section 4 provides examples of their use.

Although TAME was originally developed for reasoning about LV timed automata, it is equally useful for non-timed IO automata [4] and easily adapted to many other automaton models. We have begun to apply TAME to SCR automata (see Section 2.4). For this new application, a slight modification to the TAME template has proved important for proof efficiency. Additional strategies that complete many state invariant proofs totally automatically seem possible. Section 5 presents the results of our initial experiments.

2.4 The SCR Requirements Method and Toolset

The SCR (Software Cost Reduction) requirements method is a formal method for specifying and analyzing the requirements of safety-critical control systems. Since its introduction in 1980 [10], SCR has been applied successfully to a wide range of critical systems, including avionics systems, space systems, telephone networks, and control systems for nuclear power plants. A set of software tools, called SCR* [9, 6, 7], has been constructed to support the SCR method. In addition to a *specification editor* for creating a specification and a *dependency graph browser* to display the dependencies among the variables in the specification, the toolset includes an automated *consistency checker* to detect type errors, missing cases, circular definitions, and other types of application-independent errors, a *simulator* to allow users to symbolically execute the specification to ensure that it captures their intent, and an interface to a model checker called Spin [11, 12] that detects certain safety property violations.

An SCR requirements specification describes both the required system behavior and the system environment in terms of *monitored variables*, quantities that the system monitors, and *controlled variables*, quantities that the system controls. To specify the required behavior concisely, SCR specifications use two types of auxiliary variables, *mode classes* and *terms*. Mode classes, whose values are called *system modes* (or simply *modes*), capture historical information, whereas terms have very general utility.

SCR requirements specifications contain a set of dictionaries and a set of tables. The dictionaries, which contain the static information in the specification, include a *variable dictionary*, which lists the name, data type, and initial value of each variable; the *type dictionary*, which provides the data type definitions; the *constant dictionary*, which defines the names and values of constants; the *specification assertion dictionary*, which contains statements of properties such as state invariants; and the *environmental assertion dictionary*, which describes constraints on the behavior of the monitored variables. For every variable other than a monitored variable, there is a corresponding *condition*, *event*, or *mode transition* table. Each table defines a mathematical function called a *table function*. For example, an event table describes the (post-transition) value of a controlled variable or term as a function of a mode and an event. The notation "`@T(c)`" denotes an *event*, defined as `@T(c)` $= \neg c \wedge c'$, where the unprimed condition c is evaluated in the *current* state, and the primed condition c' is evaluated in the *new* state. Informally, "`@T(c)`" means that condition c becomes true.

To provide formal underpinnings for the SCR specifications, a formal model has been developed [9]. In the SCR model, the system is represented as a state machine that begins execution in some initial state and then responds to a sequence of input events, where an *input event* is an event that signals a change in some monitored variable. In particular, a system Σ is represented as a 4-tuple, $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is a function that maps an input event and the current state to a new state. The transform T is obtained from an SCR specification as the composition of the table functions. For T to be well-defined, the "direct" dependencies in the specification of a given variable in the new state on other variables in the new state must define a partial order. In SCR*, this partial order is verified to exist by the consistency checker, and is represented in the dependency graph browser as the *new state dependency graph*.

3 Recent Improvements to TAME

3.1 An Improved Induction Strategy

The induction strategy is the major TAME strategy for proving state invariants. This strategy sets up an induction proof for a state invariant by breaking the proof up into a base case and induction steps for the transitions, one for each kind of action. In the case where the invariant involves quantification over variables other than the automaton state, it is frequently the case that one wants to skolemize these variables in the inductive conclusion, and instantiate them in the inductive hypothesis with the resulting skolem constants. After some standard simplification on each branch, the induction strategy incorporates this skolemize-instantiate step on each induction branch of the proof. It then probes the branch to see whether a simple decomposition into cases followed by the application of the arithmetic, propositional logic, and other decision procedures of PVS will complete the proof of that branch—i.e., it checks to see if the branch is "trivial". If so, the branch is proved; otherwise, the branch is unchanged. The induction strategy then returns the unproved branches to the user.

The induction strategy was an appropriate candidate for improvement using several of the PVS enhancements. Originally, an appropriate variant for each specific automaton required external compilation. Multiple versions were needed when there were state invariant lemmas both with and without quantification of non-state variables. Standard simplification of the branches expands certain definitions, including the automaton's transition function. However, PVS's proof rule EXPAND, normally used for this purpose, does not simply expand a definition, but does some additional steps that often, though not invariably, produce a desirable simplification. One case where there may not be a desirable simplification is when the expanded definition contains an IF-THEN-ELSE expression under a quantifier. In this case, the additional steps result in lifting the IF-THEN-ELSE to the top level, with the quantifier appearing in both the THEN and ELSE branches. When this happens in expanding the transition function in the inductive conclusion, performing the skolemize-instantiate step is difficult. Using REWRITE in place of EXPAND was one solution, but not completely satisfactory because REWRITE, too, comes with baggage, and a resulting loss in efficiency. In any case, automated support for the skolemize-instantiate step had to rely on knowing the exact formula numbers of the inductive conclusion and the inductive hypothesis. Finally, because the induction strategy only produces subgoals for the nontrivial proof branches, it was difficult to examine a saved proof and determine the correspondence of branches to cases. Knowing this correspondence is important in several contexts, including the case when one wants to go back and complete a partial proof. Previously, labeling the branches was best done interactively using PVS's original, rather primitive, comment facility: incorporating a comment as an extra argument in an APPLY.

The improved induction strategy, which we call AUTO_INDUCT, avoids these problems. It need not be compiled; rather, it probes the body of the state invariant lemma being proved to retrieve data structures from which the necessary proof steps can be computed. This information includes the list of actions, with their arguments, that correspond to the transitions of the subject automaton. The new induction strategy then computes the strategy that previously was compiled externally, and applies it. Part of the computed strategy is a call to another strategy used in simplifying the proof branches. At the point where this auxiliary strategy is called, there is enough information in the current proof goal to determine whether and how to do any coordinated skolemization-instantiation. The auxiliary strategy computes a sequence of steps to accomplish this, and then applies them.

The improvements in the induction strategy described above rely on the additional documentation and access functions for the PVS internals. Eliminating the remaining problems in the original induction strategy required several of the PVS enhancements. The problem with EXPAND was solved by using one of the new finer-grained steps that simply expands a definition, and does no more. The problem of knowing the locations of the inductive conclusion and inductive hypothesis was solved by using the new labeling capabilities in PVS: AUTO_INDUCT labels various parts of an induction goal as "inductive-conclusion", "inductive-hypothesis", "specific-precondition", and so on. The auxiliary strategy called by AUTO_INDUCT can then skolemize "inductive-conclusion" and instantiate "inductive-hypothesis". The new comment facility is used by AUTO_INDUCT to automatically label proof branches with their corresponding cases. An example proof goal with labels and comments is given in Section 4.

3.2 Other Improvements in the TAME Strategy Set

With the combination of tools to access and analyze PVS sequents, formulae, and expressions plus the enhancements to PVS, we have succeeded in implementing many of the improved or new TAME strategies discussed in [2]. The resulting TAME strategies are designed for simplicity of use. We list a sample below. We illustrate the usefulness of most of these examples in Section 4.

- The *invariant-lemma strategy* for applying previously proved state invariant lemmas in a proof formerly had two versions; it now has a single version called APPLY_INV_LEMMA. Its first argument is the name of a state invariant. It checks to see whether its second argument (if any) is a state, and if so, applies the corresponding state invariant lemma to that state. All remaining arguments except the state argument (if there is one) are used as instantiations to any top level universal quantifier in the invariant lemma. APPLY_INV_LEMMA also displays the invariant being applied as a comment.
- The *precondition-strategy* for introducing the explicit form of the precondition for the action of an inductive step into the hypotheses of the current proof goal is now called APPLY_SPECIFIC_PRECOND. It now provides two additional services. First, it displays this explicit form of the precondition as a comment in the proof. Second, if the precondition is a conjunction, APPLY_SPECIFIC_PRECOND separates the conjunct conditions into a list, and gives them additional labels of the form *specific-precondition_part_i*, where the index *i* indicates their original position in the conjunction.
- The strategy TIME_ETC_SIMP, intended to complete the proof in a branch when a human might say "it is now obvious", has been replaced by the strategy TRY_SIMP. TRY_SIMP first hides formulae containing quantifiers, applies TIME_ETC_SIMP, and if the current subgoal is not proved, reveals the formulae it hid and applies TIME_ETC_SIMP again. Experimentation has shown TRY_SIMP to be frequently more efficient than TIME_ETC_SIMP, and never measurably less efficient.
- The strategy USE_EPSILON simplifies the application of the Hilbert ϵ-axiom. We have found USE_EPSILON useful in establishing certain properties of nondeterministic automata (e.g., hybrid automata in which there are tolerances in the amount of change in some state variables in a time-passage step; see [3]). Being able to analyze formulae and expressions in the proof goal with respect to content and type has enabled us to simplify the application of USE_EPSILON so that the user need no longer supply the domain type of the predicate to which the axiom is being applied, nor the full, sometimes complex, expression representing the predicate. Rather, the user need now only supply a hint as to which predicate to choose.
- The strategy DISCHARGE_HYPOTHESES is new. It removes hypotheses from any implications in the antecedent of a proof goal that can be deduced from the contents of the goal. While the standard PVS strategy ASSERT can sometimes be used to this purpose, it does not work if the hypotheses are of a certain complexity. DISCHARGE_HYPOTHESES can be used to avoid unnecessary case splits in the course of a proof.

Section 5 discusses a few new strategies useful in proving properties of SCR specifications.

3.3 Improvements in the Literacy of TAME Proofs

With the current set of TAME strategies, it is possible to maintain meaningful (sometimes multiple) labels on most if not all of the formulae in a proof goal. We have found labels to be useful not only in the support of more intelligent strategies, but in the course of an interactive proof.

Labels can serve as reminders of the source of a formula, and therefore aid in determining what point in reasoning has been reached in the proof, and in choosing the appropriate steps to take next. For example, we have replaced PVS's CASE rule with the TAME rule SUPPOSE. The only difference between the two rules is that SUPPOSE labels the introduced assumption *Suppose*, and in the companion proof branch in which the assumption must be proved, *Suppose not*. This documents the intended purposes of the formulae introduced.

Labels can also be used to increase proof portability. For example, the PVS proof rule INST ("instantiate") frequently must specify the formula to instantiate, and previously, this could only be done by giving the formula number. We note that maintaining unique labels for individual formulae can be important for an analogous reason.

The current set of TAME strategies also makes liberal use of comments. As noted in Sections 3.1 and 3.2, our strategy AUTO_INDUCT labels proof branches with comments, and the strategies APPLY_INV_LEMMA and APPLY_SPECIFIC_PRECOND introduce comments that spell out the actual facts being introduced. Several other TAME strategies do this also. This information can be useful interactively in noting how the facts have been simplified. One place where it is particularly useful in proof scripts is in clarifying the effects of certain later proof steps—e.g., in determining exactly which general fact has been instantiated. Without also incorporating the sequent at the beginning of each proof branch as a comment, there is not yet enough information in TAME scripts to completely follow the reasoning in a proof. Exactly how much information to incorporate into a proof script is an open question, and is largely a matter of taste.

4 Examples

Figure 1 shows an example of how labels are used in induction case proof goals of TAME induction proofs. Note that the inductive conclusion has been "flattened" into three parts.

```
lemma_6_1.1 :
;;;Case nu(timeof_action)
 {-1,pre-state-reachable}
   reachable(prestate)
 {-2,inductive-hypothesis}
   (((trains_part(basic(prestate))(r_theorem) = P)
        & ((gate_part(basic(prestate)) = fully_up) OR (gate_part(basic(prestate)) = going_up)))
       => first(prestate)(enterI(r_theorem)) > now(prestate) + gamma_down)
 {-3,general-precondition}
   enabled_general(nu(timeof_action), prestate)
 {-4,specific-precondition}
   enabled_specific(nu(timeof_action), prestate)
 {-5,post-state-reachable}
   reachable(prestate WITH [now := now(prestate) + timeof_action])
 {-6,inductive-conclusion_part_1,inductive-conclusion}
   (trains_part(basic(prestate))(r_theorem) = P)
 {-7,inductive-conclusion_part_2,inductive-conclusion}
   ((gate_part(basic(prestate)) = fully_up) OR (gate_part(basic(prestate)) = going_up))
  |-------
 {1,inductive-conclusion_part_3,inductive-conclusion}
   first(prestate)(enterI(r_theorem)) > now(prestate) + timeof_action + gamma_down
```

Figure 1. Example of a labeled proof subgoal returned by AUTO_INDUCT.

The changes in the TAME proof of the subgoal in Figure 1 are a good example of the improvements now possible in TAME proof scripts. The proof of this subgoal now has only six steps where it previously had nine. The first step, an application of APPLY_SPECIFIC_PRECOND, results in an expansion and decomposition of the formula labeled *specific-precondition* into six individually labeled parts and the inclusion of a corresponding comment into the proof script. The next two steps are calls to APPLY_INV_LEMMA; previously one of these steps invoked the special version of the invariant-lemma strategy for universally quantified invariants. The next step, DISCHARGE_HYPOTHESES, now accomplishes simplifications that previously required three steps, two of which relied on formula numbers. Next, the step (INST "specific-precondition_part_5" "r_theorem") replaces an INST step that required a formula number. The comment introduced by APPLY_SPECIFIC_PRECOND allows one to see from the proof script exactly which formula is instantiated by INST. Finally, the step TRY_SIMP now replaces two steps: a HIDE step that relies on a formula number, and a step TIME_ETC_SIMP.

5 Integrating TAME with SCR*

Currently, SCR specifications define deterministic automata, so their transitions can be represented by a function. Because this function is computed from many incremental updates in the state variables, proofs that use the function representation for the transitions of an SCR machine are inefficient. In fact, dramatic increases in the speed of proofs have resulted from representing transitions by a relation rather than a function, in contrast with our experience with LV timed automata in [3]. Therefore, we adapted the TAME template, supporting theory **machine**, and induction strategy to accomodate this variation.

Translating SCR specifications into TAME specifications is straightforward. TAME specifications are simply PVS specifications with a special structure. Information in the type dictionary, constant dictionary, and variable dictionary of an SCR specification is translated into PVS type, constant, and variable declarations and the initial state predicate. The monitored and controlled variables, terms, and mode classes become the "basic" state variables (there are additional standard state variables reserved for encoding timing information). The "actions" of an SCR automaton are input events, which represent some change in a monitored quantity. Such actions have as a parameter the new value of the monitored quantity. The environmental assertion dictionary contains all information about constraints on how monitored quantities can change, and this information is translated into preconditions on the actions. The assertions in the specification assertion dictionary are translated into invariant lemmas in TAME format. Finally, the definition of the transition relation is obtained using the event, condition, and mode transition tables and the new state dependency graph, and is represented in terms of "update" functions for variables that are affected by a given action (input event). The update function for a variable is a translation of its table, based upon translations of predicates and expressions in the table that are indexed numerically in the natural way. This translation scheme has been automated for a significant subset of the SCR specification language. Figure 3 shows part of the automated translation of the event table for the state variable ("term") `Overridden` shown in Figure 2, and how the corresponding update function for `Overridden` is used in the definition of the transition relation.

Experimentation with proving state and transition invariants for several example SCR specifications strongly suggests that there exists a uniform strategy that, for typical SCR applications, will suffice to prove many invariants automatically. We are continuing experiments aimed at the development of such a strategy.

Mode	Events	
`High`	False	@T(Pressure = TooLow OR Pressure = Permitted)
`TooLow`, `Permitted`	@T(`Block=On`) WHEN `Reset=Off`	@T(Pressure = High) OR @T(`Reset=On`)
`Overridden`	True	False

Figure 2. Event table for `Overridden`.

```
statepred : TYPE = [ states -> bool ];
atT(P : statepred, s1,s2 : states):bool = P(s2) AND NOT P(s1);
Overridden_0_0(s1,s2 : states) : bool = FALSE;
Overridden_0_1(s1,s2 : states): bool =
 let e0 = (LAMBDA (s : states): Pressure(s) = TooLow OR Pressure(s) = Permitted) in atT(e0, s1, s2);
 ...
Overridden_assignment_0 (s1, s2 : states) : bool = TRUE;
Overridden_assignment_1 (s1, s2 : states) : bool = FALSE;
update_Overridden(s1,s2 : states) : bool =
 IF Pressure(s1) = High
  THEN IF Overridden_0_0(s1,s2) THEN Overridden_assignment_0(s1,s2)
    ELSIF Overridden_0_1(s1,s2) THEN Overridden_assignment_1(s1,s2)
    ELSE Overridden(s1)
  ENDIF
 ELSIF
   ...
 ELSE Overridden(s1)
 ENDIF;
trans(s_old : states, A : actions, s_new : states) : bool =
 s_new = CASES A OF Block(Block_value): s_old WITH [basic := basic(s_old) WITH
                            [Block_part := Block_value,
                             Overridden_part := update_Overridden(s_old,s_new),
                             SafetyInjection_part := update_SafetyInjection(s_old,s_new)]],
            ...
          ENDCASES;
```

Figure 3. Fragments of TAME translation of `Overridden` event table and the transition relation.

6 Discussion and Future Plans

Providing an interface such as TAME for a theorem proving system requires several capabilities within, or in relation to, that system. These include support for user-defined strategies or tactics and access to the data structures and some of the internal analysis tools used by the proof engine. TAME also relies on the term rewriting facilities in PVS. For some provers, it may be necessary to add capabilities: for PVS, we required additional low-level proof steps plus labeling and comment facilities.

Capabilities needed to provide what we call first-level interface support for a prover were noted in [21]. These include formal languages and parsers for the data structures used in the prover and its outputs, and a means to communicate changes of state between the prover and the interface. As indicated in the introduction, we plan to provide first-level interface support for TAME and to support the use of TAME through the SCR* toolset interface. To do so, we will need some additional capabilities along these lines.

With the capabilities we now have, we expect to improve our PVS interface further by providing strategies for PVS steps that we noted in [2] as potentially useful, e.g., in avoiding unnecessary case splitting. Examples are strategies for the skolemization and instantiation of embedded quantified formulae. The Lisp code needed in the strategies for analyzing formulae would almost certainly be simpler to create given access to analysis tools such as parsers already present in PVS.

Acknowledgments

We wish to thank our colleague Ralph Jeffords for helpful discussions about proof efficiency and SCR specifications, and our colleague Ramesh Bharadwaj for insightful comments on an earlier version of this paper. We also thank Natarajan Shankar and Sam Owre of SRI International, who implemented the enhancements to PVS that allowed us to improve TAME.

References

[1] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*. IEEE Computer Society Press, 1996.

[2] M. Archer and C. Heitmeyer. Human-style theorem proving using PVS. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 33–48. Springer-Verlag, 1997.

[3] M. Archer and C. Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.

[4] S. J. Garland, N. A. Lynch, and M. Vaziri. IOA: A Language for Specifying, Programming, and Validating Distributed Systems. Draft. MIT Laboratory for Computer Science, December, 1997.

[5] J. Harrison. A Mizar mode for HOL. In *Proc. 9th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 203–220. Springer-Verlag, 1996.

[6] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 109–122, Gaithersburg, MD, June 1995.

[7] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.

[8] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, Dec. 1994.

[9] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[10] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, Jan. 1980.

[11] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997.

[13] S. Kalvala. Annotations in formal specifications and proofs. *Formal Methods in System Design*, 5(1/2), 1994.

[14] S. Kalvala. A formulation of TLA in Isabelle. In *Higher Order Logic Theorem Proving and Its Applications (HOL'95)*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 46–57. Springer-Verlag, 1995.

[15] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[16] J. Rushby. Private communication. NRL, Jan. 1997.

[17] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.

[18] J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems, Lect. Notes in Comp. Sci. 863*. Springer-Verlag, 1994.

[19] D. Syme. A new interface for HOL – ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications (HOL'95)*, volume 971 of *Lect. Notes in Comp. Sci.*, pages 324–339. Springer-Verlag, 1995.

[20] L. Théry. A proof development system for the HOL theorem prover. In *Higher Order Logic Theorem Proving and Its Applications (HUG'93)*, volume 780 of *Lect. Notes in Comp. Sci.*, pages 115–128. Springer-Verlag, 1993.

[21] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. *Proc. Fifth ACM SIGSOFT Symp. on Software Development Environments, Software Engineering Notes*, 17(5), 1992.

Extracting Proofs from Documents

Roland Backhouse and Richard Verhoeven*
Department of Mathematics and Computing Science
Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
Email: {rolandb,river}@win.tue.nl

March 27, 1998

Abstract

Often, theorem checkers like PVS are used to check an existing proof, which is part of some document. Since there is a large difference between the notations used in the documents and the notations used in the theorem checkers, it is usually a laborious task to convert an existing proof into a format which can be checked by a machine. In the system that we propose, the author is assisted in the process of converting an existing proof into the PVS language and having it checked by PVS.

1 Introduction

The now-classic ALGOL 60 report [5] recognized three different levels of language: a reference language, a publication language and several hardware representations, whereby the publication language was intended to admit variations on the reference language and was to be used for stating and communicating processes. The importance of publication language —often referred to nowadays as "pseudo-code"— is difficult to exaggerate since a publication language is the most effective way of communicating among *people*. Unfortunately, the example of ALGOL 60 has not been followed in the ensuing thirty years. Modern programming languages (C++, Java, Haskell etc.) do not admit variations. Worse still they are based on the ASCII standard which itself is based on teletype technology.

The 30-year tyranny of teletype technology also extends to theorem proving systems which, like programming languages, are typically ASCII-based and do not admit notational variation. It is no wonder that theorem provers are hard to use, even for checking theorems or programs that are already known to be correct.

In the recently begun research project entitled "publication-style programming" we are aiming to contribute to the development of systems in which programs and specifications are written in a publication language and then transformed to the hardware representation required by existing implementations of compilers and theorem provers. Such a system will allow programmers to write programs and specifications in application-specific (mathematical) languages, thus facilitating communication with the client and reducing the risk of misunderstandings that can bedevil large-scale projects. A similar project (on a much larger scale) is the Intentional Programming project being conducted by Microsoft under the direction of Charles Simonyi. Ultimately, as argued by Simonyi [13], such systems will turn programmers into language designers.

The basis for our own work in this direction is the Math∫pad editor [3, 2] which we have developed as a tool to assist those writing articles containing a large amount of advanced mathematics. The priorities set for the Math∫pad editor were readability of on-screen documents (thus enabling the use of sophisticated two-dimensional mathematical notation), flexibility (no built-in notational

*Research supported by the Dutch Organisation for Scientific Research (NWO) under contract SION 612-14-001

conventions but, instead, a flexible system for on-the-fly, user-supplied notational definitions) and writability (the ability to mix prose and mathematics freely in the text). Our next priority — the subject of the current project— is to explore architectures that will enable the coupling of Math∫pad, or a similar system, to compilers and theorem provers.

The recent Ph.D. thesis by Matteo Vaccari [14] is illustrative of what we ultimately want to achieve. In his thesis, Vaccari discusses the calculational construction of hardware circuits, where the first 6 chapters contain theoretical discussions of relation algebra, circuits and regular language recognizers, while the later chapters contain simulations of circuits using Ruby [11] and Tangram [10] and a machine verification of the theory using PVS [7]. Vaccari used Math∫pad in the process of developing and documenting the "theoretical" designs in the initial chapters, and then hand-coded these into the forms acceptable to Ruby, Tangram and PVS. Inevitably there is a lot of duplication of information since each system requires its own format. Moreover, the manual transformation from one format to another is a source of errors and adjustments in one format requires the adjustment to the other formats in a similar way. Since each system uses its own format and files, incorporating those files in the main document is often clumsy and also error sensitive. Furthermore, the user has to learn the different formats and systems, which takes a lot of time. It would be better if all interaction could be directed with one interface and one format, which takes care of updating and manipulating all the files needed by the different systems.

In an initial investigation [9] we have looked into linking Math∫pad with Jones' Gofer interpreter with the goal of allowing publication-style development of functional programs. The current paper is a preliminary investigation of the problems that occur in trying to link a Math∫pad-like system to a theorem prover. We use PVS as example theorem prover, simply because Vaccari used PVS to check his thesis and thus both a document containing structured proofs and a PVS version of those proofs are available to us.

2 The Formatted Proof

The following text is a part of Vaccari's thesis. It defines a law with a condition and gives a proof by induction of that law:

A law about *map* and *fold* is the following: given R and S such that

$$R \circ S{\times}S \;=\; S \circ R$$

then

$$\mathit{fold}_n.R \circ \mathit{map}_n.S \;=\; S \circ \mathit{fold}_n.R$$

The proof is by induction on n; for $n = 1$ it is trivially true. For $n + 1$ we have

$$\begin{array}{cl}
 & \mathit{fold}_{n+1}.R \circ \mathit{map}_{n+1}.S \\
= & \qquad \{ \text{ definitions } \} \\
 & R \;\circ\; \iota \times \mathit{fold}_n.R \;\circ\; S \times \mathit{map}_n.S \\
= & \qquad \{ \text{ fusion, induction hypothesis } \} \\
 & R \circ S{\times}(S \circ \mathit{fold}_n.R) \\
= & \qquad \{ \text{ proviso: } R \circ S{\times}S \;=\; S \circ R\text{; fusion } \} \\
 & S \;\circ\; R \;\circ\; \iota \times \mathit{fold}_n.R \\
= & \qquad \{ \text{ definition } \} \\
 & S \circ \mathit{fold}_{n+1}.R
\end{array}$$

The law is written informally and the proof is formatted with a standard proof notation. That is, each step in the proof is augmented with a comment, explaining why the two expressions in that step relate to each other according to the given operator. We will refer to such a proof as "a formatted proof".

Although the text is concise, there are still some unclear points. The variable n is not specified in the definition of the law and the fact that n has to be positive is part of the proof. Since n is not introduced, there is an ambiguity whether the quantification over n is part of the consequent or also part of the antecedent, which influences both the definition and the use of the law in a theorem prover.

Furthermore, the author assumes that the reader has some basic knowledge on the subject, so trivial rules are not mentioned. The trivial rules in this case are the associativity of composition and the unit of composition ι. In this case, the second "fusion step" can not be applied correctly without first adding the unit of composition to the right hand side of the first S, replacing it with $S \circ \iota$.

The document is written with the Math∫pad system and the structure of the proof and the formulas is known, that is, no additional parsing is needed to construct that structure. Due to the structure, additional operations like formula extraction and matching are available for free.

3 The PVS Proof

The law in the previous section can easily be proven in PVS if the correct environment is available, that is, if all the definitions, theorems and proofs up to this law are already available in PVS. If that is the case, the above law can be entered as the theorem:

```
fold_map:  THEOREM  R o (S*S) = S o R
               IMPLIES fold(n,R) o map(n,S) = S o fold(n,R)
```

This resembles the formatted text closely, because the binary operators o and * are overloaded and the variables R, S and n are already defined in the PVS context. Otherwise, the operators need to be replaced by functions with two arguments and the variables need to be introduced with a universal quantifier, which would result in the following PVS input:

```
fold_map:  THEOREM  (FORALL (R:rel, S:rel, n:upfrom(1)):
         comp(R,prod(S,S)) = comp(S,R)
            IMPLIES comp(fold(n,R),map(n,S)) = comp(S,fold(n,R)) )
```

Since only a limited number of operators can be overloaded in PVS, it is most likely that, in a general case, a PVS formula will look like this last formula, thereby creating a gap between the familiar syntax used in the document and the unfamiliar syntax used by PVS. The translation from the formatted document to the PVS formula is still straightforward and easy to do automatically once the structure is known.

Following the formatted proof, it is quite easy to construct a standard PVS proof of the theorem:

```
(induct "n" 1)                        %induction
1  (grind)                            %basis: trivial
   (rewrite "id0")
   (rewrite "id1")
2  (skolem!) (ground)                 %step
   (skolem!) (ground)
   (expand "fold" :if-simplifies t)   %definition
   (expand "map" :if-simplifies t)    %definition
   (assoc-rewrite "fusion" :dir RL)   %fusion lemma
   (inst?) (ground)                   %induction hypothesis
   (replace*)
   (rewrite "id1")                    %remove unit of composition
   (rewrite "id0" 1 ("R" "S!1") 1 RL) %add unit of composition
```

```
(assoc-rewrite "fusion")          %fusion lemma
(rewrite "id0")                   %remove unit of composition
(replace*)                        %proviso
(rewrite "comp_assoc")            %associative of composition
```

Using PVS all the trivial steps have to be provided. In this case, the trivial lemmas are:

```
id0:        THEOREM R o I = R
id1:        THEOREM I o R = R
comp_assoc: THEOREM (R o S) o T = R o (S o T)
```

Since the last lemma is difficult to instantiate if multiple matches occur, a PVS rewrite rule had been defined that rewrites modulo associativity of composition, called assoc-rewrite.

As this example clearly shows, the PVS version is less readable than the formatted version, partly because the formulas are missing. However, adding the formulas that PVS generates in its output would not really improve the readability, because notational conventions are different and it would increase the length of the proof dramatically, especially if all the premises are added.

The structure of the PVS version resembles the structure of the formatted proof quite closely, except on some small points. In PVS, you prove that the equation is true, while in the formatted text, you prove that the left side is equal to the right side by transitivity of equality. As a result, the last step in the formatted proof (using the definition of *fold*) is already applied at the earlier step in the proof. Furthermore, in the formatted proof, a definition can be used in two directions, while in PVS, a definition can only be expanded. Another small difference is the changed order of the second use of fusion and the proviso, due to the fact that the formatted proof is less strict in giving the hints.

Since PVS provides very powerful strategies, the PVS proof could be made shorter by using strategies like induct-and-simplify and installing automatic rewrite rules. These strategies were not used in order to be able to follow the formatted version and to identify problems and solutions.

4 The Scenario

Since the PVS proof resembles the formatted proof, system support for semi-automatically generating the PVS version should be available. After the formatted proof is finished, the user would like to press a "Check" button to see if the proof is correct according to PVS. The system should generate the PVS files containing the theorem and the proof attempt and use PVS to check if it is correct. If errors or ambiguities occur during the generation of those files, the system asks the user for assistance. If PVS is unable to process the files, the user is pointed to the part of the document which is responsible for the error. The user can then add hints to the document to adjust the PVS input that will be generated. Once the PVS input is correct, PVS will check the proof and errors in the proof steps are redirected by the system to the hints in the formatted proof. Again, the user can add PVS specific hints to adjust the behaviour of PVS.

In this feedback loop, the user should see as little PVS syntax as possible and all PVS results should be shown in the syntax preferred by the user. The user should of course be able to add raw PVS commands to the generated PVS files, to make sure that the user has full control over both PVS and the system itself. However, the user should not have to adjust the generated PVS files by hand, as that would introduce synchronisation problems and, at any time, the document itself should be sufficient to regenerate the correct PVS files.

The system would analyse a part of the document and generate as much relevant PVS input as reasonably possible. Since the document is already structured in Math∫pad, much of the structure for the PVS input is already available. The formulas are converted easily as long as the PVS versions use a similar structure. In order to fill in the missing gaps, databases with common keywords, lemmas, definitions and variables are used to collect information and select PVS tactics and strategies. The user and system can extend and adjust these databases to optimise the automatic generation of PVS input or to adjust it to the language native to the user. In the example proof, the known keywords would be 'given', 'such that', 'then', 'proof', 'induction', 'trivially', 'definition', 'induction hypothesis' and 'proviso', the known lemma would be 'fusion', the known definitions would be 'fold' and 'map' and the known variables would be 'R', 'S' and 'n'. With this knowledge, much of the PVS input could be extracted. This form of extracting information can be compared with the methods used in the Eliza system [15] and other natural language processing systems.

5 Problems and Solutions

It is obvious that constructing such a system is not trivial and that a lot of problems will occur while trying. In this section, some of the foreseeable problems are discussed, together with some solutions.

5.1 Extracting the Theorem

As the text is already available, it would be nice if the system can partially extract the theorems and definitions and convert them to the correct PVS syntax. However, a natural language contains a lot of ambiguity, even in a mathematical or technical document, where the author carefully weighs ambiguity with readability. As a result, small ambiguities might be introduced which are not noticed by the reader but are important for a theorem prover. In the example, the variable n is never introduced and as a result, the extracted law might be either

$$\forall(R,S :: \ R \circ S{\times}S \ = \ S \circ R \ \Rightarrow \ \forall(n : \ n \geq 1 : \ \mathit{fold}_n.R \circ \mathit{map}_n.S \ = \ S \circ \mathit{fold}_n.R))$$

or

$$\forall(R,S,n : \ n \geq 1 : \ R \circ S{\times}S \ = \ S \circ R \ \Rightarrow \ \mathit{fold}_n.R \circ \mathit{map}_n.S \ = \ S \circ \mathit{fold}_n.R)$$

In this situation, the meaning of the two expressions is equal, since n does not occur in $R \circ S\times S \ = \ S \circ R$ (which might not be true in a more general case). However, the PVS proofs of the two laws are different and, more importantly, applying the laws is different. Therefore, interaction between the user and the system is needed to extract the final unambiguous interpretation of the text. This interaction could be based on a natural language, describing the meaning of the text unambiguously or could be based on the native syntax, possibly extended with standard logical notations, as shown above. The user should be able to select one of the possible interpretations or change the contents of the document to remove the ambiguity.

To reduce the number of the errors that PVS reports, the system has to make sure that all dependencies of the formulas are resolved by including the definitions of all referred functions, lemmas and variables. However, PVS remembers which lemmas are already proven and which theory contains them. Copying a lemma to a working document will result in error messages from PVS about multiple occurrences of the same lemma, with the result that the copied version is not proven. Therefore, the proven lemmas have to be collected in theory files, which are included by the temporary PVS file containing the unproven theorem. After this theorem is proven, it can be added to the appropriate theory file, where it has to be proven once again to update the PVS state and its databases.

5.2 Proving the Formatted Proof

In order to prove that the formatted proof is correct, a proof has to be given for each step in the formatted proof, that is, each consecutive pair of formulas is indeed related to each other according to the given relator (and this proof should be based on the hints that are given). For example, the proof step

$$R \ \circ \ \iota \times \mathit{fold}_n.R \ \circ \ S \times \mathit{map}_n.S$$

$$
\begin{array}{ll}
= & \quad \{ \text{ fusion, induction hypothesis } \} \\
 & R \circ S{\times}(S \circ \mathit{fold}_n.R)
\end{array}
$$

is proven in PVS by showing that the two formulas are equal, by using fusion and the induction hypothesis. This would result in a small lemma stating that this equality is a valid consequent of the induction hypothesis, as in:

```
pstep2:  LEMMA fold(n,R) o map(n,S) = S o fold(n,R)
           IMPLIES R o (I*fold(n,R)) o (S*map(n,S)) = R o (S*(S o fold(n,R)))
```

Again, the translation of the formulas is straightforward and can be applied to each step of the formatted proof and, since the document is already structured, determining the steps and formulas in a proof is trivial. Such a small lemma can be used to check if the corresponding step is correct and together these small lemmas are used to verify that the proof of the law is correct. The advantage of using these small lemmas is that unclear hints and problems in PVS are directly related to specific steps in the formatted proof. A disadvantage would be that a large collection of small lemmas results in large PVS files, what would effect the performance of PVS. This problem might be resolved by combining the proofs of the several small lemmas into one larger proof, thereby removing the need to define those lemmas.

A different approach to proving that the formatted proof is correct would be to prove that the first formula is related to the last formula and check the PVS output to verify that the intermediate formulas do occur at the correct positions and that the PVS tactics have some effect on the formulas. However, this approach does not work in a general case. Each of the following proofs is in fact a combination of two proofs.

$$
\begin{array}{ll}
 & P \\
\Rightarrow & \quad \{ \ \vee \text{ introduction } \} \\
 & P \vee (Q \wedge P) \\
\equiv & \quad \{ \text{ DeMorgan } \} \\
 & (P \vee Q) \wedge P \\
\Rightarrow & \quad \{ \ \wedge \text{ elimination } \} \\
 & P
\end{array}
\qquad\qquad
\begin{array}{ll}
 & P \\
\equiv & \quad \{ \ \Rightarrow\text{: } \vee \text{ introduction} \\
 & \quad\quad \Leftarrow\text{: DeMorgan, } \wedge \text{ elimination } \} \\
 & P \vee (Q \wedge P)
\end{array}
$$

In the proof on the left, the purpose is not to show that $P{\Rightarrow}P$, but to show that all the intermediate formulas are equal. In the proof on the right, the hint contains an explanation why the equality holds, by reasoning about both directions. In both cases the proof of $P \Rightarrow P \vee (Q \wedge P)$ and $P \vee (Q \wedge P) \Rightarrow P$ is combined into a single proof, which is difficult to extract in a more complicated case.

5.3 Hiding Information

After the PVS files are generated, PVS can be used to check if the definition is correct. In case of errors, the system should start an interaction with the user to resolve the problems. During this interaction, the user should not be exposed to the PVS syntax, as it is often awkward, unfamiliar and unreadable in comparison with the standard notational conventions used in mathematics and other technical fields. Instead, the system should translate all the PVS communication to the notational conventions native to the user. This requires origin tracing and, since PVS can not be adjusted freely, a parser for the PVS messages and formulas.

An interesting proof contains a complification step, that is, a step where a simplification is used in the opposite direction or where a smart case analysis is applied. Such steps can not be determined automatically, since there are usually many possible steps, but only few of them will have a positive effect. Since the formatted proof already exists, the user knows where each complification step occurs and how each simplification lemma is applied. Somehow, the user has to specify these steps to the system and add them to the generated PVS input, without being exposed to the PVS syntax. This can be achieved by showing the formula and the tactic that failed on the formula. The user can modify the formula with a restricted set of actions, all corresponding to actions in PVS, until the failed tactic can be applied successfully. Alternatively, the user can adjust the proof, if it seems to be incorrect. The actions needed to adjust the formula will be stored in the hints in the proof, such that a regeneration of the PVS proof will be correct the next time.

All formulas are given a number in PVS, to allow the user to refer to them when a tactic is applied. In the formatted proof these numbers are not available and the user is usually not interested in these numbers. In fact, the PVS user is advised to apply the tactics without specifying these numbers, as they might change if the proof or the theorem is edited afterwards. Instead, the user should only specify whether to apply the tactic to the premises or the consequents. However, such general tactics might result in changes to formulas which should not change. In the example, the general tactic `(replace*)` will use the premise to replace matching parts of the consequent.

While proving something in PVS, PVS will introduce variables with ugly names. These names are usually not needed in the tactics, except when a rewrite rule is applied and the specific substitution is needed. Since the user should not be exposed to the PVS syntax, the user should be able to construct such substitutions with the familiar syntax, where the system will translate it back to PVS syntax. In the example, the complification step with lemma `id0` contains such a substitution.

Due to tactics in PVS, the goal might get split into several subgoals. Some of these subgoals could be trivial if a powerful strategy is applied, such as `grind`. If such a strategy can successfully prove a subgoal, the system should not mention such a subgoal to the user, who could get disappointed by a large collection of unproven but trivial subgoals. Furthermore, the system should warn the user if the number of non-trivial subgoals exceeds a certain threshold, as it could indicate that an additional lemma or case analysis is needed to make the proof more intuitive.

5.4 Defining Tactics

When the formatted proof is used to construct a PVS proof, the PVS proof will start with a formula of the form $A \trianglelefteq B$ and should end with a formula of the form $B \trianglelefteq B$. However, while applying PVS tactics, it is very likely that certain actions will change formula B, which could result in a failed proof. To prevent this, the right hand side of the formula, B in this case, can be replaced using a name definition before applying the tactics and hide it from PVS. After all the tactics are applied, the name definition can be used to check if the proof has succeeded or not. This approach has one disadvantage: a definition can only be expanded in PVS, which, in the example, prevents the last step of the proof. To solve this, a trivial lemma can be defined to reverse the expansion of a definition and an additional tactic can be used to apply the rule.

In the formatted proof, trivial steps are often not mentioned, since it would make the proof less readable. In the PVS proof, these steps are needed before the next tactic can be applied. An obvious solution would be to install automatic rewrite rules and allow the user to specify which rules can be regarded as trivial. However, if a trivial rule is applied as a complification step, the automatic rewrite rules should not undo such a step, such as the complex rewrite step in the example. Therefore, the trivial rewrite rules should only be applied if the next tactic would fail without them. This can be achieved by defining new PVS tactics.

A lemma or rewrite rule is often used in two directions. In PVS, the direction of the rewrite rule has to be specified in order to apply it successfully. In the formatted proof, the hints do not contain this information, as it is obvious to the reader. The solution would be to define a new PVS strategy to apply the rewrite rule in the default direction and, if that fails, apply it in the opposite direction. The drawback of such a strategy would be that the result is unpredictable, as the rule might be applicable in both directions. In the example proof, the `fusion` lemma is applied twice in different directions and the PVS strategy would remove the requirement to specify the direction.

5.5 The PVS Interface

The PVS system itself also causes some problems. The PVS core system is written in ilisp and uses the Emacs editor as a user interface. The communication between this core system and Emacs is hidden from the normal user and is not well documented. Since the Emacs system is too large to be used as a communication layer and would introduce more problems than that it would solve, the PVS core system has to be extract and documented. Such documentation is of course also helpful for other systems that try to interact with PVS.

The PVS core system is rather large and calculations might be time consuming, especially if advanced and powerful strategies are used. Therefore, the system has to protect the user from trying to proceed with the document preparation while PVS is busy, but it should allow the user to perform other tasks. The best solution would be to block editing actions on certain parts of the document, such that PVS is always computing results which are still valid and needed.

6 Related work and conclusions

There are already some projects to improve the interface of PVS. TAME [1] is a layer on top of PVS for reasoning about timed automata and consists of a number of strategies to reduce the number of steps made in a typical PVS proof to the number of steps made in a hand-made proof. With these additional strategies, the user of TAME will not be exposed to the low-level steps and commands needed in PVS, thereby making the commands field specific. However, since the PVS interface is used, there is still a gap between the notational conventions used by PVS and those used in the documentation.

The system PAMELA [4] is designed to check partial correctness of VDM-like specifications in the area of code generators. By providing a connection with PVS, the system supports a larger class of specifications, using PVS to discharge proof obligations. Since PAMELA provides its own interface, communication between PVS and another system seems to be feasible and some experience is available.

Simons [12] has been working on a system to combine proofs in Isabelle [8] with documentation. The system uses the structured documentation technique introduced by Knuth [6] to allow one file to contain both the proofs and the documentation and uses programs to separate those. This solves the problem of combining several files into one document, at the expense of using different languages in a single file, namely, LaTeX for formatting the document, Isabelle for specifying the proof and the meta language to instruct the programs. For a user, this might be confusing.

Constructing a tool which extracts a PVS proof from a document seems to be feasible, although there are a lot of problems to overcome. We already have a collection of electronic documents which are structured and a tool to edit and manipulate those documents. For some documents, the PVS definitions and proofs are already available, which is very helpful to get started.

Since PVS is not the only available theorem prover, the connection with PVS should be made general enough, such that a connection with a different theorem prover would not be too difficult. The fact that PVS is a closed system could in this case be helpful, as modifications to PVS would indicate that modifications to other theorem provers are needed if such a connection is required.

References

[1] Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 33–48, Murray Hill, NJ, August 1997. Springer-Verlag.

[2] R.C. Backhouse, R. Verhoeven, and O. Weber. Mathſpad: A system for on-line preparation of mathematical documents. *Software – Concepts and Tools*, 18:80–89, 1997.

[3] Roland Backhouse and Richard Verhoeven. Math∫pad *Ergonomic Document Preparation*, version 0.60 edition, February 1996. Manual of the Math∫pad system. See also: `http://www.win.tue.nl/cs/wp/mathspad/`.

[4] Bettina Buth. *Operation Refinement Proofs for VDM-like Specifications.* PhD thesis, Institute of Computer Science and Practical Mathematics of the Christian-Albrechts-University Kiel, February 1995. See also: `http://www.informatik.uni-bremen.de/ bb`.

[5] P. Naur (Ed.). Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6:1–17, January 1963. Also in *The Computer Journal*, 5: 349–67 (1963); *Numerische Mathematik*, 4: 420–52 (1963).

[6] D.E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[7] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. See also: `http://www.csl.sri.com/pvs.html`.

[8] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover.* Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[9] Daniël Peeters. Towards a publication style of functional programming. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, June 1997.

[10] Frits D. Schalij. Tangram manual. Technical Report UR 008/93, Philips Electronics N.V., 1996.

[11] Robin Sharp and Ole Rasmussen. An introduction to Ruby. 2nd edition. Technical report, Dept. of Computer Science, Technical University of Denmark, 1995. Available at `ftp://ftp.it.dtu.dk/pub/Ruby/intro.ps.Z`.

[12] Martin Simons. Proof presentation for Isabelle. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 259–274, Murray Hill, NJ, August 1997. Springer-Verlag.

[13] Charles Simonyi. The death of computer languages, the birth of intentional programming. Proceedings of the 28th Annual International Seminar on the Teaching of Computing Science at University Level, Sponsored by ICL and University of Newcastle upon Tyne, Department of Computing Science, September 1995.

[14] Matteo Vaccari. *Calculational Derivation of Circuits.* PhD thesis, Department of Computing Science, Università degli Studi di Milan, December 1997. Draft version, to be published.

[15] Joseph Weizenbaum. Contextual understanding by computers. *CACM*, 10(8):474–480, 1967.

Using gestures to disambiguate unification

Richard Bornat[1] and Bernard Sufrin[2]

March 1998

The purpose of a user interface to a theorem prover is to do with easy gestures what would otherwise be done with long-winded commands – use *this* instance of *that* rule to attack *this* subproblem – and to display information so clearly that alternatives become as easy to see as to invoke.

There isn't a unique design solution to the UITP problem and – *pace* those outsiders who believe that it must just be an afternoon's work to bolt a bit of Tcl/tk onto a text-command-driven theorem prover – it is probable that good solutions will always require specially designed and carefully integrated components. UITP is as much about TP as about UI, then, but it's worthwhile to attempt to abstract from our UI experience, even though the designs may be difficult to export to conventional TP practice.

Jape's[1] GUI uses the present-day gestural vocabulary – click, double-click, press-and-drag – and restricts itself to use no more than the present-day GUI vernacular of selection and command. Our aim is to make the tool easier for the novice to use, because the gesture vocabulary and language is familiar, but it has required considerable ingenuity to allow an expert user to exert sufficiently subtle control through the same kinds of gesture.

Jape uses unification, but it has to deal with problems which don't have determinate solutions: higher-order unification (because it deals with explicit substitution formulæ in rules and/or proofs); list and bag unification (when unifying contexts, and perhaps one day when dealing with associative/commutative operators). In principle and in general Jape is designed to defer decisions about ambiguous unifications, by introducing provisos and unknowns as necessary. In practice that deferral can introduce confusing and misleading detail to a proof, so it is worthwhile introducing gestures which disambiguate unification directly. In certain cases, especially when deciding on the formula instance which should match a principal formula of a rule, it makes little sense to defer the decision and direct gesturing is required.

Indicating position and point of interest

A natural first requirement is to be able to indicate just where in a partially-completed proof an action should be carried out. There is a well-attested HCI principle[2] that *postfix* command construction – select first, then act – is preferable to prefix, infix or modal dialogue. Jape's first gesture, then, must be one which commands the visible *selection* of a point of interest, and in the best GUI design tradition Jape's users can move the selection by making just the same gesture at an alternative point. In both the current implementations of our GUI the selection gesture is a mouse-click over a single formula, and the visible effect is that the selected formula is enclosed in a box.

$$\dfrac{\dfrac{\forall x.R\to S(x),\boxed{\forall x.\neg R\to S(x)}, R\vdash S(m) \qquad \forall x.R\to S(x), \forall x.\neg R\to S(x), \neg R\vdash S(m)}{R\vee\neg R, \forall x.R\to S(x), \forall x.\neg R\to S(x)\vdash S(m)}\,{\scriptstyle \vee\vdash}}{R\vee\neg R, \forall x.R\to S(x), \forall x.\neg R\to S(x)\vdash \forall x.S(x)}\,{\scriptstyle \vdash\forall}$$

fig 1: selection indicates a subtree, sequent or formula of interest

[1] richard@dcs.qmw.ac.uk; Department of Computer Science, Queen Mary and Westfield College, University of London, LONDON E1 4NS, UK

[2] sufrin@comlab.ox.ac.uk; Programming Research Group, Wolfson Building, OXFORD OX1 3QD, UK; Worcester College, Oxford.

Formula-selection is taken to indicate not only a particular instance of a formula, but also the particular sequent which contains it as well as the subtree rooted at that sequent instance. Depending on the command which is applied to the selection, Jape will choose the relevant indication: if the command is tree- or display-editing – prune, hide or display the subtree above the node containing the selected formula – then take it that a subtree is intended; if it is a proof action – applying a rule or a tactic – then take it that a sequent is intended (and, in that case, be sure that it is at a tip of the tree). Thus we satisfy the simplest requirement of the user interface, to make the tool work *here* rather than *there* in the tree[1].

The formula-selection gesture makes a finer distinction than would be required to indicate a sequent or a position in the tree. When used before a rule application it is taken to indicate a problem formula which must be matched by a principal formula of the rule.

$$\dfrac{\neg(P\land Q)\vdash\neg P,\boxed{\neg Q}}{\neg(P\land Q)\vdash\neg P\lor\neg Q}\,\vdash\lor \qquad\qquad \dfrac{\Gamma,A\vdash\Delta}{\Gamma\vdash\neg A,\Delta}\,\vdash\neg$$

fig 2: selection disambiguates rule-unification by indicating a principal formula

Without formula-selection there is more than one way in which the ⊢¬ rule can match the problem sequent; after selection there is only one. Thus formula-selection resolves an ambiguity of list (or bag) unification, and determines it uniquely.

Principal formulæ can occur on the left or the right of a sequent, or both. Jape therefore permits both hypothesis (left-hand) and conclusion (right-hand) formula selection, up to one of each simultaneously. Simultaneous selection is occasionally essential in box-and-line presentation of a single-conclusion calculus: because hypotheses are written only once, at the head of the box which corresponds to the subtree for which they are introduced, hypothesis selection will often indicate a sequent at the root of a subtree, not one of the tips of that subtree.

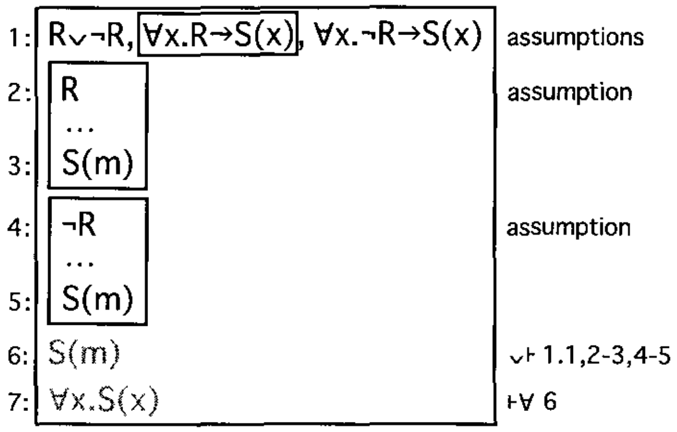
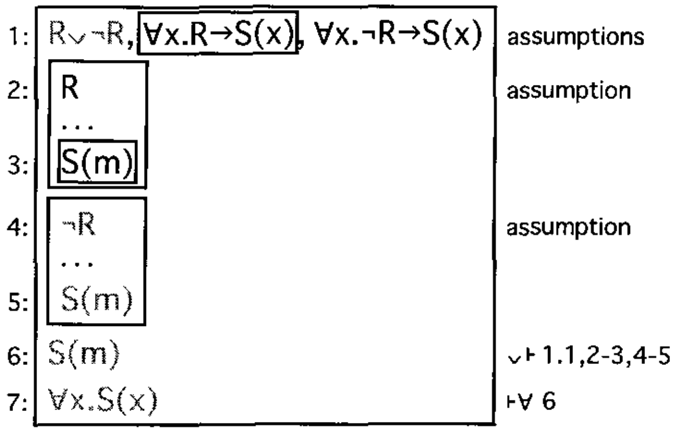


fig 3: hypothesis selection in the box-and-line presentation of the proof from fig 1 selects a subtree, not a sequent

Figure 3 shows the box-and-line version of the tree of figure 1. Lines 1 and 7 correspond to the root of the tree; lines 1 and 6 to the antecedent of the root; lines 1, 2 and 3 to the left tip and lines 1, 4 and 5 to the right tip. Selecting a hypothesis on line 1 doesn't, therefore, indicate either of the tips, which correspond to lines 3 and 2 in one case, 5 and 4 in the other (each to be read, in the box-and-line convention, in conjunction with line 1). The solution is simple and obvious – allow the user to disambiguate the selection by selecting a conclusion line at the same time, thus indicating both the

[1] Jape allows application without pre-selection in the case that there is only one tip in the tree. HCI purists may shudder, and indeed we have not experimentally evaluated this feature of the interface, but on the face of it it seems a sensible and natural convenience. Then again, it may be a source of confusion. An upcoming evaluation project may supply us with data which will help us to decide.

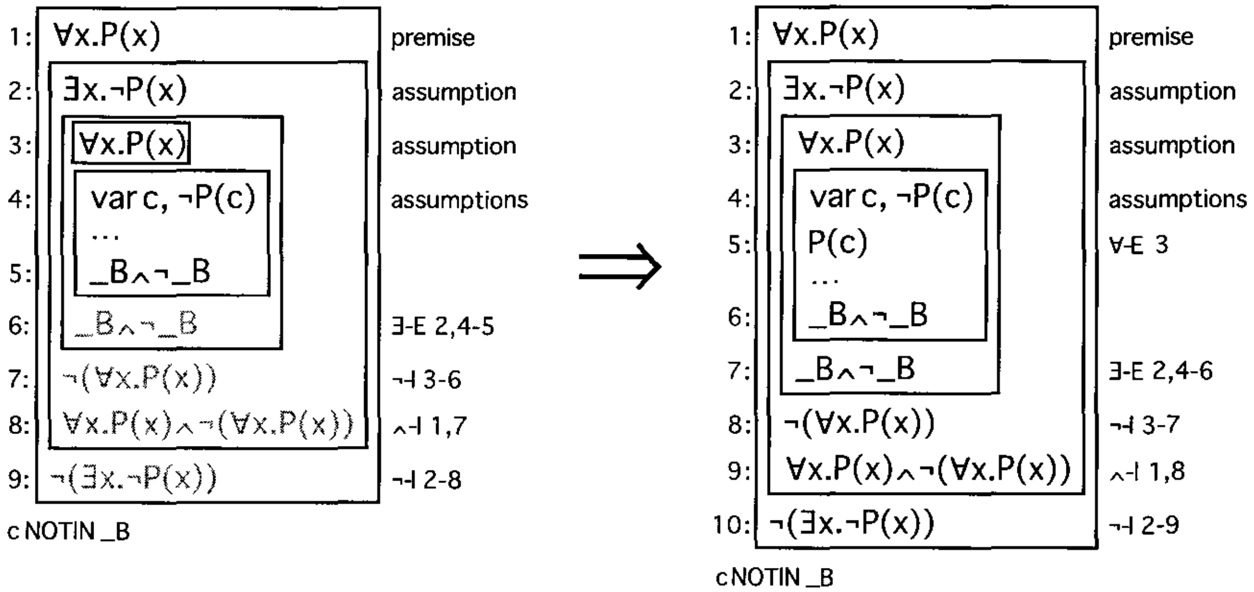
hypothesis which is to be worked on and the tip at which action should take place[1], as illustrated in figure 4[2].



1:	R∨¬R, ∀x.R→S(x), ∀x.¬R→S(x)	assumptions
2:	R	assumption
	...	
3:	S(m)	
4:	¬R	assumption
	...	
5:	S(m)	
6:	S(m)	∨⊢ 1.1,2-3,4-5
7:	∀x.S(x)	⊢∀ 6

fig 4: conclusion selection disambiguates subtree selection of fig 3

There would be a visual ambiguity in box-and-line hypothesis selection if Jape did not insist that the user choose an *instance* of a formula in the proof. Figure 5 shows a proof in which there is more than one instance of the same hypothesis formula in scope at once – which means, more than one instance of the same hypothesis formula in the context. It is important that the justification of the proof step refers to the instance selected and, as illustrated in figure 5, Jape obliges.



1:	∀x.P(x)	premise
2:	∃x.¬P(x)	assumption
3:	∀x.P(x)	assumption
4:	var c, ¬P(c)	assumptions
	...	
5:	_B∧¬_B	
6:	_B∧¬_B	∃-E 2,4-5
7:	¬(∀x.P(x))	¬-I 3-6
8:	∀x.P(x)∧¬(∀x.P(x))	∧-I 1,7
9:	¬(∃x.¬P(x))	¬-I 2-8

c NOTIN _B

⇒

1:	∀x.P(x)	premise
2:	∃x.¬P(x)	assumption
3:	∀x.P(x)	assumption
4:	var c, ¬P(c)	assumptions
5:	P(c)	∀-E 3
	...	
6:	_B∧¬_B	
7:	_B∧¬_B	∃-E 2,4-6
8:	¬(∀x.P(x))	¬-I 3-7
9:	∀x.P(x)∧¬(∀x.P(x))	∧-I 1,8
10:	¬(∃x.¬P(x))	¬-I 2-9

c NOTIN _B

fig 5: proof steps credit formula instances, not formulæ

Jape uses a resource-labelling mechanism to achieve this effect: each formula instance in the proof tree is invisibly labelled with a resource number[3]. The resource number is copied from consequent to antecedent(s) whenever a formula is not the principal formula in a rule. This enables the proof engine, when an instance of a formula at the root of a subtree is selected, to identify corresponding instances in the tips. The mechanism makes the displayed proof stick closely to the user's intentions: they can't help but indicate a formula instance, and the proof step records the instance that was used.

$$\frac{S \succ T}{C, x \mapsto S, C' \vdash x : T}\ (x \text{ NOTIN } C')\ \text{var lookup}$$

fig 6: a rule in which the principal formula determines a section of the context

[1] Jape doesn't require the user to indicate a tip if there is only one open tip in the selected subtree. Perhaps another HCI shudder?

[2] Note that part of line 1 is greyed out when line 3 is selected: the first hypothesis in line 1 is 'used up' in the ∨⊢ step which produces lines 2-5.

[3] We haven't found any need to reveal them to the user; even the implementors hardly ever want to see them.

The resource mechanism has more than cosmetic advantages: it is essential in situations which require a much finer disambiguation of unification. When the position of a formula instance in a sequent is important, this device allows just the control that would be required. This permits, for example, the use of rules which treat the left-hand side as a sequence of type judgements, each with a preceding context. Figure 6 shows a rule whose application might require careful pre-selection of a principal formula, because the context to the right of that formula must have certain properties. Ambiguous selection would be more than an irritation and must be resolved: formula-instance selection does the job where necessary.

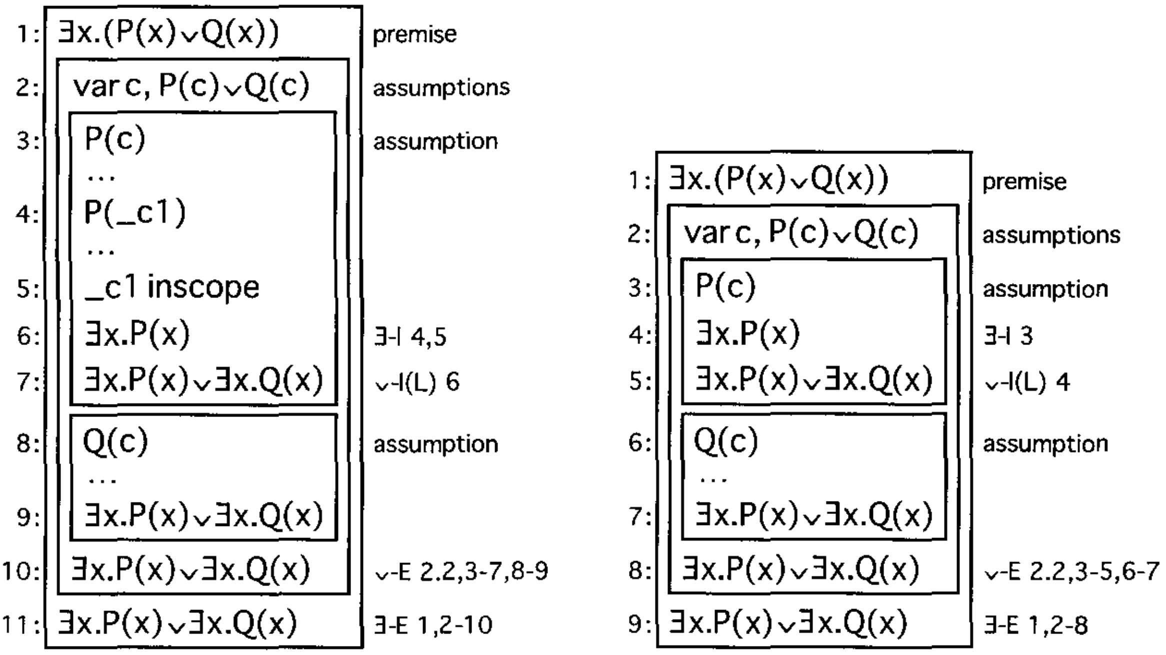
As with all the best GUI mechanisms, formula-instance selection has an easy explanation. We tell our users *"select the formula you want to work on"*. That makes good sense in logics whose rules each work on a single principal formula, exposed at the level of the sequent. It doesn't do so well with logics – such as the BAN logic of authentication protocols – where rules work on formulæ inside tuples, or on more than one principal hypothesis formula. Even the simple formula-selection gesture needs further work.

Acting on formulæ

Jape makes very little use of the double-click at present. In the vernacular, double-click means something like *"do the right thing to this object"*, where the right thing is usually decided according to the object's type – document type, icon type – or sometimes by the style of the click – which button, what modifier keys. A double-click in Jape's GUI causes the proof engine to look through a list of formula-patterns till it finds one which matches, and to run a corresponding tactic. This has nothing to do with disambiguating unification, so we dwell on it no further in this description.

Indicating points of interest within formulæ

Jape uses unification in every proof step, and it is prepared to defer some of its unification decisions until more information is available. This is fine for proof exploration, and in particular it permits use of Jape as a calculator (for example, in the Hindley-Milner proof inference algorithm), but the effects can sometimes be confusing to the novice. A simple example of this is the use of the ∀-*elim* rule backwards in a logic which demands a version of variable scoping.



```
 1: ∃x.(P(x)∨Q(x))          premise
 2: var c, P(c)∨Q(c)        assumptions
 3: P(c)                    assumption
    ...
 4: P(_c1)
    ...
 5: _c1 inscope
 6: ∃x.P(x)                 ∃-I 4,5
 7: ∃x.P(x)∨∃x.Q(x)         ∨-I(L) 6
 8: Q(c)                    assumption
    ...
 9: ∃x.P(x)∨∃x.Q(x)
10: ∃x.P(x)∨∃x.Q(x)         ∨-E 2.2,3-7,8-9
11: ∃x.P(x)∨∃x.Q(x)         ∃-E 1,2-10
```

```
1: ∃x.(P(x)∨Q(x))          premise
2: var c, P(c)∨Q(c)        assumptions
3: P(c)                    assumption
4: ∃x.P(x)                 ∃-I 3
5: ∃x.P(x)∨∃x.Q(x)         ∨-I(L) 4
6: Q(c)                    assumption
   ...
7: ∃x.P(x)∨∃x.Q(x)
8: ∃x.P(x)∨∃x.Q(x)         ∨-E 2.2,3-5,6-7
9: ∃x.P(x)∨∃x.Q(x)         ∃-E 1,2-8
```

fig 7: application of ∃-I rule without (left) and with (right) selection of argument 'c'

Jape will, quite naturally, introduce an unknown into a proof whenever a rule is used backwards which hasn't the subformula property (when a name in the antecedents doesn't appear in the consequent). The novice user, when working straightforwardly with formulæ that don't include unknowns, usually wants to avoid the introduction of unknowns into the proof. They appear as a kind of visual clutter, justified only when there is a real search for a genuinely unknown formula – for example when working with classical negation axioms. In this case we've chosen to highlight the cluttering problem by illustrating use of a rule which includes a normally-hidden antecedent, a kind of side-condition. If the user provides a variable name as an argument to the rule application, Jape will automatically satisfy that antecedent and it will remain hidden; if no argument is provided, Jape will provide an unknown and the unsatisfied antecedent must be shown.

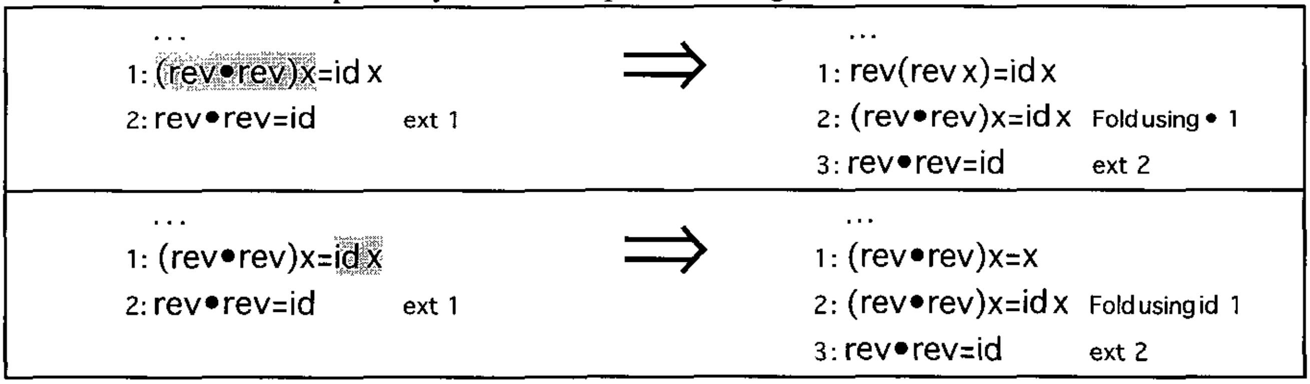
A second gesture, then, must be used for argument selection. It can't be the same gesture as formula selection, and it can't be indicated in the same way. In both current implementations of the Jape GUI we have chosen press-and-drag within a formula, indicated by textual highlighting (using the background OS's GUI convention)[1]. If there is a single textual selection which parses as a formula and if the tactic invoked by the user doesn't interpret the text selection specially, then that selection is taken to be an argument formula which modifies the instance of the rule to be applied. Tactics can be written so that the same argument can be applied to more than one rule invocation, to none, to other than the first rule invoked by the tactic, and so on.

Historically the argument-selection gesture was invented to solve the problem of rules which have substitution formulæ in their consequent, but this use has been superseded by a mechanism which allows a user to indicate more directly the particular substitution unification which is to be used.

$$\frac{\Gamma \vdash X = Y \quad \Gamma \vdash P[v \backslash Y]}{\Gamma \vdash P[v \backslash X]}\text{ rewrite} \qquad \frac{\Gamma \vdash P\{v \backslash []\} \quad \Gamma \vdash P[v \backslash [x]] \quad \Gamma, P[v \backslash xs], P[v \backslash ys] \vdash P[v \backslash xs +\!\!+ ys]}{\Gamma \vdash P[v \backslash Q]}\text{ list induction}$$

fig 8: some rules naturally include a substitution in their consequent

Using the argument-selection gesture, the original unification mechanism would accept an argument A to a rule such as one of those in figure 8; it would match the principal formula $\_P[v \backslash A]$ with a formula R from the problem sequent by constructing R' such that $R'[v \backslash A]$ was equivalent to R. This *abstraction* mechanism found as many occurrences as possible of A within R, thus making the unification determinate. But it was a poor way to solve the problem of higher-order unification.



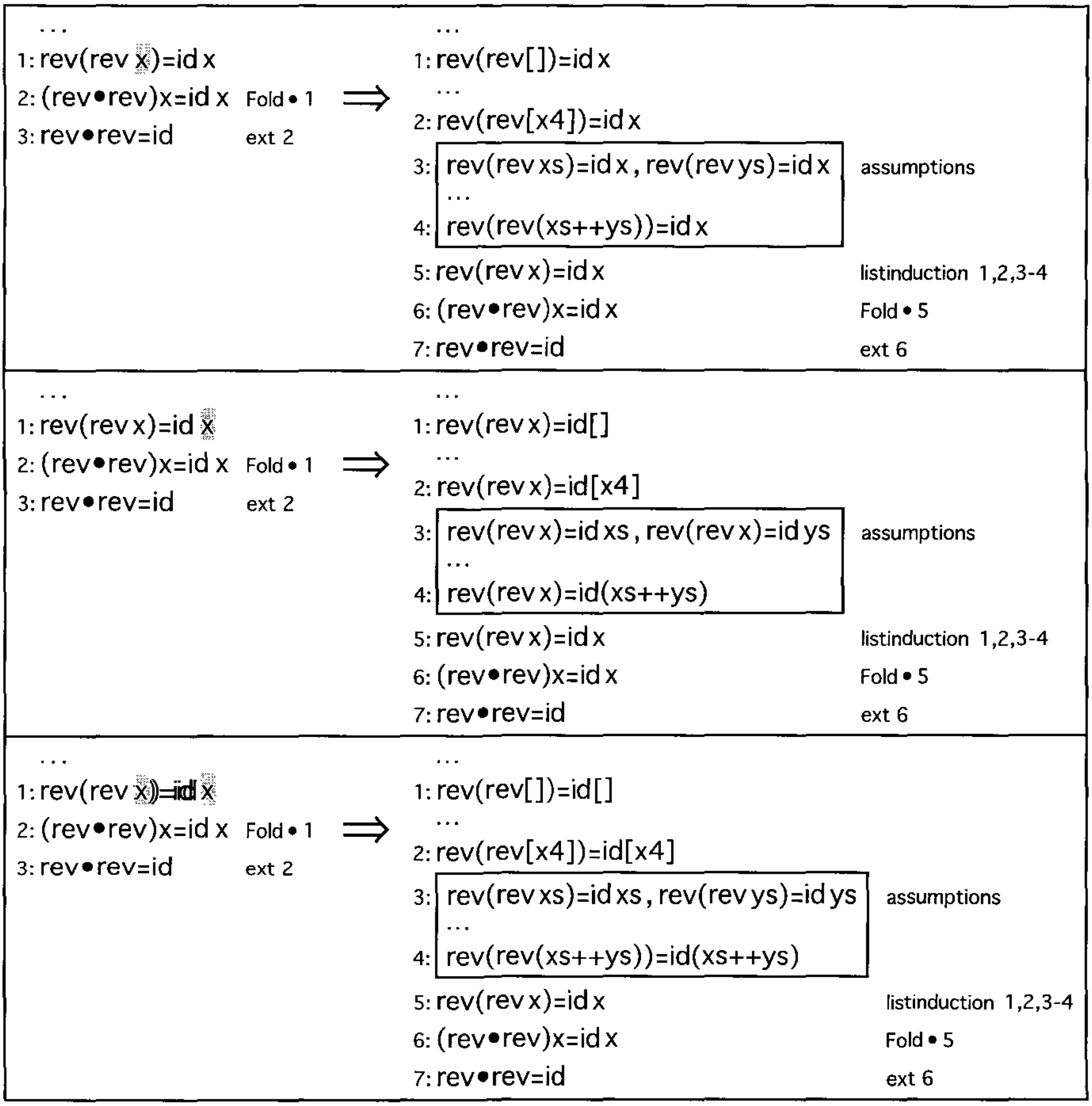
```
...                                          ...
1: (rev•rev)x=id x            ⟹             1: rev(rev x)=id x
2: rev•rev=id        ext 1                   2: (rev•rev)x=id x   Foldusing• 1
                                             3: rev•rev=id        ext 2

...                                          ...
1: (rev•rev)x=id x            ⟹             1: (rev•rev)x=x
2: rev•rev=id        ext 1                   2: (rev•rev)x=id x   Foldusingid 1
                                             3: rev•rev=id        ext 2
```

fig 9: alternative argument selections permit alternative proof steps (using rewrite behind the scenes)

Nowadays Jape will – under tactical direction – interpret the user's textual selection as describing a substitution formula directly. The selected subformula is just the one which is to be replaced by the substitution variable v to form $R'[v \backslash A]$. It is now meaningful to select more than one instance of a particular subformula: in $rev(rev\ x) = id\ x$ we can select one or the other or both xs, giving the user a choice between $(rev(rev\ v) = id\ x)[v \backslash x]$, $(rev(rev\ x) = id\ v)[v \backslash x]$ and $(rev(rev\ v) = id\ v)[v \backslash x]$[2]. The original problem formula is replaced by the user-specified selection, specially labelled as a 'stable substitution' so that the unifier won't immediately reduce it to the original formula. Unification between the stable substitution and the consequent substitution is by structure, and the ambiguous higher-order unification is bypassed.

[1] We have not implemented in either GUI any mechanism which allows easy selection of subformulæ. The reason is partly lack of effort, partly the occasional accidental convenience of the textual mechanism (for example, we sometimes select just 'T' from the the unknown '_T3'), but mainly a desire to keep our options open. We have to be able to deal with formulæ which contain associative operators, where selection has to be able to cross conventional structural boundaries, and probably always will. Even the simple subformula-selection gesture needs further research.

[2] We can't, at present, let the user select the null substitution, because we don't know how to do it gesturally. This means that some theoretically-possible proof trees are difficult to generate: to date it hasn't proved a problem.



...
1: rev(rev x)=id x
2: (rev•rev)x=id x Fold • 1
3: rev•rev=id ext 2

⟹

...
1: rev(rev[])=id x
...
2: rev(rev[x4])=id x
3: rev(rev xs)=id x , rev(rev ys)=id x assumptions
...
4: rev(rev(xs++ys))=id x
5: rev(rev x)=id x listinduction 1,2,3-4
6: (rev•rev)x=id x Fold • 5
7: rev•rev=id ext 6

...
1: rev(rev x)=id x
2: (rev•rev)x=id x Fold • 1
3: rev•rev=id ext 2

⟹

...
1: rev(rev x)=id[]
...
2: rev(rev x)=id[x4]
3: rev(rev x)=id xs , rev(rev x)=id ys assumptions
...
4: rev(rev x)=id(xs++ys)
5: rev(rev x)=id x listinduction 1,2,3-4
6: (rev•rev)x=id x Fold • 5
7: rev•rev=id ext 6

...
1: rev(rev x)=id x
2: (rev•rev)x=id x Fold • 1
3: rev•rev=id ext 2

⟹

...
1: rev(rev[])=id[]
...
2: rev(rev[x4])=id[x4]
3: rev(rev xs)=id xs , rev(rev ys)=id ys assumptions
...
4: rev(rev(xs++ys))=id(xs++ys)
5: rev(rev x)=id x listinduction 1,2,3-4
6: (rev•rev)x=id x Fold • 5
7: rev•rev=id ext 6

figure 10: three alternative substitution-selections, and the corresponding proof steps using the list induction rule

Naturally we don't tell our users that they are constructing substitutions (although they can read about it on our web site[4]); they need not even write the rule in terms of substitution but in a kind of 'abstraction' or 'predicate' notation (figure 11). We explain the gesture as *"select the subformula you want to work on"*, and to date it has fitted well into all of the logics we have encoded.

$$\frac{\Gamma \vdash X = Y \quad \Gamma \vdash P(Y)}{\Gamma \vdash P(Y)}\text{ rewrite} \qquad \frac{\Gamma \vdash P([]) \quad \Gamma \vdash P([x]) \quad \Gamma, P(xs), P(ys) \vdash P(xs +\!\!+ ys)}{\Gamma \vdash P(Q)}\text{ list induction}$$

fig 11: optional alternative notation allows description of rule without substitution notation

This technique of user-directed substitution makes it much easier to use Jape as a rewrite engine, and extended use has exposed one interesting problem – not a UI problem this time but a TP one. In a classical logic the equivalence of $P \to Q$ and $\neg P \vee Q$ is provable, and many users would like to use that equivalence in a rewrite rule. There's a problem when attempting to rewrite within a binding formula: $\forall y(A \to B)$ isn't equivalent to $(\forall y(v))[v \backslash A \to B]$ without extensive assurances that y doesn't occur within A or B, and if those assurances were given then the proof would probably be useless. A rule which uses formal substitution, like the one in figure 8, doesn't capture the notion of rewriting sufficiently well.

We can see, meta-logically, that if $P \rightarrow Q$ is equivalent to $\neg P \vee Q$ in any context or none, then it's safe to rewrite with that equivalence at any subformula position, no matter what its binding context. But how to tell that to Jape? Even the use of substitution formulæ needs further research.

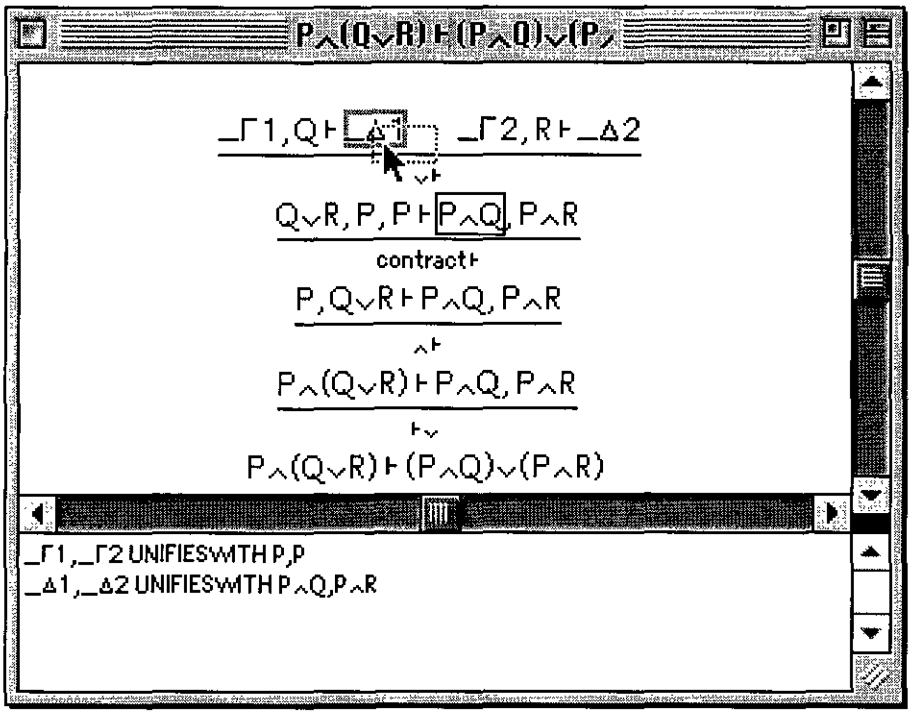
Resolving context splits

Many logical descriptions use context-splitting rules to illustrate how hypotheses are 'resources' for use in the proof of antecedents. This is taken seriously in linear logic[3] where *multiplicative* rules split both left and right contexts[1].

$$\frac{\Gamma,A\vdash\Delta \quad \Gamma',B\vdash\Delta'}{\Gamma,\Gamma',A\vee B\vdash\Delta,\Delta'}\ \text{multiplicative}\ \vee\vdash$$

fig 12: a multiplicative rule

Jape permits this kind of rule, and when it is applied defers a decision about the division of formulæ between left and right subtrees. It uses a proviso mechanism: the proof is acceptable provided that some way is found to make the split so that a specified collection of formulæ unifies with a specified collection of unknown contexts.



```
_Γ1,Q⊢_Δ1    _Γ2,R⊢_Δ2
          ∨⊢
  Q∨R,P,P⊢P∧Q,P∧R
        contract⊢
   P,Q∨R⊢P∧Q,P∧R
          ∧⊢
  P∧(Q∨R)⊢P∧Q,P∧R
          ⊢∨
 P∧(Q∨R)⊢(P∧Q)∨(P∧R)

_Γ1,_Γ2 UNIFIESWITH P,P
_Δ1,_Δ2 UNIFIESWITH P∧Q,P∧R
```

fig 13: a deferred context split, and use of drag-and-drop to resolve the ambiguity

It's possible, using Jape's previously-existing mechanisms, to complete many proofs, using identity rules (axiom, hyp) to demand unifications which resolve the context-splitting ambiguity. But it's confusing, it introduces a lot of unknowns onto the screen, and it requires the user to hold in their head a complex unification and search strategy. Better to use gestures to resolve the ambiguity, and Jape now employs drag-and-drop. When a user drags a formula instance[2] the engine consults its collection of provisos. If that instance is part of an unresolved context split, the interface is told the 'destination' variables into which the formula instance can be dropped[3]. Those variables – for example, Δ1 and Δ2 in figure 13 – highlight as the dragged formula passes by, and receive it if it is dropped into them. Evidently the gesture would be more efficient if the user could select more than one formula for dragging at a time;

[1] Linear logic is not yet encoded into Jape, because we haven't worked on the treatment of modalities in the last eighteen months. Writing this footnote is a spur: real soon now it will be done.

[2] GUI purists will note that we have two kinds of dragging: one to select text, the other to move a formula. This is to be regretted, but it seems to be as simple as it can be.

[3] If it isn't part of a split context, the engine tells the interface to ignore it; the user is allowed to drag it around, for example into a neighbouring window or onto the desktop, and whatever the OS does with dragged bits of text, it does.

perhaps our concentration on formula *instances* is a little fussy (for example when you have to drag one instance of *P* left and the other right, as in figure 13). But we feel that in principle the gesture is just the right one to resolve this kind of ambiguity, making context-splitting unification determinate.

Conclusions

If UIs are to affect the behaviour of TPs efficiently, we have to devise gestures which capture a user's intentions effectively and efficiently. It's necessary to do more than to resolve search ambiguity, or to describe in a single gesture a sequence of rule applications. It's necessary also to resolve the ambiguity inherent in rule application with a variety of gestures rich enough to convey those intentions concisely, but which still speak no more than the parsimonious vernacular of point and click, press and drag which is all that users, operating systems and programming-language libraries can deal with at present. Jape's formula-selection, substitution-selection (or subformula-selection) and context-dragging are a first step towards the goal of a UI that can truly communicate with a TP.

References

1 The Jape program, several logic encodings and an encoder's manual are available via the World Wide Web at two sites. Consult ftp://ftp.dcs.qmw.ac.uk/programming/jape/index.html and http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/jape.html.

2 Thimbleby H.W. (1990) *User Interface Design.* ACM press, New York.

3 Girard J-Y et al (1989) *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press

4 Bornat R and Sufrin B (1998) Roll your own Jape logic: encoding logics for the Jape proof engine. Available from ftp://ftp.dcs.qmw.ac.uk/programming/jape/papers

Mathematical Vernacular in Type Theory-based Proof Assistants*

Paul Callaghan and Zhaohui Luo
Department of Computer Science, University of Durham,
South Road, Durham DH1 3LE, U.K.
{P.C.Callaghan, Zhaohui.Luo}@durham.ac.uk

March 27, 1998

Abstract

In this paper we present the Durham Mathematical Vernacular (MV) project, discuss the general design of a prototype to support experimentation with issues of MV, explain current work on the prototype – specifically in the type theory basis of the work, and end with a brief discussion of methodology and future directions.

The current work concerns an implementation of Luo's typed logical framework LF, and making it more flexible with respect to the demands of implementing MV – in particular, meta-variables, multiple contexts, subtyping, and automation. This part of the project may be of particular interest to the general theorem proving community.

We will demonstrate a prototype at the conference.

1 Introduction: Defining a Mathematical Vernacular

The long term aim of the project is to develop theory and techniques with which the complementary strengths of NLP (Natural Language Processing) and CAFR (Computer-Assisted Formal Reasoning) can be combined to support computer-assisted reasoning. Our chosen area is mathematical reasoning, with the goal of implementing proof systems that allow mathematicians to reason in a "mathematical vernacular" (MV). Thus, mathematicians may work in a style and a language which is closer to traditional practice, rather than in a formal setting.

The MV project is a natural consequence of existing projects at Durham on Type Theory and on Natural Language Processing. The work on type theory includes coercive subtyping [10, 8], implementation of automatic decision procedures [21], and applications of type theory, eg in concurrency. The NL work is represented by the LOLITA system [19, 4]: this large project aims to build a general purpose platform with which specific NL applications can be built, eg information extraction [14], or object-oriented requirements analysis [12].

In the remainder of this section, we briefly discuss the notion of mathematical vernacular, and our type-theoretic approach.

Mathematical Vernacular The term "Mathematical Vernacular" has been used before with varying meanings, eg in [13] and [5]. We define it to mean a language which is suitable for ordinary mathematical practice, and which can be implemented with current technology and under the guidance of a formal semantics. The basis for implementation will be constructive type theory and its associated technology.

This MV is not necessarily the same as the language mathematicians commonly use, which we shall call "Informal Mathematical Language" (IML). Thus, we do not intend to model completely what mathematicians do, for several reasons. Firstly, IML is an informal solution to the problem

*This work is supported partly by the University of Durham Mathematical Vernacular project funded by the Leverhulme Trust (see http://www.dur.ac.uk/~dcs0zl/mv.html) and partly by the project on Subtyping, Inheritance, and Reuse, funded by UK EPSRC (GR/K79130, see http://www.dur.ac.uk/~dcs0zl/sir.html).

of communicating mathematics, and we can't be sure that it is the most effective way. Secondly, groups of people have their own idea about IML, of what is acceptable and what isn't, and so on. It would be hard to study such variation in opinion. Lastly, IML does not take account of the new technology for proof checking. This is an important point: we are still learning how to use the formal side of the technology, so an examination of how natural language may be productively used with it is certainly worthwhile. We discuss further methodological issues in section 4.1.

A Type-theoretic Approach Type theory provides a flexible language for expressing and checking a lot of non-trivial mathematics (eg [3]). The process of checking proofs (via type checking) is decidable for the type theories we will use, so it may be mechanised (implemented) straightforwardly. However, type theory and the technology based on it is hard to use: humans require a lot of training to use it effectively, and experts are not fully satisfied with the current implementations. One aim of our project is to contribute to this technology, especially in terms of being able to support the demands of MV.

Constructive type theory with dependent types is also a useful framework for natural language, eg in semantics (see Ranta [15]). The use of coercive subtyping may add flexibility to formal accounts of NL [11].

2 A Prototype for Experimentation

It is difficult to design an interactive system based on MV in a top-down style. We have ideas on various pieces of the problem, but tying these together is difficult. Thus, we need a prototype which will allow experimentation with different ideas. This prototype will also be a useful intermediate tool. Bear in mind that our interest is not only practical – we would also like to understand mathematical language and how it differs from general language – so prototype development will not be a pure 'engineering' matter.

This section outlines our initial design of such a prototype, discussing the basic requirements of functionality, and sketching the architecture which we will use to achieve this. More ambitious aims and future developments are discussed in section 4.2.

2.1 Establishing Requirements

We are taking a simple piece of mathematics as a guide in this early work. This step gives substance to our initial aim: it helps us gauge progress and to 'characterise' what the prototype is capable of. It is in general difficult to write exact requirement specifications for NL systems [4]. One possible reason for this is the sheer variability possible in NL and the complexity thus required of systems to handle this variability, and a general lack of techniques to talk precisely about NL.

Our informal aim is then that the prototype can accept and use a good selection of the possible ways of presenting the material in this simple development. We do not quantify this condition: it does not seem possible that we could do so in a meaningful way. Hence it is not a strict minimum requirement. Neither is it a maximum requirement. We shall generalise when reasonable and 'over-implement' to a certain extent based on our expectations for the next level of the prototype, eg with more complex grammar or more powerful logical facilities. Such considerations also affect our choice of architecture.

2.2 Required Functionality

For this prototype, we take as basis a simple mathematical development: of simple definitions on binary relations, and three proofs concerning those definitions. The development may be briefly summarised as:

- We introduce the notion of binary relations on a type or set.
- We define what it means for a relation to be reflexive, irreflexive, symmetric, and transitive. Two less common properties are also defined: 'separating' and 'apartness'[1].
- The negation of a relation is defined.

[1] The former defined as $\forall x,y \in U \cdot (Rxy) \Rightarrow \forall z \in U \cdot (Rxz) \vee (Ryz)$, and the latter that a relation is both separating and irreflexive.

- Theorem 1: the negation of a symmetric relation is symmetric.
- Theorem 2: if there is a 'top' element to which all others are related by some relation R, and R is symmetric and transitive, then it is also reflexive.
- Theorem 3: an apartness is symmetric.

We remark:

- It is mathematically simple, hence formalisation is straightforward and it can be expressed in simple natural language. Note that the above statements of the three theorems are the kind of language we would like to implement.

- Versions of this material appear in many text books in some form, eg introductions to discrete mathematics. It has also been formalised under several approaches, including Mizar [13] (comparing their MV to ours will be interesting).

- The mathematical basis is quite fundamental, so we do not need a large 'library' of basic concepts first. A key point of MV is that users define the terminology they are about to use. If we need an non-trivial library before studying a development, then the library itself should be used as the case study.

- Language-wise, there is potential for a good range of phenomena, such as mixing symbolic expressions and NL, different ways of referring to properties of relations (ie, some are simple adjectives which have noun forms – eg 'transitive' and 'transitivity', others as nouns without adjectival forms – eg 'apartness'), and the proofs are non-trivial enough to admit variations in presentation.

- Technically, the development includes quantifier reasoning, some equality reasoning, an operation on relations (negation), properties defined both independently and in terms of others. The proofs are short and simple.

There is no fixed NL version of this development[2]. In addition to textbook and Mizar presentations, we are conducting an informal empirical study of how people would express the proofs, including how type-theorists would explain the formalised version, how mathematicians would express that piece of mathematics, and how non-experts would do it. Thus, we have access to several independently-produced ways of expressing the ideas in IML, which we can use for development and testing.

2.3 Basic Architecture and Development Method

This discusses how we produce a prototype to satisfy the above aim, in terms of what components and techniques we will use. There are two aspects to this (type theory and natural language), which we discuss separately; then we discuss how we envisage the two interacting. As noted before, we will implement the prototype more generally than needed for short term goals, so in places we discuss techniques which are more complex than immediately required. The prototype will be constructed "bottom up", ie starting with the type theory levels and then implementing the NL functionality. The main implementation language is the non-strict functional programming language Haskell [6].

Implementing Luo's LF The type theory aspects will be based on Luo's typed logical framework LF (see [9], chapter 9). LF has not been implemented in any form before. The plan is to first implement a very basic proof assistant for LF, with a single context, definition mechanisms, basic refinement proof etc. We will then be able to experiment with ways of using type theory, as explained in section 3. A minor goal is to implement an efficient type checker in Haskell which will

[2] And there is no unique formalisation of the mathematics. This raises the interesting question of how we actually demarcate the development: in a sense, it is the mathematics involved, but in what form is this to be expressed? The various NL and formal versions are just expressions of some underlying idea. What is this underlying form? We leave this question open, since our concern is with how the manifestation in NL corresponds to a manifestation in a formal language (which then may be checked etc), and not with what may or may not be the 'real' process in-between.

be suitable for large-scale work, eg by implementing inductive types as a primitive notion, rather than as an extension to an existing system.

Using LF avoids the particularities of specific proof assistants (which are oriented towards helping experts prove things quickly) and of specific type theories (we can use LF to implement just the type theory we require). Thus we have more control over what happens – for example, we don't have to attempt to explain to the user when the proof assistant makes some complicated inference which would not be understandable to a novice type theorist. The LF implementation will also be useful in other projects, eg the coercive subtyping research at Durham.

Processing MV We will use a conventional NL analysis architecture[3]: essentially this is lexical analysis (normalising and categorising word forms), parsing (building the structure of a sentence), semantics (converting the tree to an expression in the semantic language), pragmatics (checking restrictions are observed and propagating other restrictions), and discourse (connecting the information from several clauses or sentences). Generating replies to the user will be implemented by using simple sentence patterns.

Parsing is (algorithmically) the most complex component: for the moment we shall use an implementation of Tomita's algorithm [20] due to Hopkins [7], as is used in LOLITA. This will allow us to write and parse heavily ambiguous grammars. It is almost certain that syntactic ambiguity will be present. For example, even the simple grammar in Ranta's prototype [16] contains ambiguity. However, a powerful parser introduces problems of (syntactic) disambiguation, of choosing which interpretation to use in further analysis. This will be considered when we get evidence about the behaviour of a realistic grammar.

Semantic and pragmatic interpretation will be implemented as operations on an abstract machine. This will hide detail of the type theory underneath, such as details of how concepts are actually represented. Part of this design is considered in [11]. Discourse will be implemented as a simple post-processor of semantic information.

As to the strategy for populating this infrastructure with rules, we shall start with the basic language sufficient to communicate the development, even if it is cumbersome to use, and then look at adding "surface language" conveniences such as forms of deixis[4] and more varied syntax.

Particular *new* problems to be solved include: mechanisms for introducing new terminology and for getting that recognised as such even with standard transformations of language[5]; interaction with symbolic expressions – particularly the understanding of these in order to perform semantic checking and interpretation, plus the translation of these to formal notation.

One distinction that may be useful is the language used for definitions vs. that used for proofs. Definitions and statements of theorems must be expressed with precision – they are statements of ideas or concepts in a mathematician's mind, and it is generally impossible to infer missing details. Contrast this with proof, where competent mathematicians can fill in the missing details by inference, thus the communication does not need to be as precise.

How the Prototype will Work The type checker will be used during NL analysis, thus interspersing analysis and interpretation to a degree (the alternative is to perform all type checking once analysis has finished). One consequence is that type information is potentially available for disambiguation – although we need to experiment to see how effective or useful this is. Another consequence is that errors can be detected quickly and the user given useful error messages, rather than just saying 'yes' or 'no' to a sentence.

We would also like to use the type theory level to handle parts of NL analysis where appropriate. For example, anaphora could be attempted purely at the NL level, but some anaphoric references can be interpreted as meta-variables and attempted with the automatic reasoning procedures. (NB especially above sentence level, the factors governing anaphora are heavily dependent on domain

[3] See [1, 19] for more information.

[4] Indirect references to objects – eg use of pronouns and partial expressions like 'it', "the group", etc. Anaphora is the common case where the referent occurs in previous context.

[5] A simple example: transitivity being introduced as an adjective 'transitive', and later being referred to with the noun 'transitivity'. Where we allow phrases containing more than one word, more complexity is possible, eg "monotonically increasing with respect to x", "increases monotonically wrt. x", "monotonically increases" (with the variable left implicit), etc.

factors, such as types of objects – hence this is reasonable.) Coercive subtyping is another good example [11]: we can handle aspects of NL analysis at the type theory level (see section 3.4).

The style of operation is expected as follows. Users will make statements, which are possibly incomplete – eg requiring pre-conditions or additional formal detail. The NL analysis translates such sentences into a representation of this imperfect information, using meta-variables to represent the incompleteness; eg steps in proofs will give rise (internally) to proof terms. Cues from the sentence will be used to classify the meta-variables, and depending on the classification, different kinds of automatic reasoning will be employed to resolve the omissions. For example, statements marked with phrases like "obviously" or "qed" can be viewed as assertions that a proof exists and is easy to obtain in the current mathematical context – automatic reasoning will try to find one by combining the proof steps (ie the internal proof terms) the user has previously described.

3 Technical Issues for Supporting Implementation of MV

This section outlines work in progress on the type theory side of the system. The basis of this is an implementation of Luo's LF in Haskell, as explained above. We discuss issues of how this type theory is used, ie what support we give the user. These are points which we have also identified as being necessary or extremely useful in an implementation of MV; they are also open questions in contemporary proof assistants.

3.1 Meta-variables

Meta-variables are 'holes' or 'place-holders' in a proof. They must be replaced with a concrete term before a judgement can be decided. They have known types, but of course the types may contain other meta-variables. There are several uses of meta-variables:

- Productivity: less important details of a proof can be left whilst the key details are explored. For example, proofs which are simple but tedious to do formally can be left, and supplied when the main proof is complete.
- Flexibility in creation of a proof. Lemmas which are found to be useful can be 'assumed' for the purposes of a proof, then proved separately when convenient.
- Interface with automation. Meta-variables can be used to represent information which the user believes should be inferable. Hence automation can be used to try to remove (or satisfy) meta-variables.
- For MV, to allow direct expression of NL statements in the formal system. The alternative is to implement some mediator which collects NL information and when possible outputs completed formal expressions. Clearly, use of meta-variables is more flexible.

However, meta-variables cannot be used like normal variables because of logical difficulties. In particular, they should not be abstracted over – they cannot appear in the body of a newly created lambda abstraction, for example. The term used to satisfy a meta-variable must be constructed from the entities available when the meta-variable was introduced. Abstraction (and then application of this abstraction) alters this context – the information that the metavariable could have used some x is lost when x is abstracted and then replaced by some arbitrary term E.

One solution is the restriction that meta-variables are eliminated (ie satisfied) before abstraction occurs. Another solution is to annotate meta-variables with the terms that can be used in their satisfaction – but this is much more complex than the first solution.

Meta-variables will be implemented as variables with restrictions on how they are used, and their elimination as a Cut operation. Cut is, however, an expensive operation – it essentially involves global substitution of the meta-variable, hence a rewrite of the relevant context. This could be made less expensive through lazy evaluation (ie the non-strictness of Haskell), and by the structure of multiple contexts restricting the amount of rewriting needed.

3.2 Multiple Contexts

Most proof assistants implement a *single* context, that is, development proceeds in a single line, and a new proof cannot be started until the previous one is completed or abandoned. Clearly, this is inflexible. However, how to design an alternative is not clear, and like meta-variables, the concept needs careful design. Multiple contexts may have a deep effect on treatment of meta-variables, and vice-versa. Allowing multiple contexts could have several benefits, depending on the scheme chosen:

- It will allow flexible development of proofs – in particular, allowing the user to introduce lemmas freely when needed, or to develop several proofs in parallel. This is very powerful.
- For NL, different contexts of development can be used to represent scoping of names and concepts, eg separating re-use of standard names like x, R.
- Libraries: can be represented as parts of the context, and 'imported' when names from them are used. One could also extend or create new libraries more easily. The problem of organising a large development, which includes formation of libraries, is frequently mentioned (eg [3]).
- The structure of the multiple contexts is likely to be a graph which represents dependency between objects of the development (see below), thus it can represent association between objects. This can replace fixed linear structure in type theory to provide a more natural view. For example, a prototype group g can be introduced, and then the components of g introduced and noted as being related to, or dependent on, g. Then, when a theorem based on g is used in another context for another group g', the user would have to supply objects which are related to g' in analogous fashion.

We are considering the following scheme. A 'context' is the set of *objects* which are in scope at a particular time. Objects can be in several scopes simultaneously, eg a lemma which is used in several theorems. Objects in a given context have unique *names*; obviously, identical names in different contexts can refer to different objects. Contexts can be joined by importing objects from other contexts, eg by applying a theorem in the current proof. (Obviously the context used in developing the applied proof should remain hidden.)

Many issues still need to be considered and experimented with. One is how theorems developed inside a multiple contexts framework are used. We could do a conventional 'abstraction' or discharge operation, and then reapply to local terms. Note that this operation is not a procedure a mathematician would recognise: they seem to use a Cut-like operation, essentially substituting local terms for the variables in a proof statement. This cut operation seems more natural than abstraction for multiple contexts.

3.3 Questions of Automation

A useful implementation of MV should contain automatic reasoning, in order to support the user. For MV, techniques which help a formal mathematician may not be useful. What we need is techniques to fill in the small details a mathematician would typically omit, and to tie steps of proofs together to prove the whole.

Meta-variables provide 'hooks' for automatic reasoning, by representing the places which need attention. We can also differentiate between kinds of meta-variable to provide appropriate treatment for certain cases of reasoning. As mentioned above, the user's statements will be translated in to terms containing meta-variables, and automation will attempt to build complete proof terms from these, eg when the user claims a proof is complete.

Several kinds of automation will be required, such as type-theoretic model checking [21], or the various procedures implemented in the verification tool PVS [18]. Interfacing to computer algebra systems will also be useful for heavy calculations.

3.4 Use of Coercive Subtyping

Coercive subtyping aids use of a type theory by providing a type-secure abbreviational mechanism [10]. As noted in [11], it can be used to simplify implementation of MV, in particular expressions denoting classes of mathematical object. For example, if 'finite' is defined for sets, and 'group' is declared as a subtype of 'set', then the construction representing "finite group" is well-typed and immediately understandable as meaning "group whose set is finite". No further action is necessary, eg NL analysis does not need to make this inference.

Forms of coercive subtyping have been implemented in Lego [2] and Coq [17], but have certain restrictions, such as working on syntactically equal terms rather than computationally equal terms (the latter being more general). Implementing subtyping at a more fundamental level, ie in LF, avoids such restrictions [10]. The implementation will also complement theoretical work on subtyping, eg [8], by helping to explore ideas.

4 Discussion

We have presented the initial design of a prototype for interactive development of mathematics, in terms of a first goal of functionality and the architecture we will use to achieve it. We discuss a few methodological points, and then outline future directions for the project.

4.1 A Bottom-Up Strategy

There are many methodological issues to be considered in an interdisciplinary project like this. For example, how do we define what our requirements are, and how do we evaluate the results, or quantify the worth of the work? De Bruijn noted in his design of an MV that there were many arbitrary decisions to make [5]; what guidance do we have in such cases? These questions are hard to answer.

One way in which we tackle them is by adopting a "bottom-up" strategy of working. That is, we look at pieces of the problem that we can understand, and fit them together to establish a coherent, if limited, whole. The prototype is one example of this. We shall use it to explore further issues. Also bear in mind that our intention is to develop the *technology* for a successful future system, and not to create a system in the short term.

4.2 Future Developments

The chosen development is a modest start, so clearly there are many aspects of mathematics which it does not contain. The following two aspects are the obvious next steps, and will allow us to do more substantial examples.

Algebraic Structures Definition of structures that satisfy certain axioms, and examination of how such structures relate to other structures is a central part of mathematics. An obvious example is of 'group'. MV must allow us to define the notion of group, and then to use that definition in the usual ways, including access to the component parts and use of the axioms of groups. A particular problem is correct handling of proof terms, especially since these will not be visible to users. We may also allow different axiomatisations of structures; this is useful if mathematicians wish to explore the relationships between axiomatisations. Multiple contexts may support such activity.

Induction We shall concentrate on induction with natural numbers, this being the form most used by mathematicians. We will need to implement the basic language, and the necessary automatic reasoning to support it. Induction is a standard form of argument, so in an interactive system we could prompt the user for the necessary cases, and calculate what those cases are. Note that a proof by induction often begins by the user stating this is the method he will use.

References

[1] James Allen. *Natural Language Understanding.* Addison Wesley, 1995. (2nd Ed.).

[2] A. Bailey. Lego with coercion synthesis. `ftp://ftp.cs.man.ac.uk/pub/baileya/-Coercions/LEGO-with-coercions.dvi`, 1996.

[3] A. Bailey. *The Machine-checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998. (submitted).

[4] Paul Callaghan. *An Evaluation of LOLITA and Related Natural Language Processing Systems.* PhD thesis, Department of Computer Science, University of Durham, 1998.

[5] N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In Nederpelt, Geuvers, and de Vrijer, editors, *Selected Papers on Automath.* 1994.

[6] Haskell. Haskell report 1.4. `http://www.haskell.org`, 1998.

[7] Mark Hopkins. Demonstration of the Tomita parsing algorithm. `ftp://iecc.com/pub/-file/tomita.tar.gz`, 1993.

[8] A. Jones, Z. Luo, and S. Soloviev. Some proof-theoretic and algorithmic aspects of coercive subtyping. *Proc. of the Annual Conf on Types and Proofs (TYPES'96)*, 1997. To appear.

[9] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[10] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 1998. To appear.

[11] Z. Luo and P. Callaghan. Mathematical vernacular and conceptual well-formedness in mathematical language. In *Proceedings: Logical Aspects of Computational Linguistics*, 1997. (forthcoming in LNAI series).

[12] L. Mich. NL-OOPS: From Natural Natural Language to Object Oriented Requirements using the Natural Language Processing System LOLITA. *J. Natural Language Engineering*, 2(2):161–187, 1996.

[13] Mizar. Mizar WWW Page. `http://mizar.uw.bialystok.pl/`.

[14] R. Morgan, R. Garigliano, P. Callaghan, S. Poria, M. Smith, A. Urbanowicz, R. Collingham, M. Costantino, C. Cooper, and the LOLITA Group. Description of the LOLITA System as Used in MUC-6. In *Proceedings: The Sixth Message Understanding Conference*, pages 71–87. Morgan Kaufman, Nov 1995.

[15] A. Ranta. *Type-theoretical Grammar.* Oxford University Press, 1994.

[16] A. Ranta. A grammatical framework (some notes on the source files), 1997.

[17] A. Saibi. Typing algorithm in type theory with inheritance. *Proc of POPL'97*, 1997.

[18] SRI. The Prototype Verification System. `http://www.csl.sri.com/pvs.html`, 1998.

[19] The LOLITA Group. *The LOLITA Project.* Springer Verlag. (forthcoming in 3 vols: "The System Core", "Applications", "Philosophy and Methodology").

[20] M. Tomita. *Efficient Parsing of Natural Language: A Fast Algorithm for Practical Systems.* Kluwer Academic Publishers, Boston, Ma, 1986.

[21] S. Yu and Z. Luo. Implementing a model checker for Lego. *Proc. of the 4th Inter Symp. of Formal Methods Europe, FME'97: Industrial Applications and Strengthened Foundations of Formal Methods, Graz, Austria. LNCS 1313*, 1997.

Using ILF as an Interface to Many Theorem Provers

Ingo Dahn
Mathematical Institute
Humboldt-University Berlin
Ziegelstr. 13a
D-10099 Berlin
dahn@mathematik.hu-berlin.de

May 28, 1998

Abstract

The ILF system offers a uniform interface for several automated theorem provers. It does not assume that the user has any knowledge of the particular language and options of the single provers. The menu system is restricted to setting the most general parameters, common to all provers. More specific options can be set by configuration files. Use of the provers is facilitated by a sorted input language, visualization of proof structures and generic natural language proof presentation in LaTeX and HTML.

1 Introduction

One of the major aims of the German research program "Deduktion" (1992 - 1998) was to combine the power of several automated theorem provers. The ILF system [DG97] attempts to realize this. Moreover it intends to make current reasoning technology available to the non-specialist. Its interface has been designed to simplify

- the use of many theorem provers
- by users familiar with informal proofs but with no acquaintance with theorem proving technology
- in a large variety of situations, possibly in cooperation with other systems.

During the early years of the research program it became clear that the integration of technologies from different provers within a single new prover could not be achieved. At that time, the ILF system, which has been developed at the Humboldt University Berlin since 1988 under the name FLEXLOG, offered the possibility to call the automated theorem prover OTTER from within several small interactive theorem provers. The system, previously developed under MS-DOS, had been redesigned and implemented to work under UNIX. This made it possible, to run several copies of OTTER simultaneously in the background while the user could continue his work with an interactive theorem prover in the foreground.

It was only natural to use the experience gained with the integration of OTTER [MC90], to integrate other provers developed within the "Deduktion" program in very much the same way. The equational prover DISCOUNT [De95], developed at the University of Kaiserslautern, was the first German prover used this way. Soon thereafter, it was followed by the provers SETHEO [GL94] (Technical University Munich) and KoMeT [BB94](Technical University Darmstadt). Later, the provers PROTEIN [BF94] (University Koblenz), SPASS [WG96] (Max-Planck-Institute Saarbrücken) and CM (Humboldt University Berlin) have been added. The integration of the equational prover WALDMEISTER (University Kaiserslautern) and the tableau prover 3TAP (University Karlsruhe) has been recently completed. We mention that ILF has also a COMPETITOR prover which runs several of the aforementioned provers in a competition and a COOP prover which lets two of these provers exchange information at run time.

Each of these provers has its own input and output syntax. Each has its own variety of switches, parameters and flags. None of them had a graphical user interface. When deductive systems are to be used outside their own circles they are confronted with potential users without knowledge about logical calculi. These users will not be willing to memorize a variety of input languages and parameter settings. Only accidentally their proof problems will be in clausal form. They just want to specify a proof problem, give it to the system and see the answer. Even inside the deduction community, the situation is not very much different when developers of theorem provers test provers from other groups.

Obviously, a unified interface was needed that hides the differences between the provers as much as possible from the user. However, it was another reason that enforced the development of a graphical user interface for the ILF system in an early stage of the development.

Recall that we have a user in mind who considers theorem provers just as a tool for his work. Consequently, he will not allow the tool to interrupt the main work he is doing, i.e. the provers should do their job whenever they can quietly. Hence, normally, the ILF user launches several automated theorem provers simultaneously and these provers run asynchronously in the background on several machines in the local network. Each of them generates output files and messages. While most of the output files are kept for automated post-processing, a selection of the messages - especially those concerning success or failure of the single proof jobs - has to be presented to the user. It is quite inappropriate to let these messages interfere with user input. Hence, a separate window for prover messages is needed. At that time, Tcl/Tk was not yet available. Therefore, the user interface was implemented using X11/Motif. This interface will be explained in the following sections.

ILF is available for education and research free of charge with update and installation service. The update service includes updates of the integrated automated theorem provers.

The major design principles of ILF were

- integration of several automated theorem provers without any change of their code
- support for asynchronous work of several provers in the local network
- common parameters of provers can be configured by common commands, specific parameters can be configured by configuration files

- reuse of preprocessing steps (e.g. clause generation) for several provers
- user access to Prolog to control prover systems
- high level typed input language with automated translation into an untyped first order language
- support for a variety of user interfaces (GUI, command line, EMACS, WWW, Mathematica)

2 The Windows

The user interface of the ILF system can be configured in many ways. Especially, it is possible to work without the graphical user interface, directing all output into the same stream and reading from standard input. This variant of ILF is used as part of scripts when ILF works as a server for other systems. Also interactive use is possible when automated provers are not used, e.g. to generate natural language presentations of existing proofs. However, the default user interface comes with four windows.

The *main window* is used for the dialogue with the user. It has a menu bar, an output area and an editable command line. Lines can be copied for reuse from the output area to the command line.

Though ILF can generate LaTeX presentations on the fly, the main window can be also used for a quick review of proofs or axiom systems. Through this window the ILF user has also complete access to the underlying Prolog system. Any Prolog command he enters will be immediately executed. It is possible to write Prolog commands which call ILF procedures to load theories, configure automated provers, launch them, analyze their output and generate natural language proof presentations. User defined commands can be re-loaded at run time or auto-loaded at start-up.

Also system messages go to the main window. ILF comes with the interactive theorem prover PROOFPAD. The PROOFPAD calls automated provers in order to close gaps in the proof edited by the user. The main window is also the user interface of the PROOFPAD, which adds its own special menu.

It was mentioned above that the ILF core system consists of two separate parts - the foreground system communicating with the user and the background system communicating with automated provers (called experts). The ILF user will not see prover output. Nevertheless, this output is kept in files for some time for use by the prover expert. ILF analyzes prover output and displays appropriate messages in the *expert window.* This window does not have menus and cannot be used for input. Normally the user is informed which prover is working on a particular job and whether it succeeded or not. For provers running on a single machine, the name of the selected host is also displayed. These informations are sufficient to review the native prover output files if necessary. Output from the foreground and background system is logged for later inspection.

The remaining two windows of the ILF system are controlled by a separate program - the TREEVIEWER. This tool can display labelled directed acyclic graphs. It is used to display proofs, term structures and hierarchies of theories. When proofs are displayed, nodes of the graph are labelled with formulas. Since these can be quite complex, they are displayed in a separate window. Both windows are synchronized, i.e. labels are hidden in the second window when their nodes are hidden in the first. Nodes can be colored (e.g. to characterize nodes belonging to the same subproof) and marked with one out of ten user-editable symbols (e.g. to distinguish proved nodes from unproved).

It is of special importance that the TREEVIEWER can be used as a graphic input device for the client program (ILF in our case). So the user can select a location in a proof with the mouse or select a formula, edit it and send it to the ILF core system. The client program can add menus at run time to each of the two windows of the TREEVIEWER. It can also request from the TREEVIEWER informations on the status of each node (hidden or displayed, contracted or separated).

When the graph is changed by adding or removing leafs, changing labels, colors or attached symbols, only the changes have to be retransmitted. This enables the TREEVIEWER to display dynamically a proof in progress.

Internally, ILF transforms proofs found automatically into a standard format describing their structure as a labelled directed acyclic graph. Then, uniform procedures can be applied to display all these proofs from different sources with the TREEVIEWER.

ILF has its own input language to specify proof problems. This language has an order-sorted polymorphic type system. Theories are automatically translated into input files for the single provers. This includes the conversion into clauses with the normal form transformation of the KOMET prover, the addition of equality axioms and the encoding of type checking into unification if necessary and possible. Of course, this is kept transparent for the user.

The input language is an important part of the user interface. In software and hardware verification it is frequent that quite complex expressions have to be written. This is facilitated by the ILF input language. So names of variables can be reserved for specific types and the user can specify abbreviations which are unfolded during the parsing process. When a proof is edited interactively within ILF, the same abbreviations are available too. Also when theories or proofs are displayed, the abbreviations will be applied. Sometimes this can have the effect, that a formula is displayed in a more compact way than entered by the user. The ILF type system requires that each term has a unique minimal type. When it is necessary and possible to add missing type parameters to achieve this, ILF will do it tacitly. If not, a parser error is generated. The use of types is not necessary. There is a more simple format for untyped theories which are automatically converted into typed theories.

There is a separate small generic editor GENEDIT installed with the system which supports user defined templates. Templates for editing typed and untyped ILF theories are included.

3 The Menus

By default there are three menus - a command menu, a menu for working with theories and proofs and a menu for configuring the background system. The command menu has items for switching the Prolog occur check and for closing the system.

The theories menu contains items and submenus to edit load and display typed and untyped theories. Theories are stored in a database on disk. This database caches also preprocessing steps for the single automated provers for later use. This accelerates the generation of prover input files when several proof jobs use similar axiom systems. Consequently, there is a menu item which enables the user to erase the database. This can be automated - if desired - by including an appropriate command in a theory or in a file which is auto-loaded on start-up.

Proofs can be loaded from the same menu for display with the TREEVIEWER or in natural language with LaTeX or HTML. These proof presentations transform the native proofs into block structured proofs [DW96] and apply a series of further proof transformations to enhance the readability of proofs. This sequence of transformations depends on the prover which has found the proof. It can be changed by the user. Proof presentation technology was described in [DW96]. It is currently available for untyped proof problems for all integrated provers except OTTER. Since anything can be proved correctly from an inconsistent theory, proof presentation has turned out to be a valuable tool to discover errors in formal specifications. It is also useful to demonstrate the results of the work of theorem provers to the ILF user without requiring acquaintance with a formal calculus.

By varying the typesetting declarations (see next section) the user can easily change the form in which proofs are displayed. These declarations can be edited in a file which can be reloaded with an appropriate menu item without restarting the system. Also the level of detail used in proof presentations can be set by the user.

The second default menu has items to connect with any of the integrated provers. When such an item is selected, the prover isn't launched yet. The background system only creates a data module for a proof job for this prover and prepares a default configuration. Moreover, the foreground system adds a menu to access basic properties of the prover configuration. It is possible to prepare several proof jobs for the same prover at the same time.

In earlier versions it was possible to change each prover setting by a menu item. However, it turned out that users, inexperienced with the possible settings of all the available provers, where overwhelmed by the many parameters and switches that could be set. On the other hand, specialized users being very familiar with the particular possibilities of a single prover, do not need support by an extensive menu system. They are used to write configurations or command line options. Therefore most prover menus have only items to change settings which concern the integration of the prover into the ILF system. This includes an item to load a suite of prover settings from a configuration file which has been prepared by the prover authors, possibly modified by the user himself.

In order to explain the possibilities common to all provers it is necessary, to say a few words about the way ILF handles proof jobs. Such jobs can be created and launched automatically. For each of these jobs a prover is launched on one or more machines in the local network. ILF guards the network resources and delays jobs if necessary. Each prover is guaranteed a minimal wall-clock time to work (*flexykill-time*). If this time has expired and there are more proof jobs waiting, the prover is killed in order to free resources. Moreover, there is a maximal amount of time (*max-time*) which the prover is allowed to work. flexykill-time and max-time can be modified for each job by menu items.

For some provers there are advanced menu items which are invisible at start-up. For example, the selection of such an item can modify the way ILF adds equality axioms to input files generated for the prover SETHEO. Which of these advanced menu item are shown depends on a numeric parameter called *menu-level*. There is a menu item for changing this parameter and there are *refresh*-items on the prover menus to extend or shrink the menu.

The menu system is read from a file. By editing this file, the user has complete control over all menus. For example he can easily add menu items to call procedures which he added to the ILF system. Each line in this file describes a menu or submenu or a menu item. Each of these entries specifies at which menu level the item occurs, what is written in the menu and which action has to be taken when the item is selected. For the leafs of the menu tree these actions insert a command into the command line of the main window. This helps the user to memorize the main ILF commands. The user can modify the command line, e.g. by adding parameters. For some commands which do not require parameters it can be specified that the selected command is immediately executed, without an additional confirmation by the user.

4 Configuring the Systems

Each ILF installation has a central part and a user specific part. As a principle, each required information for configuring ILF or the integrated provers, is taken from the user specific part whenever possible. This concerns especially the basic configuration file which can include other user specific files like menu settings. In this file it is specified which components of the ILF system are to be used, especially which automated provers are made available. Also the use of the ILF type encoding mechanism is specified here.

This is also the place to modify the details of natural language proof presentation. ILF uses context-dependent templates to determine the typesetting of formulas and proofs. The user can easily add templates to determine the typesetting of the formulas he is actually working with. In fact, ILF has possibilities to create these templates for presentations in LaTeX and HTML from a single generic format. Formulas or rules of inference that are to be suppressed in proof presentations, as well as the amount of explicit references can be set by the user. Advanced users can modify the default sequence of proof transformations which is applied before a proof is presented. For example, by switching off all proof transformations it is possible to inspect the proof in a form very close to that generated by the prover.

The configuration of automated provers is determined by configuration files. In fact, a default configuration is considered first which can be later modified by user specific settings. These can change a large variety of prover specific parameters. Especially it is possible to specify strings which are directly inserted by ILF into prover input files or prover command lines. Thus prover-internal changes concerning these parts will not affect the integration of the prover into the ILF system.

Recently, ILF has been extended by the DBFW tool [JW97] developed at the Technical University Munich. This tool extracts automatically statistical informations of runs of provers within ILF. These informations are regularly mailed to the developers of the prover for support purposes, provided that the ILF user has enabled this procedure.

Thus, besides its main purpose to facilitate the use of a variety of provers, ILF can help prover authors to develop their products in a way more directed to user needs.

5 Interfaces in Use

Most users prefer the graphical user interface, switching off particular components which they don't need for a specific job. It is very popular to modify ILF's menus by inserting individual commands, especially to call Prolog programs for preparing individual prover jobs. On the other hand, the possibility to insert user specific menus in the TREEVIEWER and to modify them dynamically, is seldom used. Also, users tend to stay with the default prover configuration. This emphasizes the value of provers which are able to configure themselves, like in the automode of the OTTER prover. In case of the PROTEIN prover, ILF will modify the configuration automatically to select the right way to treat equality in PROTEIN.

The way ILF outputs formulas can be easily modified by the user. This is widely used. The procedures used to transform proofs for better readability can be also selected by the user. However, this requires acquaintance with the available procedures and is hardly applied to replace the default procedures.

Though ILF works internally with a typed language, many users from the deduction community have only first order problems. Therefore, it was necessary to design a more simple input language for first order theories together with an automated translation into typed theories. But for more realistic problems, polymorphic types are necessary. The ILF input language supports such types. The parser will automatically augment type parameters when they can be inferred. Writing complex terms in theories is facilitated by user-defined abbreviations which are automatically unfolded by the parser and folded back by the ILF presentation tools. Note that this has the consequence, that terms are sometimes presented in a form different from the form in which they were entered. However, the resulting reliability in communicating complex terms with the user is a more than sufficient compensation for this.disadvantage.

The command line interface is mainly used to run large experiments with provers from skripts. ILF has also a possibility to communicate over the internet, either via a web server using a cgi skript or directly using socket communication. This is mainly applied to give remote users access to the ILF mathematical library from a NETSCAPE web browser or from within the computer algebra system MATHEMATICA.

6 Work Accomplished with ILF

ILF has been used in a variety of ways. Perhaps most popular is its use as a proof presentation tool. This has turned out to be a valuable help in discovering deficiencies in the work of provers as well as inconsistencies in formal specifications. It was also an indispensible tool to extract the mathematical content of the solution of ROBBINS' problem by BILL MCCUNE's prover EQP, since it permitted to look at this proof from several views.

The interactive prover PROOFPAD has been used e. g. to verify some communication protocols, to edit a proof of the intermediate value theorem in calculus and a proof of BUSULINI's theorem on lattice-ordered groups. These experiments confirmed that automated provers make it possible to use an interactive prover without knowledge in formal logic.

Many experiments have been made with ILF to evaluate the power of the integrated provers for various applications. In cooperation with a software engineering group at Braunschweig university, they have been tested on problems to be solved for selecting components from a software library. Within these experiments, the provers SPASS and DISCOUNT have been used also in an ILF configuration where they communicated at run time. This was developed in cooperation with DIRK FUCHS from Kaiserslautern university.

ILF has a mathematical library which consists of a data base, extracted from 50 articles from the mathematical library of the MIZAR system. ILF's proof presentation technology has been used to provide Web access to this library. This has demonstrated the benefits of an electronic library which genereates texts dynamically on user demands. Automated theorem provers have been used in a competitive way in ILF to search the library for theorems in an intelligent way from within MATHEMATICA. The contents of the library can be exported in a format that can be easily parsed by Prolog or Lisp based systems. This has been used to generate a series of about 100 proof problems as a test suite for automated theorem provers.

References

[BF94] P. Baumgarter, U. Furbach: Protein: A PROver with a Therory Extension INterface. In Proc. CADE-12, pp. 769–773, Springer, 1994.

[BB94] W. Bibel, S. Brüning, U. Egly, T. Rath: KoMeT, in Proc. CADE-12, pp. 783–787, Springer, 1994.

I. Dahn, C. Wernhard: First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. RISC-Linz Report Series, No. 97-50, pp. 58–62, Johannes Kepler Universität Linz, 1997.

[DW96] B. I. Dahn, A. Wolf: A Calculus Supporting Structured Proofs. Journal for Information Processing and Cybernetics(EIK), (5–6): pp. 261–276, 1994.

[DW96] B. I. Dahn, A. Wolf: Natural Language Presentation and Combination of Automatically Generated Proofs. In Proc. FroCoS'96, pp. 175- -192, Kluwer, 1996.

[DG97] B. I. Dahn, J. Gehne, Th. Honigmann, A. Wolf: Integration of Automated and Interactive Theorem Proving in ILF. In Proc. CADE-14, pp. 57-60, Springer, 1997.

[De95] Denzinger, J.: Knowledge-Based Distributed Search Using Teamwork, Proc. ICMAS-95, San Francisco, AAAI-Press, 1995, pp. 81-88.

[GL94] C. Goller, R. Letz, K. Mayr, J. Schumann: SETHEO V3.2: Recent Developments (System Abstract). In Proc. CADE-12, pp. 778–782, Springer, 1994.

[JW97] P. Jakobi, A. Wolf: DBFW: A Simple DataBase FrameWork for the Evaluation and Maintenance of Automated Theorem Prover Data (incl. Documentation), SFB-Bericht 342/28/97 A, Technical University Munich, 1997.

[MC90] W. McCune: Otter 2.0. In Proc. CADE-10, pp. 663–664, Springer, 1990.

[WG96] C. Weidenbach, B. Gaede, G. Rock: Spass & Flotter, Version 0.42. In Proc. CADE-13, pp. 141–145, Springer, 1996.

User interfaces in natural deduction programs

Hans van Ditmarsch[1,2,3,4]
University of Groningen
Cognitive Science and Engineering / Computing Science
Grote Kruisstraat 2/1
9712 TS Groningen
The Netherlands
hans@tcw2.ppsw.rug.nl / hans@cs.rug.nl

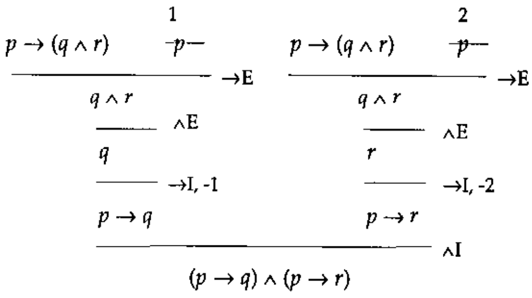
Abstract

Various programs to assist learning natural deduction for first order logic are presently available. Relevant issues when comparing them are: the version of natural deduction being taught, how proofs are visualized (notably whether proofs are trees or sequences), whether both forward and backward reasoning are supported, the availability of global, tactical and strategic help and 'debugging' facilities, and proof checking. Five natural deduction proof assistants are compared within this framework: MacLogic, Jape, Hyperproof, Carnegie Mellon University Proof Tutor, and Symlog. Other programs are briefly mentioned. Jape's interface is visually most appealing and least biased towards forward or backward reasoning. Hyperproof's natural deduction assistant suits proof checking very well. Symlog's help facilities are remarkable. Interface design for full-fledged theorem provers can profit from experiences with simple proof assistants.

1 Introduction

A natural deduction proof can be visualized in different ways. As an example of that we give two different visualizations of the same proof of $p \to (q \wedge r) \vdash (p \to q) \wedge (p \to r)$:

1	1	$p \to (q \wedge r)$	assumption
2	2	p	assumption
1,2	3	$q \wedge r$	2, →E, 1, 2
1,2	4	q	∧E, 3
1	5	$p \to q$	→I, 2, 4, -p
3	6	p	assumption
1,3	7	$q \wedge r$	2, →E, 1, 6
1,3	8	r	∧E, 7
1	9	$p \to r$	→I, 6, 8, -p
1	10	$(p \to q) \wedge (p \to r)$	∧I, 5, 9



What are the differences? The proof on the right has a tree-like representation, the one on the left a linear representation. In the linear proof assumptions are withdrawn by repeating them, as -p in line 5, in the proof tree they are referred to by number. Assumption $p \to (q \wedge r)$ occurs once in the linear proof, but occures twice in the proof tree. A certain two-dimensional proof symmetry is noticeable in the proof tree but less so in the linear proof, this being a one-dimensional proof format. Assumptions are explicit (by reference) in every step of the linear proof, e.g. 1, 3 in line 8, whereas in the proof tree they remain implicit on the level of a single node.

Proof visualization is just one issue when comparing interfaces for natural deduction proof assistants. In section 2 we will discuss the issue of visualization and other relevant issues. In section 3 we will

[1] The author thanks the following persons for their comments: Richard Bornat, Roy Dyckhoff, Keith Stenning, Johan van Benthem, Gerard Renardel, Jon Oberlander, Dave Barker-Plummer, Richard Scheines, Rix Groenboom, Rineke Verbrugge, Pietro Cenciarelli.

[2] Part of this research has been carried out at the Human Communication Research Centre, Edinburgh University, on a grant by the Netherlands Organization for Scientific Research (NWO).

[3] A strong motivation for this paper is [Goldson et al. 93].

[4] Links to further information on software discussed and mentioned here can be found on `http://tcw2.ppsw.rug.nl/~hans/logiccourseware.html`.

compare five natural deduction programs. The conclusions are in section 4. We finish with conjectures on future developments.

2 How to compare natural deduction programs

We will not address general grounds on which to compare educational software, such as platform portability, and the kind of window management facilities. We restrict ourselves to interface issues that are related to proof assistance. Unless expressly mentioned, we will refrain from value judgements and present different options as the alternatives they are. Some relevant issues are:

- the version of natural deduction
- proof visualization
- forward and backward reasoning
- proof help
- the method of proof checking

We relate the issues to the example above. Below the level of mathematical interest of users of theorem provers but well above the level of student perception is what we call a *version* of natural deduction: instead of a two-case $\wedge$I-rule, as in step 10 of the linear proof above, one might prefer a multiple-case $\wedge$I-rule. The matter of *visualization* will be already clear. Naturally, we'd want a program to allow one to introduce steps 5 and 9 given 10, thus constructing the proof by *backward reasoning*, as well as to proceed by *forward reasoning*, going from say step 1 and 2 to step 3. And what kind of *proof help* is available when one is stuck: is there truly sensible advice such as in step 1 of the proof 'assume the antecedent of a given implication, in order to apply $\rightarrow$E'? How about *proof checking*: suppose the user has entered q instead of p as assumption in step 2, does the system allow him to proceed beyond step 3, so that he can at least finish what he *thinks* to be the entire proof, or not? That some of these issues are truly 'issues' might come as a surprise to a regular user of theorem provers. It turns out that what is normal for theorem provers often is not realized in natural deduction courseware. We will now discuss the issues in some detail.

2.1 Version of natural deduction

A multiple-case rule for $\wedge$I is more 'natural' than repeated application of a two-case rule, which is to be preferred for proof theoretical reasons. Some other determinants of versions of natural deduction are: the treatment and appearance of a contradiction in a proof, minimizing assumption-withdrawing rules, and the kind of variable constraints on quantifier rules. Some versions can be criticized, and some software rejected for that reason. We have restricted the comparison to programs that use acceptable versions of natural deduction, and we will therefore not separately comment on that issue. A further issue is whether proof assistants can handle *equality*. Except for one, they all can. A side issue is whether theorems and derived rules (e.g. 'De Morgan') are allowed as a single step in a derivation. When relevant, this is mentioned.

2.2 Proof visualization

We have seen one basic distinction: one can show a natural deduction proof either as a tree or as a sequence. A tree is a two-dimensional representation, a sequence is one-dimensional. A tree is to be preferred. A derivation rule, although always having one conclusion, can have two or more premises. In a sequence, where the obvious visual reference is the preceding proof line, one has therefore to *refer* to other premises of rule application (generally, but not necessarily, by the *numbers* of the relevant proof lines). In tree format a reference isn't necessary. Determinants of proof visualization are:

- tree or linear
- rule name explicit when applying rules (yes, no)
- assumptions explicit ('Gentzen-style') in a proof line/node (yes, no)
- reference to active assumptions at each rule application (by repetition, by number)
- reference to withdrawn assumptions (by repetition, by number, by boxing subproof)
- reference to rule application premises (various visual ways, by number)

Notice that the two example proofs differ on all but the second of these six issues. *Proof schemata* and *derivation rules* are often visualized differently from actual proofs. This can be confusing to students.

2.3 Forward and backward reasoning

When making a natural deduction proof, one can proceed either forward or backward. In order to prove $(p \vee q) \wedge (r \vee s)$ from assumption $p \wedge r$ one can either start by applying $\wedge$E on $p \wedge r$ and proceed by attempting to derive $(p \vee q) \wedge (r \vee s)$ from assumptions p and r, or one can start by applying $\wedge$I on $p \vee q$ and $r \vee s$ resulting in $(p \vee q) \wedge (r \vee s)$ and then proceed by attempting to derive both $p \vee q$ and $r \vee s$ from $p \wedge r$. The first is forward reasoning (or assumption-driven), the second backward (or goal-driven). The best strategy is generally considered backward reasoning. Best is a program that allows both forward and backward reasoning, and has no bias towards one or the other. Bias can result from proof visualization, interface constraints, and the underlying theorem prover. An example: a linear, numbered, proof format is biased towards forward reasoning, as one does not know the last proof line's number when reasoning backward.

2.4 Proof help

We distinguish four different kinds or proof help, namely: global help, tactical help, strategic help, and debugging.

Global help is proof help unrelated to a particular proof. An example is on-line advice on how to apply a derivation rule, regardless of any stage in the proof. Being insecure about which rule to apply, a student might look up the rule for $\wedge$I when having to proceed from an assumption $p \wedge q$. This will not help a lot! Whether one needs such an on-line manual, also depends on whether the proof assistant comes with a textbook.

Tactical help is advice on how to extend an unfinished proof by one step. In the example above: 'try $\wedge$E'. This is a forward (assumption-driven) tactic. This is different from backward (goal-driven) tactics. A further option is to refer explicitly to relevant subformulas: 'Try applying $\wedge$E on $(p \wedge q) \wedge r$. You can derive either $p \wedge q$ or r.' In theorem proving terminology: tactical help is advice on what simple tactics to apply, and how.

Tactical advice can be wrong: when proving s from $p \wedge q$, $(p \wedge q) \rightarrow r$ and $r \rightarrow s$, we do not want to be advised to derive p or q from $p \wedge q$. More helpful is *strategic help*, in theorem proving terminology: advice based on a proof plan. In this case we expect the advice that we have to proceed by forward reasoning, by applying $\rightarrow$E on $p \wedge q$ and $(p \wedge q) \rightarrow r$. More 'dangerous' advice in this case is: 'in case of atomic conclusions, proceed by backward reasoning, by trying to apply an elimination rule' (in this case $\rightarrow$E).

It is unclear how much 'proof help' really helps. There are two reasons why it might *not* help: detailed advice is essentially if not practically incomplete and thus possibly incorrect; but even when it is correct, too much guidance makes a student lazy. On the other hand, a program that instead of an interactive fully guided proof, presents the entire proof without interaction at all, isn't very helpful either.

A somewhat different issue is what we call '*debugging*': when wrongly applying a rule, or making other mistakes during execution, sensible advice on what went wrong is very helpful. In other words: debugging is advice on proof repair.

2.5 Proof checking

A truly helpful proof assistant allows one to make possibly incorrect entire 'proofs', that are to be criticized *after* finishing them. Just as a student submits an entire proof on paper to a live teacher. Some proof assistants criticize only stepwise: they do not allow the user to proceed beyond one incorrect proof step. Some go beyond that: when the student selects a rule, they automatically generate its output.

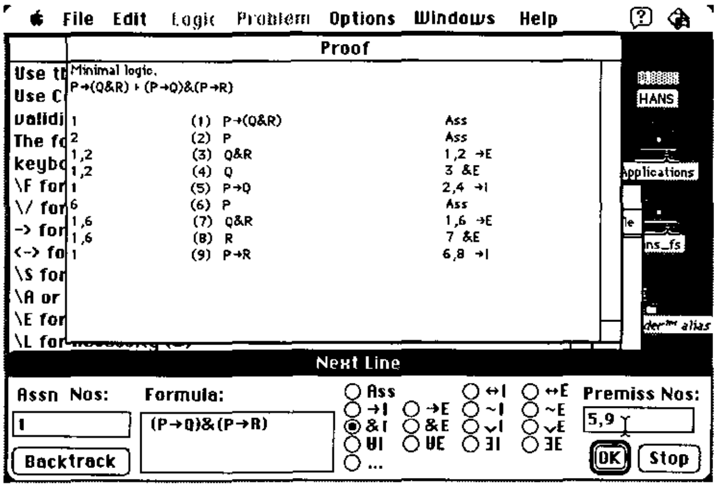
3 Comparison of natural deduction software

We will now discuss MacLogic, Jape, Hyperproof, CMU Proof Tutor and Symlog, in this order. After that we will comment on some other software for natural deduction. The programs will be compared on the issues from section 2, except for the issue 'version of natural deduction', as already mentioned. For the sake of brevity, we restrict ourselves to remarkable pluses or minuses. Accompanying pictures show a stage in the proof of $p \rightarrow (q \wedge r) \vdash (p \rightarrow q) \wedge (p \rightarrow r)$. This is the example we already know from section 1. For product information we refer to the web site mentioned on the first page.

3.1 MacLogic

MacLogic (Roy Dyckhoff, St Andrews University, Scotland) is the oldest of the five compared. Apart from natural deduction for classical predicate logic, it supports other proof systems and logics. The user can choose between two bracket conventions (enclosing quantifiers in brackets, or not), a useful feature. MacLogic has two modes for making proofs: *check mode* and *construct mode*, displayed in different windows. These modes differ in several of the investigated issues.

- *Proof visualization* Both in check and construct mode the proof representation is linear[1], with assumptions explicit in every proof step, i.e. sequent notation. In check mode assumptions are referred to by number (see picture below), in construct mode they are explicit.
- *Forward and backward reasoning* In check mode only forward reasoning is possible. In construct mode both forward and backward reasoning are possible.
- *Proof help* MacLogic gives extensive global help on how to apply the different derivation rules. It gives no tactical help[2]. However, it can be quite helpful in debugging: explaining why user input is *not* applicable (e.g. that application of ¬E requires two premises, in case the user has supplied only one). Its validity checker gives some kind of strategic advice: it warns students if their input is 'probably not provable in the current logic'.
- *Proof checking* MacLogic is proof checking while constructing proof.



Proof

```
Minimal logic.
P→(Q&R) ⊢ (P→Q)&(P→R)

1        (1)  P→(Q&R)        Ass
2        (2)  P              Ass
1,2      (3)  Q&R            1,2 →E
1,2      (4)  Q              3 &E
1        (5)  P→Q            2,4 →I
6        (6)  P              Ass
1,6      (7)  Q&R            1,6 →E
1,6      (8)  R              7 &E
1        (9)  P→R            6,8 →I
```

Assn Nos: 1

Formula: (P→Q)&(P→R)

○ Ass ○ ↔I ○ ↔E
○ →I ○ →E ○ ~I ○ ~E
● &I ○ &E ○ ∨I ○ ∨E
○ ∀I ○ ∀E ○ ∃I ○ ∃E
○ ...

Premiss Nos: 5,9

3.2 Jape

Jape (Richard Bornat, QMW, and Bernard Sufrin, Oxford, England) is acronym for 'Just Another Proof Editor'. The natural deduction part is called ItL Jape. Apart from natural deduction for classical predicate logic it contains other proof systems and logics and allows for user-defined logics. In both Jape and MacLogic natural deduction is based on an underlying sequent theorem prover.

In Jape the required action for rule application is minimal (and not unlike MacLogic in construct mode): the user selects a formula and a rule; the position of the formula in the subderivation determines whether a forward or backward proof step results; on 'return' the proof step (if applicable) is automatically executed. Based on the formula's main logical connective, in case of forward reasoning the execution automatically introduces another assumption, if so required, and the conclusion; in case of a backward step it introduces the assumption(s). Both might as well result in a new subproof being created. There is also a 'double click' mode, for which just formula selection suffices. Essential to this approach is the introduction of formula *variables* in the proof, that at a later stage must be *unified* with actual formulas.

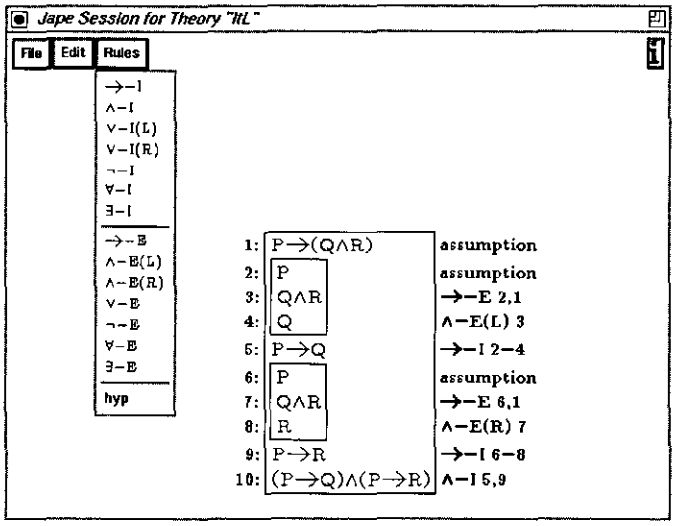
The required user interaction for this semi-automatic procedure is not entirely satisfactory. For an example we refer to the Jape Session picture below. Formula $q \wedge r$ in line 3 of the proof results from applying →E to assumption $p \rightarrow (q \wedge r)$ and assumption p in the first subproof. For this application the

[1] Roy Dyckhoff regrets not having developed a tree format interface (personal communication).

[2] Possibly different for a later version of MacLogic.

user has to select $p \rightarrow (q \wedge r)$ and q: in order to determine the relevant subproof one has to select its goal. One might prefer having to select both assumptions $p \rightarrow (q \wedge r)$ and p instead. Also, when (really...) badly structuring proofs, the program may worsen the situation by incorrectly pasting the resulting formula of rule application in another part of the proof.

- *Proof visualization* Jape's proof construction interface exceeds anything achieved in other programs. Not just because of its functionality but because of its simplicity (or, as Richard Bornat rightly calls it: its *quiet* interface). For natural deduction the proof format is 'Fitch-style', i.e. boxed linear proofs (as in the figure below). Worth mentioning is, that in Jape's sequent proof systems one can switch between tree and linear representation at any stage of proof construction. If only this were possible as well for natural deduction! Unfinished proofs or subproofs are visualized schematically, by using dots. Something similar holds for parts of a proof outside the scope of the subproof one is working in. This gives some screen unrest.
- *Forward and backward reasoning* Jape allows for both forward and backward reasoning. In some rules Jape is remarkably 'backward'. We give three examples. (1) The rule ∀I cannot generally be use in a forward way. Jape is too restrictive here. This is mentioned by the authors of Jape. It is because the underlying proof engine is sequent-based. (2) Given two assumptions, one cannot join them by applying ∧I, unless the relevant conjunction is already present as a goal. So ∧I can only be applied backward. (3) The same holds for →E: given an assumption $p \rightarrow q$, one cannot proceed forward by assuming p. Therefore, Jape is biased towards backward reasoning. This seems partly a didactic choice by the designers.
- *Proof help* There is no proof help, apart from 'debugging' help: helpful warning messages when wrongly applying rules.
- *Proof checking* It is not possible to submit entire proofs. It is not even possible to make an incorrect proof step, as rule execution is automatic given a formula and a rule. This makes Jape less fit for teaching natural deduction to 'absolute beginners'.



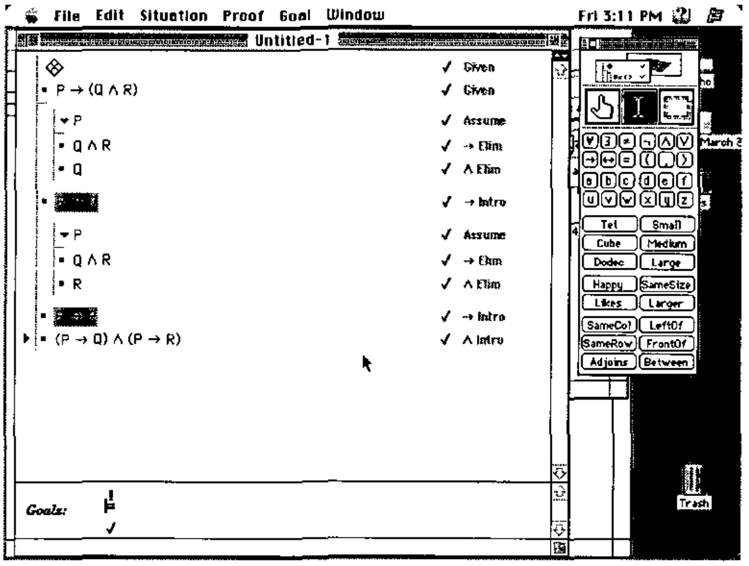
```
 1: P→(Q∧R)          assumption
 2: P                assumption
 3: Q∧R              →-E 2,1
 4: Q                ∧-E(L) 3
 5: P→Q              →-I 2-4
 6: P                assumption
 7: Q∧R              →-E 6,1
 8: R                ∧-E(R) 7
 9: P→R              →-I 6-8
10: (P→Q)∧(P→R)      ∧-I 5,9
```

3.3 Hyperproof

Hyperproof (Jon Barwise, Indiana, and John Etchemendy, Stanford, USA) is mainly known for its diagrammatic reasoning and not as a natural deduction program. We do not discuss diagrammatic reasoning. This kind of visual inference is unrelated to visualization of natural deduction proof. Hyperproof is accompanied by a textbook [Barwise & Etchemendy 94]. Various rules are available in more general, multiple-case versions. One can postpone proof obligations by applying a (first order) 'logical consequence' rule in any part of the proof. A successor of Hyperproof, called OpenProof, is in the making. This will contain more domains for visual inference and allow user-defined interpretation. Just the natural deduction part of Hyperproof is going to be incorporated in the successor of Tarski's World (a different program, for interpreting formulas in a blocks world and vice versa), called Fitch Xtreme, probably (and hopefully!) to be released soon.

- *Proof visualization* The visualization is linear, with boxed proofs, as in Jape. Application of the quantifier rules ∀I and ∃E comes with introducing subproofs assuming an arbitrary fresh variable that is a visual primitive in the proof (this is also called Fitch-style). Instead of referring by *number* to earlier proof lines in rule applications, proof lines are referred to in a visually direct way, by appearing black on the screen when the user has selected the proof line containing the rule's conclusion (see figure below). As this is only the case when the relevant rule is selected, this makes the entire proof less readable. When a proof window is printed though, number references are automatically added.

- *Forward and backward reasoning* Both are fully functional. Hyperproof seems somewhat biased to forward reasoning, because its proof representation is linear. This bias is enhanced by the interface. The textbook gives extensive advice on applying backward reasoning as well (see e.g. [Barwise & Etchemendy 94], 148).
- *Proof help* There is only global advice on how to apply various deduction rules. For help while wrongly applying proof steps, see *proof checking*.
- *Proof checking* Anything goes: one can either check the entire proof or check proof steps. Finishing one's proof entirely before checking it seems, rightly so, the recommended procedure. Some elementary proof 'checking', such as whether an application of →I has two assumptions, is *not* delayed. If only one assumption is given, the user gets a debugging advice. Further, the system response to proof checking can be to propose a countermodel that invalidates a supposed consequence. The proof checking mechanism in Hyperproof is very useful and helpful.



```
  ◈                          ✓ Given
  • P → (Q ∧ R)              ✓ Given
     ▾ P                     ✓ Assume
     • Q ∧ R                 ✓ → Elim
     • Q                     ✓ ∧ Elim
  • P → Q                    ✓ → Intro
     ▾ P                     ✓ Assume
     • Q ∧ R                 ✓ → Elim
     • R                     ✓ ∧ Elim
  • P → R                    ✓ → Intro
▸ • (P → Q) ∧ (P → R)        ✓ ∧ Intro

Goals:  ⊨
        ✓
```

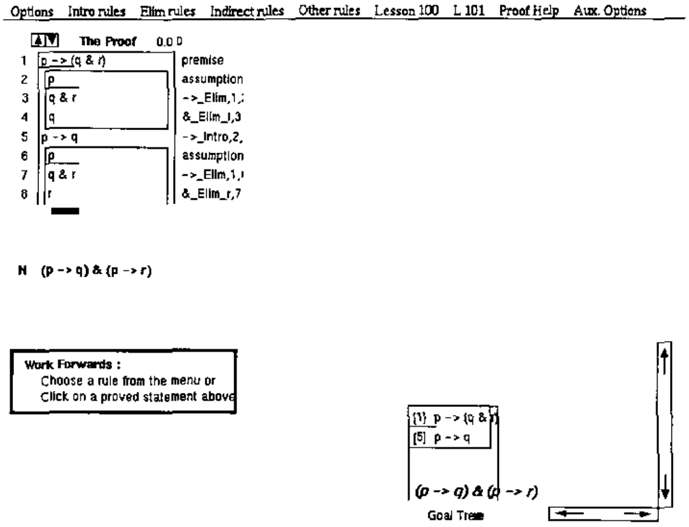
3.4 Carnegie Mellon University Proof Tutor

The CMU Proof Tutor (Richard Scheines and Wilfried Sieg, Carnegie Mellon University, USA) is for natural deduction in propositional logic only. A predicate logic extension is under construction. Its interface is primitive: logical connectives have ASCII key combination approximations, such as -> for implication.

- *Proof visualization* Proofs are visualized in linear style with boxed subproofs. Tree visualization is a simultaneous option, being the result of (and only of) backward reasoning.
- *Forward and backward reasoning* Both forward and backward reasoning seem to be supported equally. Surprisingly though, the system is strongly biased to forward reasoning: even a complete proof produced by backward reasoning has then to be 'verified' by the user step by step in a forward way. Unverified proof nodes are output in italics, and change to normal when verified. The single-node goal tree in the down-right corner of the figure below is therefore in italics. The verification procedure seems superfluous, and restricting a visualization to one strategy seems unfortunate.
- *Proof help* Global help facilities are rather extensive: there is help on rule definitions and on how to apply rules forward and backward, there are example proof runs of complete proofs, and there is an on-line tutorial. Supposedly, there is also tactical and strategic advice. We couldn't assess this, as it is not functional when remotely accessing the system, but we have the impression it is

not much beyond debugging.

- *Proof checking* Proof checking occurs while executing single proof steps. Entire proofs cannot be submitted. This lack surprises in a system that is otherwise intent on maximizing help.



Options Intro rules Elim rules Indirect rules Other rules Lesson 100 L 101 Proof Help Aux. Options

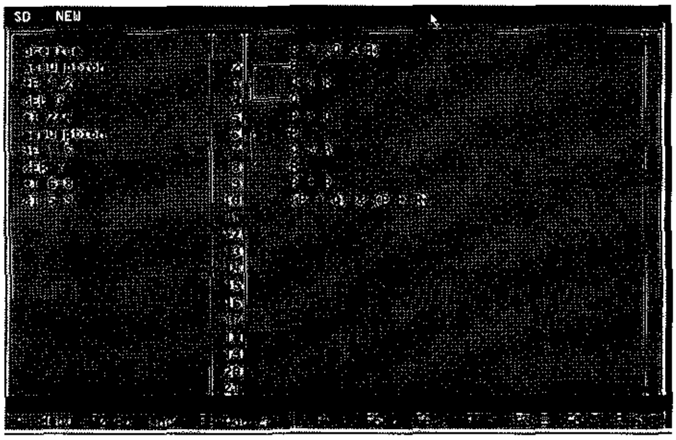
```
The Proof   0.0 0
1  p -> (q & r)     premise
2  | p              assumption
3  | q & r          ->_Elim,1,:
4  | q              &_Elim_l,3
5  p -> q           ->_Intro,2,
6  | p              assumption
7  | q & r          ->_Elim,1,ı
8  | r              &_Elim_r,7

N  (p -> q) & (p -> r)
```

Work Forwards :
Choose a rule from the menu or
Click on a proved statement above

```
{1}  p -> (q & r)
[5]  p -> q

(p -> q) & (p -> r)
Goal Tree
```

3.5 Symlog

Symlog (Frederic Portoraro and Robert Tully, Toronto, Canada), for *Sym*bolic *log*ic, comes with a textbook: *Logic with Symlog* [Portoraro & Tully 94]. It contains a standard as well as an extended natural deduction proof system. The extended system also contains derived rules such as 'quantifier negation' (¬∃ = ∀¬) and 'De Morgan', thus allowing more natural (and more 'computational style') proofs. The interface is rather (old and) primitive, e.g. quantifiers ∃ and ∀ are displayed as Σ and Π. While editing a proof, one regularly has to delete empty (unused) proof lines in completed subproofs. Deletion isn't bugfree: it is not possible to delete an unused *last* proofline in a subproof.



- *Proof visualization* The visualization is linear with 'Fitch-style' square-bracketed subproofs (which amounts to the same as boxing subproofs, but is visually less appealing).
- *Forward and backward reasoning* These are both available. Because of its linear proof visualization and design constraints, Symlog is somewhat biased towards forward reasoning.
- *Proof help* Symlog has global help on derivation rule definition and application, as well as a mixture of - very well thought out - tactical and strategic help. It is a mixture, because sometimes the program is smart enough to determine the best move and apparently uses a proof plan, whereas other times it is deficient and just gives an applicable tactic, that is apparently based on the main connective of the conclusion but is possibly the wrong one under the circumstances. An example is that when asking for advice on how to prove *p*, in the presence of an assumption $(p \wedge q)$

$\wedge r$, the system suggests to try $\neg$E and assume $\neg p$! Apart from these misses, proof help is very user friendly. An(other) example: when trying to prove $(p \wedge q) \wedge r$ the advice is not 'use $\neg$I' but 'apply $\neg$I on premises $p \wedge q$ and r'. Symlog's proof help is impressive and more developed than in the other programs discussed.
- *Proof checking* This is both stepwise during proof, and step by step after finishing an entire proof.

3.6 Other programs

Apart from the programs mentioned here, there are various others, that are less interesting from our point of view. On the one hand there are proof assistants that have been developed for purely educational purposes but that are less successful. On the other hand there are user interfaces for full-fledged theorem provers.

An example of the first kind is the 'Logics Tutor' of the Philosophy Department of Vrije Universiteit Amsterdam [Jonker & De Waal 97]. It comes with a textbook [de Jong 88]. It has been under development since 1995. The project originated from the desire to improve an existing course. Software fitting the lecturer's preferred logical language and preferred version of natural deduction was not available (personal communication). The results of this project are a Windows-based module for natural deduction in propositional logic and a similar one for natural deduction in predicate logic. It is a big improvement over the former course, that lacked computer assistance. Available user interaction is somewhat poor. It would require major and costly further development to become generally acceptable software. We assume similar projects to abound throughout the world, for the similar reason of lecturers wanting educational software to meet their preferences. What if these preferences had been facilitated by a proof assistant that allows finetuning logical notation, setting the version of the natural deduction system, choosing the kind of visualization, and so on?

Also approaching our goals but from a different perspective are products of the theorem proving community, such as Euodhilos-II [Ohtani et al. 96], XBarnacle [Lowe 97], and very many others, as e.g. presented in previous UITP workshops. Exceptions granted, and we assume Jape to count as one, they seem to fall short of being helpful to teach logic to absolute beginners. Partly this is because they are not intended for that: they cover a much wider area of 'mathematical' proof where inductive definitions play a major part, such as XBarnacle. On the other hand interfaces for theorem provers can be rather unappealing and unhelpful, as in the case of Euodhilos. Theorem provers that could work miracles if educational front-ends were developed for them, seem to abound. We think such efforts would pay off. Also, ideas from user interface design for purely educational software might prove fruitful for theorem prover interface design.

4 Conclusions

We have compared interfaces for natural deduction proof assistants from the issues: version of natural deduction system used, proof visualization, backward and forward reasoning facilities, proof help, and proof checking method. Jape's interface is visually most appealing and least biased towards forward or backward reasoning. Hyperproof's natural deduction assistant suits all kinds of proof checking very well. Symlog's help facilities are excellent but not always correct. As to date, tactical and strategic help are a rare good in a proof assistant. Flexible choice of notation might stimulate introduction of educational logic software in the classroom, as logic teachers would rather teach in their preferred logical language than take advantage of a computerized proof assistant. Developing educational front-ends to automated theorem provers also seems a fruitful approach.

5 Conjectures

In this section I allow myself a more personal address to the reader. I will comment on the situation in the Netherlands, and on some future directions.

There is a general demand for proof assistants that can be used in introductory logic education. Although there are many complaints about the limited applicability of Tarski's World (the predecessor of Hyperproof), that 'is just for interpreting sentences in a blocksworld' and does not even *have* a proof assistant apart from the Hintikka game for tableau reasoning, Tarski's World is nevertheless widely used on Dutch universities: in Groningen, Amsterdam, Utrecht, Twente, etc. There have been several smaller projects for proof assistant construction, as the mentioned project at Vrije Universiteit Amsterdam. Also, rumour spreads about some major efforts at other Dutch universities. Apparently, people are still not satisfied with available proof assistants.

Developments from the Barwise&Etchemendy team are very hopeful. A combined semantic tool and proof assistant as Fitch Xtreme promises to be, might well sweep the field. They put many person years of effort into building these products. Naturally that pays off. On the other hand a beautiful tool as Jape is presently claimed to be a 'two-persons-in-their-spare-time' effort. I sincerely hope it will be further developed, and especially its interactivity still improved. Who knows what else might be in the making.

References

[Barwise & Etchemendy 94a] Barwise, J. & Etchemendy, J. *Hyperproof, CSLI Lecture Notes No. 42*. CSLI, Stanford USA 1994.

[Bornat 96] Bornat, R. *Using ItL Jape*. Department of Computer Science, QMW College UK 1996.

[Dyckhoff 90s] Dyckhoff, R. *MacLogic (reference manual)*. St Andrews University, UK 1990s.

[Forbes 94] Forbes, G. *Modern Logic*. OUP, Oxford UK 1994.

[Goldson et al. 93] Goldson, D. & Reeves, S. & Bornat, R. *A review of several programs for the teaching of logic*. Published in *The Computer Journal* 36(4). 1993.

[Gries 94] Gries, D. & Schneider, F.B. *A Logical Approach to Discrete Math*. Springer, Berlin D 1994.

[de Jong 88] Jong, W.R. de. *Formele logika: een inleiding (Formal logic, an introduction)* Coutinho, Muiderberg NL 1988.

[Jonker & de Waal 97] Jonker, J.C. & Waal, S. de. *Computer-leerprogramma Logica (Computer logic proof assistant)*. MSc thesis. Vrije Universiteit Amsterdam, NL 1997.

[Lowe 97] Lowe, H. *The Use of Theorem Provers in the Teaching and Practice of Formal Methods*. Electronically published. Springer, 1997.

[Ohtani et al. 96] Ohtani, T., Sawamura, H. & Minami, T. *Reasoning Assistant System EUODHILOS-II: Operation Manual;* Research Report ISIS-RR-95-19E, ISIS, Japan 1996.

[Portoraro & Tully 94] Portoraro, F. D. & Tully, R. E. *Logic with Symlog; Learning Symbolic Logic by Computer*. Prentice Hall, Englewood Cliffs USA 1994.

[Scheines 90s] Scheines, R. *The Carnegie Mellon Proof Tutor; User's Manual*. CMU, USA 1990s.

Support for Interactive Theorem Proving: Some Design Principles and Their Application

Katherine A. Eastaughffe

Katherine.Eastaughffe@cl.cam.ac.uk
Computer Laboratory, Cambridge University
Pembroke Street, Cambridge CB2 3QG
United Kingdom

Abstract. This paper proposes a set of guidelines for use in the design of automated support for theorem proving. In particular they are aimed at graphical user interfaces to existing interactive proof engines. The application of these guidelines to the design of a graphical user interface to Isabelle is described.

1 Introduction

This paper presents a number of principles formulated to guide the design of enhancements to a graphical user interface of an interactive theorem prover. An interactive theorem prover is a tool in which a user chooses and applies proof steps to terms in a given logic, to produce theorems. The prover actually performs the proof steps and ensures that only valid chains of inference are developed.

Although there are many standards and texts which provide general guidelines for designing GUIs there is great benefit in attempting to formulate principles and guidelines that are specific to the problem domain of an application. Such specific principles can be informed by the purposes to which applications are put.

These guidelines were used to design enhancements to XIsabelle [9], a graphical user interface to Isabelle [10]. This paper considers some aspects of XIsabelle and how they relate to the guidelines.

Section 2 presents a categorisation of the purposes to which theorem provers are put. A user interface and its underlying design principles can only be evaluated with respect to which of these categories it is intended to support. Section 3 sets out a list of principles for guiding the design of interactive theorem proving GUIs. The rationale for these guidelines is discussed in terms of the aims described Section 2 and guidelines in general texts on interfaces. A description of aspects of XIsabelle that illustrate these guidelines is given in Section 4 and Section 5 discusses possible directions for future work.

2 Aims of Theorem Proving

Automated support for theorem proving must be designed with the purpose of theorem proving activities kept in mind. In this section we define four very broad categories for the use of theorem provers.

We assume there are basically two objects of interest: *goals* which are statements of theorems to be proved in a machine readable form; and *proofs* which are evidence or methods which show a goal or set of goals to be true.

We can distinguish between the first three of these categories by what objects are inputs (on the left of the arrows) and outputs (on the right of the arrows) to the system using a typing notation.

Truth Validity: Goal → [*True* | *False*]. To find out whether a goal is true or false.

This aim is usually the domain of automatic theorem provers and model-checking systems. However, the logic for which truth validity is decidable or mechanisable is fairly limited. For interactive theorem proving this aim is equivalent to proof discovery except that the form of the proof is not important and it is more usual to invoke "heavy duty" tactics that perform many low-level steps and tend to be iterative, perhaps using automatic proof procedures to solve subgoals of the problem. However, it is very rare for the proof to be completely unimportant, since at least it could be re-used to establish the validity of another goal. Therefore we shall consider this purpose to be subsumed by the aim of proof discovery.

Proof Checking: Goal × *Proof* → [*True* | *False*]. To check that a proof does prove a goal.

Here a proof may be at any level of formality, for example: a proof in the head of a user; a proof in a text book; a formal proof which is not rendered in a machine-readable format; or a machine-readable proof. As the level of formality decreases proof checking moves closer to being proof discovery.

Proof Discovery: Goal → [*None* | *Some Proof*]. To find a proof of a goal.

The difference between proof discovery and truth validity is that here the proof itself is also an important artefact. Most interactive theorem proving GUIs appear to be aimed at this category.

GUIs in the past have been criticised for not supporting re-use or replay very well, especially after modification of goals or assumptions [7]. This results from the difference between proof discovery and proof checking. Design of GUIs tends to aim at support for proof discovery. A separate activity is the manipulation of resulting proofs into those that have more general applicability. One proves a theorem however one can and then later sees if the proof can be generalised. Further goals may be proved by proof checking with the more general proof. One compelling reason for keeping separate the aim of making a proof re-usable is the application of theorem proving to software and system design. Here the specific proof steps, theorems and assumptions which were used can provide vital information about the design. The author is aware of one project where the fact that a certain assumption about the behaviour of a device was not used in the proof of a critical property enabled the design of the device to be simplified. On the other hand highly re-usable proofs have an obvious advantage in productivity. There is some tension in what a proof tool provides in terms of proof discovery and proof checking support.

Educational Purposes To construct or check proofs as a way of learning about formal logic.

3 Principles for Theorem Proving Support

These principles are based on the assumption that the user interface is being built upon a basic proof engine which maintains the internal representation of the current state of proof and performs all logical operations associated with the proof. It is also assumed that the proof engine is tactic-based meaning that a proof is constructed by formulating proof steps (tactics) in some tactic language and applying these to a proof state resulting in a new proof state. Certain proof states are associated with successful proof and a proof is complete when such a proof state is reached.

3.1 Multiple Views

For a complex and intellectually demanding activity such as theorem proving it is often productive for the user to develop more than one model for the task at hand [11]. Supporting several models implies having several concurrent representations of the current state of computation which the user can assess and use to decide on further action.

Different views make different information and manipulation of information explicit and easy to understand. For example, the freeness of a variable as a side condition of a rule is easier to see in a sequent calculus formulation than it is in natural deduction. Trees can represent the logic dependency between proof steps whereas a linear structure records the chronological history of proof steps. A declarative representation of a proof, such as a proof term, can have advantages over a procedural representation of a proof, such as a list of tactics [3]. Representing a proof at a high level of abstraction may make its structure clearer. Domain-specific proof support may extend as far as providing views which visualise objects at the domain level.

The use of multiple views is an under-used enhancement in existing prover interfaces. Some tools have a proof tree representation but they either do not visualise it, or they do not allow appropriate commands to be invoked within that view. Some tools allow the proof state to be represented at different levels of abstraction but once again tend to only allow operations within one level.

This leads to the following guidelines:

Principle 1 *There should be a number of complementary views of the proof construction and the user should be able to choose to see any number of the views simultaneously.*

Principle 2 *Within any view the user should be able to invoke operations that are meaningful in that view.*

3.2 Default Actions

It is often convenient for the user to choose a single object and perform an action which means "execute the obvious command". The program must be able to infer the other variables of the command from the context. For example, the form or the name of a theorem often indicates its intended use: a theorem with the name `gcd_def` or of the form `"gcd(x,y) = ..."` can indicate that the theorem will be used to rewrite terms.

Principle 3 *For multiple-part commands the interface should provide defaults for any variables that the user does not specify.*

The principle of default actions is reflected in the work of Kolyang et al. [4] where an object can have one of several subtypes which determine what operation on that object should be inferred. However, object can only have one subtype at a time (unless explicitly changed) whereas this general principle of default actions allows the variables to be inferred from the whole context of the action.

As there is a greater chance than normal that a default action is not the desired step, backing up from that action should be fast and easy. This supports the general HCI principle that the ease of undoing an action should be proportional to the ease of doing the action [6].

Principle 4 *If there is a choice for the default values then the option which results in the simplest proof step and the easiest to undo should be chosen.*

The implicit nature of the action means that the behaviour can be confusing and it is rarely appropriate for proof checking. However, it is a very good way for learners to articulate proof steps easily and can be useful for proof discovery because the simple actions mean there is little cost in experimenting. The principles of default action are closely related to flexible invocation.

3.3 Flexible Invocation

Many guidelines and texts on user interface design promote flexibility in input methods because it allows users of different skills, knowledge and experience to perform tasks in the way in which they prefer [12], [13]. As users of theorem provers can have a very wide range of skill and experience levels we include a principle of flexibility.

Principle 5 *There should be a high level of flexibility in the way in which the user can articulate commands to the prover.*

Flexibility can include the ability to provide input to the prover in different ways (e.g. by selecting items in a list, by keyboard, by direct manipulation of graphical objects) and in different orders (e.g. choose a rule before the name of a proof step or vice versa).

3.4 Relevant Information

One of the difficulties that users of popular interactive theorem provers face is the sheer volume of information, from which they must choose those pieces that will advance the current proof. This includes the large number of theories each with their own set of definitions, axioms, proved theorems and proof procedures. The benefits of a user interface which can asses the potential relevance of such components is obvious for proof discovery. Even for checking of a proof, selecting from a smaller list of objects can improve the speed at which a proof progresses.

Principle 6 *The user interface should support the user by displaying only information that is relevant in the current state.*

This principle is reflected in general HCI texts by general guidelines such as *"Only Necessary Information Displayed"* [13] or more concrete rules such as greying out non-applicable menu selections.

3.5 Multiple User Threads

Many popular software packages (e.g. word processing, spread sheet, games) allow the user to swap between different execution threads of the program. For theorem provers, the principle of multiple threads allows different proof strategies to be investigated concurrently and encourages experimenting by learners.

Principle 7 *The user interface should support several concurrent proof constructions.*

4 XIsabelle

XIsabelle is a Tcl/Tk [8] implementation of a graphical user interface for Isabelle. It is implemented in a way similar to Syme's interface for HOL [14]. XIsabelle processes and records all variables as strings and all logical operations are sent to Isabelle (using Expect [5]) for processing. XIsabelle has three main components:

- the *Theory Browser*, which allows the user to view and edit the definition of theories of a logic;
- the *Theorem Browser* which displays theorems (both axioms and derived ones) of a theory; and
- the *Prover* which provides an interface to Isabelle's subgoal package for proving theorems in a goal-directed fashion.

Each of these components are windows divided into smaller frames which display various groups of information on the current state and available options to the user. The user controls it by means of a mouse, invoking commands by choosing from a menu, clicking on buttons or clicking on other objects. Shortcuts for commands and additional user input may be provided by way of the keyboard.

4.1 Multiple Views

The XIsabelle Prover has three views of the current proof state. These three views are complementary in the sense that each of them contains information that the other two views do not.

The main window of the Prover forms the primary view and is a structured textual view closely resembling that of the underlying subgoal package of Isabelle. The current subgoals, the current tactic (the last tactic applied or the tactic currently being formulated), and lists of tactics are displayed in separate labelled sub-windows.

Another view is the history view which displays the history of the proof as a list of tactics. Clicking on a tactic in the list displays the tactic in the current tactic window and double-clicking on the tactic directly applies the tactic.

The most graphical view is that of the tactic tree. Each subgoal in the history of the proof is represented by a node in the tree with the original goal at the root. Clicking on a node displays the subgoal corresponding to that node in a small text window adjacent to the tree display and displays the tactic applied to that subgoal (if there was one). Using another button, clicking on the node displays the sequence of tactics corresponding to the subtree with that node as its root. These tree operations provide a convenient means for re-use. Experience with using the Prover has shown the tactic tree view very useful for case-based reasoning in proof discovery with it being used to articulate proof commands approximately half the time.

4.2 Default Actions

This principle is already partially embraced in the Isabelle system itself with the simplification and classical reasoning tactics. Isabelle maintains two lists of theorems, the current *simpset* and the current *claset*, which are used automatically with these tactics.

For most Isabelle built-in tactics there are at most three variables: the tactic name, a theorem or a list of theorems and a subgoal or a list of subgoals. XIsabelle provides a mechanism by which any one of these variables may be given by the user (usually by double-clicking on an item from a list) and other variables will be inferred. We describe how the value of each of these variables is inferred.

Inference of the tactic name depends on what other values are supplied. If only a subgoal is chosen by the user, the simplify tactic will be applied to the subgoal. As already mentioned, the simplify tactic infers a the current simpset as an argument to the tactic. The Browser displays theorems in lists associated with a category of rule (e.g. definition, introduction rule, induction rule) If a theorem is chosen from a list then the associated category of the list is used to infer a default tactic.

Inferring theorems is a more difficult problem because there usually is a very wide choice of possibly applicable theorems. It generally involves matching, but should the inferred value be all theorems that match or just one, such as the most relevant one by some measure. In order to support Principle 4, XIsabelle takes the first matching theorem according to some heuristic measure of relevance.

During a proof, XIsabelle maintains a current *active subgoal*. This is either chosen by the user or the default is the first subgoal which is a child of the previous active subgoal. A subgoal parameter is always inferred as the current active subgoal.

It is difficult to cater for bespoke tactics that use the full range of expressiveness of ML. However, a future capability is to allow a power user to define default values for variables. The power user may also change the default variables to suit their own needs.

4.3 Flexible Invocation

A tactic can be invoked in a number of ways: double-clicking on a theorem; double-clicking on a tactic's name in a list; double-clicking on a subgoal; reusing a tactic from the current proof history or another proof history; reusing the tactic in the current window; reusing a tactic by pointing at a node of the proof tree; or hand-crafting a tactic by editing the current tactic window.

4.4 Relevant Information

XIsabelle has two mechanisms for restricting information to that which is relevant to the current state. The first is based on a theorem matcher in the Theorem Browser. This matcher finds the theorems in a list which are relevant to a given term. After each application of a tactic in a proof, the theorem matcher displays all theorems in the current view of the Browser which are relevant to the current subgoal. The way in which a theorem is determined to be relevant to the subgoal depends on the category of theorems currently being viewed. For example, if the current view is of introduction rules, then the theorems whose conclusions unify with the conclusion of the subgoal are chosen. If the current view is of constant definitions, then the definitions for constants which occur in the subgoal are chosen.

The second mechanism is a filtering of tactics to those that are applicable to the current proof state and return a new state. The way this is achieved is not very clever and tends to be computationally expensive. Therefore the user can choose, by means of an options menu, how much filtering XIsabelle does. In the case of proof-checking it is often more convenient to disable the filtering mechanisms.

4.5 Multiple User Threads

Isabelle itself supports multiple user threads with its `push`, `pop`, `rotate_proof` commands that maintain a stack of proofs, and the `save_proof` and `restore_proof` commands that allow access to proofs by name. However, the subgoal package does not record the tactic history and any history of intermediate subgoals. Therefore XIsabelle maintains an array indexed by proof name of the additional information needed to swap between proof contexts. This additional information includes the tactic history, the current active subgoal, and the proof/tactic tree structure. A menu which contains the list of currently open proofs is used to change proof contexts.

5 The Future

In this paper we have proposed seven guidelines for the design of theorem proving GUIs. These guidelines are based around five concepts: multiple views, flexible invocation, relevant information, default values and multiple threads. The guidelines are by no means complete. They are intended to be a contribution towards a larger body of guidelines for designing graphical user interfaces for theorem provers. Such a set of guidelines would provide one means for comparing and evaluating proof tools. Concerted effort should be made to compile a more complete list and investigate the validity of these guidelines by user experiments and other HCI techniques. Such a de facto standard could go a long way towards greater movement between proof tools of users, theorems and proofs.

Aitken et al. [1] define three levels of abstraction for describing user interfaces: the logical level which describes a logical view of the theorem proving task, the abstract interaction level which describes the information presented to the user and the concrete interaction level which describes the physical aspects of the interaction. General user interface guidelines focus at the lower end of this abstraction spectrum and work on theorem prover interfaces tend to focus on reducing the gap between the logical and abstract interaction levels. We can view the logical level as part of the interface too and another level can be added to this hierarchy called the domain level. This level can be used to describe proof tools in which the user can reason about and guide a proof using actions and objects that are meaningful with respect to a specific domain [2]. Examples may include visual geometric reasoning tools and hardware design tools. Although this idea does not have the appeal of genericity it is perhaps the only hope for the integration of theorem proving into mainstream design tools and other tools to aid human reasoning. The potential benefits of the multiple view, flexible invocation and default value principles seem even greater for such interfaces.

As discussed in Section 2 there is some tension between producing goal-specific proofs, which can hold important information about assumptions which have been made, and highly re-usable proofs. User interfaces should provide support to the user in managing this tension.

Finally, the implementation of the principles of relevant information and flexible invocation make heavy use of unification and theorem indexing procedures. For proof tools to provide more sophisticated support in these areas, performance of these searching procedures must continue to be improved.

Acknowledgements

Maris Ozols and Tony Cant were the authors of the original versions of XIsabelle. The implementation of tactic and theorem filtering and proof/tactic trees is largely written by them. My enhancements of XIsabelle and the principles from which they were born are the result of many discussions with them. Larry Paulson made helpful comments relating to the work described in this paper which is funded by the EPSRC grants GR/K57381 "Mechanizing Temporal Reasoning" and GR/K77051 "Authentication Logics".

References

1. Aitken, J. S., Gray, P., Melham, T., Thomas, M., Interactive theorem proving: An empirical study of user activity, *Journal of Symbolic Computation* **25**, 2 (February 1998), 263 - 284
2. Eastaughffe, K. A., Ozols, M. A., Cant, A., The visualisation of interactive proofs, In *Proceedings of the 20th Australasian Computer Science Conference* (Sydney, Australia, February 1997), vol. 19 of *Australian Computer Science Communications*, pp. 297–306
3. Harrison, J. R., Proof style, Tech. Rep. 410, Computer Laboratory, University of Cambridge, Jan. 1997
4. Kolyang, Lüth, C., Meier, T., Wolff, B., TAS and IsaWin: Generic interfaces for transformational program development and theorem proving, In *TAPSOFT '97: Theory and Practice of Software Development* (1997), M. D. M. Bidoit, Ed., LNCS 1214, Springer Verlag, pp. 855–858
5. Libes, D., *Exploring Expect*, O'Reilly & Associates, 1995
6. Mayhew, D. J., *Principles and Guidelines in Software User Interface Design*, Prentice Hall, Engelwood Cliffs, NJ, 1992
7. Merriam, N. A., Harrison, M. D., What is wrong with GUIs for theorem provers, In *Proceedings of User Interfaces for Theorem Provers* (1997)
8. Ousterhout, J., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994
9. Ozols, M. A., Eastaughffe, K. A., Cant, A., XIsabelle: A system description, In *Proceedings of 14th International Conference on Automated Deduction* (July 1997), W. McCune, Ed., vol. 1249 of *LNAI*, Springer-Verlag, pp. 366–389
10. Paulson, L. C., *Isabelle: A Generic Theorem Prover*, Springer, 1994, LNCS 828
11. Rasmussen, J., *Information Processing and Human Machine Interaction: An Approach to Cognitive Engineering*, North-Holland, New York, 1986
12. Shneiderman, B., *Designing the User Interface*, 2 ed., Addison-Wesley, 1992
13. Smith, S. L., Mosier, J. N., Guidelines for designing user interface software, Tech. Rep. ESD-TR-86-278, The MITRE Corporation, Bedford, Massachusetts, USA, August 1986
14. Syme, D., A new interface for HOL - ideas, issues and implementation, In *Proceedings of Conference on Theorem Proving and its Applications* (1995)

Using ERMIA for the Evaluation of a Theorem Prover Interface

Mike Jackson[*], David Benyon[*] and Helen Lowe[**]

[*]Napier University, Edinburgh
[**]Glasgow Caledonian University, Glasgow

{m.jackson, d.benyon}@dcs.napier.ac.uk
H.Lowe@gcal.ac.uk

Abstract

ERMIA (Entity-Relationship Modelling of Information Artefacts) provides an extension to entity-relationship modelling techniques to provide a structural representation of the interaction between people and "information artefacts". Such a representation may then be used to compare contrasting interface designs or identify potential usability problems in an existing system. In this paper we present an application of ERMIA analysis to a version of the XBarnacle semi-automated theorem proving system that features interactive proof critics.

1. Introduction

Benyon and Green have introduced a method for understanding and describing Human-Computer Interaction known as ERMIA (Entity-Relationship Modelling of Information Artefacts (Benyon and Green, 1995; Green and Benyon, 1996; Benyon, Green and Bental, in press). ERMIA uses an extended entity-relationship modelling technique to provide a structural representation of the interaction between people and computer systems or other information artefacts. This representation can then be examined and discussed between designers in order to highlight features of the interface. The construction of the model can itself reveal insights into a proposed design and the final models used to communicate between designers or between users and designers.

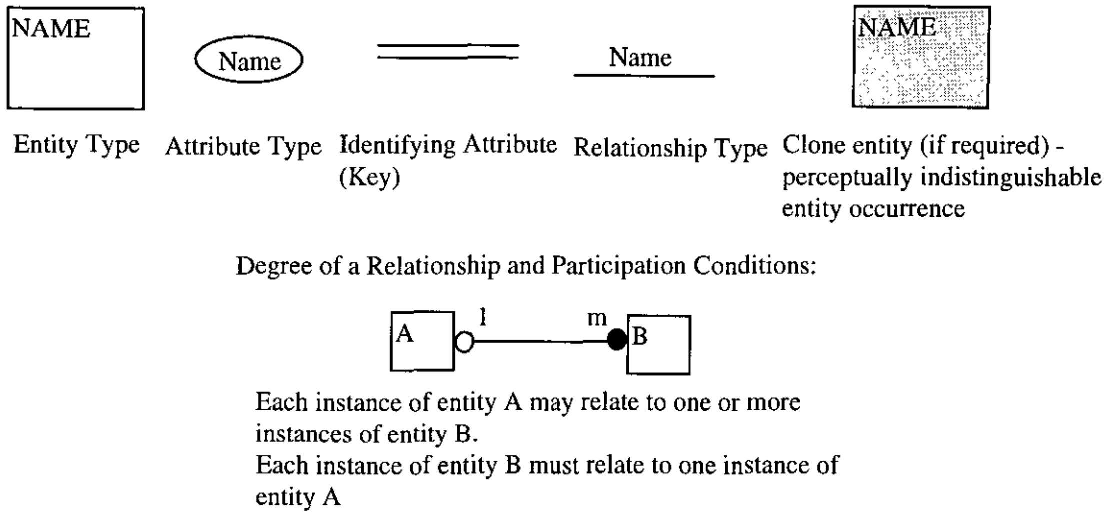
ERMIA can be used in a number of ways during interface development; to look at possible interfaces at an early stage of design, long before the final rendering has been decided on; to compare different mental models (designer's/user's, or across different users); or to analyse distributed systems, i.e. worksystems in which requisite information is distributed across different people and/or artefacts.

In this paper we show how ERMIA may be used to provide a conceptual model of a theorem prover and a perceptual model of an interface to this theorem prover. We then show how analysis of the conceptual model in itself and also with relation to the perceptual model may highlight potential usability problems. We also describe some experimental results showing how some problems identified during the ERMIA analysis then arose during an empirical evaluation of the theorem prover.

2. XBarnacle and Interactive Proof Critics

XBarnacle (Lowe and Duncan, 1997) is a version of CLaM automated proof planner (Bundy, van Harmelen, Horn and Smaill, 1990) incorporating a graphical user interface that allows users to interact with CLaM during a proof. XBarnacle is designed to allow users to step in and use their domain knowledge to guide CLaM in the search for a proof. This might be appropriate if they conclude that CLaM is pursuing an unproductive search strategy or CLaM performs a proof step the user knows is unproductive.

The version of XBarnacle described in this paper also features an implementation of interactive proof critics (Ireland, Jackson and Reid, 1997; Jackson, 1996). Proof critics (Ireland, 1992; Ireland and Bundy, 1996) provide functionality to CLaM to allow the patching of failed proof steps allowing then to succeed. Examples of proof patches include generating a required lemma, performing a case-split or revising an induction step earlier in the proof. Critics are associated with CLaM's methods and are triggered by patterns of failure of the related methods preconditions. Proof critics can extend the power of CLaM allowing it to prove theorems previously beyond its reach. Interactive proof critics allow a user to interact with a proof critic and view all the possible patches that a critic proposes and to apply, customise or reject these. Interacting with proof critics may improve the efficiency of CLaM over the purely automated critics version and also allow theorems to be proven that are beyond the reach of the automated CLaM. Part of the functionality of the interactive proof critics is an explanation facility which describes why a method failed in terms of its preconditions, why a critic was applicable, in terms of failure of the associated methods preconditions, and what the critic will do.


NAME
Name
Name
NAME
Entity Type
Attribute Type
Identifying Attribute (Key)
Relationship Type
Clone entity (if required) - perceptually indistinguishable entity occurrence
Degree of a Relationship and Participation Conditions:
A
1
m
B

Each instance of entity A may relate to one or more instances of entity B.
Each instance of entity B must relate to one instance of entity A

Figure 1: Basic Notation of Constructs in ERMIA

3. An Introduction to ERMIA

The entity-relationship (E-R) model is a graphically-based technique for representing the things of interest (entities) in an application and the associations between them (relationships). An Entity type is an aggregation of one or more property (or attribute) types. The concept of an entity provides two types of abstraction. The aggregation of properties into entities allows the designer to focus on the entities and to suppress details of the attributes. The classification of entity occurrences as entity types allows the designer to deal with a class of things rather than the individual things themselves. For example the methods (specifications of tactics) used by CLaM can usefully be viewed as instances of an entity *METHOD*, say, which has attributes *Name* and *Definition*, and so on.

Entities in the same set have the same types of attribute, though typically these attributes will take different values for different occurrences of the entity. For example, each method will have a different value for the *Name* attribute. Entities are defined by their attributes. The characteristics which define an entity are obtained by analysts in consultation with users. ERMIA does not accept that there is an objective world waiting to be carved up into a universal set of entities. Entities are subjective. Defining the entities makes such subjectivity explicit.

A further level of abstraction may be obtained by recognising that entities can have sub-types. This allows us to generalise certain characteristics or relationships between entity super-types, whilst recognising that the sub-types may differ from the super-type in some (relatively) minor respect. For example as XBarnacle allows user-CLaM collaboration during a proof we have the notion of an *AGENT* entity with sub-types *USER* and *CLAM* (the CLaM planner) as both these entities may take actions in CLaM. Each sub-type of an entity may share some attributes and/or relationships with their super-type entity but differ in others. Entities in ERMIA also demonstrate the principal of encapsulation. It is possible and often desirable to deal with quite complex artefacts as if they were a single entity, hiding the details of their construction. This type of abstraction again delivers a degree of simplification which makes for a more powerful model.

Conceptual entities, or concepts, are cognitive constructs. Conceptual entities can be seen as having some correspondence with the ideas or notions which users and/or designers have in their minds. We develop concepts in order to make sense of the experienced world. We represent those concepts and the relationships between them by developing ERMIA models. Perceptual entities are things in the experienced world which are of interest to the ERMIA modeller within the terms of some discourse. They are defined at some level of abstraction which is suitable for the intended perceivers.

In ERMIA, entities (but not relationships) have attributes (also known as properties or characteristics). An entity is the aggregation of its attributes in that it is defined as the total of its attributes. Usually one or more of the attributes are used to distinguish between entity occurrences. This attribute (or attributes) is known as the entity *identifier*. For example *Name* may be considered to be the identifying attribute of the *METHOD* entity since each method used by CLaM has a unique name. The structural attributes of perceptual entities are their perceivable characteristics (typically visual, audible or tactile properties). Behavioural attributes describe perceptual changes which occur under certain circumstances. An important

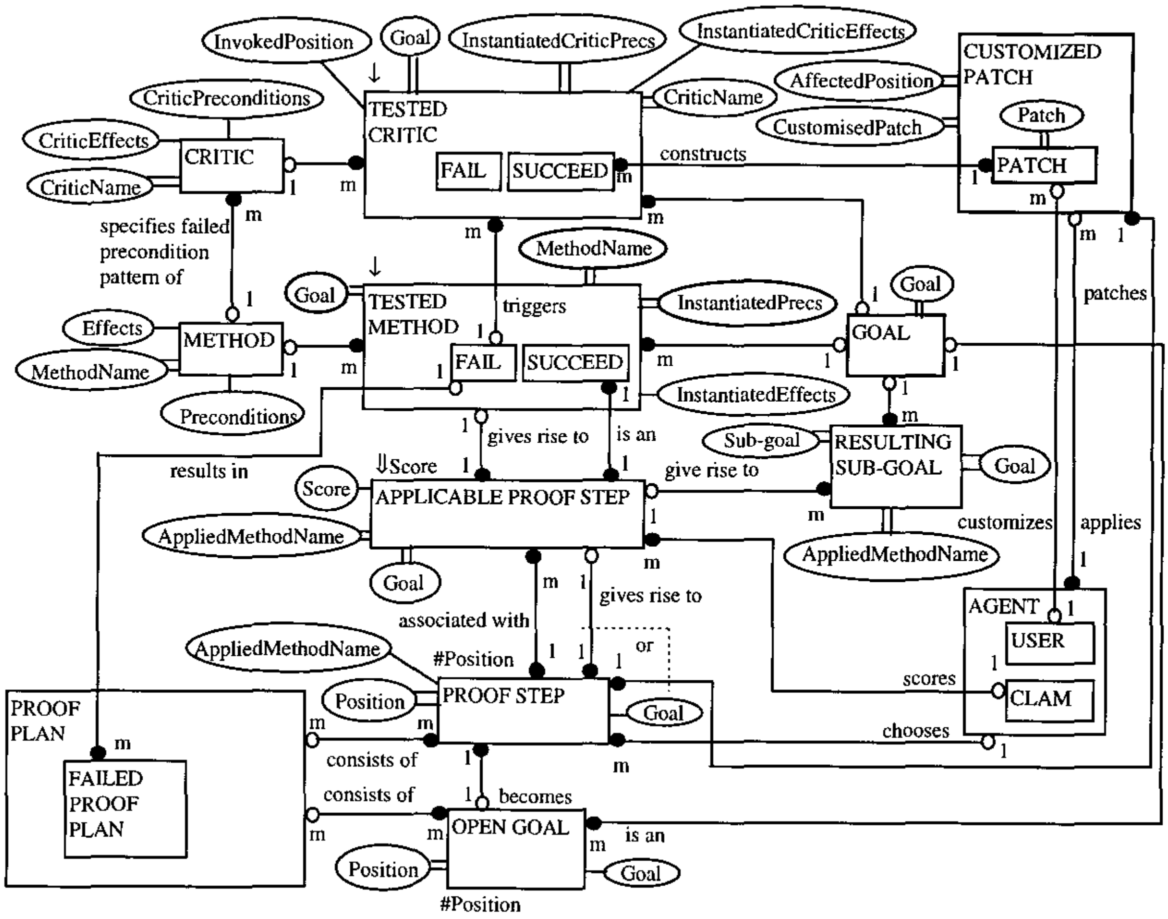


Figure 2: A Conceptual ERMIA for XBarnacle 3.2

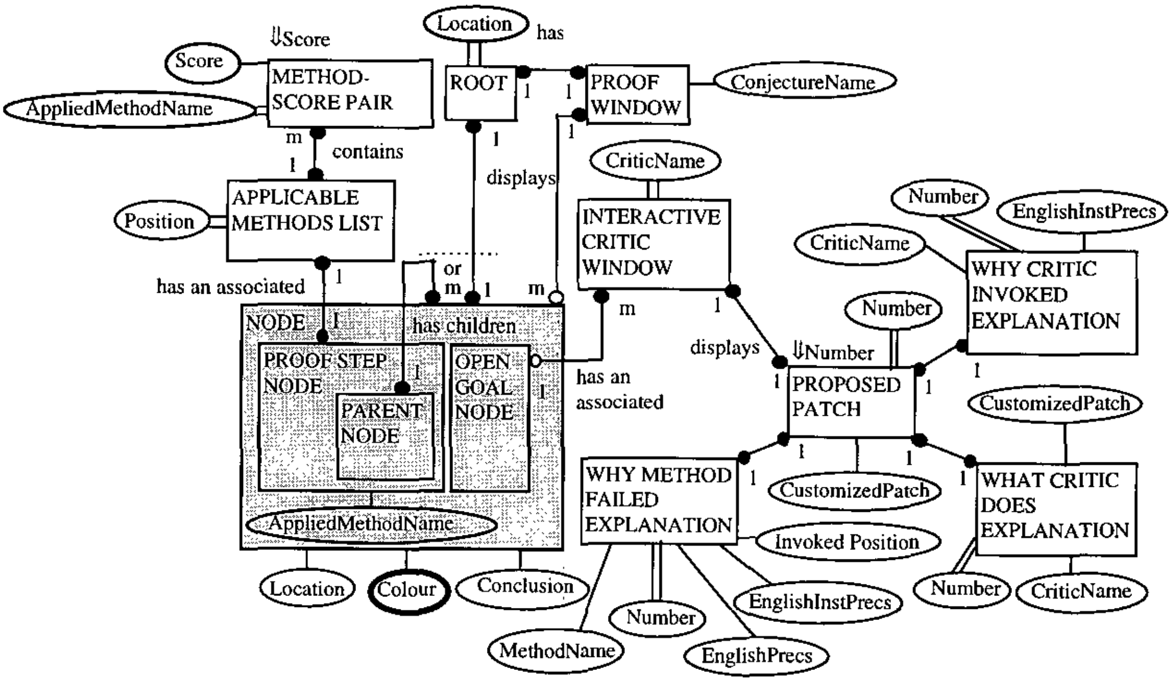
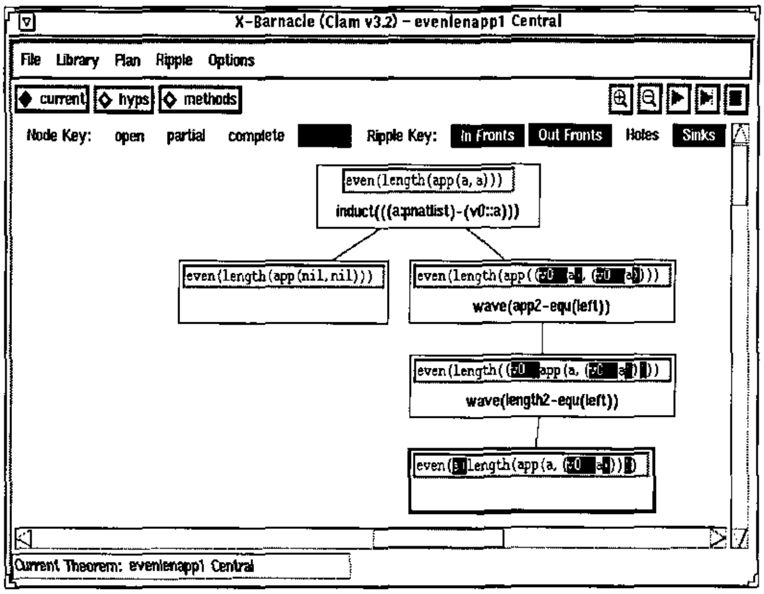
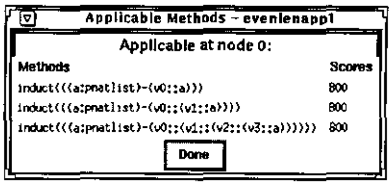


Figure 3: A Perceptual ERMIA for the XBarnacle 3.2 Interface


X-Barnacle (Clam v3.2) – evenlenapp1 Central
File Library Plan Ripple Options
◆ current ◇ hyps ◇ methods
Node Key: open partial complete
Ripple Key: In Fronts Out Fronts Holes Sinks
even(length(app(a, a)))
induct(((a:pnatlist)-(v0::a)))
even(length(app(nil, nil)))
even(length(app((v0::a), (v0::a))))
wave(app2-equ(left))
even(length((v0::app(a, (v0::a)))))
wave(length2-equ(left))
even(s(length(app(a, (v0::a)))))
Current Theorem: evenlenapp1 Central

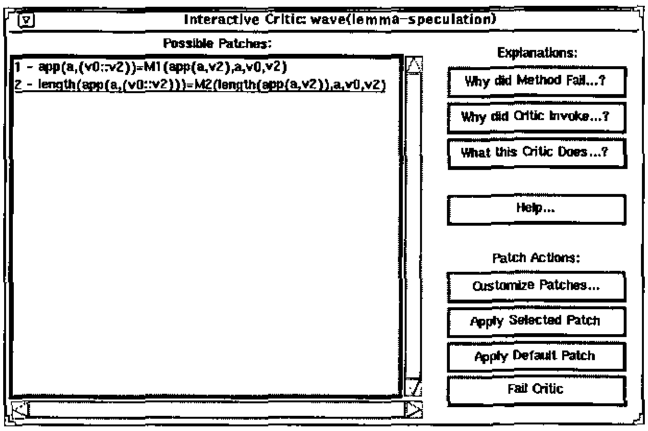
The main XBarnacle interface



Applicable at node 0:

Methods	Scores
induct(((a:pnatlist)-(v0::a)))	800
induct(((a:pnatlist)-(v0::(v1::a))))	800
induct(((a:pnatlist)-(v0::(v1::(v2::(v3::a))))))	800

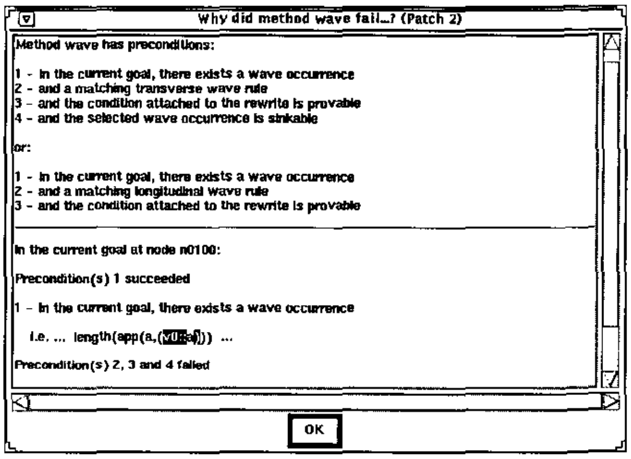
The list of methods (and heuristic applicability scores) that the user may obtain from a node



Possible Patches:

1 - app(a,(v0::v2))=M1(app(a,v2),a,v0,v2)
2 - length(app(a,(v0::v2)))=M2(length(app(a,v2)),a,v0,v2)

The interactive proof critic window which XBarnacle displays when a critic is invoked (the main part of the window displays the proposed proof patches)



Method wave has preconditions:

1 - In the current goal, there exists a wave occurrence
2 - and a matching transverse wave rule
3 - and the condition attached to the rewrite is provable
4 - and the selected wave occurrence is sinkable

or:

1 - In the current goal, there exists a wave occurrence
2 - and a matching longitudinal wave rule
3 - and the condition attached to the rewrite is provable

In the current goal at node n0100:

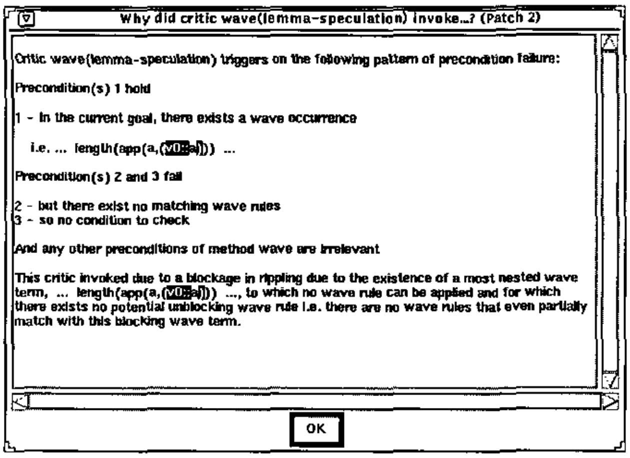
Precondition(s) 1 succeeded

1 - In the current goal, there exists a wave occurrence

i.e. ... length(app(a,(v0::a))) ...

Precondition(s) 2, 3 and 4 failed

An explanation from the interactive proof critic as to why the method failed (acquired by pressing the appropriate button)



Critic wave(lemma-speculation) triggers on the following pattern of precondition failure:

Precondition(s) 1 hold

1 - In the current goal, there exists a wave occurrence

i.e. ... length(app(a,(v0::a))) ...

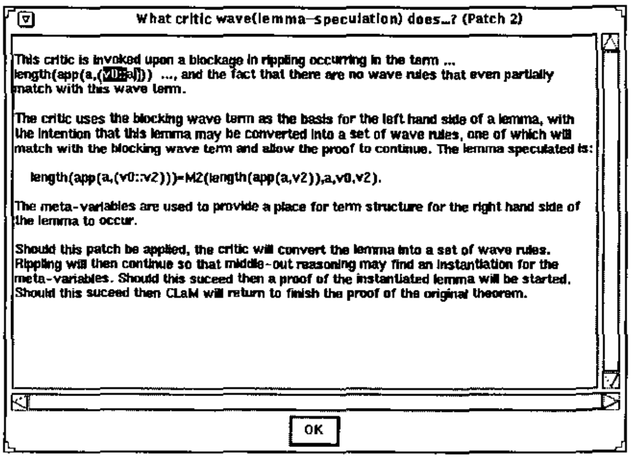
Precondition(s) 2 and 3 fail

2 - but there exist no matching wave rules
3 - so no condition to check

And any other preconditions of method wave are irrelevant

This critic invoked due to a blockage in rippling due to the existence of a most nested wave term, ... length(app(a,(v0::a))) ..., to which no wave rule can be applied and for which there exists no potential unblocking wave rule i.e. there are no wave rules that even partially match with this blocking wave term.

An explanation from the interactive proof critic as to why the critic invoked (acquired by pressing the appropriate button)



This critic is invoked upon a blockage in rippling occurring in the term ... length(app(a,(v0::a))) ..., and the fact that there are no wave rules that even partially match with this wave term.

The critic uses the blocking wave term as the basis for the left hand side of a lemma, with the intention that this lemma may be converted into a set of wave rules, one of which will match with the blocking wave term and allow the proof to continue. The lemma speculated is:

length(app(a,(v0::v2)))=M2(length(app(a,v2)),a,v0,v2).

The meta-variables are used to provide a place for term structure for the right hand side of the lemma to occur.

Should this patch be applied, the critic will convert the lemma into a set of wave rules. Rippling will then continue so that middle-out reasoning may find an instantiation for the meta-variables. Should this suceed then a proof of the instantiated lemma will be started. Should this suceed then CLaM will return to finish the proof of the original theorem.

An explanation from the interactive proof critic as to what the critic will do (acquired by pressing the appropriate button)

Figure 4: Various components of the XBarnacle interface

development for ERMIA models is that perceptual entities are not always distinguishable from one another. Thus we introduce the notion of a 'clone'; an entity type which has instances which are perceptually indistinguishable from other instances of that entity.

In ERMIA, as in ER models, entities are associated with each other and sometimes with themselves through relationships. There may be more than one relationship between entities. A one-to-one relationship *(1-1)* between entities *A* and *B* associates an occurrence of entity *A* with at most one occurrence of entity *B* and an occurrence of entity *B* with at most one occurrence of entity *A*. A one-to-many relationship *(1-m)* between entities *A* and *B* may associate many occurrences of entity *B* with each occurrence of entity *A*, but each occurrence of *B* is associated with at most one occurrence of *A*. A many-to-many relationship (*m-m*) permits many occurrences of entity *B* to be associated with each occurrence of entity *A* and many occurrences of entity *A* to be associated with each occurrence of entity *B*. It is useful to decompose *m-m* relationships by the introduction of a new entity. For example there is a potential *m-m* relation between goals and methods since a method may be applied to a number of goals and each goal may have a number of methods applicable to it. In Figure 2 we have broken up this *m-m* relationship revealing the entity *TESTED METHOD*, resulting from the application of a specific method to a specific goal.

Further semantics of relationships are represented by including participation conditions of entities in relationships. Mandatory participation constrains the entities in a set so that they must always participate in the relationship. Optional participation allows some or all occurrences of an entity not to participate in the relationship at any particular time. Sometimes it is desirable to insist that an entity must participate in two or more relationships (inclusivity). This is represented on an ERMIA diagram by suitable annotation of the diagram. Similarly we may want to represent that an entity may only participate in one of several relationships (exclusivity). Other constraints on the participation of entities in relationships may be represented by natural language annotations.

The basic notation used for ERMIA is shown in Figure 1. Figure 2 presents an ERMIA of the conceptual elements in XBarnacle.

4. A Perceptual ERMIA of the XBarnacle interface

The XBarnacle interface may be viewed as a viewport onto the underlying conceptual domain. In Figure 3 we present a perceptual ERMIA of this viewport, components of which are shown in Figure 4. Where conceptual entities and attributes are rendered at the interface we have used the same entity and attribute names as in the conceptual model of the underlying CLaM system. Note that there are new entities, however, for example *METHOD-SCORE PAIR* or *WHY METHOD FAILED EXPLANATION*, which have no specific conceptual analogue.

Note that nodes (denoting a super-type of the perceptual entities representing *PROOF STEPS* and *OPEN GOALS*) have a perceptual attribute, *Colour*, and that the value of this attribute directly reflects the type of node (i.e. is it a proof step node or an open goal node). Note also that nodes in the proof plan as displayed by XBarnacle are clones as there may be no way to tell certain occurrences of nodes apart at the interface. This may have serious implications for the user as we describe in the next section.

5. Using ERMIA to Identify Potential Usability Problems

We now give examples of how analysing the conceptual ERMIA in itself, and also comparing the conceptual ERMIA to the perceptual ERMIA of the viewport, can highlight potential usability problems. The work on ERMIA models of XBarnacle was done as part of research into the utility and usability of interactive proof critics. A co-operative style evaluation (Monk, Wright, Haber and Davenport, 1993) has been performed to address this question. One of the aims of this evaluation was to see if the problems highlighted by an ERMIA analysis undertaken prior to the evaluation arose in actual use of the interface by real users, thereby giving evidence as to the utility of conducting an ERMIA analysis. When discussing the problems highlighted by ERMIA we shall give examples where those problems arose in practice.

Problem 1. A Problem Due to the Collaborative Nature of the Interface

From our knowledge of how XBarnacle is used we know that a proof step may have been chosen by the CLaM planner or the user. However our ERMIA model shows that neither the *PROOF STEP* nor *APPLICABLE PROOF STEP* entities (of Figure 2) of XBarnacle contain any attribute to record which agent actually applied each proof step. Thus the system is limited in that there is no means of determining the division of labour (if any) between the CLaM planner and a user when performing a proof. Related to this is the fact that critics may also be responsible for applying proof steps and, again, no means of storing this fact, in such cases, is provided.

This is important since users and other interested parties may over-estimate or under-estimate the power of CLaM or may gain a false impression of the reasoning strategies used by CLaM if this information is not available to them. An example of this arose during the evaluation. One participant, an expert in CLaM and proof critics, remarked on being presented with a proof:

"...so its chosen an induction on a, double induction on a which was very clever of it. How did it manage to think of a double induction? That's cunning."

The participant was unaware that the double induction resulted not from a method application, as they assumed, but rather from a critic which may redo induction steps. Another participant, also an expert in CLaM stated during the same example:

"That's no normal induction analysis...that's somebody being clever"

This problem is an example of how providing functionality at the interface (in the case of user/CLaM collaboration) or providing conceptual and/or interface functionality (in the case of proof critics or interactive proof critics) can create the need for new attributes in certain underlying conceptual entities to support the implications of this additional functionality. In this example this would perhaps entail the addition of an attribute to the conceptual *PROOF STEP* entity to identify who executed each step in the proof (or if the proof step arose from a critic application) and the provision at the interface of a suitable presentation of this new attribute.

Problem 2. Positions in the Proof Plan

Figure 3 shows how XBarnacle displays proof steps and open goals using a node entity. Analysing the ERMIA we see that this entity (and hence its rendition at the interface) has no identifying attribute meaning that nodes at the interface are clones - node entities do have an attribute *Location*, the location of the node on the XBarnacle display, but this may change as a proof progresses and is unrelated to the underlying position of a proof step or open goal in the proof plan. This demonstrates a problem with the interface since the proof steps and open goals in the underlying theorem prover, which nodes at the interface represent, *do* have an identifying attribute - their position in the proof plan, as may be seen in Figure 2. Therefore the interface may, in certain circumstances, cause navigation problems for the user if two separate parts of a proof plan have the same sets of proof steps or open goals as these will be indistinguishable at the interface. Also, referring to proof steps or open goals in the proof plan by position may cause problems since there is no direct representation of this position in the entities that display the proof plan - the user must take extra action to display the position of a node in the proof plan.

A problem of this type arose in the evaluation. For example the induction revision critic which may propose the revision of an application of the induction method at a proof step earlier in the proof plan prints as patches to the user information of form:

```
Apply method induct(x:pnat,s(x)) at node 000
```

where *000* is a proof step/node position in the conceptual proof plan. One participant in the evaluation pointed at the displayed proof plan and remarked:

"I think you need to label these nodes if you're going to refer to them by some number ... its not obvious which one you're talking about."

despite these addresses being in a form similar to that in which node addresses are usually presented (as another participant correctly identified). Another participant stated on the same task:

"...so the question is where's node 000 ?"

and like the first participant had to head to the root of the proof plan and count down to the correct point in the proof, which would be very problematic in large proofs, as one participant stated. Another participant stated:

"I want it to do the induction that its suggesting but I want to do it on this node."

pointing to the node where the induction *would* be done and assuming wrongly that it gets done at the current node, where the critic was invoked. The participant here did not pick up the fact that *000* referred to the node at which the induction would be done.

Unlike Problem 1 this problem arises as a result of a key attribute of important theorem proving entity (proof steps and open nodes) not being rendered directly at the interface.

Problem 3. Where was the critic invoked and to what does it apply ?

In Figure 2 we see, by following the appropriate relations and examining the attributes, that critics are invoked at specific positions in the proof plan, those positions corresponding to the position of the open goal where the associated method fails. However, we also see that the effect of a critic may be to take action at a different node in a proof tree, for example the induction revision critic described above. Related to Problem 2 problems relating to positions of proof steps and open goals in the proof plan may arise due to the interface not rendering these attributes at the interface (as is highlighted by the omission of such attributes from the perceptual ERMIA of Figure 3). Firstly the interactive critics interface, when invoked by CLaM, does not display the node at which it is invoked as one participant stated:

"Which goal's it working on now...which one's it asking about ?"

The user must take extra action to elicit this information, as one participant verbalised:

"I'm hitting the "Why did method fail ?" button which tells me which node the problems at which isn't entirely clear unless you actually do something like this..."

causing the explanation as to why a method failed to be displayed, this explanation (as Figure 3 shows) rendering the conceptual attribute *InvokedPosition*, which stores the position in the proof plan of the goal to which the critic was invoked from. The participant later stated:

"...it really would be useful if the display, the interactive critic window tells you which node its looking at..."

Similarly the critic interface does not always say to which node it does apply. Nor do the explanations. This led to comments of the form:

"I think it would be quite useful if the...the display actually showed which nodes they were proposing to be applied to without actually having to hit one of the buttons to get the more detail."

and

"It certainly would be useful if you could see what nodes each of the patches were applying to.".

The utility of showing the nodes to which a critic is applied was borne out by a comment from a participant with respect to the display of patches for induction revision which do state the node they affect:

"It's more clear from these what they're going to do which is apply an induction at a particular node."

Again these problem arises from the interface not rendering certain attributes of conceptual entities in the appropriate place i.e. here the position where the critic invoked and the position to which each of the proof patches apply should be rendered in the main interactive critic window, not just in the explanation windows which pop-up only after extra action by the user.

Other Problems

(Jackson, Benyon and Lowe, 97) describes in detail other potential usability problems that may arise, most of these also relate to the standard XBarnacle system described in (Lowe and Duncan, 97). The problems include:

- As stated proof critics may create a lemma automatically. This sets up a requirement for the lemma to be proven. CLaM has functionality to prove such lemmas automatically resulting in a system where a conjecture may either have been defined by the user or a critic. This leads to a problem related to Problem 1 in that false impressions of XBarnacle's power may arise if outside observers are unaware of this fact. Therefore some means of recording who defined what conjecture should perhaps be provided.
- Each applicable proof step has an associated set of resulting sub-goals but only the sub-goals for the applicable proof step actually applied may be accessed (since these become sub-goals in the proof plan);

- Method applicability is determined by their preconditions but there is no way of accessing the preconditions of a method as they relate to a goal in the proof plan i.e. one cannot see why a method was applicable to a given goal. Nor can one see why other methods failed to be applicable to a goal i.e. the pattern of precondition failure. This is important since a failed method may lead to a failed proof plan. The exception is for methods whose critics invoke, the precondition pattern may then be viewed using the interactive critic interface. One participant in the evaluation used this feature extensively and this may give indications as to the utility of this form of explanation in general.

6. Conclusion

We have presented an introduction to ERMIA and a model of the conceptual structure of a version of XBarnacle that features interactive proof critics. We also provided a perceptual model of a viewport onto that conceptual structure and showed how analysis of the conceptual structure, both in itself and in relation to the perceptual structure, highlighted potential usability problems, some of which arose when potential users of XBarnacle participated in an evaluation of the utility and usability of interactive proof critics.

There is little doubt that developing the ERMIAs has provided an insight into XBarnacle. Whether such insight could have been gleaned through other approaches is a moot point We would argue that a task analysis approach would not have highlighted some of the usability problems, because we are not dealing with existing tasks, rather we are dealing with the distribution of knowledge throughout the underlying system and the representation of this knowledge at the user interface.

This is not however to state that task analysis approaches or other interface modelling techniques are of no use. On the contrary in many respects these approaches may be superior to ERMIA. For example one limitation of ERMIA is the problem of highlighting the fact that some entities may exist only for a certain limited period of time and then cease to exist in a conceptual system. This further serves to emphasise the fact that ERMIA is one of a number of modelling techniques of great use in interface design and that interface designers may need to consider the pro's and con's of each of techniques, in conjunction with their own areas of concern, to choose the tools most suitable for their task.

References

Benyon, D. R. (1996) Domain Models for User Interface Design. In Benyon, D. R. and Palanque, P. *Critical Issues in User Interface Systems Engineering*, Springer-Verlag

Benyon, D. R. and Green, T. R. G. (1995) Displays as Data Structures. In Nordby, K., Helmersen, P. H., Gilmore, D. J, and Arnesen, S. A. (Eds.) *Human-Computer Interaction: INTERACT-95*. London: Chapman and Hall.

Benyon, D. R. and Green, T. R. G. and Bental, D. (in press)*Conceptual Modelling for User Interface Design, using ERMIA* Springer

Bundy, A. Van Harmelen, F. Horn, C. and Smaill, A. (1990) The Oyster-CLaM System *10th International Conference on Automated Deduction* Kaiserlauten, Germany July 1990 (ed. M.E. Stickel) Springer-Verlag: London, Lecture Notes in Artificial Intelligence-449 p647-648

Green, T. R. G. and Benyon, D. R. (1996) The skull beneath the skin; Entity-relationship modelling of Information Artefacts. *International Journal of Human-Computer Studies* **44**(6) 801-828

Ireland, A. and Bundy, A. (1996) Productive Use of Failure in Inductive Proof Special edition of *Journal of Automated Reasoning* on Inductive Proof **16** (March 1996) 1996.

Ireland, A. (1992) The Use of Planning Critics in Mechanizing Inductive Proofs In *International Conference on Logic Programming and Automated Reasoning - LPAR'92* St. Petersburg July 1992 (ed. A. Voronkov) Springer-Verlag: London, Lecture Notes in Artificial Intelligence-624 p178-189

Ireland, A., Jackson, M. and Reid, G. (1997) A collaborative approach to theorem *proving Proceedings of the First International Workshop on Proof Transformation and Presentation* Schloss Dagstuhl, Germany, April 1997 (eds. X. Huang, J. Pelletier, F. Pfenning and J. Siekmann) p21-22

Jackson, M. (1996) *HCI Techniques for Theorem Proving* MSc Project Report Heriot-Watt University, Edinburgh June

Jackson, M., Benyon, D. And Lowe, H. (1997) *Evaluating an Interface to a Theorem Prover: An Application of Entity-Relationship Modelling of Information Artefacts*. Submitted to BCS-HCI 1998 Sheffield.

Lowe, H. and Duncan, D. (1997) XBarnacle: Making Theorem Provers More Accessible, *Proceedings of the Fourteenth Conference on Automated Deduction (CADE-14)* Townsville, Australia, July 1997 (ed. W. McCune) Springer-Verlag: London, Lecture Notes in Artificial Intelligence-1249 p404-407

Monk, A., Wright, P., Haber, J. and Davenport, L. (1993) *Improving Your Human-Computer Interface: A Practical Technique* Prentice-Hall International: New York.

Making Design Decisions to Support Diversity in Interactive Theorem Proving

Nicholas A. Merriam* Michael D. Harrison
Department of Computer Science, University of York
York, YO10 5DD, U.K.
Nicholas.Merriam,Michael.Harrison@cs.york.ac.uk

Abstract

We present a view of interactive theorem proving as a collection of diverse activities which impact on one another. We catalogue a number of features of existing theorem proving assistants (TPAs) and connect them with the activities which they seem to support. We then volunteer approaches to evaluating these features respecting the connectedness of the activities they support, and begin to make explicit some questions which could be passed on to psychologists, taking the perspective of a designer attempting to construct a powerful and well integrated interface for a TPA.

1 Introduction

At UITP '95 with [MDH95], we presented a paper which viewed theorem proving assistants (TPAs) as "levers" for extending human cognitive processes through the mechanisms of planning, reflection and reuse, rather than just looking at how proof commands could be articulated. At UITP '97 with [MH97], we talked about the influence which one part of the interface can have "downstream" on subsequent activities, to be precise the way that a proof created using a graphical direct manipulation interface would tend to contain more unnecessary dependencies than one created using a commandline interface. In the present paper, we attempt to integrate the insight that interactive theorem proving is about all of formalisation, planning, articulation, reflection and reuse with the understanding that these activities are interrelated. In addition to trying to understand these issues, we use our perspective to examine how designers of TPA interfaces could select from a range of functionality to provide well integrated and powerful interfaces. It has been observed that some of the issues that we consider have their roots not in the interface but in the underlying functionality of TPAs. By confining ourselves to the design of the *interface* only, we will be tackling some such issues in a rather shallow way. The constraints of time and expertise have thus far focussed our research on the interface and we are of the opinion that it is, in fact, worthwhile to consider how interfaces do support existing theorem proving technology.

To this end, in Section 2 we present a view of interactive theorem proving *activity* as made up of diverse elements which are related and interdependent, an observation which will lead us away from criteria which would consider only one activity in isolation. We then, in Section 3 examine the demands placed on the interface in supporting these activities and correlate them with some relevant *features of existing TPAs* and envisage some possible new features which might satisfy these demands. In Section 4 we look at the extent to which the technology of Section 3 provides answers to the questions of Section 2, taking the part of a designer selecting functionality to provide in a new TPA interface.

Here we consider only TPAs supporting interactive backward reasoning, systems like PVS, see [SOR95], and Jape, see [SB95].

*Employed on EPSRC grant GR/K09205

2 Interactive Proof Activities

In order to gather information about the activities involved in interactive proof we have conducted informal interviews with users of TPAs working at Universities including Cambridge, Glasgow and York. We used PVS in an attempt to prove Kleene's theorem, identifying the languages accepted by finite state automata with the languages generated by regular expressions. That exercise had the dual objective of recording data about the time spent on different kinds of activity in interactive theorem proving and also increasing our understanding of, and familiarity with, the "world" of machine assisted theorem proving.

Interactive theorem proving means different things to different people, either because they are looking at very different kinds of theorem proving assistant or because they are focussing on one particular kind of activity. We can attempt to classify TPAs by degree of automation. A TPA like CLAM, see [BvHHS90], aims to prove the theorem automatically; user intervention is required to ensure that the system is appropriately configured to start with, rather than to supply input during the proving process. At the opposite extreme, TPAs like Jape, see [SB95], and MathSpad, see [Ver], allows the editing of a proof (according to enforced rules) and it aims to make this editing process as painless as possible but not to automate the process of proof discovery. However for our purposes the distinctions along this continuum will be noted but do not change our approach.

We are more interested in the diversity of activities which comprise interactive theorem proving regardless of the degree of automation on offer:

- **Formalising** the problem, writing formal descriptions of the entities about which we wish to prove properties and collecting these together into theories, takes significant time and effort. It has much in common with programming, about which much has been written, but will be informed by the proving process. Indeed it may well be the case that the specification is the deliverable and the proofs are just there to establish certain kinds of consistency within the specification.
- Discovering the **plan** is convincingly suggested by [Pol57] as the essential and most "creative" problem solving activity.
- **Articulating** the proof plan to the TPA is the core activity of theorem proving. We can view this as taking place in two, not necessarily distinct stages, as the plan is translated to a form which *can* be communicated to the TPA and then this form is translated into a series of syntactically correct commands. For example the plan might be "prove by induction", the intermediate form might be "use the induction command, on the base case use commands ..., and on the step case use commands ..." and the final, interface articulated form might be "`(induct "i") ...`".
- Understanding the proof by **reflecting** on it is essential to answer "where am I?" questions. In the case of a very automatic theorem prover, this may be connected with monitoring, and then possibly influencing, the progress of automated procedures.
- **Reusing** proofs is vital in order to keep the time required for theorem proving within sensible bounds. This dominates serious theorem proving, which would be impossible without a host of "ready-to-wear" proven theorems both relating to the problem domain and also providing basic proof tools, such as induction.
  - We can view our theory store as a database of theorems, useful facts to be employed in the construction of the current proof.
  - We can also view the theory store as a database of proofs. We might refer to a proof as an example of how to use a particular proof command. We might also extract part or all of a proof as a starting point for the current proof. Store proofs may show common patterns, which we wish to translate to an automated procedure that can generate future proofs.
  - In order to be able to reuse theorems effectively we need tools to allow them to be combined with a particular statement quickly and easily.

In a process of discussion with with Stuart Aitken and Tom Melham at the University of Glasgow, we came to the realisation that we can identify (greatly) interleaved phases of proof which include the actual *proving* but also expressing the theorem to be proved, *formalisation*, and in "tidying up" the results of proving into orderly theories of efficient proofs, *proof management*, see [AGMT96]. Whilst this concept of phases provides a useful tool for understanding what is going on during interaction with a TPA, we have presented the above, less separate, categorisations because they seem to give us a more pragmatic foundation for looking at design questions. There is no clear relationship between our categories and the phases of proof, since our categories are neither a direct expansion of the phases nor an orthogonal classification.

3 Tools for Supporting Proving

We can interpret our list of activities as a list of questions about how we support those activities and then correlate this list with a list of interface tools provided by currently available TPAs which support those activities. In order to collect examples, we have looked at CADiZ [TV97], CLAM [BvHHS90], CtCoq [BBC+96], Isabelle [Pau94], HOL (unmodified [GM93] and with additional interfaces such as CHOL [TBK92] and TkHOLWorkbench [Sym]), IMPS [FGT98], Jape [SB95], MURAL [JJLM91] and PVS [SOR95] and the toolkit of [CH95]. Of course, it is not necessarily the case that one tool will support only activity in one of our categories but for the convenience of the presentation we will tolerate some duplication across categories.

3.1 Formalisation

In order to commence theorem proving, we require, in the TPA's specification language, a statement of the theorem to be proved, which will typically rely on the definition of some functions. For example, if we want to prove that every natural number greater than one has a unique bag of prime factors, we must at least define, or reuse from somewhere, a predicate which characterises prime numbers and a datatype for bags. Questions which arise with regard to formalisation include that of how the human prover knows the correct syntax for expressions and they find and browse existing specifications to find useful material. With regard to the first question, there is little support in the interface with existing systems. PVS uses the emacs font lock mode to offer the possibility of highlighting correctly spelt keywords, and emacs bracket matching to show how correctly balanced parentheses match up. These simple features mean that the user gets some immediate feedback about the syntactic accuracy of their specification as they are entering it and would seem to make an appreciable contribution to usability. PVS also has online help for the specification language with such information as the precedence of infix operators.

All TPAs perform some checking of specification files, usually for syntax and then for type-correctness. CADiZ requires the creation of a text file with embedded codes outside the TPA environment which is then read, checked and represented to the human prover with the codes replaced by appropriate non-text symbols of the Z specification language, described in [Spi92]. PVS, with its text-based interface, offers an integrated process of editing, checking and error reporting for the construction of specification files, obviating the need for two dissimilar forms of the information.

With regard to the second question, many TPAs provide displays of a graph of dependencies between theory files which can be used to drive a browser. However, naming conventions and text searches also provide a powerful way of investigating theories. For example, we can be fairly certain that a theory file which deals with prime numbers will contain the string "prime". Of course a search for that string will also turn up all other files which refer to prime numbers and it may be possible to refine the search to also look for a characteristic expression associated with the definition of predicates.

3.2 Planning

As an aid toward the goal of completing proof, TPAs typically provide decision procedures and automated search strategies in order to automate some theorem proving work. It may be that these obviate the need for the human prover to think about the proof plan but when this is not so, the automation needs to be carefully deployed by the human prover. It seems appropriate to ask the question, how can the interface support the user in planning a proof using such automatic tools and there is no clear answer. There are no simple metaphors which the interface might attach to such complicated functions. We are reliant on appropriate naming and on-line help in order to assist the human prover in understanding the functions available and how to deploy them. Therefore we will not attempt to address questions of how an interface can support the *discovery* the plan of a proof.

However, in order to carry out such planning of a proof effectively, the plan must not only be discovered but recorded and effectively implemented. It may be entirely appropriate to record the emerging proof plan on paper before and during the interactive proving process but, as we will see when we come to reflect on the proof part way through, it can be very useful to be able to connect informal (or extra-formal) information about the proof plan with the interface representation of the proof. One suggestion for achieving this in some measure is Kalvala's annotations for proofs and specifications, see [Kal94]. This allows annotations to be attached to proofs and to be propagated through them, a facility which could be used to record the role that component parts play in the overall proof plan.

3.3 Articulation

Compared to planning, it is perhaps easier to see what the user-interface might provide to support the articulation of proof commands. Although different interfaces exhibit a rich diversity of different components, we can identify almost all of them as employing primarily command-line interaction or graphical direct manipulation. If we look at the example of the PVS proof checker, we find a host of tools supporting the command-line interaction, including cut-and-paste, history mechanisms, bracket matching, short-cuts. TPAs such as CADiZ, see [TV97] and Jape have interfaces which break away from the limitations of the ASCII character set and exploit high resolution displays to use more conventional mathematical notation in the representations of terms. The human prover can directly interact with these by selecting (sub)terms on which to perform commands from menus or tool-bars.

Some TPAs attempt to aid in the *selection* of proof commands. In such a case, articulating a proof command may mean making a selection from a menu. It is our belief that such a tool is provided not in order to help the human prover decide how to progress the proof but either to make the work of articulating a particular command easier or to make the human prover aware of the *range* of commands and arguments, especially theorems, which are applicable. It is quite possible that the human prover may be unaware of some commands or theorems, or at least have overlooked the possibility that they may be useful in the current context. In this case this tool could be viewed as an aid to planning. However, we have chiefly observed the use of such a tool when the human prover was having difficulty in articulating a proof command any other way, that is as an aid to articulation and not selection of commands.

We have not yet discussed the higher-level issue of articulation, that of transforming the kind of proof plan that humans are trained to devise into a sequence of steps which can be achieved using the particular TPA. At this level, the human prover is reliant on their ability to comprehend the behaviour of the proof tools provided by the TPA. As we mentioned with regard to planning, more powerful automatic proof tools may mean that this becomes more difficult, diminishing to some extent the advantage they offer.

3.4 Reflection

Most TPAs allow a proof to be viewed as a tree, with the goal statement at the root and supporting statements, fanning out, connected by the appropriate inference commands. These tree form proof displays are at least based on Gentzen trees, see [Gen69]. Such a display is generally thought to be helpful in answering "what am I doing now?" questions, when the human prover's understanding of the current context has been lost due to a pause in proving, or because they have been attending to one branch of the proof tree and are now turning their attention to one which was suspended some time ago. The shape of the tree, provided that a sufficiently large portion can be viewed on the screen at one time, allows the human prover to comprehend the structure of the proof and the location of the current goal within this structure. Such tree displays will typically elide information about the content of each statement, just showing the structure of the proof and some clues as to the commands which have been entered. This allows much more of a proof tree to be displayed than if all the information were displayed in a tree form. An exception to this is Jape and in order to overcome the problem of very wide proof tree displays Jape provides an alternative, Fitch box presentation, see [Fit52], which ensures that long proofs only exceed screen size in the vertical dimension. TPAs other than Jape have to provide a display which actually shows the information content of the proof, not just its tree structure. In the case of command-line based interface TPAs this is provided by the scrolling transcript of interaction. It is useful to be able to ask of the tree display what is the elided information and both PVS and CADiZ provide the facility to inspect the hidden goal information.

3.5 Reuse

Existing TPAs exhibit a host of tools for supporting reuse of proofs. PVS provides many of these, allowing theorems to be reused conveniently either singly, with a *rewrite* command, or composed into groups for "automatic" rewriting. PVS also provides the means to edit proofs off-line, whilst not actually within any proving process. Arbitrary parts of any proof may be inserted into any other proof. In addition, parts of other proofs may be "replayed" during the current proof, where they can be single-stepped. The ability to replay a proof admits the possibility of not only adding new parts to the proof tree through proof commands but of editing arbitrary parts of the tree and replaying the resulting proof. In [MH97], we discussed the impact of different kinds of proof command on the *viscosity* of a proof, our ability to make small changes to a proof without having to make large numbers of edits to maintain consistency, see [Gre89]. We can recast this issue as one of reuse, with the observation that when we make a small change to a proof we are reusing the old parts in this new and slightly different context. Thus proofs with high viscosity are proofs which are difficult to reuse in all but the most similar contexts. We made the observation that with a direct manipulation interface, articulating a command may be made easy by the ability to point to a particular (sub)term but that the record of the proof will then be littered with references to (sub)terms in a very specific context and will be highly viscous. Conversely, with a command line interface, the human prover tends to indicate information in a convoluted, indirect way, using constraints such as matching rather than specifying specific (sub)terms, which can be painfully awkward and require an understanding of the internal representation of displayed terms. However the resulting proof is far less viscous and therefore far easier to reuse than that created with direct manipulation interaction. We also indicated that if it were possible, as suggested by [Ste96], to automatically "renumber" references to (sub)terms as needed, the viscosity problem with direct manipulation proofs would be greatly reduced. An analogue of this is the way that direct manipulation word processors can renumber list items, avoiding the need to change numbers by hand when adding an entry.

Another possibility is that, rather than altering such specific commands each time, a tool be provided to generalise commands once and for all. It would be possible for specific references to (sub)terms to be replaced by the use of minimal constraints sufficient to achieve the same selection, especially if commands are combined into more meaningful groups. An example of this is rewriting, which is the composition of introducing an equality-based lemma, instantiating its universally quantified variables and then replacing some occurrence of the left hand side of equality with the right hand side. As separate steps it is hard to see how the variable instantiation could be expressed in terms of constraints but in conjunction, it is clear that the values to be introduced should be exactly those which will permit the replacement to happen by generating a match with the target expression.

In fact, we have identified three ways of thinking about how a lemma is used in another proof. Firstly, it may represent a key step in the proof, like an induction theorem. The requirement of the interface is to allow the human prover to be able to be aware of, locate and properly deploy such theorems. There are several ways in which we might want to try to locate a theorem. We might wish to search for all theorem which match some part of the current goal, we might use knowledge about the overall theory structure to look through certain theories, or we might take advantage of naming conventions in order to perform textual searches for a theorem.

Secondly, a lemma may be used, probably in conjunction with others, in order to reduce some term to a normal form. In this case, we are not interested in each individual application of each theorem, we simply want to understand that the result is in a normal form with respect to a certain set of theorems. Here, we require the interface to allow us to assemble such sets of rewrites, perhaps to perform certain kinds of analysis on those sets, such as an investigation of confluence, to be able to automatically apply the set of theorem repeatedly and to be able to interrupt the process if it is taking too long, perhaps because it is non-terminating. Industrial strength theorem provers, such as PVS, support this kind of use to some degree.

There is a third way of using theorems, especially those which capture associativity, commutativity, distributivity and absorption properties, and this is to allow a term to be easily rearranged into a new form. Here the requirement of the interface is that it provide efficient commands for articulating these relatively simple transformations. We are only aware of one TPA which currently supports which way of using theorems, which is a version of CtCoq, see [Ber97a].

4 Integrating Tools for a Single TPA

Here we consider how we can evaluate the effectiveness of the support that the TPA features listed above provide for the activities we have identified. We offer not so much criteria against which each can be scored but rather concepts which help in assessing the trade-offs.

With regard to support for formalisation, this is sufficiently close to the task of programming that we are tempted to borrow from the literature on that subject. We need to consult with psychologists about the extent to which results for one be applicable for the other. Assuming that this can, by and large, be done, we note that colourisation and formatting according to syntactic rules have proved very popular in the programming community and there is evidence that formatting has an effect, see for example [MMNS83].

Furthermore there is no clear winner for formalisation between the three options of an ASCII-only specification language, a specification language with more traditionally mathematical notation compiled from a separate text file, and a mathematical notation language using menus or key-sequences to express non-keyboard symbols. Most importantly, we believe that if one is to be chosen in preference to others, it must be respectful of its impact on other activities, such as searching to support reuse, or the ability to reflect on a proof in progress, where the goals in that proof are expressed using the same specification language. This makes it hard to imagine a rigorous psychological experiment that would identify one option as the strongest. It seems more practical to make a study based on cooperative evaluation, see [MWHD93], for a particular application of theorem proving.

The various ways of articulating proof commands, through gesture, by menu selection, or with textual commands all have merits both for articulation and in terms of their impact on other activities. It is not simply the case that command-lines are out of date and that we now have access to better technology. As well as affecting reuse through the viscosity issue raised above, the choice of commands provided and the way in which they can be articulated affects planning. There is already a recognition of this interconnection in the field of psychology and some work has been done to explore the relationship, see [GGC96]. Also we may look to the resources model described in [FWH97, FWH96]. Among other things, the ability of the user to predict the outcome of commands and to use this information to select an action is captured by the resources model's notion of an action-effect model.

There is anecdotal evidence for the value of tree displays of proofs and, since they are quite easy to implement and can be hidden or switched off, they are almost certain to be part of new TPAs. Thus, from our design perspective, we are not concerned with debate about the intrinsic value of such displays. What is more interesting is how such a display is linked with the possibly separate parts of the interface which display goals and allow commands to be articulated. CADiZ allows the human prover to select a part of the tree display to summon a display of that goal and make that goal the object of the next proof command. With the PVS tree display, the human prover must enter some number of textual "(postpone)" commands in order to perform such a change of focus and it may be difficult to decide the right number of such commands, looking at a large proof tree display. The introduction of tree displays admits new kinds of manipulation based on a view of the tree structure of the proof, such as large scale copying and moving of subproofs which has previously only been possible with textual representations of the proof. It is, of course, hard to phrase specific questions in the absence of more implemented examples.

With regard to the reuse of theorems, we believe that it is important to recognise the three different ways in which lemmas may be used: as key results, for normalisation or for algebraic rearrangements. Tools must be evaluated with respect to all three of these unless it is clear that some are irrelevant for some reason. It seems reasonable to use psychological experiments to evaluate such software, since metrics such as error rates and performance times would seem to be sensible indicators of their value. The retrieval of theorems can be awkward, to the point that simple theorems might be proved from scratch rather than located within the existing corpus, see [AGMT95]. In general information retrieval is the subject of considerable current research effort and it is beyond the scope of this paper to properly explore that field. We must be content with recognising that simplistic matching for retrieval can be a powerful tool but that the effective practical strategies that we have observed are principally reliant on naming conventions and flexible textual searches.

With regard to the reuse of proofs, however, we again find that the representation of the proof has a great impact on other activities as well as reuse and we need more holistic methods. The provision of some benchmarks for the reuse of proofs is unrealistic because they the logics used by difference TPAs vary too much.

5 Summary

We have presented a view of interactive theorem proving as a diverse collection of closely interrelated activities. We have catalogued a number of features of existing TPAs and connected them with the activities which they seem to support. We have then volunteered approaches to evaluating these features and begun to make explicit those questions which need to be passed on to psychologists. As a whole this lays down foundations for the design of new TPAs whose interfaces effectively support the range of activities involved in interacting theorem proving.

6 Acknowledgements

We would like to thank the UITP '98 reviewers for their contribution to this final version of the present paper.

References

[AGMT95] S. Aitken, P. Gray, T. Melham, and M. Thomas. Interactive Proof Discovery: An Empirical Study of HOL Users. In Philip Gray, editor, *User Interface Design for Theorem Proving Systems.* University of Glasgow, 1995.

[AGMT96] S. Aitken, P. Gray, T. Melham, and M. Thomas. Phases, Modes and Information Flow in Theory Development. In Merriam [Mer96].

[BBC+96] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. User guide to the CtCoq proof environment. Online at URL `http://www.inria.fr/croap/publications.html`, 1996.

[Ber97a] Y. Bertot. Direct manipulation of algebraic formulae in interactive proof systems. In *International Workshop on User Interfaces for Theorem Provers* [Ber97b], pages 17–24.

[Ber97b] Y. Bertot, editor. *International Workshop on User Interfaces for Theorem Provers.* INRIA, Sophia-Antipolis, 1997.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648. Springer-Verlag, 1990.

[CH95] R. Caferra and M. Herment. A generic graphic framework for combining inference tools and editing proofs and formulae. *Journal of Symbolic Computation*, 19(2):217–243, 1995.

[FGT98] W. M. Farmer, J. D. Guttman, and F. J. Thayer. *The IMPS User's Maunual.* The MITRE Corporation, version 2 of first edition, 1998. Online at `ftp://math.harvard.edu/imps/doc/imps-2.0-manual.ps.gz`.

[Fit52] F. B. Fitch. *Symbolic Logic: an introduction.* Ronald, 1952.

[FWH96] R. Fields, P. Wright, and M. Harrison. Designing human-system interaction using the resource model. In *APCHI'96: First Asia Pacific Conference on Human Computer Interaction*, pages 181–191, Singapore, June 1996. Information Technology Institute.

[FWH97] B. Fields, P. Wright, and M. Harrison. Objectives, strategies and resources as design drivers. In Steve Howard, Judy Hammond, and Gitte Lindgaard, editors, *Human-Computer Interaction INTERACT'97*, pages 164–171. Chapman and Hall, July 1997.

[Gen69] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen.* North-Holland, 1969.

[GGC96] D. Golightly, D. Gilmore, and E. Churchill. Puzzling interfaces: The relationship between manipulation and problem solving. Online at URL `http://www.psyc.nott.ac.uk/~dag/HCI96_txt.htm`, 1996. Presented at HCI '96.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Gre89] T. R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *Fifth Conference of the BCS HCI Special Interest Group*, volume V of *People and Computers*, pages 443–460. Cambridge University Press, 1989.

[JJLM91] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. mural: *A Formal Development Support System.* Springer-Verlag, 1991.

[Kal94] Sara Kalvala. Annotations in formal specifications and proofs. *Formal Methods in Systems Design*, 5(1/2):119–144, July 1994.

[MDH95] N. A. Merriam, A. M. Dearden, and M. D. Harrison. Assessing Theorem Proving Assistants: Concepts and Criteria. In P. Gray, editor, *User Interface Design for Theorem Proving Systems.* Department of Computer Science, University of Glasgow, 1995.

[Mer96] N. A. Merriam, editor. *International Workshop on User Interfaces for Theorem Provers.* Department of Computer Science, University of York, 1996.

[MH97] N. A. Merriam and M. D. Harrison. What is Wrong with GUIs for Theorem Provers? In Bertot [Ber97b].

[MMNS83] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Schneiderman. Program indentation and comprehensibility. *Human Aspects of Computing*, 26(11):861–867, November 1983.

[MWHD93] A. Monk, P. Wright, J. Haber, and L. Davenport. *Improving Your Human Computer Interface: A Practical Approach.* BCS Practitioner Series. Prentice-Hall International, Hemel Hempstead, 1993.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[Pol57] G. Polya. *How To Solve It.* Princeton University Press, second edition, 1957.

[SB95] B. Sufrin and R. Bornat. *The Gist of Jape*, 1995. Online at URL `http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/UNIXJAPEDOCHTML/gi%st.html`.

[SOR95] N. Shankar, S. Owre, and J. Rushby. *The PVS Proof Checker: A Reference Manual.* SRI International, 1995.

[Spi92] J. M. Spivey. *The Z Notation.* Prentice-Hall International, 2nd edition, 1992.

[Ste96] A. Stevens. Toward mechanised revision of proofs and theories. In Merriam [Mer96].

[Sym] D. Syme. TkHOL world-wide web site. Online at `http://www.cl.cam.ac.uk/users/drs1004/TkHol.html`.

[TBK92] L. Théry, Y. Bertot, and G. Kahn. Real theorem provers deserve real user-interfaces. In *The Fifth ACM Symposium on Software Development Environments (SDE5).* ACM, 1992.

[TV97] I. Toyn and S. H. Valentine. Proving conjectures using CADiZ. Technical report, Dept. of Computer Science, University of York, UK, March 1997. (Under ongoing revision).

[Ver] Richard Verhoeven. Mathspad homepage. Online at `http://www.win.tue.nl/cs/wp/mathspad/`.

Grail

An Automated Proof Assistant for Categorial Grammar Logics

Richard Moot

Utrecht institute of Linguistics OTS
Trans 10
3512 JK Utrecht
Richard.Moot@let.ruu.nl

Abstract

This paper gives an overview of the Grail system and its use as a tool for the development and prototyping of grammar fragments for categorial logics.

Grail is an automated theorem prover based on proof nets, a graph-based representation of proofs, and labeled deduction. The theorem prover is implemented in SICStus Prolog, the user interface in TclTk.

Though the underlying logic is decidable, and the theorem prover can operate automatically, user guidance is often desirable during the proof search. It can increase the performance of the algorithm and, more importantly, help the user visualize the status of the proof attempt thereby showing *why* a given statement is provable or not.

The Grail user interface is based on the Prolog debugger. At each proof step the user can take one of the following actions: *select* allows the user to select an inference step, *leap* performs automatic proof search until a proof is found, *fail* marks the current branch of the search tree as unsuccessful and *abort* abandons the entire proof attempt.

We have found the interface gives users better insight in the operation of the theorem prover and greatly enhances its facilities for prototyping and debugging of categorial grammars.

1 Introduction

Since 1958 [Lambek 58], linguists and logicians have been trying to specify grammars as logical theories (see e.g. [Morrill 94] or [Moortgat 97] for an overview). An advantage of this approach over more traditional approaches to linguistics is that we can prove our grammars have abstract properties like soundness, completeness and consistency.

In the logical framework we are proposing, the grammaticality of sentence should correspond to the derivability of a logical statement. More specifically a string of words $w_0, \ldots, w_n$ is a sentence of our language if and only if the lexicon l maps these words to formulas such that $l(w_0), \ldots, l(w_n) \vdash s$ is a theorem of our logic.

Grail is a tool which allows you to specify such logical theories together with a lexicon and see what kinds of sentences are derivable for the given logic.

2 Logical Background

Much of the inspiration for the Grail theorem prover comes from recent advances in linear logic [Girard e.a. 95] and labeled deduction [Gabbay 94], so I will briefly touch on these subjects.

2.1 Linear Logic

Linear logic, introduced in [Girard 87], was developed as a logic where the use of the structural rules of contraction and weakening, which apply freely in classical logic, is restricted. Without these structural rules the meaning of our connectives changes. Linear implication, written as '$\multimap$', is perhaps the clearest illustration of this. A formula $A \multimap B$ consumes or destroys an A formula in order to produce a B formula.

The sequent rules for (intuitionistic) linear implication are essentially the same as those of intuitionistic logic. Commutativity of the sequent comma is implicit.

Sequent Rules

$$\frac{}{A \vdash A}\,[Ax] \qquad\qquad \frac{\Delta \vdash A \quad \Gamma, A \vdash C}{\Gamma, \Delta \vdash C}\,[Cut]$$

$$\frac{\Delta \vdash A \quad \Gamma, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\,[L\multimap] \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,[R\multimap]$$

Example 1 *If we assign the formula $np \multimap (np \multimap s)$ to a transitive verb like 'likes', we say it combines with two np formulas like those assigned to 'Nixon' or 'cigars' to produce an s formula.*

$$\frac{\dfrac{}{np \vdash np}\,[Ax] \quad \dfrac{\dfrac{}{np \vdash np}\,[Ax] \quad \dfrac{}{s \vdash s}\,[Ax]}{np, np \multimap s \vdash s}\,[L\multimap]}{np, np \multimap (np \multimap s), np \vdash s}\,[L\multimap]$$

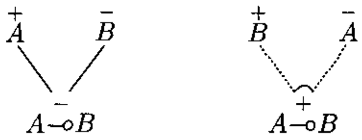
This allows us to derive 'Nixon likes cigars' as a sentence, but not, in the absence of weakening, 'Nixon likes cigars special relativity', nor, in the absence of contraction, 'Nixon likes'. Unfortunately, as commutativity still applies, the current logic makes the incorrect linguistic prediction that any permutation of a sentence is also grammatical. We will remedy this in section 2.2

2.1.1 Proof Nets

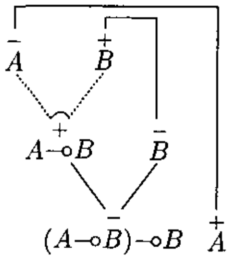
A clear advantage of the (minimal) fragment of linear logic introduced above is that it has an exceptionally elegant proof theory, called proof nets.

For a given logical statement we can obtain a graph by taking the formula decomposition trees and linking atomic formulas to their negations using axiom links, which correspond to the sequent axiom rule. We call these graphs *proof structures.* Proof structures are a superset of proof nets and not all proof structures will correspond to sequent proofs. However, it turns out that we can distinguish proof nets from other proof structures by looking only at graph theoretic properties of the proof structure.

Linear implication is decomposed as follows, depending on whether it occurs in a positive (non-negated or succedent) or negative (negated or antecedent) position.



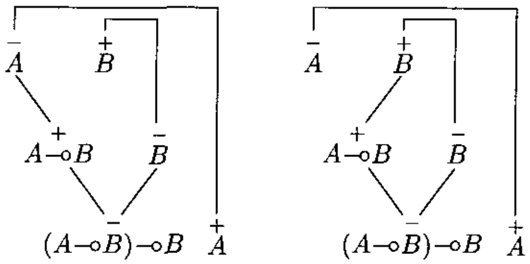
Example 2 *A proof structure for the sequent* $(A \multimap B) \multimap B \vdash A$.



$\overset{-}{A}$
$\overset{+}{B}$
$A \overset{+}{\multimap} B$
$\overset{-}{B}$
$(A \multimap B) \overset{-}{\multimap} B$
$\overset{+}{A}$

From a proof structure we can obtain a *correction graph* by 'switching' each of the dotted links either to the left or to the right. A proof structure is a *proof net* if and only if all its correction graphs are acyclic and connected.

Example 3 *The proof structure of example 2 is* not *a proof net, as shown by the correction graph on the right.*



Proof search for a given logical statement consists merely of trying all linkings of atomic formulas. A practical downside is that for $2n$ atomic formulas there can be $O(n!)$ of these linkings, so while computing them all is possible, it is unlikely that it can always been done efficiently.

2.2 Categorial Grammars and Labeled Deduction

Although the proof net approach described above gives us a very elegant proof theory, it does so only for associative, commutative logics. For serious linguistics, however, we will want the structural rules of associativity and commutativity, like weakening and contraction, to be *optional.*

In categorial grammars we drop commutativity and associativity as global options. Without commutativity the connective $\multimap$ will split into two versions, depending on whether the implication looks for its argument to the left or right. We will write A/B (resp. $B\backslash A$) for the formula which yields an A when it finds a B to its right (resp. left).

Because different linguistic constructions may require access to different sets of structural rules, we will also use a multimodal logic. We will indicate the modes of connectives by using an index as subscript (e.g. $A/_iB$, $A\backslash_iB$).

In order to build upon the proof theoretic advances of linear logic, we embed the categorial logic into linear logic by means of labeling. Instead of using formulas A as our basic declarative unit we use labeled formulas $X : A$, where the label X represents structural information. The label assigned to the (single) succedent formula of the proof represents the way the formulas in the antecedent are organized. Labeled sequent rules for '/' are the following (the rules for '\' are symmetric).

$$\frac{\Delta \vdash Y : B \quad \Gamma, X \bullet_i Y : A \vdash Z : C}{\Gamma, \Delta, X : A/_iB \vdash Z : C}\,[L/] \qquad \frac{\Gamma, \mathrm{x} : B \vdash X \bullet_i \mathrm{x} : A}{\Gamma \vdash X : A/_iB}\,[R/]$$

Compared to the sequent rules for linear implication, we note that the only difference is in the labels. For the $[L/]$ rule, the label $X \bullet_i Y$ assigned to the result category A is a structured term indicating that the label Y computed for the formula B occurs to the direct right of structure label X assigned to formula $A/_iB$. The $[R/]$ rule says that in order to prove a formula $A/_iB$ we assign B a new label x and demand it occurs on the direct right of the structure label assigned to formula A.

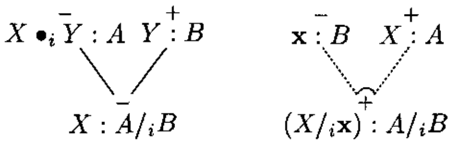
Structural rules operate on the labels only. For example, we can declare a mode c to be commutative by means of a structural rule which allows us to replace sublabels $X \bullet_c Y$ by $Y \bullet_c X$.

$$\frac{\Gamma \vdash Z[Y \bullet_c X] : C}{\Gamma \vdash Z[X \bullet_c Y] : C}\,[Com]$$

Example 4 *When we have a commutativity rule for mode* c, $\mathbf{x} : A/_cB \vdash \mathbf{x} : B\backslash_c A$ *is a theorem.*

$$\cfrac{\cfrac{\cfrac{}{\mathbf{y} : B \vdash \mathbf{y} : B}\,[Ax] \qquad \cfrac{\cfrac{}{\mathbf{x} \bullet_c \mathbf{y} : A \vdash \mathbf{x} \bullet_c \mathbf{y} : A}\,[Ax]}{\mathbf{x} \bullet_c \mathbf{y} : A \vdash \mathbf{y} \bullet_c \mathbf{x} : A}\,[Com]}{\mathbf{x} : A/_cB, \mathbf{y} : B \vdash \mathbf{y} \bullet_c \mathbf{x} : A}\,[L/]}{\mathbf{x} : A/_cB \vdash \mathbf{x} : A\backslash_c B}\,[R\backslash]$$

Moortgat [Moortgat 97, section 7] proposes the following way to add labeling to proof nets (the cases for '\' are again symmetric).



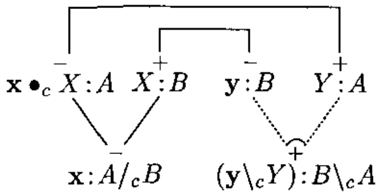
We can see the similarity between the labeled sequents and the labeled proof nets. Only the case for positive occurrences is somewhat different from the $[R/]$ rule: the label $X/_i\mathbf{x}$ should be seen as a constraint specifying that $\mathbf{x}$ should be a sublabel occurring on the immediate right of X. We check this constraint by means of a conversion:

$$(X \bullet_i Y)/_iY \rightarrow_{R/} X$$

Similarly, we will have a label conversion for each of the structural rules.

Using this labeling we can generate an acyclic, connected proof structure, compute the label of the succedent formula and check if we can satisfy all constraints on this label by means of the conversion rules.

Example 5 *A proof net corresponding to the sequent proof above would look like*



$\mathbf{x} \bullet_c X : A$
$X : B$
$\mathbf{y} : B$
$Y : A$
$\mathbf{x} : A/_cB$
$(\mathbf{y}\backslash_c Y) : B\backslash_c A$
−
+
−
+
−
+

The unifications $X := \mathbf{y}$ *and* $Y := \mathbf{x} \bullet_c X$ *will produce the label* $\mathbf{y}\backslash_c(\mathbf{x} \bullet_c \mathbf{y})$ *for the conclusion. We can then apply the conversions*

$$\mathbf{y}\backslash_c(\mathbf{x} \bullet_c \mathbf{y}) \rightarrow_{Com} \mathbf{y}\backslash_c(\mathbf{y} \bullet_c \mathbf{x}) \rightarrow_{R\backslash} \mathbf{x}$$

showing $\mathbf{x} : A/_cB \vdash \mathbf{x} : B\backslash_c A$ *is a theorem.*

3 The Grail Theorem Prover

3.1 Brief history

In the end of '95, the first incarnation of Grail was a piece of Prolog code of some 250 lines. You could enter a logical statement and wait until it produced an answer in the form of yes or no, or until you got bored (which happened a lot those days). In spite of a number of improvements to the efficiency of the original code, several grammar fragments designed in it could not handle longer sentences in a reasonable amount of time.

I therefore tried to give a user-friendly representation of the computation state with which the user can inspect and guide the computation. The benefits of this are twofold: firstly, the user can select a promising state from the possible states and abandon hopeless subgoals, and secondly, it gives the user insight into *why* specific statements are underivable, without having to use the Prolog debugger where tracing the execution is difficult even for the programmer.

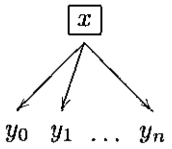
The current version is some 8000 lines of mixed Prolog and TclTk code, using the TclTk library included with SICStus Prolog. It can be used without knowledge about Prolog and produces output in human-friendly natural deduction format.

Grail is used as a research tool and as courseware for introductory to advanced level courses in categorial grammar.

3.2 Execution Model

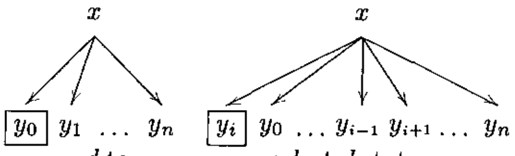
Though we work in a hybrid system of proof nets and labeling, the basic execution model of the interactive mode is the same in both cases. It is based on a simplified version of the Prolog debugger.

From the current state x we can apply a finite number of steps to get in a next state y_i. If no rules are applicable or the user selects *fail*, we backtrack to the first parent of the current state which still has unvisited daughter states and proceed from there. When the current state is a successful state (i.e. we have found a proof) we can either *abort* the computation or continue to search for more proofs.



When we select *creep* the proof continues with y_0 as its next state, as shown below. This step is nondeterministic; when no proof can be found for descendants of state y_0, computation will continue with state y_1.

The *select* step, of which creep is a special case, enables the user to reorder the goals, such that a state y_i of his choice will be the next state. When no proof can be found for descendants of this state, control is passed to the user who is then allowed to select a next state.



creeped to y_0 *selected* state y_i

There are two other safe commands we can use: *leap*, after which Grail will compute until it has found the first solution and *nonstop* which will cause Grail to grind on until all proofs have been found.

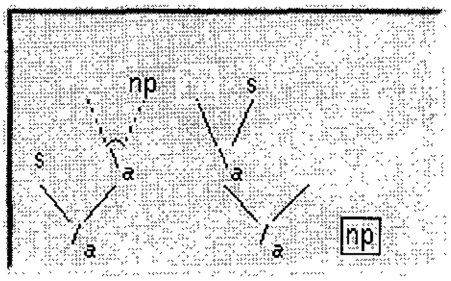
The final two commands are unsafe in that they can prevent proofs from being found by pruning the search space. *fail* will mark the current state as unsuccessful and continue computation on the next unvisited state, and *abort* will completely abandon the current proof attempt even if in leap or nonstop mode.

3.3 The Proof Net Window

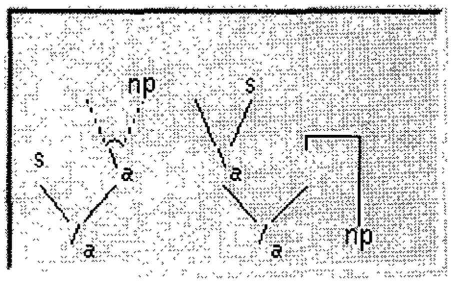
In the proof net window we see the current partial proof structure, with positive atomic formulas drawn in white and negative atomic formulas drawn in black. Here atomic formulas of opposite polarity are linked until we find a proof structure which is both acyclic and connected, after which we can try to satisfy the label constraints.

As an example we give a proof of the theorem $s/_a(np\backslash_a s), (np\backslash_a s)/_a np, np \vdash s$ which the lexicon would produce for 'Someone killed JFK'.

We first select an atom. This is a committed choice step; after we select the atom we are forced to link it before we can select another atom. After this selection all possible ways to link it by an axiom link will appear. As shown below, the selected *np* will appear in a black box and the atoms of opposite polarity which have not been tried before will appear in a white box. Each of these atoms will represent a possible next computation state and they will normally be searched from left to right. For the moment however, we use some user guidance and select the rightmost atom, knowing we can always try the other possibility later.

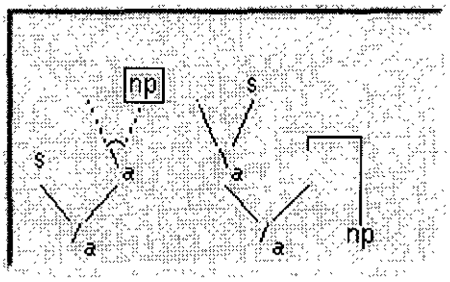


1. Selecting *np*

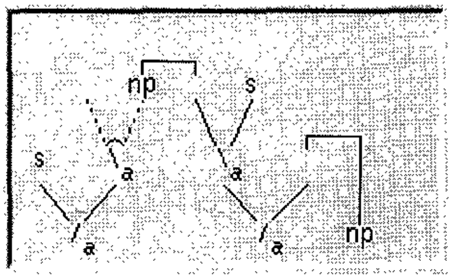


2. Linking

Next, we select the second negative *np*, which we are forced to link to the final *np*.

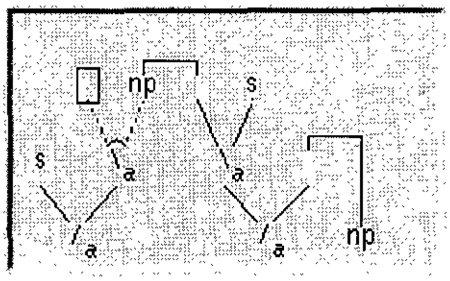


3. Selecting other *np*

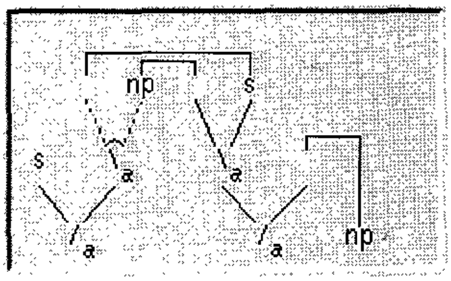


4. Linking

We select one of the *s* formulas. Linking it to the rightmost *s* would create a cycle with the switch set to the left and cause that branch of the search space to fail immediately, so we pick the other *s*.

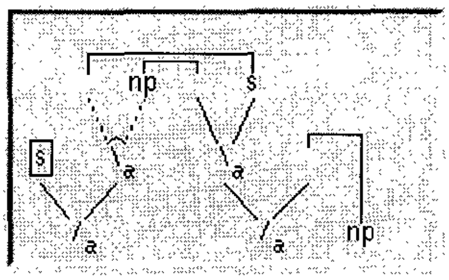


5. Selecting *s*

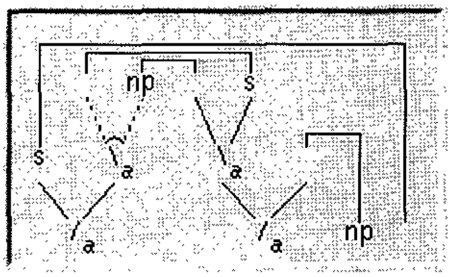


6. Performing noncyclic link

Finally, we connect the last two formulas.



7. Selecting final *s*



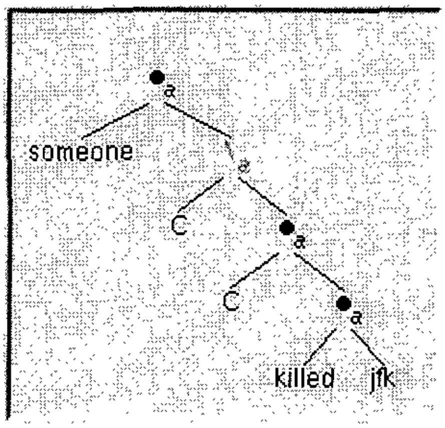
8. Done

3.4 The Rewrite Window

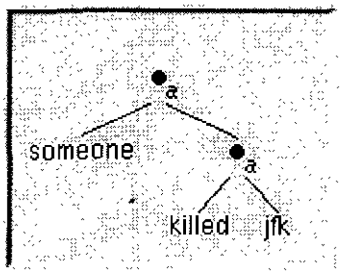
We check the label constraints by means of a very simple term rewrite system. Because a typical grammar fragment will have a number of rules (like e.g. commutativity) for which a naive algorithm would loop, Grail keeps track of visited labels in a closed set. Again, we have the option to either creep through Grail's built-in rewrite system, which uses breadth first search with local heuristic ordering, or to select the conversion we want to apply ourselves.

Grail will draw the current label as a tree. Next states will be all labels which can be obtained by a single rewrite step from the current label. Clicking on a node in the tree will cause a pop-up menu to appear with the possible conversion steps rooted at this node. Here we can select the conversion we want to apply, and Grail will keep track of the other possibilities.

For the example we used in the previous section, we will start with the label below.



The constructors denoting unsatisfied constraints, like the '$\backslash_a$' above, will be drawn in dark grey. In this case we can immediately convert this label to



which satisfies all constraints.

Typically there will be a large number of language specific structural rules in a grammar fragment and checking the label constraints will be a lot more complicated than the one step reduction above, making user interaction a valuable asset.

4 Conclusions

We have given an overview of the Grail interactive theorem prover and its underlying logical theory. Grail displays an intuitive representation of the state of the computation and allows the user to guide the computation by interacting with this representation.

On the proof net level, an advantage over sequent or natural deduction systems is that linking atomic formulas is a relatively trivial way to generate all proofs for a given statement. User guidance allows more experienced users to perform the axiom links they are interested in immediately, thereby sidestepping the $O(n!)$ complexity.

On the label rewrite level, it is often enlightening to see Grail (ab)use your carefully chosen structural rules in unintended ways, showing linguistically incorrect predictions of your logical theory, or to see it fail to satisfy a critical constraint, pointing to a missing or not sufficiently general structural rule. User interaction can considerably improve the performance by allowing the user to perform the intended label conversions himself.

Finally, though proof nets are in many ways an optimal proof theory for proof *search*, natural deduction is generally a better theory to *display* them. Therefore, source code which transforms the completed proof net into LaTeX natural deduction output is included with the release.

References

[BtM 97] Benthem, J. van, and A. ter Meulen (eds.), *Handbook of Logic and Language*, Elsevier, 1997.

[Gabbay 94] Gabbay, D. *Labeled Deductive Systems*, Report MPI-I-94-223, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[Girard 87] Girard, J.Y. *Linear Logic*, Theoretical Computer Science 50, 1987, pp. 1-102.

[Girard e.a. 95] Girard, J.Y., Y. Lafont and L. Regnier (eds.), *Advances in Linear Logic*, London Mathematical Society Lecture Notes, Cambridge University Press, 1995

[Lambek 58] Lambek, J. *The Mathematics of Sentence Structure*, American Mathematical Monthly 65, 1958, pp. 154-170.

[Moortgat 97] Moortgat, M. *Categorial Type Logics*, chapter 2 of [BtM 97].

[Morrill 94] Morrill, G. *Type Logical Grammar. Categorial Logic of Signs*, Kluwer, Dordrecht, 1994.

Notions of dependency in proof assistants

Olivier Pons, Yves Bertot and Laurence Rideau

INRIA Sophia Antipolis - BP 93
06902 Sophia Antipolis - France
{Olivier.Pons, Yves.Bertot, Laurence Rideau}@sophia.inria.fr

Abstract. This article describes uses of dependencies in tools to maintain big proofs in interactive proof assistants.

1 Introduction

When developping a large proof or a large theory in an interactive proof system, it is often necessary to modify previous results and axioms or to move theorems from one context to another. To maintain the consistency of the mathematical development after such an operation it is necessary to take care of the results that depend on the modified objects.

Our objective is to use notions of dependency to develop general tools to help users in the task of maintaining their large proof developments.

We have concentrated on two activities, which we believe to be the most frequent ones for maintenance tasks, and which could benefit from automatic support.

Changing axioms and basic definitions. During a proof or theory development, a user often has to modify some of the basic facts. For instance, if the objective is to re-use a theorem in a more general context, one may need to weaken the assumptions on which it relies. It is then necessary to know which part of the development may be reached by these modifications and must be checked again. We want to produce tools to guide the user in this task.

Code motion. A development is a set of definitions and theorems, but it is generally organized so that related parts are grouped together. Ensuring that related theorems are all stored in the same area is important for the long term re-usability of the proofs. Unfortunately, users are sometimes reluctant to organize correctly their mathematical results, as they are intimidated by the task of restoring the consistency of their proof development.

A situation that frequently leads to mathematical results being misplaced is that users discover that a theorem is missing only when they need it. Users then tend to prove the missing theorem in their own context, and store it there. This often leads to duplicated efforts, as other users will not know that the theorem has already been proved if it is not stored in its natural place. However, putting a theorem in its natural place requires an effort, as adding a theorem in a well-used theory may have consequences in very large contexts, due to automatic proof procedures.

Tools described in this paper are not fully implemented, but some experiments have already been performed to understand the needs and design these tools in detail. In the rest of this paper, there are five sections. Section 2 describes the dependencies that we use in our tools and the data on which they are computed. Section 3 describes the most basic tool that uses these dependencies: a visualizer. Section 4 describes how these dependencies are used to support the work that is necessary after modifying axioms or lemmas. Section 5 describes the issues involved when moving lemmas around. Section 6 brings a conclusion to this work.

2 Computing dependencies

Computing dependencies is a well known issue for people interested in the maintenance of large software systems. The usual approach is to consider files as the basic units (for instance a compiler processes one file at a time), and tools like make [Fel79] use descriptions of the dependencies between files to process them in order, while tools like makedepend[Wal84] produce these descriptions by analyzing the C code they contain. But we need a finer grain.

Pure object versus object used to build it. Some provers maintain a concept of proof objects (in the form of λ-terms as in Coq[CCF^{+}95], Lego[LP92], Alf [MN94], in the form of a derivation as in Jape[BS], EUODHILOS[SMO92] ...). The relations between different proof objects, which we will call logical relations of dependency, make it possible to understand which objects are necessary to construct a given object.

Some provers (Coq,Lego,Hol [GM93] ...) provide concepts of tactics and proof scripts. A tactic is just a command which transforms an initial goal into zero or several sub-goals remaining to prove. The proof is finished when there are no more sub-goals. A sequence of tactics constitutes a proof script.

In some systems (Coq, Lego) scripts can be used in to construct the proof object which is the only thing stored in the system.

In fact, it is the proof script itself that we want to maintain. The proof script is the real source code that was initially given by the user. Because it may use automatic proof search procedures, this proof script may be more concise and abstract than the resulting proof object.

That will be mainly done by using the logical dependencies. To compute those dependencies we must use the most informative structure. This is generally the proof object when it exists. In fact proof script usually do not contain all the information which is hidden by complicated and automatic tactics.

Computing the dependencies is done by going throw the object and finding all the object used in it simply by scanning term for identifiers.

Example. Consider the following Coq proof script:

```
Lemma ass_app : (l,m,n : list)(app l (app m n))=(app (app l m) n).
Intros.
Apply sym_equal.
Auto.
```

This script leads the system to construct the following proof object:

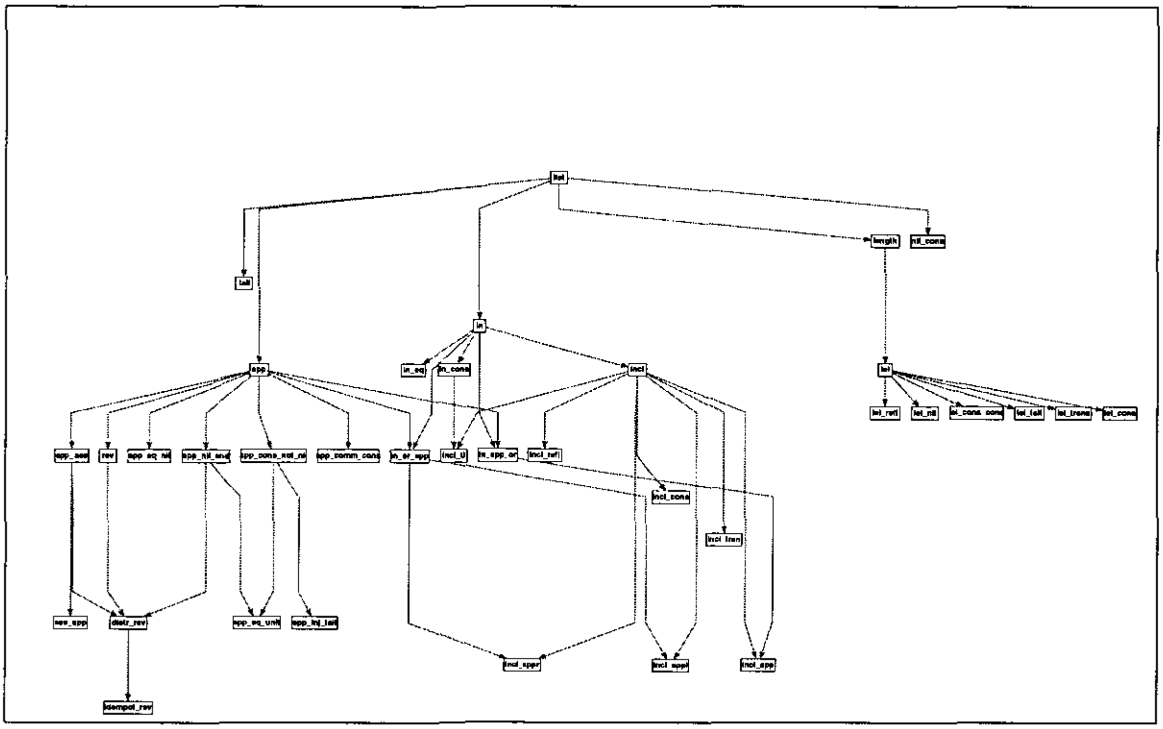
```
ass_app =
  [l,m,n:list]
    (sym_equal list (app (app l m) n) (app l (app m n)) (app_ass l m n))
```

Going through this proof term we can say that it depends on `sym_equal, list, app, app_ass`. Even if these name do not appear in the proof script the objects are necessary for the construction of the proof.

Performing this kind of analysis for all the objects of the development makes it possible to construct a complete dependency graph.

3 Visualizing and navigating dependency links

With a dependency graph, it is possible to allow the user to understand and check the structure of his development. We can provide a global visualization of dependencies by compiling our graph data structure into a formalism (like [Him97a]) understood by a graph displaying program [Him97b,Frö97,MF96]. For example, the following figure gives the dependency graph between the theorems and definitions of the `List` theory provided in the standard library of Coq.



For large developments, this display mechanism becomes quickly very obscure and it is relevant to provide tools to navigate this graph. When selecting an object and with a single key stroke, we want to highlight all the objects on which it depends, the objects that directly depend on it, the objects that transitively depend on it, etc.

4 Changing a set of definitions

Now that we have tools to understand the structure of a development we discuss the design of a tool to help the user restore the consistency of his development after changing axioms or lemmas.

4.1 Initial operation of the user

When the user needs to modify established results or axioms, he can require the help of the dependency tool to evaluate the amount of theorems that will need to be adapted to the new context. The tool simply computes the set of all the objects that can be reached through the dependency links from the modified objects. The tool can even be used *before* actually committing the modification, to evaluate the amount of extra work it will imply.

4.2 Working method

The dependency tool answers by proposing a list of theorems to be modified. It can propose extra information to ease the choice of the user (visualizing the statement of theorems, counting the number of descendants, ...). The user chooses and modifies a theorem taken from this list. The dependency tool then re-computes its graph to take into account the new modified object and proposes a new selection of theorems that must be redone, or informs that the end of the modifications propagation process has been reached.

4.3 Opaque and transparent dependencies

For a given modification, the sub-graph of the dependency graph that contains all the objects that depend transitively on the modified objects is often a gross over-estimation of the amount of proofs that need to be re-done. Theorems depend on each other only through their statements and not their proof. This is the opaque dependency. As a consequence, the need to propagate modifications will stop as soon as one can adapt the proof of a theorem without changing its statement. For instance, if theorem `T` depends on theorem `U` that depends on `V`, a change in the statement of `V` will require a change in the proof of `U`, but not in `T` if `U`'s statement is left unchanged. In systems based on type theory, an opaque dependency is a dependency on an object's type.

On the other hand a dependency on the term is called a transparent dependency. Transparent dependencies correspond to objects whose type gives too little information on their actual value (for instance, `plus`, `mult`, and `power` have the same type `nat->nat->nat`).

Instead of analyzing separately each dependency, we consider transparent and opaque objects: all the dependencies on opaque objects are opaque. It is harder to determine the status of a dependency on a transparent object, but a safe approximation is to consider such dependencies as transparent. Practically, the problem is to tell apart opaques and transparent objects. Usually theorems are opaque, while definitions are transparent. This can usually be determined with a good knowledge of the proof system behavior. The dependency tool simply considers that an opaque node has not been modified if its statement does not change after the modification. By analogy, separate compilation uses interfaces to describe modules. These interfaces play the same role as theorem statements and procedures declared but not described in the interfaces are opaque. In the `C` programming language for instance, procedures are usually perceived as opaque while macros would be transparent.

4.4 Incompleteness with respect to scripts

Theoretically, the propagation should stop when we have gone through all the nodes reachable from the modified nodes.

Nevertheless the presence of automatic decision procedures that can be seen as a "black boxes" complicates the matter. A modification of the code or the context in which these procedures execute can change their behavior and thus invalidate the proof script. Thus, we cannot be sure to be able to replay the code after that all the modifications provided by the calculation of the logical dependencies have been done. We come back later to this issue, as it is strongly related to that of code motion.

4.5 Necessary data structures

We maintain a graph of dependencies. The algorithm of propagation marks on the nodes of the graph (so that we constantly know their status). The graph is doubly linked so that each node can immediately know the status of its parents.

Recovering a proof development is a longlasting activity, that may span over several days. It is necessary to allow for interruptions and store the current state of the task.

The information that needs to be kept consists in three parts:

1. the whole consistent script before the first modification,
2. the script that contains all the objects modified so far,
3. the dependency graph, annotated to indicate the object being modified at the moment of interruption.

4.6 Automatic support for adapting individual objects

When updating an individual theorem, it is useful to replay the proof commands that were used to prove this theorem in the old context.

There are two possible cases:

1. The replay succeeds. In this case, there is not much more to do. If the theorem's statement is unchanged and the theorem is opaque then the theorems that depend only on the current one do not even need to be studied. Their proof can also be replayed automatically.
2. The replay fails. In this case, the script, and maybe the theorem's statement, need to be updated. A simultaneous replay of the previous script in the old context and the new context, in two different sessions, will help reuse a large part of this script. The local dependencies as described in [Pon97] can also be used for this purpose.

5 Moving data around

We consider the reordering of a set of theorems by moving proof script fragments from one place to the other. We only want to consider moves that respect the structure imposed by the dependency graph. If two theorems T and U are unrelated by the dependency relation, then U can be moved freely around T. Theoretically, this move should not have any consequence on the validity of the proof script. In the presence of automatic procedures whose behavior depends on the context, this will not be true.

5.1 Working method

The user marks a theorem or a definition in the script by a mouse click and then select a new position for this theorem. There is censor that uses the dependency graph to check if this move is coherent so that those code motion can not introduce incoherent state of script.

We can move an object in two direction (upward or backward) and there is two level of code motion, displacement in files or from one file to another.

When we move an object backward, there may be two cases. If it is independent from all the objects of the script between its old position and the new one then we can move it without additional work. If it dependson any objects inhis interval those objects also have to move to keep the rightorder.

The system computes the intersection of the dependency graph of the moved theorem and the set of object present in the part of the script between the old position and the new one.ă

For an upward move the principle is the same with the object of the script that depend on the moved theorem. The computation is done in the same way.

5.2 Failure due to code motion

Let us consider a proof script that contains the proofs of two theorems T and U in that order, and so that the proof of U does not depend on the proof of T. If U is moved in front of T and the proof of T uses automatic procedures, the behavior of these automatic procedures may be changed by the presence of U in a way that adds a dependency on U for the proof of T.

For instance, the proof of T may contain a subgoal that is solved easily when U is in the context and not otherwise. In this case, the initial proof of T contains commands to solve this subgoal the hard way, while this subgoal simply vanishes when executing the same proof script in the new context. The useless commands interfere with the rest of the proof and make the proof of T fail.

It seems that the problem is only that automatic procedures become more powerful when one adds theorems in their context. In this case, it should be possible to automatically clean the proof script for T, by removing the now useless proof commands. To do this, the tool must compare the runs of the proof script in the two contexts.

This solution assumes that the subgoals for the proof of T in the new context were already present in the old context. The next section shows that this property is not always achieved

5.3 A counter example in Coq.

In this example, we use a few tactics from the Coq system. The automatic tactic Auto uses all the theorems declared with a Hint command. The tactic Apply thm where thm has the statement A_1 -> ...-> A_k -> B, matches the current goal with B and returns the k sub-goals corresponding to A_1, ..., A_k. The tactic Split breaks goals corresponding to inductive inductions with one constructor (conjunctions fall in this category) and fails on other kinds of goals. Tactics can be composed with a semi-column, where $\texttt{tactic}_1$;$\texttt{tactic}_2$ applies $\texttt{tactic}_2$ to all the subgoals generated by $\texttt{tactic}_1$. Tactics can also be combined using an Orelse combinator with an obvious meaning.

We first construct a context by defining some theorems and definitions:

```
Inductive B: Prop -> Prop -> Prop -> Prop :=
   B_intro: (a, b, c:Prop) a -> b -> c ->(B a b c).
Parameters H1, H2, H3, H4:Prop.
Parameters th1:H1; th3:H3.
Hint th1.
Parameter th:(B H1 H2 H3 /\ H3).
```

Then, in this context, we state a specially crafted goal and apply a compound tactic.

```
Goal ((H4 /\ H4) /\ (H4 /\ H4)) /\ (B H1 H2 H3 /\ H3).

Split; (Split; Auto; Split) Orelse Apply th.
```

This command leaves four subgoals with the same statement: H4. As a quick explanation, we can see that the first Split generates two subgoals. On the first one the first branch of the Orelse compound succeeds and produces four subgoals of statement H4. On the second one, the second Split produces three new subgoals: H1, H2 and H3 /\ H3. The tactic Auto solves the first one and does nothing for the other two. Then the Split command fails on the subgoal of statement H2. The whole first branch of the Orelse compound fails, but the second branch succeeds.

Now let us consider that we move theorems that prove H2 and H4 before this proof and make them availlable to Auto.

The compound tactic produces completely different subgoals. The first Split still produces two subgoals, but the first branch of the Orelse compound does not fail anymore on these subgoals, thanks to the Auto command. So the whole command produces two subgoals with the same statement: H3. We see here, that Auto has become more powerful due to the motion of theorems th2 and th4 and that the subgoals generated by the compound tactic are not a subset of the subgoals that existed before. These goals will not be solved by script fragments coming from the previous script.

This is due to the fact that the Orelse combinator is not monotonous: if Auto solves more goals in the new context, we are not ensured that Auto Orelse tac will solve more goals. However, Orelse is seldom used and an automatic approach like the one given in the previous section will give good results in many cases.

6 Conclusion

In this paper we address two important tasks for computer-verified proof maintenance. The data we want to maintain is the proof scripts, that is, the sequences of commands that can be sent to a proof system to make it verify the proof.

The first task we consider concerns the modification of logical statements for established facts or axioms. When users need to perform this kind of modification, they have to propagate the modification in the proof development and this can be very time-consuming. The second task concerns code re-organization. This activity is important to ensure the readability and the long-term usability of the proof development.

We show that a notion of dependency is central in these issues. We propose to construct tools around a dependency graph. The first tool makes it possible to visualize dependencies and to navigate the graph. This helps the user understand the structure of the proof development. The information this tool can bring is useful before taking the decision to modify or move an artefact. A second tool helps the user when he has modified the statement of a lemma or an axiom. It uses the dependency graph to indicate to the user the theorems and proofs that need updating. While this tool does not directly help the user making the right modification, it helps propagating all its consequences to other objects. A third tool supports the activity of code motion, where dependencies can be used to indicate the set of theorems that must be moved together.

This work shows that there are two notions of dependencies. The dependency graph that we study uses *direct* dependencies, where a theorem depends on another only if the proof of the former refers to the latter. However, since we consider proof script maintenance, we are also faced with indirect dependencies, where theorems may influence the behavior of proof search procedures, so that the move of a theorem may invalidate proofs that were initially unrelated.

We believe that this case is rare, and we think that a more thorough study of the abstract properties of tactics and tactic combinators is needed.

References

[BBC^{+}96] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CtCoq Proof Environment.* INRIA, Feb 1996.

[BS] Richard Bornat and Bernard Sufrin. Jape - a framework for building interactive proof editors. Available at http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/jape.shtml.

[BT98] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 22, 1998.

[CCF^{+}95] C. Cornes, J. Courant, J.C. Filliatre, G. Huet, P Manoury, C. Munoz, C. Murthy, C. Parent C. Paulin-Mohring, A. Saibi, and B. Werner. *The COQ Proof assistant, Reference Manual, Version 5.10.* INRIA, Le Chesnay Cedex, France, July 1995.

[Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *spe*, 9(4):255–65, April 1979.

[Frö97] M. Fröhlich. *Incremental Graphlayout in the Visualization System daVinci (in german language).* PhD thesis, Department of Computer Science; University of Bremen, November 1997.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[Har97] John Harrison. Proof style. Technical Report 410, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1997. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/style.html`. Revised version to appear in the proceedings of TYPES'96.

[Him97a] M. Himsolt. Gml: A portable graph file format. Technical report, Universität Passau, 1997.

[Him97b] M. Himsolt. Graphlet manuals: Gml, graphscript, c++ interface. Technical report, Universität Passau, 1997.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[MF96] M. Werner M. Fröhlich. davinci v2.0 online documentation, 1996. Available at http://www.informatik.uni-bremen.de/~davinci/doc_V2.0.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[Pon97] Olivier Pons. Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997. Available at http://www.inria.fr/croap/-events/uitp97-papers.html.

[SMO92] Hajime Sawamura, Toshiro Minami, and Kyoko Ohashi. Euodhilos: A general reasoning system for a variety of logics. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR 92),St. Petersburg, Russia*, volume 624. SpringerVerlag LNCS, 1992.

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.

[Wal84] Kim Walden. Automatic generation of make dependencies. *spe*, 14(6):575–585, June 1984.

$\mathcal{L}\Omega\mathcal{UI}$: A Distributed Graphical User Interface for the Interactive Proof System ΩMEGA

Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou,
Detlef Fehrer, Armin Fiedler, Helmut Horacek, Michael Kohlhase,
Karsten Konrad, Andreas Meier, Erica Melis, Volker Sorge,
FB Informatik, Universität des Saarlandes, Germany
`http://www.ags.uni-sb.de`

Abstract

Most interactive proof development environments are insufficient to handle the complexity of the information to be conveyed to the user and to support his orientation in large-scale proofs. In this paper we present a distributed client-server extension of the ΩMEGA proof development system, focusing on the $\mathcal{L}\Omega\mathcal{UI}$ (Lovely ΩMEGA User Interface) client. This graphical user interface provides advanced communication facilities through an adaptable proof tree visualization and through various selective proof object display methods. Some of $\mathcal{L}\Omega\mathcal{UI}$'s main features are the graphical display of co-references in proof graphs, a selective term browser, and support for dynamically adding knowledge to partial proofs – all based upon and implemented in a client-server architecture.

1 Introduction

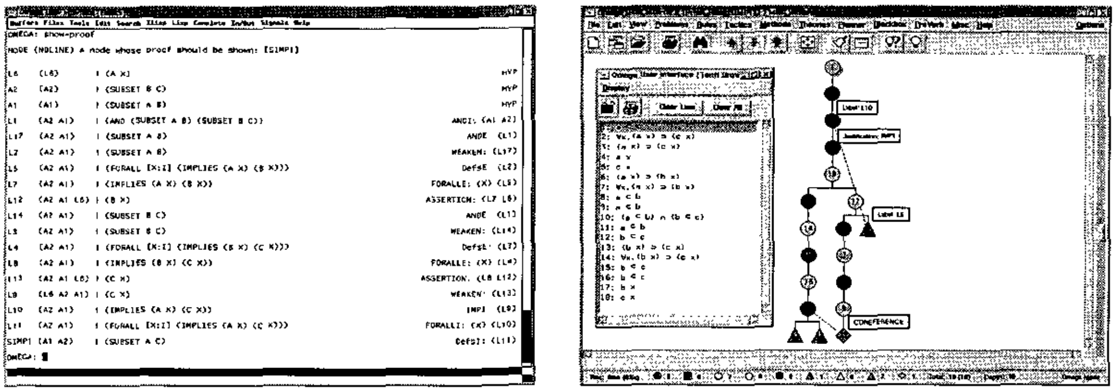
One (of several) reasons, why current deduction systems have not found a wider acceptance in mathematical practice is that they are too inconvenient to use. The ΩMEGA system [BCF$^+$97] – an interactive, plan-based deduction system with the ultimate goal of supporting theorem proving in main-stream mathematics and mathematics education must address this, in order to reach its goal. In order to provide a conceptually structured, understandable and easily usable front-end, the interface $\mathcal{L}\Omega\mathcal{UI}$ of the ΩMEGA system is designed with respect to the following requirements:

- In any proof state the system should display the proof information to the user at different levels of abstraction and detail and furthermore in different modes (e.g. as a proof tree, as a linearized proof, or in verbalization mode, etc.).

- The system should minimize the necessary interaction by suggesting commands and parameters to the user in each proof step. Optimally, the system should be able to do all straight-forward steps autonomously.

- The interface should work reasonably fast, and its installation in other environments should be possible with minimal effort and storage requirement.

These issue are elaborated in detail in the following three sections. We will only discuss the ΩMEGA proof system (the current system consists of a proof planner and an integrated collection of tools for formulating problems, proving subproblems, and proof presentation) where it becomes necessary to understand the interface issues.



```
OMEGA: show-proof
NODE (NDLINE) A node whose proof should be shown: [SIMP1]

L6    (L6)         ⊢ (A X)                                          HYP
A2    (A2)         ⊢ (SUBSET B C)                                   HYP
A1    (A1)         ⊢ (SUBSET A B)                                   HYP
L1    (A2 A1)      ⊢ (AND (SUBSET A B) (SUBSET B C))       ANDI: (A1 A2)
L17   (A2 A1)      ⊢ (SUBSET A B)                              ANDE (L1)
L2    (A2 A1)      ⊢ (SUBSET A B)                          WEAKEN: (L17)
L5    (A2 A1)      ⊢ (FORALL [X:I] (IMPLIES (A X) (B X)))       DefsE (L2)
L7    (A2 A1)      ⊢ (IMPLIES (A X) (B X))                FORALLE: (X) (L5)
L12   (A2 A1 L6)   ⊢ (B X)                              ASSERTION: (L7 L6)
L14   (A2 A1)      ⊢ (SUBSET B C)                              ANDE (L1)
L3    (A2 A1)      ⊢ (SUBSET B C)                          WEAKEN: (L14)
L4    (A2 A1)      ⊢ (FORALL [X:I] (IMPLIES (B X) (C X)))       DefsE: (L3)
L8    (A2 A1)      ⊢ (IMPLIES (B X) (C X))                FORALLE: (X) (L4)
L13   (A2 A1 L6)   ⊢ (C X)                              ASSERTION. (L8 L12)
L9    (L6 A2 A1)   ⊢ (C X)                                 WEAKEN: (L13)
L10   (A2 A1)      ⊢ (IMPLIES (A X) (C X))                       IMPI (L9)
L11   (A2 A1)      ⊢ (FORALL [X:I] (IMPLIES (A X) (C X)))  FORALLI: (X) (L10)
SIMP1 (A1 A2)      ⊢ (SUBSET A C)                             DefsI: (L11)
OMEGA: ▮
```

```
2: ∀x.(a x) ⊃ (c x)
3: (a x) ⊃ (c x)
4: a x
5: c x
6: (a x) ⊃ (b x)
7: ∀x.(a x) ⊃ (b x)
8: a ⊂ b
9: a ⊂ b
10: (a ⊂ b) ∧ (b ⊂ c)
11: a ⊂ b
12: b ⊂ c
13: (b x) ⊃ (c x)
14: ∀x.(b x) ⊃ (c x)
15: b ⊂ c
16: b ⊂ c
17: b x
18: c x
```

Figure 1: A linearized proof and its graphical tree-representation in $\mathcal{L}\Omega\mathcal{UI}$

2 Multi-modal Views: Proof Tree Visualization and Proof Content Display

The ΩMEGA system provides different techniques to analyze partial or complete proofs. As in traditional theorem proving systems, $\mathcal{L}\Omega\mathcal{UI}$ can present a proof in a linearized form, in our case as a higher-order variant of Gentzen's Natural Deduction (ND) calculus (see figure 1). For long proofs, such a presentation lacks transparency and structure. Therefore $\mathcal{L}\Omega\mathcal{UI}$ offers two additional ways of representing proofs: as a tree that models the logical dependencies between the different proof lines and as a text in natural language, as it would appear in a mathematical textbook. Before we go into details let us look at an example:

*Example 2.1 (**Proof Representations in** $\mathcal{L}\Omega\mathcal{UI}$).* The left window in figure 1 shows the linearized ND format of a simple proof of the transitivity of the subset relation, while the right window shows $\mathcal{L}\Omega\mathcal{UI}$'s main window with a tree representation of the proof. Below, we have a natural-language representation of the same proof that has been automatically generated by the ΩMEGA system.

Assumptions:
(1) $a \subset b$.
(2) $b \subset c$.
Theorem: $a \subset c$.

Proof:
Let $x \in a$. That implies that we have $x \in b$. That leads to $x \in c$. We have $a \subset c$ since $\forall x.\ x \in a \Rightarrow x \in c$. □

2.1 Hierarchical Plan Data Structure

The entire process of theorem proving in ΩMEGA can be viewed as an interleaving process of proof planning, plan execution, and verification that is centered around the so-called *Proof Plan Data Structure* ($\mathcal{PDS}$).

The hierarchical data structure represents a (partial) proof at different levels of abstraction (called *proof plans*). It is represented as a directed acyclic graph, where the nodes are justified by methods. Conceptually, each justification represents a proof plan (the *expansion* of the justification) at a lower level of abstraction that is computed when the method is expanded. A proof plan can be recursively expanded, until a fully explicit proof on the calculus level (ND) has been reached. In ΩMEGA, we keep the original proof plan in an expansion hierarchy. Thus the $\mathcal{PDS}$ makes explicit the hierarchical structure of proof plans and retains it for further applications such as proof explanation or analogical transfer of plans.

Once a proof plan is completed, its justifications can successively be expanded to verify the well-formedness of the ensuing $\mathcal{PDS}$. When the expansion process is completed, the establishment of correctness of the ND proof relies solely on the correctness of the verifier and the calculus. This approach also provides a basis for a controlled integration of external reasoning components – such as an automated theorem prover or a computer algebra system – if each reasoner's results can (on demand) be transformed into a sub-$\mathcal{PDS}$.

A $\mathcal{PDS}$ can be constructed by automated or mixed-initiative planning, or by pure user interaction. In particular, new pieces of the $\mathcal{PDS}$ can be added by directly calling tactics, by inserting facts from a data base, or by calling some external reasoner. Automated proof planning is only adequate for problem classes for which method and control knowledge have already been established.

2.2 Visualization – Proofs as Trees

In the main display window of $\mathcal{L}\Omega\mathcal{UI}$, the structure of proofs is shown in a pure tree format, independently of the logical terms associated with the nodes (see the central part in Figure 1). Since logical proofs are in general acyclic directed graphs and not trees, $\mathcal{L}\Omega\mathcal{UI}$ represents nodes with multiple predecessors (i.e. subproofs used more than once) as *co-reference* nodes: The subproof is displayed only in one place, and the other occurrences are represented as a special node – the co-reference node – that points to the root of the displayed subproof. Thus the resulting structure is a proper tree, which is displayed in such a way that node categories are expressed by color and shape (see the front panel of the window in the right part of figure 1):

Terminal nodes are represented by triangles, with assumptions, assertions, and hypotheses distinguished according to their color (green, yellow, and violet).

Intermediate nodes are represented as circles, with ground, expanded, unexpanded, and open nodes distinguished according to their color (dark blue, bright blue, yellow, and red).

Untested nodes are represented by red squares. A node is considered untested in case ΩMEGA assumes an external reasoner to be able to solve the associated sub-problem but the assumption is not yet verified.

Co-reference nodes, which may or may not be terminal nodes, are represented by diamonds and uniquely colored in orange.

The categories of intermediate nodes need some explanation. While open nodes are subject to further derivations, the other nodes are distinguished by their respective level of abstraction in the $\mathcal{PDS}$. Ground nodes are at the ND level, while all others are on higher levels of abstraction; Expanded nodes are nodes, where the expansion to the natural deduction level is known, but not displayed. The user has the following possibilities to manipulate the appearance of the proof tree:

zooming between tree overviews and enlarged tree parts,

scrolling to a desired tree part,

focusing on a subtree by cutting off the remaining tree parts,

abstracting away from details of a subtree derivation by hiding the display of that subtree, which then appears as a double-sized red triangle.

2.3 Term and Proof Content Display

The design decision to separate the tree structure from the terms associated with individual nodes enables the display of large trees without crowds of annotations. The connection between the tree structure and the associated content can be selectively re-established by the user. One possibility to achieve this is the introduction of annotations by clicking at a node, so that a yellow box enclosing a label and a justification appears besides that node (four such boxes appear in Figure 1). Another possibility is to apply the *term browser* (see the smaller window beside of the proof tree in Figure 1): by double-clicking at some node the associated term is displayed in the term browser. Nodes whose terms appear in the term browser are numbered dynamically in the displayed proof tree. Pointing to either a node or a term leads to both objects being highlighted in their respective windows. Co-references are not handled by the term browser. Instead, pointing to a co-reference node leads to the temporary appearance of a line between the co-reference node and the node it co-refers to.

2.4 Proofs in Natural Language: PROVERB

ΩMEGA uses an extension of the PROVERB system [HF97] developed in our group that presents proofs and proof plans in natural language. In order to produce coherent texts that resemble those found in mathematical textbooks, PROVERB employs state-of-the-art techniques of natural language processing and generation.

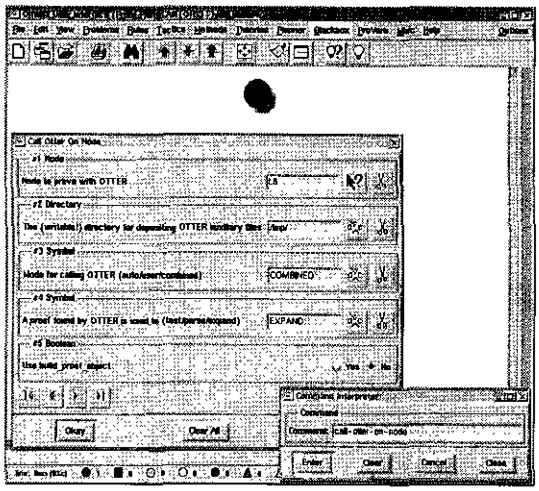
Due to the possibly hierarchical nature of $\mathcal{PDS}$ proofs, these can be verbalized at more than one level of abstraction, which can be selected by the user. Since a user will normally want to vary the level of abstraction in the course of a proof, the current verbalization facility will be extended to one that explains proofs to users guided by their feedback.

3 Controlling ΩMEGA

ΩMEGA's main functionality – especially including those commands and facilities important for interactive proof development – is available via the structured menu bar in $\mathcal{L}\Omega\mathcal{UI}$'s main window. Its entries reflect the conceptually different facilities of the ΩMEGA-system. For instance, there are a menu entities *Black-box* and *Planner* providing all useful commands of these conceptual categories to the user.

For non-experts and especially for novices, a graphical user interface has many advantages over a purely command-shell based user interface, as it provides a steady overview on the – mostly unknown – system commands to the user and thus relieves him from searching for appropriate commands in an interactive shell. Experts, who are familiar with nearly all of the commands, may prefer the interaction via a command-shell. Therefore, $\mathcal{L}\Omega\mathcal{UI}$ also provides a command shell for expert users (see the bottom-right part of the figure).

One important feature of $\mathcal{L}\Omega\mathcal{UI}$ is its dynamic and generic menu extension, i.e. $\mathcal{L}\Omega\mathcal{UI}$ selectively offers commands to the user depending on the current system state. This is in contrast to most systems which always present all commands even if some of them do not make sense within the current state. For instance, when working on a problem within a given theory, only those commands will be offered which belong to this theory (or it's parent theories) or which are defined to be always applicable. This generic approach eases the integration of new commands as they can be either integrated fully automatically by connecting them with a certain theory or by just adding the command-name to one of the non-dynamic menu entities.


File Edit View Problems Rules Tactics Methods Theories Planner Blackbox Proverb Misc Help
Options
Call Otter On Node
Okay
Clear All
Command Interpreter
Enter
Clear
Cancel
Close

#1 Node
Node to prove with OTTER: L6

#2 Directory
The (writable!) directory for depositing OTTER auxiliary files: /tmp/

#3 Symbol
Mode for calling OTTER (auto/user/combined): COMBINED

#4 Symbol
A proof found by OTTER is used to (test/parse/expand): EXPAND

#5 Boolean
Use build_proof_object: ✓ Yes ● No

Command
Command: call-otter-on-node

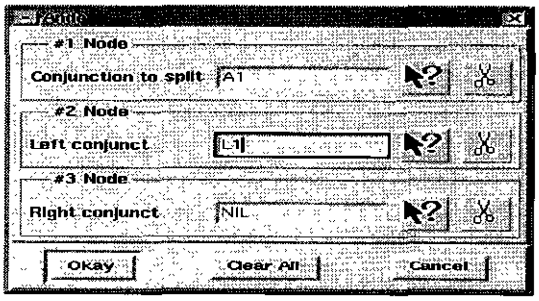
In the following subsections, we illustrate the connection of $\mathcal{L}\Omega\mathcal{UI}$ to major parts of ΩMEGA.

Theories In ΩMEGA, mathematical knowledge is structured with respect to mathematical domains and is therefore organized in a hierarchy of theories. Theories represent signature extensions, axioms, definitions, theorems, lemmata, and the basic means to construct proofs, namely rules, tactics, planning methods, and control knowledge for guiding the proof planner. Each theorem $\mathcal{T}$ has its home theory and therefore a proof of $\mathcal{T}$ can use the theory's signature extensions, axioms, definitions, and lemmata without explicitly introducing them. A simple inheritance mechanism allows the user to incrementally build larger theories.

The user can both use and manage ΩMEGA's knowledge base through $\mathcal{L}\Omega\mathcal{UI}$. In particular, it is possible to load theories or their single components incrementally and separately, browse through available assertions and import them into the active proof. Furthermore, if a problem has been proven by constructing and verifying a proof it can be stored into the theory where it was proven.

Rules, Tactics and Methods The hierarchic organization of theories and their incremental importation does not only affect their availability for a proof but also $\mathcal{L}\Omega\mathcal{UI}$'s menu structure. Since theories contain rules, tactics and planning methods, these are also incrementally loaded. To each inference method there is an attached command which is not statically contained in the interface but is dynamically appended to the menu structure. Commands for inference methods are inserted into the respective menu topic for rules, tactics, and methods. Within these topics, the commands are ordered in additional sub-menus. Since rules are always defined in ΩMEGA's base theory, they are just sorted by their type: elimination rules, introduction rules, structural rules, etc. The menus for both, methods and tactics, are divided into sub-menus according to the theories the inference methods belong to. These sub-menus can be further divided by categories specified within these theories. Moreover, each inference method can be listed in several subtopics in the menus.

Inference rules are applied by executing the attached commands. In general, it is necessary to provide some arguments for the application of a rule, which can be specified inside a generic command window. The command window adjusts itself automatically to the number of required arguments and provides some help for the requested parameters. The user can then specify the arguments either by manually entering them or by referring to certain nodes with a mouse-click.



#1 Node
Conjunction to split: A1

#2 Node
Left conjunct: L1

#3 Node
Right conjunct: NIL

In order to provide further support for interactive proof development, ΩMEGA uses a multi-layered focusing technique to compute suitable default values for rule applications [BS98]. These default values are suggested to the user as arguments in the command window.

Planner ΩMEGA's proof planner is based on an extension of the well-known STRIPS algorithm. It constructs a proof plan for a node g (the *goal node*) from a set I of *supporting nodes* (the initial state) using a set Ops of proof planning operators, called methods. The plans found by this procedure can be incorporated into the $\mathcal{PDS}$ as a separate level of abstraction. Furthermore, the proof planner also stores the reasons for its decisions for later use in proof explanation and analogy.

The ΩMEGA commands for evoking the planner and changing some settings relevant to the planner are provided by $\mathcal{L}\Omega\mathcal{UI}$ as menu items, such as applying the planner step by step, to do a certain number of planning steps, or to change the list of the proof operators (methods) considered by the planner. When the planner succeeds to find a plan, one can apply this plan to the $\mathcal{PDS}$. The graphical representation of the resulted $\mathcal{PDS}$ in $\mathcal{L}\Omega\mathcal{UI}$ shows the proof part of the $\mathcal{PDS}$ constructed by the planner.

In the near future, we intend to extend the planner so that it can be run in a reactive modus, i.e. reacting to user suggestions, such as to consider a given task next, or to take back some planner decision, and to continue with the next possible alternative. For this, the graphical representation of the $\mathcal{PDS}$ in $\mathcal{L}\Omega\mathcal{UI}$ must reflect the progress of the planner. Furthermore, the current agenda of planning goals must be displayed in parallel. This extension is facilitated by the client-server architecture (see section 4) of ΩMEGA that allows the user to enter suggestions to the planning process asynchronically with the help of appropriate Po-pup-menus.

External Systems - Automated Theorem Provers and Computer Algebra Systems ΩMEGA employs several automated theorem provers and computer algebra systems (for details cf. [KKS98]) as modules that can be applied to special-purpose proof problems. ΩMEGA uses for example OTTER [McC94], an automated theorem prover based on first order clause set resolution. We have described the integration of OTTER in [HKK+94] and the proof transformation necessary for incorporation of the result into the $\mathcal{PDS}$ in [HF96]; the methods described there also apply for the other theorem provers (SPASS, PROTEIN, and LEO) available in ΩMEGA. These systems can prove first-order theorems using various flags that control the search and the proof strategies. If for instance OTTER is called from ΩMEGA, some of these flags are set automatically, but others must be set by the user individually every time he uses OTTER.

To set these flags directly in ΩMEGA is laborious because it is necessary to know all valid values. $\mathcal{L}\Omega\mathcal{UI}$ provides an input mask that contains all flags with short descriptions of their valid values and what they will affect. Additionally, $\mathcal{L}\Omega\mathcal{UI}$ stores the last settings and offers it as a default value in the next call. $\mathcal{L}\Omega\mathcal{UI}$ controls several such "computational modules" by mapping their interface functionality into flexible input masks.

4 The Client-Server Architecture

A client-server architecture that separates ΩMEGA's logical kernel from its graphical user interface has increased its efficiency and maintainability.

In local computer networks the situation is quite common that users have relatively low-speed machines on their desktop, whereas some high-speed servers that are accessible for everyone operate in the background. Running the user interface on the local machine uses the local resources that are sufficient for this task while the more powerful servers can be exploited for the really complex task of actually proving theorems.

The maintenance advantage applies to both the user's and the developer's side. ΩMEGA is a rather large system (roughly 17 MB of COMMON LISP (CLOS) code for the main body in the current version), comprising numerous associated modules (such as the integrated automated theorem provers and a small computer algebra system) from different original sources, written in various programming languages. For the user it is a difficult task to install the complete system. In particular successful installation depends on the presence of (proprietary) compilers or interpreters for the respective programming languages.

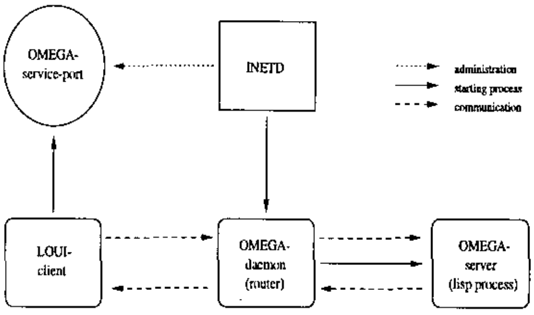
In the current client-server architecture, the user only has to install the $\mathcal{L}\Omega\mathcal{UI}$ client, which connects to the main system and exchanges data with it via the Internet. Thus the user interacts with the client, which can be physically anywhere in the world, while the ΩMEGA kernel is still on our server (here in Saarbrücken, where it is maintained and developed). Since $\mathcal{L}\Omega\mathcal{UI}$ is implemented in the OZ programming language [Saa98], which is freely available for various platforms, including UNIX and Windows95, this keeps the software and hardware requirements of the user moderate. The installation of the client is further simplified by the possibility of running $\mathcal{L}\Omega\mathcal{UI}$ as a Netscape applet, i.e. $\mathcal{L}\Omega\mathcal{UI}$ is automatically downloaded via the Internet. Thus we are able to provide current versions of ΩMEGA and $\mathcal{L}\Omega\mathcal{UI}$ without need for re-installation at the user's site.

Technical Realization $\mathcal{L}\Omega\mathcal{UI}$ is realized via a distributed programming system, called MOZART, which is an interactive distributed implementation of Oz. Mozart provides the full infrastructure to write distributed applications. Its main strength comes from its network transparency and network awareness.

Network transparency means that the semantics of Oz programs does not change if you distribute computations among different sites. For example, the programmer can use lexical scoping, logical variables, objects, etc. in distributed applications.

Network awareness means that the programmer has full control over the network operations. The language provides mobile and stationary objects, i.e. methods are executed locally (the object moves) or remotely (the message moves). The programmer has control over structure copying among sites. Structures may be copied eagerly or lazily. To reduce the bandwidth needed for communication, ΩMEGA implements an incremental approach based on SMALLTALK's MVC triad[1], which only transmits the parts of the $\mathcal{PDS}$ that are changed by a user action. This not only improves response times for low-bandwidth Internet connection but also focuses the user's attention to the effects of an action.

The Omega Client/Server Network A service called OMEGA is established on the server side, to which clients in form of $\mathcal{L}\Omega\mathcal{UI}$ applets can connect to. By addressing the port of the service an ΩMEGA daemon is started, the connection to the client is fulfilled and the main ΩMEGA LISP process comes up. This LISP process is also connected to the ΩMEGA daemon by an Internet socket. The administration and monitoring of the service's port is done by the Internet super server INETD, which listens for connections at certain Internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. Therefore, each client using the OMEGA service, has its own ΩMEGA daemon and LISP process running. As mentioned the whole communication between the client and the server process is realized via Internet sockets using strings. The above figure illustrates the client-server architecture.



Since the presentation of the proof tree is defined by a context-free grammar, it should be easy to connect $\mathcal{L}\Omega\mathcal{UI}$ to different kind of provers. In this sense $\mathcal{L}\Omega\mathcal{UI}$ can be seen as a generic proof viewer.

Distributing ΩMEGA Up to this point, we have considered a client-server network with one server that is dedicated to ΩMEGA itself and several clients that use this server. In reality, a ΩMEGA network may consist of several servers that can be accessed via a gateway service. The gateway daemon runs on one machine that provides the ΩMEGA service. It can start the actual ΩMEGA process and its the associated modules on any of the servers, depending on their current work load. In this way, we are able to employ the whole computational power of a local area network with a background of several larger servers.

5 Related Work

User interfaces for theorem provers are credited with increasing importance in the field. These interfaces comprise graphical illustrations of proof structures and their elements, and facilities to set up commands in the proof environment.

Some special modes of proof types express part of the semantics of proof steps by graphical objects and annotations. Examples of this sort of visualization are binary decision diagrams for first-order deduction systems [PS95], which have special display facilities for the relation between quantified formulae and their instantiation, and natural deduction displays of sequent proofs [Bor97] where the scoping structure of the proof is visualized by adjacent and by nested boxes enclosing segments of proof lines. Another presentation technique displays proof steps in an appropriately formatted and interactive way. [BJK$^+$97] is able to present a proof in natural language, to a certain level of detail with deeper levels indented. In addition, levels of detail temporarily hidden can be exposed by clicking on the corresponding root proof line. A rather elaborate presentation system is CTCOQ [BKT94] which distributes the information about a proof over three sections of a multi-paned window: a *Command* window records the script of commands sent to the proof engine, a *State* window contains the current goals to be proved, and a *Theorems* window contains the results of queries into the proof engine's theorem database. Some other approaches put particular

[1] See for instance `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html` for an overview.

emphasis on visualization by making the tree format of proof structures explicit in the display. The user interface for the SEAMLESS system [EM97] provides display facilities for a proof graph at different levels of abstraction in a framed window: a variety of lay-out operations including zooming and reuse of multiple appearances of lemmas. The user interface of INKA [HS96] allows for the display of induction proof sketches at varying levels of detail. Its features include status information, typically expressed by different coloring, and context-sensitive menus of possible user actions.

In comparison to these systems, ΩMEGA in some sense combines features of SEAMLESS and CTCOQ. Its graphical display is similar to that of SEAMLESS, but the set of node categories and their display is fixed to the particular proof environment. However, $\mathcal{L}\Omega\mathcal{UI}$'s tree visualization can easily be adapted to a different set of node categories and display options. Its status information display is similar to that of CtCoq, but the database window is handled differently. Apart from that, the strict separation of visualizing the proof tree structure and browsing the terms associated with individual nodes selectively, handling of co-references, and the client-server architecture are unique features in OMEGA.

References

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In William McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*, number 1249 in LNAI, pages 252–255, Townsville, Australia, 1997. Springer Verlag.

[BJK+97] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. An Overview of the Theorema Project. In *ISSAC'97*, Hawaii, 1997.

[BKT94] Y. Bertot, G. Kahn, and L. Therry. Proof by pointing. *Theoretical Aspects of Computer Software*, 789:141–160, 1994.

[Bor97] R. Bornat. Natural Deduction Displays of Sequent Proofs: Experience with the Jape Calculator. In *First International Workshop on Proof Transformation and Presentation*, Dagstuhl Castle, 1997.

[BS98] Christoph Benzmüller and Volker Sorge. A Focusing Technique for Guiding Interactive Proofs. Submitted to the 8th International Conference on Artificial Intelligence: Methodology, Systems, Applications, 1998.

[EM97] J. Eusterbrock and N. Michalis. A World-Wide Web Interface for the Visualization of Constructive Proofs at Different Abstraction Layers. In *First International Workshop on Proof Transformation and Presentation*, Dagstuhl Castle, 1997.

[HF96] Xiaorong Huang and Armin Fiedler. Presenting machine-found proofs. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th Conference on Automated Deduction*, number 1104 in LNAI, pages 221–225, New Brunswick, NJ, USA, 1996. Springer Verlag.

[HF97] Xiaorong Huang and Armin Fiedler. Proof verbalization as an application of NLG. In Martha E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)*, Nagoya, Japan, 1997. Morgan Kaufmann.

[HKK+94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Daniel Nesmith, Jörn Richts, and Jörg Siekmann. Ω-MKRP a proof development environment. In Alan Bundy, editor, *Proceedings of the 12th Conference on Automated Deduction*, number 814 in LNAI, pages 788–792, Nancy, France, 1994. Springer Verlag.

[HS96] D. Hutter and C. Sengler. A Graphical User Interface for an Inductive Theorem Prover. In *International Workshop on User Interface Design for Theorem Proving Systems*, 1996.

[KKS98] Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, 1998. Special Issue on the Integration of Computer Algebra and Automated Deduction; forthcoming.

[McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

[PS95] J. Posegga and K. Schneider. Interactive First-Order Deduction with BDDs. In *International Workshop on User Interface Design for Theorem Proving Systems*, Glasgow, 1995.

[Saa98] Programming Systems Lab Saarbrücken, 1998. The Oz Webpage: `http://www.ps.uni-sb.de/ns3/oz/`.

User Interfaces for Generic Proof Assistants
Part II: Displaying Proofs*

Bernard Sufrin [†]   Richard Bornat [‡]

June 1998

Abstract

JAPE is a generic proof editor which (amongst other things) offers the teacher, user, or logic designer the opportunity of constructing a direct manipulation interface for proofs.

In the first part of this paper we introduced JAPE, illustrated some aspects of proof development by direct manipulation, and described some of the infrastructure provided for interpreting the gestures which users make at proofs. In this part we discuss the problem of displaying proofs at appropriate levels of abstraction, and describe the infrastructure JAPE provides for doing this. We also show how specialised modes of display for proofs which use *cut* and *identity* rules can be exploited to give the illusion of forward (natural deduction style) proof.

1 Introduction

In the first part of this paper we gave an account of the interaction tactics which are used by JAPE to interpret users' gestures in context in order to invoke proof tactics and rules. We also explained how such tactics provide support for interactive proof activity at level of abstraction closer to provers' intuitions than that of the basic inference rules of a logic might be.

Just as it is important to be able to define tactic programs which implement intuitive proof moves, so it is important to be able to *show* proofs at an appropriate level of abstraction: for otherwise the human engaged in a proof has to abstract the essence of a proof situation from a display which may well present too much detail.

We start this part of the paper by describing the mechanisms which JAPE currently provides for displaying proofs at a level of abstraction chosen by the interaction designer. Then we describe JAPE's linear presentation style for proofs in which *identity* and *cut* rules have been used – and show how the use of a simple display transformation coupled with appropriate interaction tactics can be used to give the illusion of forward proof.

2 Displaying Equational Proofs

Logical presentation masks equational essence

Consider the following proof state, reached during a proof that list-reversal is self-inverting:

[*] This is the second part of a paper whose first part[1] was presented at the workshop "User Interfaces for Theorem Provers", York, U.K., July 1996.

[†] Computing Laboratory and Worcester College, Oxford

[‡] Queen Mary and Westfield College, London

$$\dfrac{\dfrac{}{id\,x = x}\,id \qquad \dfrac{\dfrac{}{(rev \bullet rev)x = rev(rev\,x)}\,\bullet \quad \dfrac{}{rev(rev\,x) = x}}{(rev \bullet rev)x = x}\,rewrite}{\dfrac{(rev \bullet rev)x = id\,x}{rev \bullet rev = id}\,ext}\,rewrite$$

At an intuitive level we might describe the history of this proof as follows: "The extensionality rule is followed by unfolding the term *id x* into *x*, and this is followed by unfolding the term *(rev • rev) x* into *rev(rev x)*."

But the display shows so much detail that it can be hard for someone who didn't do the proof (or who doesn't care that an unfolding proof step happens to be implemented by an application of the *rewrite* rule) to follow what went on. What's needed is a way of suppressing the irrelevant logical detail from the proof display whilst preserving its equational essence.

One way of doing so would be to augment the basic rules (and definitions) of the the logic in question with specialised derived rules. For example, from the rule `id X = X` which defines *id*, and the *rewrite* rule

```
RULE  rewrite (X, Y, ABSTRACTION P)
FROM  X=Y
AND   P(Y)
INFER P(X)
```

we can derive the rule[1]

```
RULE  "Fold id"(X, Y, ABSTRACTION P)
FROM  P(X)
INFER P(id X)
```

Using this rule, and the corresponding rule for compositions, the proof would become.

$$\dfrac{\dfrac{\dfrac{rev(rev\,x) = x}{(rev \circ rev)x = x}\,\textit{Fold}\circ}{(rev \circ rev)x = id\,x}\,\textit{Fold id}}{rev \circ rev = id}\,ext$$

But this isn't really a practical proposition: in any interesting logic there will simply be too many derived rules for us to install before we can start work on the conjectures which really interest us. What is needed is a way of transforming the *presentation* of the *logical* proof state into a presentation in the equational style.

Subtree Selection – a partial solution

In order to explain the partial solution which JAPE currently offers we shall first have to explain the machinery for displaying proofs. In effect there's a two-stage pipeline.

$$\begin{array}{c}\text{Logical}\\\text{Proof}\end{array} \longrightarrow \boxed{\text{Proof Abstraction}} \longrightarrow \begin{array}{c}\text{Abstracted}\\\text{Proof}\end{array} \longrightarrow \boxed{\text{Proof Rendering}} \longrightarrow \begin{array}{c}\text{Displayed}\\\text{Proof}\end{array}$$

The rendering stage depends on the proof display style currently in use, and constructs a proof layout from the abstracted proof. The details of this stage of the transformation will become important when we come to discuss automatic elision of redundant formulae, but they're not important here.

The abstraction stage is guided by abstraction descriptions which are installed in the logical proof tree as it is being constructed. This is done by tactics of the form

[1] Why is the rule called `Fold id`? Because when we read the proof *forward*, the transformation is from *x* to *id x*. Such a right-to-left use of a defining equation is generally referred to as a folding transformation.

(LAYOUT *reason branch–list* $tactic_1$ $tactic_2$ ...)

A tactic of this form is executed by running its component *tactic*s in sequence. If this succeeds, then the proof subtree which it generates is marked to be displayed as an abstraction of the logical proof tree, explained with (a text derived from) the given *reason*. The only form of abstraction currently provided is *subtree selection*, and the *branch–list* is a (possibly empty) list of the numbers of the branches of the generated subtree which are to be displayed.

For example, if the tactic form

```
(LAYOUT "Fold id" (1) (rewrite (id x)) id)
```

is executed in the proof state

$$\frac{(rev \circ rev)x = id\,x}{rev \circ rev = id}\,ext$$

it transforms the logical proof into

$$\dfrac{\dfrac{\dfrac{}{id\,x = x}\,id \qquad \dfrac{}{(rev \circ rev)x = x}}{(rev \circ rev)x = id\,x}\,rewrite}{rev \circ rev = id}\,ext$$

but the proof abstraction stage of the display process ensures that only the second branch (numbered 1) of the new proof subtree is selected for rendering, as shown below.

$$\dfrac{\dfrac{(rev \circ rev)x = x}{(rev \circ rev)x = id\,x}\,Fold\ id}{rev \circ rev = id}\,ext$$

It makes sense to generalise this tactic over the rule (or tactic) used to close the left branch of the *rewrite* tree, so we define:

```
TACTIC Unfold(term, rule) (LAYOUT "Fold %s" (1) (rewrite term) rule)
```

When an application of this tactic is succesful, the reason annotation of the displayed proof node is constructed by expanding `"Fold %s"` in the layout tactic – replacing the `%s` by the name of the rule used at the base of the branch which is suppressed. Although this simplistic way of constructing the annotation is nearly always good enough, we shall see later that there are circumstances where it can be uninformative or misleading.

In part 1 we developed a tactic, `UnfoldSelectionOrSearch`, to support the construction of "fold" moves in equational proofs. The production version of that tactic – presented below – reports proof steps at exactly the right level of abstraction for someone who is conducting an equational proof.

```
TACTIC UnfoldSelectionOrSearch(rulebase) IS
(WHEN
  (LETHYP (_L=_R)
          (LETSUBSTSEL _TERM
                (LAYOUT "Fold with hyp" (1) (WITHSUBSTSEL rewrite) (hyp (_L=_R)))))
  (LETSUBSTSEL _TERM
        (LAYOUT "Fold %s" (1) (WITHSUBSTSEL rewrite) rulebase))
  (LAYOUT "Fold %s" (1) (UNFOLD rewrite rulebase)))
```

The last stage of development of the tactic was very simple: the three underlined sequential tactic forms of the original which are underlined below were simply embedded in LAYOUT forms which select their second branch.

Shallow and weak though it is, the form of tree abstraction provided by LAYOUT has been sufficiently powerful to support coherent proof presentations in several theories based on equational reasoning. These range from a simple untyped theory of functional programming, to a reasoning system for typed category theory. In the latter theory, the typing antecedents of rules are decided automatically by tactic, but the proofs of well-typing are suppressed (unless they fail) in the presentation.

Universally Applied Tree Transformations

In one experimental JAPE implementation, the annotations which guide proof abstraction were described declaratively in a tree-transformation notation. For example, the following transformation could be used to present an unfolding – an application of rewrite whose left subgoal is closed directly by a rule.

```
TRANSFORM Unfolding (rulename, proof)
  Γ ⊢ Q BY  rewrite
  FROM
    Γ ⊢ X=Y BY rulename.
  AND
    Γ ⊢ P FROM proof.
  END
INTO
  Γ ⊢ Q BY (Fold rulename)
  FROM
    Γ ⊢ P FROM proof
  END
```

The following transformation could be used to present a more deeply-nested proof – a fold, in fact.

```
TRANSFORM Folding (rulename, proof)
  Γ ⊢ Q BY rewrite
  FROM
    Γ ⊢ X=Y BY "= Symmetric"
    FROM
      Γ ⊢ Y=X BY rulename.
    AND
      Γ ⊢ P FROM proof.
  END
INTO
  Γ ⊢ Q BY (Unfold rulename)
  FROM
    Γ ⊢ P FROM proof
  END
```

The declarative style seemed, for a while, to be inappropriate: partly because the notation is rather long-winded, but mainly because transformations were applied without discrimination to all proof trees of the right form – even if the tree in question was constructed by applying primitive rules by hand. The resulting display could misleadingly suggest that high-level proof steps like "unfold" have been employed when they hadn't.

Selectively Applied Tree Transformations

It took us a while to realize that the *selective* application of declarative transformations could help us solve a class of awkward problems which arise from the simpleminded way in which LAYOUT forms construct their explanations.

Consider programming a *Fold* tactic – designed to replace an occurence of the right hand side of a definition by the corresponding left hand side. This tactic is much the same as the *Unfold* described earlier, but we will use the rule `"= symmetric"` (defined as `FROM X=Y INFER Y=X`) before applying the definitional rule.

```
TACTIC Fold(term, rule)
       (LAYOUT "Unfold %s" (1) (rewrite term) "= symmetric" rule)
```

The tactic does the right job, but unfortunately the "%s" in the reason component of this tactic picks up the name of the "= symmetric" rule – and explains the move as Unfold "= symmetric" when what we really want is the name of the rule which was actually used.

In situations like this the LAYOUT form is just inadequate. The problem can be remedied by introducing the TRANSFORM tactic form, which is analogous to LAYOUT save that the tree abstraction it performs is described by a declaratively described transformation of the kind discussed above. The Fold tactic is now written as follows[2]

```
TACTIC Fold(term, rule)
       (TRANSFORM Folding (rewrite term) "= symmetric" rule)
```

3 Natural Deduction Displays of Sequent Proofs

As we demonstrated in the previous section, there are certain proofs whose presentations can be improved if the interaction designer takes the trouble to program interface tactics in a certain way. In this section we show how JAPE can *automatically* exploit the presence of certain kinds of structural rule in a logic to eliminate uninformative or redundant material from proof displays. One consequence of the display transformations we describe in this section is that it is easy to present a convincing simulacrum of forward proof in the natural deduction-style – despite the fact that JAPE is a backward proof system based on sequents!

The simplest automatically applicable display transformation is the removal of uninformative identity rules. We can best explain it by giving a concrete example.

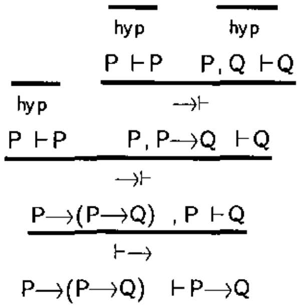


Figure 1: A proof presented in the tree style

The proof tree in figure 1 built using the following rules is shown rendered in the (untransformed) linear style in figure 2.

```
RULE    "⊢→"    FROM Γ, A ⊢ B INFER Γ ⊢A→B
RULE    "→⊢"    FROM Γ, A→B ⊢ A AND Γ, B ⊢ C INFER Γ, A→B ⊢ C
RULE    hyp(A)  INFER Γ, A ⊢ A
```

In this style the proof of each sequent $\gamma_1, ...\gamma_n \vdash C$ is shown in a box, with the hypotheses at the top and the conclusion at the bottom, and the proofs of any antecedent sub-sequents recursively shown between them (ellipsis represents an unclosed subtree). If any of the hypotheses have already appeared at the top of an enclosing box, then they are omitted, and if this results in a box with no hypotheses at the top, then the conclusion is presented unboxed.[3]

[2] "Folding" is the proof transform we described on the previous page.

[3] We note, in passing, that this presentation style is ambiguous for some logics: just because a hypothesis appears at the head of a box, that doesn't mean that it's in scope at the conclusion of all the boxes nested within. For example, the hypothesis $P\rightarrow(P\rightarrow Q)$ is not in scope at line 8 of the proof displayed above, because that line comes from the proof of the sequent $P, P\rightarrow Q \vdash Q$. In JAPE we resolve the ambiguity dynamically: when a particular conclusion is selected we "grey out" the hypotheses which aren't in scope at that conclusuion. Dually, when a particular hypothesis is selected we grey out those parts of the proof at which it isn't in scope.

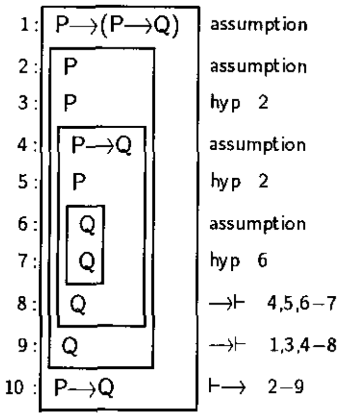


Figure 2: The same proof presented in the linear style

Notice that line 8 is justified in part by the *P* established on line 5 by *hyp* from the *identical* assumption on line 2. Likewise line 8 is justified in part by the *P* established on line 3 by *hyp* from the assumption on line 2. Finally notice that line 7 simply duplicates the assumption *Q* made on line 6. We say that *hyp* is an *identity* rule because its conclusion is identical to one of its assumptions. It seems fair to say that the use of *hyp* at lines 3, 5, and 7 do not really add to our understanding of the proof, and that these lines could therefore be eliminated. Figure 3 shows what happens when this is done.

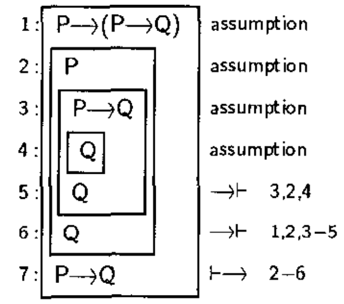


Figure 3: The same proof with *hyp* elided

Removing Cuts

A more interesting automatic display transformation is the elimination of *cut* rules: these take the form

FROM $\Gamma \vdash \mu$ AND $\Gamma, \mu \vdash C$ INFER $\Gamma \vdash C$

It is used to "cut" the task of finding a proof of $\Gamma \vdash C$ into two parts: that of establishing an intermediate formula μ from the hypotheses Γ, and that of establishing the conclusion C from the hypotheses augmented with the intermediate formula. In figure 4 we show a *Cut* proof, and its standard linear display. Notice that the intermediate formula μ appears once at the foot of its own proof, and again – as the new hypothesis opening the subproof which establishes C. The cut display transformation can sometimes replace the vertically adjacent (but structurally separate) occurences of μ with a single occurence, as shown in figure 5.

The following concrete example demonstrates the *cut* transformation in action during a proof of $p \rightarrow (p \rightarrow q) \vdash p \rightarrow q$ in a natural-deduction style logic, JnJ[3], which includes the following rules:

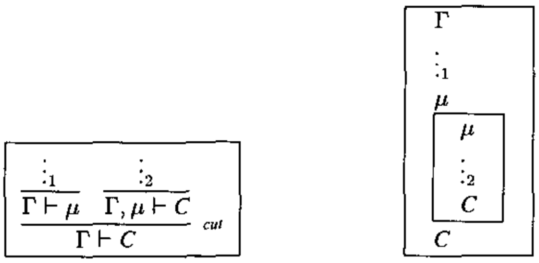


Figure 4: A *cut* proof and its standard linear display

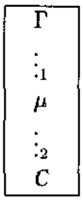
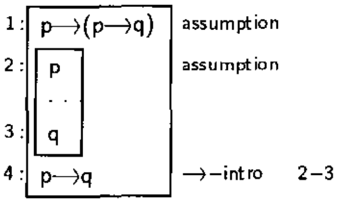


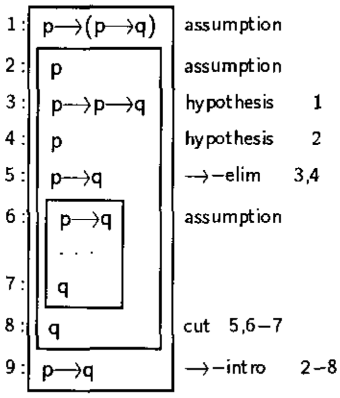
Figure 5: Linear display of a *cut* proof after transformation

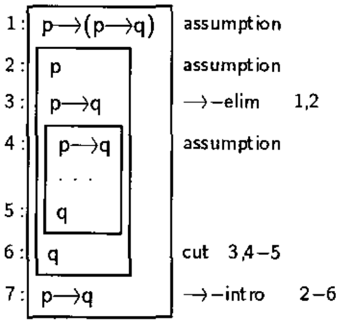
```
RULE  "→-intro"  FROM Γ, p ⊢ q INFER Γ ⊢ p→q
RULE  "→-elim"   FROM Γ ⊢ p→q AND Γ ⊢ p INFER Γ ⊢ q
RULE  hypothesis(p) INFER Γ, p ⊢ p
```

The intuitive approach to this proof for someone who has learned natural deduction is to use "→-intro" to establish the proof state



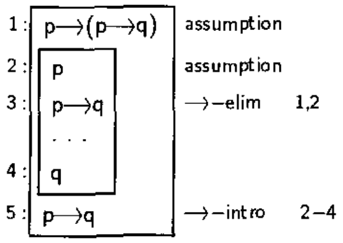
then to use implication elimination to establish *p→q*. In a backward proof engine such as ours this has to be done by *cut*ting the proof first, then applying implication elimation. The resulting state is shown in complete detail in the diagram below on the left, and with applications of the *hypothesis* rule suppressed on the right.





Whilst these details may be illuminating to a specialist, someone who's simply trying to do a natural deduction proof would almost certainly be overwhelmed by the technicalities of the left-hand display, and would find the right-hand display uninformative.

Fortunately, application of the identity and cut-suppression transformations reveals just how simple the (natural deduction) proof situation is[4]



```
1: p→(p→q)   assumption
2:   p        assumption
3:   p→q      →-elim    1,2
     ...
4:   q
5: p→q        →-intro   2-4
```

Now novices shouldn't have to know that forward application of elimination rules requires the use of *cut*! How do we arrange that simply invoking the implication elimination rule ends up doing the right thing? We do so by defining an *interaction tactic* [1] which assumes that when both a hypothesis and a conclusion are selected, rules are to be applied in the forward direction, but when only a conclusion is selected, rules are to be applied backwards.

The forward application of a proof rule *R*, from a hypothesis γ is implemented by the sequential composition of *cut*, *R*, and *hypothesis*(γ). This is easily arranged: an approximation to the interaction tactic used for this purpose in the presentation[4] of the logic is shown below

```
TACTIC ElimRule(rule)
(WHEN
  (LETHYP _p
    (ALT (SEQ  cut (WITHARGSEL rule) (WITHHYPSEL hypothesis))
         (FAIL ("%s is not applicable to %s", rule, _p))))
  (FAIL ("Select an assumption before applying %s", rule)))
```

The sequence of proof states which this evokes during the forward application of `"→-elim"` by `(ElimRule "→-elim")` described above is shown in Figures 8 through 10. Before the user sees the proof again, the automatic invocation of the *hypothesis* rule specified by the presentation designer will have disposed of the outstanding proof obligation in the left-hand subproof of figure 10.

One difficulty is the problem of deciding what is meant when the middle formula (μ) is selected in a *cut*-transformed display such as the one in figure 5. Did the user intend to select the μ *conclusion* (in the left antecedent of *cut*) or the μ *hypothesis* (in the right antecedent). A simpleminded approach to the problem might be as follows: if there's a selection below it in the display, then it is the hypothesis; if there is no other selection, or if the other selection is above it in the display, then it is the conclusion. Unfortunately this approach would make it impossible to interpret certain kinds of lone selection made in the presence of the *cut* transformation, and this would dramatically complicate the programming of interaction tactics. In fact we restrict the application of the transformation to situations in which one or both of the antecedents of the *cut* rule have already been proven. If μ is selected when the left antecedent is proven and the right antecedent is open, then it is the hypothesis for the right antecedent. If it is selected when the right antecedent is proven and the left antecedent is open, then it is the conclusion. The details are explained at length in [2].

4 Conclusion and Prospects

Although JAPE is a generic tool, we believe that the techniques we have described here could easily be adapted for use in provers designed for specific logics, and would be of considerable benefit to users. Without the complexities induced by JAPE's generic nature they might also be much easier to implement!

[4] JAPE automatically applies the display transformations described above only when declarations are made which inform it of the existence (and names) of the structural rules in question.

We have just started work on extending the *cut*, and *hyp* transformations to rules of transitivity and reflexivity. Resolving the meaning of selections in the presence of these extended transformations imposes its own difficulties, but we are beguiled by the prospect of presenting an transitivity proof which would normally take the form shown in Figure 6 without the "transitivity noise", and without the explicit movement of formulae from the right hand side of one line to the left hand side of the next.

1	$\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x + x \rightarrow * (\lambda x \bullet x + x)(1 + 2)$	Let
2	$(\lambda x \bullet x + x)(1 + 2) \rightarrow * (\lambda x \bullet x + x)3$	Rand, Add'3
3	$(\lambda x \bullet x + x)3 \rightarrow * 3 + 3$	Beta
4	$3 + 3 \rightarrow * 6$	Add'10
	...	
5	$6 \rightarrow * \_T$	
6	$3 + 3 \rightarrow * \_T$	Transitive 4, 5
7	$(\lambda x \bullet x + x)3 \rightarrow * \_T$	Transitive 3, 6
8	$(\lambda x \bullet x + x)(1 + 2) \rightarrow * \_T$	Transitive 2, 7
9	$\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x + x \rightarrow * \_T$	Transitive 1, 8

Figure 6: A "noisy" transitivity proof

1	$\mathsf{let}\ x = 1 + 2\ \mathsf{in}\ x + x \overset{*}{\rightarrow}$	Let
2	$(\lambda x \bullet x + x)(1 + 2) \overset{*}{\rightarrow}$	Rand, Add'3
3	$(\lambda x \bullet x + x)3 \overset{*}{\rightarrow}$	Beta
4	$3 + 3 \overset{*}{\rightarrow}$	Add'10
	...	
5	6	

Figure 7: A transformed transitivity proof

References

[1] Richard Bornat & Bernard Sufrin. *User Interfaces for Generic Proof Assistants: Part I.* Presented at UITP-96, York, England
`http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/JAPE/PAPERS/UITP96-paper.ps.gz.`

[2] Richard Bornat & Bernard Sufrin. *Displaying sequent-calculus proofs in natural-deduction style: experience with the Jape proof calculator.*
Presented at PTP-97, Schloss Dagstuhl, Germany
`http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/JAPE/PAPERS/PTP97-paper.ps.gz.`

[3] Jim Woodcock and Jim Davies. *Using Z.* Prentice-Hall International.

[4] Bernard Sufrin & Richard Bornat. *JnJ in Jape.*
`http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/JAPE/PAPERS/jnj.ps.gz.`

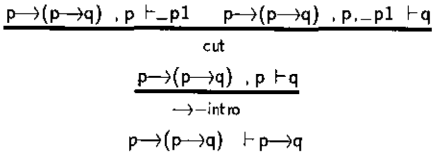


Figure 8: Proof state after `cut`

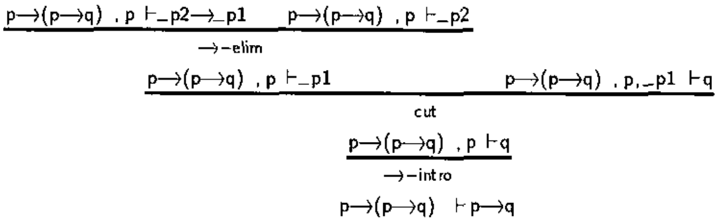


Figure 9: Proof state after `(WITHARGSEL →-elim)`

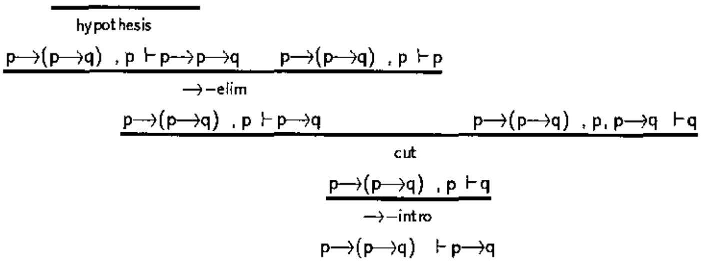


Figure 10: Proof state after `(WITHHYPSEL (p→(p→q)))`

Proving as Editing HOL Tactics

Koichi Takahashi* and Masami Hagiya†

Abstract

We introduce an Emacs interface for writing HOL proof scripts in SML based on the Computing-as-Editing paradigm. Tactics in a proof script are considered as constraints, and the process of interactive theorem proving becomes that of solving constraints. In addition, constraint solving is subsumed by the process of editing a proof script. Tactics are executed while the script is being edited. The user does not have to pay attention to the status of the HOL prover. In our interface, the user can also enjoy proof-by-pointing. The result of proof-by-pointing is inserted as a tactic into a proof script. We expect that our interface will be widely used as an extension of the familiar HOL mode on Emacs.

1 Introduction

With a tactic-based proof assistant such as HOL [4] or Isabelle [7], the user (i.e., the writer of a proof script) interactively inputs tactics to decompose the top goal to subgoals. In this process of interactive theorem proving, the current goal of the proof assistant is always printed out when the user inputs a tactic (Figure 1). The history of interaction therefore becomes very long, but it is difficult to understand the proof with only the sequence of input tactics (Figure 2).

After finishing the entire proof of the top goal, many users of a tactic-based proof assistant then compose a large structured tactic from the sequence of interactively input tactics using tacticals such as THEN and THENL in HOL, which represent control structures. Such a structured tactic can prove the top goal at one time. Some users also try to write a structured tactic as they prove the top goal. While editing a structured tactic, they successively send fragments of the tactic to the proof assistant by hand. Of course, they have to keep track of the relationship between the edited tactic and the status of the proof assistant. The reason to write structured tactics is to gain reusability of proof scripts, but the readability of a structured tactic is relatively low because no subgoals are shown.

```
val it =
  ``c + SUC x = SUC x``
  ____________________________
      ``c = 0``
      ``c + x = x``
  : goalstack
```

Figure 1: A printed current goal

In this paper, we introduce an interface for writing HOL proof scripts in SML based on the Computing-as-Editing paradigm (CAEP) [5]. In this approach, tactics in a proof script are considered as constraints, and the process of interactive theorem proving becomes that of solving constraints. In addition, constraint solving is subsumed by the process of editing a proof script. Tactics are executed while a proof script is being edited.

In contrast to other interfaces based on the CAEP (such as that for computer algebra [5]), the interface in this paper does not require a new format for expressing constraints. Users can directly edit HOL tactics written in the syntax of SML. A proof script in SML is regarded as a constrained document, where each tactic in the document is solved as a constraint by the HOL prover.

With our interface, the user can write a proof script while interactively executing tactics. When a command called check is invoked under the text editor, the tactic just before the text cursor is locally executed and confirmed by the HOL prover. According to the result of this execution, the text of the proof

*Electrotechnical Laboratory. takahasi@etl.go.jp

†Department of Information Science, University of Tokyo. hagiya@is.s.u-tokyo.ac.jp

```
g `(c=0) ==> !x . c + x = x`;
e UNDISCH_TAC;
e INDUCT_TAC;
e (ASM_REWRITE_TAC [theorem "arithmetic" "ADD_0"]);
e (REWRITE_TAC [GSYM (theorem "arithmetic" "ADD_SUC")]);
e (POP_ASSUM (fn th => REWRITE_TAC [th]));
save_thm("a_thm", top_thm());
```

Figure 2: A sequence of tactics

```
val a_thm = store_thm("a_thm",
  ``(c=0) ==> !x . c + x = x``,
  DISCH_TAC THEN
  INDUCT_TAC THENL [
    ASM_REWRITE_TAC [theorem "arithmetic" "ADD_0"]
  ,
    REWRITE_TAC [GSYM (theorem "arithmetic" "ADD_SUC")] THEN
    POP_ASSUM (fn th => REWRITE_TAC [th])
  ]);
```

Figure 3: A structural tactic

script is changed; when the tactic has produced subgoals, an appropriate template for solving the subgoals is inserted.

Using our interface, the user can avoid the bugs that arise when he or she composes a structured tactic from a successful sequence of tactics. In addition, the user can write a structured tactic in a very flexible manner. No specific order is imposed on which subgoal to solve first, and the user can switch the current subgoal simply by moving the text cursor. Since a tactic at the position of the text cursor is executed and confirmed locally, the user can check the slight change of a tactic immediately after he or she has modified the text of the tactic. This greatly helps the user to reuse existing proof scripts.

In our interface, to make local execution of a tactic efficient, subgoals (i.e., intermediate goals) can be explicitly embedded in a proof script, which also improves the readability of the proof script. It is not necessary to write such subgoals by hand, as they are usually inserted as a result of the local execution of a tactic.

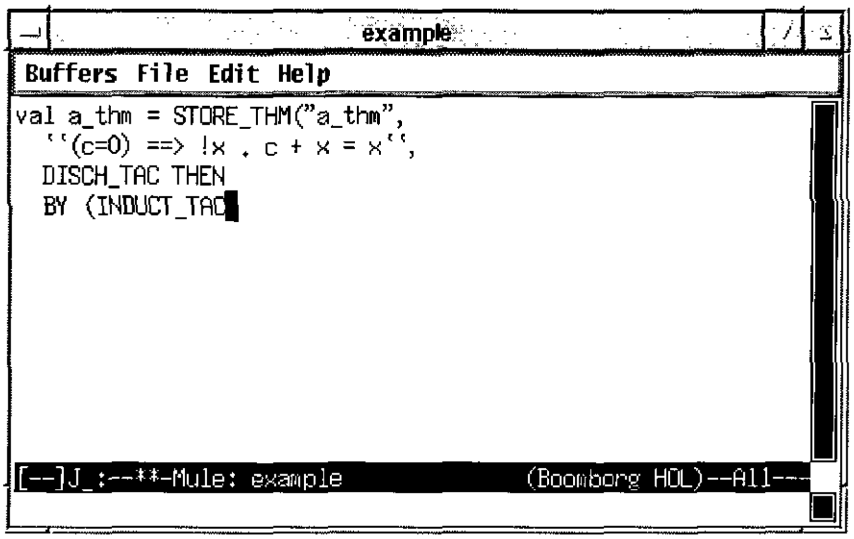
TkHol [8] is a graphical user interface for HOL. By using TkHol, the user automatically gets a structured proof script from a sequence of input tactics, but its design does not take into account the reusability and readability of proof scripts. We think that directly editing tactics under Emacs is more flexible, and allows for reusing existing tactics.

In our interface, the user can also employ proof-by-pointing [1]. At any position in a proof script, the user can pop up the current goal window (Figure 7), and point out a subterm in the current goal using a mouse. A tactic called a proof-by-pointing tactic is then generated and inserted into the proof script (Figure 8). Since the result of proof-by-pointing is expressed in the form of text, it is possible to undo, reuse or modify the result by deleting, copying or editing the corresponding text. In addition, the user can replay proof-by-pointing by specifying the proof-by-pointing tactic in a proof script.

In the next section, we describe our interface, called `Boomborg-HOL`, in detail. In Section 3, we explain how proof-by-pointing is performed in our interface. Section 4 is about the implementation of our interface. In Section 5, we discuss how the problems of graphical user interfaces pointed out by Merriam and Harrison [6] are solved in our interface. Section 6 summarizes the merits and demerits of our interface that we found through our experience in using it. In the last section, we give plans for future work. In the Appendix, we list some additional example scripts written with our interface.

2 Boomborg-HOL

In our interface (`Boomborg-HOL`) two functions, `CLAIM` and `BY`, are prepared to make subgoals (i.e. intermediate goals) in a proof script explicit. They are tacticals since they take a tactic as an argument. They



```
val a_thm = STORE_THM("a_thm",
  ‘‘(c=0) ==> !x . c + x = x‘‘,
  DISCH_TAC THEN
  BY (INDUCT_TAC
```

Figure 4: Before the check command

are used as follows.

CLAIM “assumption part” “conclusion part”
“tactic”

BY “tactic”

The assumption part and the conclusion part of a CLAIM specify the assumption and the conclusion of a goal as compared with the goal at its outer surrounding CLAIM or BY (or STORE_THM, a variant of store_thm in HOL). The assumption part is a list whose elements are either of the form

assume A

or of the form

forget A.

The form assume A means that the assumption A is added to the goal at the outer CLAIM or BY. The form forget A means that the assumption A is deleted from the goal at the outer CLAIM or BY. The conclusion part is either of the form

holds C

or of the form

thesis.

The former simply specifies C as the conclusion of the goal at this CLAIM. The latter means that the conclusion of the goal at the outer CLAIM or BY has not been changed.

BYs are used for specifying goals at inner CLAIMs (refer to a later example).

In our interface, the user edits a proof script that may contain CLAIMs and BYs in an Emacs buffer. While editing the script, the user can use the following command.

- check

The check command executes a tactic just before the text cursor and inserts each subgoal produced by the tactic in the form of a CLAIM.

After the check command is invoked in Figure 4, a CLAIM is inserted as in Figure 5. The check command first calculates the goal at the outer CLAIM or BY, and sends it to the HOL prover with the tactic between the text cursor and the outer CLAIM or BY.

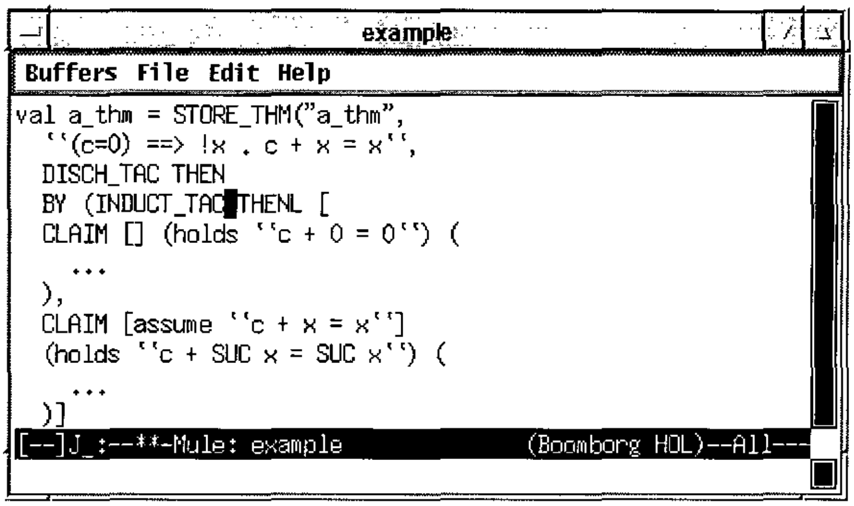
In Figure 5, the CLAIM

```
CLAIM [] (holds ‘‘c + 0 = c‘‘) (
```

means that there is no change in the assumption part of its goal as compared with the goal at the outer BY, while the conclusion is specified as ‘‘c + 0 = 0‘‘. Notice that the goal at the outer BY has the assumption ‘‘c = 0‘‘. This is not explicitly specified by the CLAIM. Using BYs, the user can avoid specifying evident changes of goals. If there was no BY, the CLAIMs in Figure 5 would be

```
CLAIM [assume ‘‘c = 0‘‘]
(holds ‘‘c + 0 = c‘‘) (
```

and



```
val a_thm = STORE_THM("a_thm",
  ``(c=0) ==> !x . c + x = x'',
  DISCH_TAC THEN
  BY (INDUCT_TAC THENL [
  CLAIM [] (holds ``c + 0 = 0'') (
    ...
  ),
  CLAIM [assume ``c + x = x'']
  (holds ``c + SUC x = SUC x'') (
    ...
  )]
[--]J_:--**-Mule: example          (Boomborg HOL)--All---
```

Figure 5: After the check command

```
CLAIM [assume ``c + x = x'',
       assume ``c = 0'']
(holds ``c + SUC x = SUC x'') (
```

In Figure 6, the check command shows the message OK at the message line of the editor because no subgoals remain. As in this example, the check command can be executed at any position in a proof script, even if the script is incomplete.

In our interface, the user does not directly interact with the HOL prover. The user just edits a proof script, and sends a fragment of the script to the HOL prover from time to time by invoking the check command. This process can be considered as constraint solving in the sense of the CAEP. The user does not have to pay attention to the status of the HOL prover.

Specification of subgoals by CLAIMs has two purposes. One is readability, as it is easier to understand the proof if the subgoals are explicitly embedded in the proof script. Another is to minimize the size of the code sent to the HOL prover. The check command assumes that the subgoal specified by an outer CLAIM is correct. This makes it only necessary to send the tactic between the text cursor and the outer CLAIM.

There are a few more commands supported by our interface.

- check-claim-body
- check-claim-specs

These two commands are available at CLAIMs.

The check-claim-body command checks whether the body tactic of the CLAIM proves the goal specified by the CLAIM. If the tactic can prove the goal, the message OK appears at the message line. Otherwise, an error is reported.

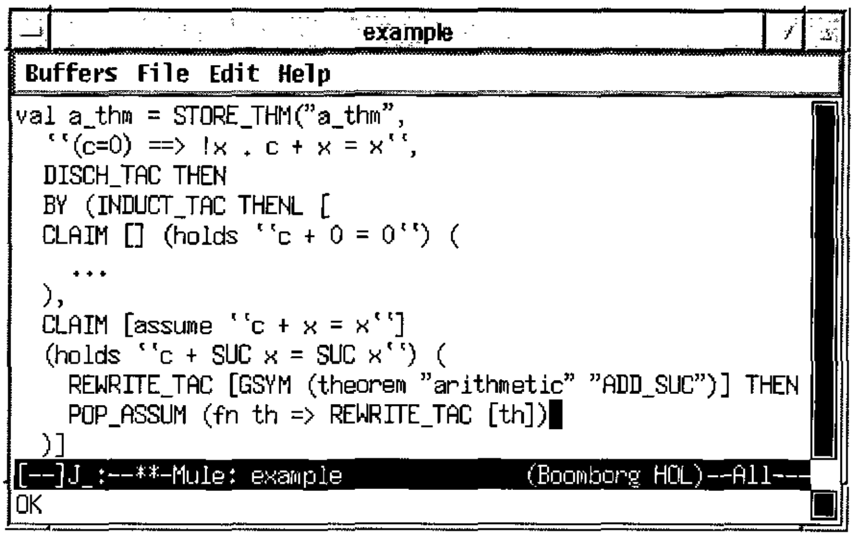
The check-claim-specs command recalculates the goal at the CLAIM by executing the tactic just before the CLAIM. If the obtained goal is as specified by the CLAIM, it merely shows the message OK. Otherwise, it replaces the assumption part and the conclusion part of the CLAIM so that they correctly specify the goal. This command is intended to be used for reusing a proof script. When the top goal of a proof script has been changed, it is possible to propagate the change through the script by repeatedly invoking this command. It is also considered as solving the constraint between a CLAIM and a tactic before it.

3 Proof-by-pointing

Proof-by-pointing [1] is a very useful interface for interactive theorem proving. Proof-by-pointing starts with pointing to a subterm in a goal. The goal is repeatedly decomposed by inference rules until the subterm appears at the top-level. For example, when the user points to the subterm B in the goal ?- A ==> B \/ C, the implication and disjunction are decomposed and the user gets the new goal A ?- B. Many proof steps are executed by one pointing. Note that pointing is a graphical operation.

Our interface also supports proof-by-pointing. We first prepare a tactic to perform decomposition, called a proof-by-pointing tactic:

PROOF_BY_POINTING "subterm" "position" "instances".



```
val a_thm = STORE_THM("a_thm",
  ''(c=0) ==> !x . c + x = x'',
  DISCH_TAC THEN
  BY (INDUCT_TAC THENL [
  CLAIM [] (holds ''c + 0 = 0'') (
    ...
  ),
  CLAIM [assume ''c + x = x'']
  (holds ''c + SUC x = SUC x'') (
    REWRITE_TAC [GSYM (theorem "arithmetic" "ADD_SUC")] THEN
    POP_ASSUM (fn th => REWRITE_TAC [th])
  )]
[--]J_:--**-Mule: example                 (Boomborg HOL)--All---
OK
```

Figure 6: An example of using the check command

The "subterm" specifies a subterm in the conclusion of the current goal. The "position" is the index of the occurrence of the subterm; 0 means the first occurrence, 1 the second, etc. The "instances" are terms that instantiate quantified variables during decomposition. The call

PROOF_BY_POINTING s n $[u_1, u_2, \cdots]$

means that the n-th subterm of s in the conclusion of the current goal is pointed to and quantified variables are instantiated by u_1, u_2, $\cdots$ in this order. For example, when B is pointed to in the goal ?- A ==> B \/ C, the corresponding proof-by-pointing tactic is

PROOF_BY_POINTING ''B :bool'' 0 [].

In our interface, the user can pop up a window showing the current goal at any position in a proof script (Figure 7). In the window of the current goal, the conclusion appears at the top line, and the assumptions are represented below the underlines, as is HOL. The user can point to a subterm in the pop-up window using a mouse. A proof-by-pointing tactic is then generated and inserted at the given position. In Figure 7, if the user points to the first occurrence of x, then a proof-by-pointing tactic is generated and inserted as in Figure 8. The "position" is 0, meaning the first occurrence. The "instances" of an inserted tactic are always a null list; the user is expected to fill in this list as needed. The proof-by-pointing tactic is only inserted, and not executed. Our interface for proof-by-pointing is only for inputting a tactic; the check command is used to execute it. A similar approach to proof-by-pointing has been taken by Bertot, Schreiber and Sequeira [3]. We discuss the differences between our approach and theirs in Section 5.

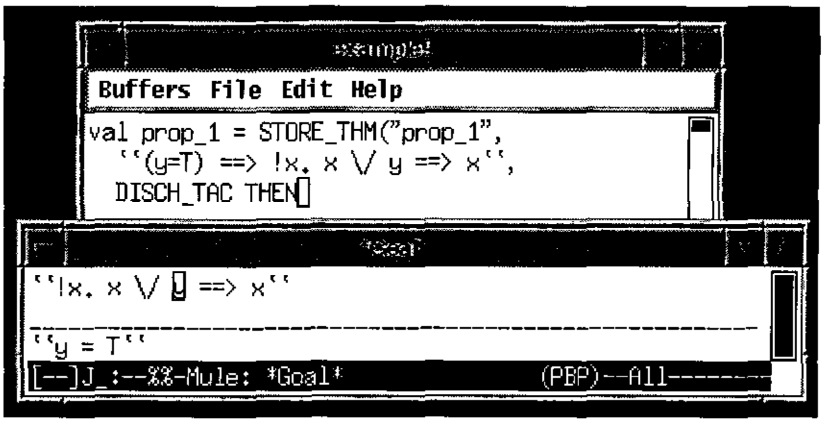
```
val a_thm = STORE_THM("a_thm",
  ''(y = T) ==> !x . x \/ y ==> x'',
  DISCH_TAC THEN
  PROOF_BY_POINTING ''x'' 0 []
```

Figure 8: A proof-by-pointing tactic inserted

4 Implementation

Since our interface is implemented on Emacs and communicates with a HOL prover, its implementation consists of an Emacs Lisp program and an SML program for communication and manipulation of goals, including the definitions of CLAIM and BY as functions in SML.

Before describing the implementation of the check command, let us explain an Emacs Lisp function called get-goal-at-point. The get-goal-at-point function calculates the goal at a CLAIM or BY, or at the beginning of a proof (i.e., at STORE_THM). If the text cursor is at the beginning of a proof, it returns the goal as specified by STORE_THM. If the text cursor is at a CLAIM, since the difference between the goal at the CLAIM and the goal at the outer CLAIM or BY is specified by the arguments



```
val prop_1 = STORE_THM("prop_1",
  ``(y=T) ==> !x. x \/ y ==> x``,
  DISCH_TAC THEN
```

```
``!x. x \/ y ==> x``
------------------------------------------------
``y = T``
[--]J_:--%%-Mule: *Goal*          (PBP)--All-------
```

Figure 7: A window of the current goal

of the CLAIM, it returns the goal at the CLAIM by calculating it with the difference; the goal at the outer CLAIM or BY is calculated by a recursive call of get-goal-at-point). If the text cursor is at a BY, the get-goal-at-point essentially uses the HOL prover. It sends to the HOL prover the goal at the outer CLAIM or BY (obtained by a recursive call) and the tactic between the BY and the outer CLAIM or BY, and returns the result of the tactic.

The check command executes the tactic between the text cursor and the outer CLAIM or BY by sending it to the HOL prover with the goal at the outer CLAIM or BY (obtained by get-goal-at-point). It then inserts the subgoals returned by the tactic into the buffer with appropriate control structures using THEN or THENL.

To implement the proof-by-pointing interface, we need to know the position of each subterm in a goal that is shown in the pop up window. We modified the pretty printer of HOL so that it records the position of each subterm as it prints out a given goal. (Unfortunately, we could not make use of the extensible pretty printer of HOL for this purpose. We had to directly modify the source code of the pretty printer.)

In HOL, every term surrounded by `` must be completely typed by itself after it is parsed; it cannot be polymorphic in the sense of Milner and Damas. So terms that appear in the arguments of a CLAIM must also be completely typed. The default HOL pretty printer has two modes. In one mode it never shows the types in a term, while in the other mode it prints every variable or constant with its type. If arguments of a CLAIM are printed in the former mode, they may cause errors when they are sent to the HOL prover because they may not be completely typed. In the latter mode, however, outputs become too noisy. In order to solve this problem, we modified the original HOL pretty printer and made a new one that shows types as little as possible. This pretty printer shows a type only on one occurrence of each variable or constant; and if the type of a variable or constant can be easily inferred, the type is not shown.

5 Discussion

As Merriam and Harrison [6] pointed out, one problem of proving with graphical user interfaces (GUIs) has been their weakness in reusability. When the user has modified the top goal of a proof script, he or she may want to reuse as many parts of the old proof script as possible, but it is difficult to do so under GUIs because the history of interaction is either lost or not editable. Supporting editable records of interaction is essential for reuse. In our proof-by-pointing interface, a user's pointing is inserted into a proof script as a tactic, and the user can reuse these tactics as other parts of a proof script. This suggests a way to utilize various kinds of GUI (e.g., drag-and-drop rewriting [2]) in our interface.

In order to utilize GUIs, some conditions are required:

- one operation on a GUI has to be translated into one tactic,
- the translated tactic has to cause the same effect as the operation, and
- from the translated tactic, it should be possible to replay the operation.

Although there are many ways to satisfy the first condition, the reusability of translated tactics is heavily affected by the design of the translation. In our case of proof-by-pointing, "pointing of a subterm" could be represented by one number, the position of the subterm from the beginning of the entire goal. A tactic with such limited information can hardly be reused. For such reasons, independence of translation from the context of interaction is important.

To achieve the second condition in the case of proof-by-pointing was relatively easy, and we were able to implement the required function in only a hundred lines.

Once translation satisfying these conditions is implemented, a new GUI becomes available in our interface. We expect that various kinds of GUI will be introduced in this manner.

Bertot, Schreiber and Sequeira have developed an interface for proof-by-pointing on XEmacs [3]. In their interface, the current goal is displayed in the goal buffer and pointing in the goal buffer by the user generates plain proof commands for the LEGO proof assistant, and inserts them in a buffer for saving in a proof script. While their interface translates a pointing to a sequence of primitive commands such as `intros` and `impE`, ours inserts a proof-by-pointing tactic, so that it is possible to replay the operation for proof-by-pointing using the inserted tactic. As mentioned above, we think that replayability is a key to reusing graphical operations. Proof-by-pointing tactics also make proof scripts more compact and readable.

6 Experience

In this section, we describe some of the insights we have gained through the experience of writing HOL proof scripts using our interface. We found the merits of our interface to be as follows.

The most important is that there is little overhead for the user in retrying tactics. In the course of constructing a proof, repeated trial and error is often used, and this prevents the user from concentrating on the activity of proving. With our interface, in order to try a new tactic, the user merely rewrites the old tactic to a new one, and invokes the `check` command. When the user interacts directly with HOL, he or she must recover the status of HOL before a new tactic can be tried. The merit of our interface is particularly evident when the user wants to change a part of a finished proof script, because setting up the status of HOL properly by hand is very difficult.

When the `check` command is executed, control structures (`THEN` or `THENL`) and parentheses are automatically inserted, meaning that the user does not need to pay attention to the structure of the proof script.

We also noticed some demerits of our interface. One is that proof scripts written with our interface tend to become long because they contain many `CLAIM`s. In particular, when a goal term is large, each `CLAIM` may consume many lines. In such cases, the user has to delete `CLAIM`s by hand. But the user has to pay attention to the effect of deleting a `CLAIM` from a finished script; replacing a `CLAIM` with a `BY` is safe, but deleting a `CLAIM` may be harmful because the specification of the goal at a `CLAIM` is relative. The user can resolve this conflict by using the `check-claim-specs` command.

Another inconvenient point is that when an error arises, the user has to investigate the cause of the error by looking at the output of HOL. Since typographic errors or careless mistakes often occur, we consider that error handling is a serious problem to be solved.

7 Conclusion

We developed an Emacs interface for writing HOL proof scripts based on the Computing-as-Editing paradigm. By using this interface, the user can edit structural proof scripts while interacting with the HOL prover. The interface also supports proof-by-pointing to input tactics.

In the future, we plan to implement drag-and-drop rewriting [2] and other kinds of GUI under our framework.

Our experience so far suggests that the cost of communication between Emacs and HOL is not large, but it is important to investigate how the communication cost increases as we treat huge goals with our interface.

It is also important to solve the problem that for huge goals the text inserted by `check` becomes too long. The representation of terms needs to be improved, for example, by embedding huge terms into hyper text [9].

As mentioned in Section 6, error handling is another important issue.

The developed interface will be available from the following URL in the near future.

`http://nicosia.is.s.u-tokyo.ac.jp/boomborg/`

References

[1] Yves Bertot, Gilles Kahn, and Laurent Théry: Proof by Pointing. In *Theoretical Aspects of Com-*

puter Software, volume 789 of Lecture Notes in Computer Science, 1994, pp.141–160.

[2] Yves Bertot: Direct manipulation of Algebraic Formulae in Interactive Proof, *User Interfaces for Theorem Provers UITP'97*, INRIA Sophia-Antipolis, 1997, pp.17–24.

[3] Yves Bertot, Thomas Kleymann-Schreiber and Dilip Sequeira: Implementing Proof by Pointing without a Structure Editor. Technical Report ECS-LFCS-97-368, University of Edinburgh, 1997.

[4] M. J. C. Gordon and T. F. Melham: *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*, Cambridge University Press, 1993.

[5] Masami Hagiya and Hiroshi Kakuno: Proving as Editing, *User-Interfaces for Theorem Provers UITP'96*, Department of Computer Science, University of York, 1996, pp.35–42.

[6] Nicholas A. Merriam and Michael D. Harrison: What is Wrong with GUIs for Theorem Provers, *User Interfaces for Theorem Provers UITP'97*, INRIA Sophia-Antipolis, 1997, pp.67–74.

[7] Lawrence C. Paulson: Isabelle: A Generic Theorem Prover, volume 828 of Lecture Notes in Computer Science, 1994.

[8] Donald Syme:
http://www.cl.cam.ac.uk/users/drs1004/TkHol.html.

[9] Yasuaki Takebe and Masami Hagiya: A user Interface for Controlling Term Rewriting Based on Computing-as-Editing Paradigm, *User-Interfaces for Theorem Provers UITP'97*, Sophia-Antipolis, INRIA, 1997, pp.93–100.

A An example of script

```
(*
 *  DESCENDING_CHAIN, NOETHERIAN_CHAIN
 *)

use "bbg-hol.sml";
new_theory "example";
add_theory_to_sml "arithmetic";

val NOETHERIAN_CHAIN_DEF =
    new_definition(
   "NOETHERIAN_CHAIN_DEF",
   ``!(k_:num->num). NOETHERIAN_CHAIN k_ =
          ?n. !m. (n < m) ==> (k_ m = k_ n)``);

val noetherian_prop_1 =
    STORE_THM("noetherian_prop_1",
    ``!(k_:num->num).
        NOETHERIAN_CHAIN k_ = ?n. !m. (n <= m) ==> (k_ m = k_ n)``,
    REWRITE_TAC [NOETHERIAN_CHAIN_DEF, LESS_OR_EQ] THEN
    GEN_TAC THEN
    EQ_TAC THEN
    STRIP_TAC THEN
    EXISTS_TAC ``n:num`` THEN
    REPEAT STRIP_TAC THENL [
    CLAIM [assume ``n < m``,
           assume ``!m. n < m ==> ((k_:num->num) m = k_ n)``]
    (holds ``(k_:num->num) m = k_ n``) (
      RES_TAC
    ),
    CLAIM [assume ``n:num = m``,
           assume ``!m. n < m ==> ((k_:num->num) m = k_ n)``]
    (holds ``(k_:num->num) m = k_ n``) (
      ASM_REWRITE_TAC[]
    ),
    CLAIM [assume ``n < m``,
           assume ``!m. n < m \/ (n = m) ==> ((k_:num->num) m = k_ n)``
    (holds ``(k_:num->num) m = k_ n``) (
      RES_TAC
    )]
    );

val DESCENDING_CHAIN_DEF =
    new_definition(
    "DESCENDING_CHAIN_DEF",
    ``!k_. DESCENDING_CHAIN k_ =
          !(n:num). (k_ (SUC n)) <= (k_ n)``);

val CHAIN_lemma_2 =
    STORE_THM("CHAIN_lemma_2",
    ``!k_.
       DESCENDING_CHAIN k_ ==>
        !n m. n <= m ==> k_ m <= k_ n``,
    GEN_TAC THEN REWRITE_TAC[DESCENDING_CHAIN_DEF] THEN
    DISCH_TAC THEN
    CLAIM [assume ``!n. k_ (SUC n) <= k_ n``]
    (holds ``!n m. n <= m ==> k_ m <= k_ n``) (
      REWRITE_TAC[LESS_EQ_EXISTS] THEN
      CONV_TAC (DEPTH_CONV LEFT_IMP_EXISTS_CONV) THEN
      REWRITE_TAC[GSYM LESS_EQ_EXISTS] THEN
      REPEAT GEN_TAC THEN
      MAP_EVERY SPEC_TAC
      [(``m:num``,``m:num``),(``n:num``,``n:num``),
       (``p:num``,``p:num``)] THEN
      CLAIM [] (holds ``!p n m. (m = n + p) ==> k_ m <= k_ n``) (
        INDUCT_TAC THENL [
        CLAIM [] (holds ``!n m. (m = n + 0) ==> k_ m <= k_ n``) (
          SIMP_TAC hol_ss []
        ),
        CLAIM [assume ``!n m. (m = n + p) ==> k_ m <= k_ n``]
        (holds ``!n m. (m = n + SUC p) ==> k_ m <= k_ n``) (
          REPEAT STRIP_TAC THEN
          ASM_REWRITE_TAC[GSYM ADD_SUC] THEN
          MATCH_MP_TAC
          (SPEC ``k_ (n + p):num``
          (SPEC ``k_ (SUC (n + p)):num`` LESS_EQ_TRANS)) THEN
          STRIP_TAC THENL [
          CLAIM [assume ``m = n + SUC p``]
          (holds ``k_ (SUC (n + p)) <= k_ (n + p)``) (
            FIRST_ASSUM MATCH_ACCEPT_TAC
          ),
          CLAIM [assume ``m = n + SUC p``]
          (holds ``k_ (n + p) <= k_ n``) (
            FIRST_ASSUM MATCH_MP_TAC THEN
            CLAIM [] (holds ``n + p = n + p``) (
              SIMP_TAC hol_ss []
            )
          )]
        )]
      )
    )
);
```

A user-interface for Knuth-Bendix completion

Patrick Viry
viry@kurims.kyoto-u.ac.jp

March 25, 1998

1 Introduction

The Knuth-Bendix completion procedure, or more precisely the family of completion procedures, is at the heart of algebraic theorem proving and equational programming.

In a theorem proving setting, it can be used for solving the word problem in a given algebra (is s equal to t modulo some given equations?) by first trying to determine a convergent rewrite system equivalent to the equations and then rewriting using these rewrite rules. It also provides a semi-decision procedure for solving arbitrary equations (unfailing completion [2]).

In a programming environment, it can be seen as a program transformation technique for constructing programs with desired properties (usually convergence, confluence or coherence [10, 16]). Equational programming, or its generalization rewriting logic, is a very high level specification language with a direct operational counterpart, ie. specifications are immediately executable *as long as the rules and equations verify some of those properties.*

Ongoing research on completion has led to many tools implementing variations of the completion procedure (for instance REVE [12], Orme [13], Cime [3]). On the other hand, programming environments based on term rewriting have also become available (OBJ3 [5], Maude [14], ELAN [11], CafeOBJ [4]), with various semantic extensions to the basic model and with fast implementations [15].

However, these programming environments remain *static* in the sense that they provide a descriptive framework and an implementation, but do not integrate program transformation. Namely, it is up to the user to provide systems of rules and equations meeting the required properties: no facility is provided for the construction of the program, which usually implies using some kind of completion procedure.

Finding a system of rules and equations verifying the desired properties is not an easy task. First of all, it is undecidable in general whether such a system exists, hence one cannot expect an automated procedure to produce the desired result: user interaction is a necessity.

Then, reaching the solution (when it exists) implies making the right choices at the right time. Because the number of rules and equations generated by the completion procedure may grow very quickly, it is very important to present information in a way easy to understand when asking the user for interaction. Because a choice may lead to a dead end, it is also important to allow a possibility of backtracking for undoing previous choices. A well designed user-interface is of upmost importance.

Existing theorem provers dedicated to completion were often designed to implement and test theoretical ideas, but are usually weak from a user interface point of view. Most run in some kind of batch mode, taking a text file as input and producing a text file, with little interaction and often no possibility of backtracking.

Reusing interfaces developed for other kinds of theorem provers is not an option, because completion is quite different from other theorem proving techniques. The objects on which it operates are sets of rules and equations, with a complex structure and the notion of *critical pair* at the heart of the deduction process. It is not goal directed, all consequences of the initial set of rules and equations must be generated in order to find a solution.

I will present here some preliminary ideas towards a user interface for completion and its integration within programming environments. These ideas mainly stem from the "proving as editing" paradigm [7, 8], which considers that developing a proof consists of editing the *proof object* (typically in a buffer of GNU emacs), where the basic editing operations correspond to deduction steps.

An important feature of the user interface presented here is dependence-based backtracking, that is the ability to undo an operation and all the subsequent operations dependent on it, without undoing unrelated operations even if they took place later in time. This makes proof reuse easy and intuitive.

Because of space limitations, I will assume familiarity with term rewriting and completion. A gentle introduction to these concepts can be found in [1].

2 Definition

I will present here a "typical" completion procedure. There are many variations or extensions of this procedure, but they all work along the same basic principles and should be able to accomodate the same interface.

A completion procedure starts with a finite set of equations E and tries to find a convergent rewriting system R equivalent to E, that is such that the relations $=_E$ and $=_R$ are the same. It works on pairs (E_i, R_i), with $E_0 = E$ and $R_0 = \emptyset$, and each deduction step

$$(E_i, R_i) \vdash (E_{i+1}, R_{i+1})$$

is an application of one of the inference rules given in figure 1. A completion succeeds if $E_n = \emptyset$ for some n, in which case R_n is a convergent rewrite system equivalent to E. It may also end in failure if $E_n \neq \emptyset$ and no inference rule applies, or it may not terminate.

Note that the order $>$ on terms (in the condition part of ORIENT) is a parameter of the completion procedure.

In the case of DEDUCE, the terms s and t are normally defined using the notion of *critical pairs.* For two rules $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$, if there is a position p in l_1 such that $l_1|p$ is not a variable and there exists a most general unifier σ of $l_1|p$ and l_2 (with their variables renamed), then $(\sigma r_1, (\sigma l_1)[\sigma r_2]_p)$ is a critical pair. A critical pair (s, t) is *trivial* if there exists v such that $s \longrightarrow v \longleftarrow t$. The set of all critical pairs between rules of R is denoted $CP(R)$; the condition part of DEDUCE could thus be rewritten as $(s, t) \in CP(R)$.

Other variations or extensions of this basic completion procedures have similar sets of rules. The main difference relevant to our purpose is that some of them use to mark and unmark rules or equations.

The inference rules can be classified according to their shape :

Type 1. moving one rule of R to an equation of E or vice-versa (ORIENT, L-SIMPLIFY-RULE), or modifying one rule or equation (SIMPLIFY-IDENTITY, R-SIMPLIFY-RULE)

Type 2. removing a rule or equation (DELETE)

DEDUCE	$\dfrac{E,R}{E\cup\{s=t\},R}$	if $s \longleftarrow_R u \longrightarrow_R t$
ORIENT	$\dfrac{E\cup\{s=t\},R}{E,R\cup\{s\longrightarrow t\}}$	if $s>t$
DELETE	$\dfrac{E\cup\{s=s\},R}{E,R}$	
SIMPLIFY-IDENTITY	$\dfrac{E\cup\{s=t\},R}{E\cup\{u=t\},R}$	if $s \longrightarrow_R u$
R-SIMPLIFY-RULE	$\dfrac{E,R\cup\{s\longrightarrow t\}}{E,R\cup\{s\longrightarrow u\}}$	if $t \longrightarrow_R u$
L-SIMPLIFY-RULE	$\dfrac{E,R\cup\{s\longrightarrow t\}}{E\cup\{u=t\},R}$	if $s \xrightarrow{\sqsupset}_R u$

Figure 1: The inference rules for completion as presented in [1]. Equations $s=t$ are considered non oriented, in order to avoid two copies of ORIENT and SIMPLIFY-IDENTITY

Type 3. introducing a new rule or equation (DEDUCE, as well as the introduction of equations by the user)

An application of an inference rule of type 1 and 2 is uniquely determined by the equation or rule on which it "applies", ie. the equation or rule singled out in the premisse (this is not exactly true because the reduct u in the simplification rules may not be unique, but the representation we will choose for the proof object allows to recover u). *Applications* of inference rules will thus be denoted ORIENT(e), L-SIMPLIFY-RULE(ρ), SIMPLIFY-IDENTITY(e), R-SIMPLIFY-RULE(ρ), and DELETE(e).

An application of DEDUCE is uniquely determined by a critical pair in $CP(R)$, that is two rules $l_1 \longrightarrow r_1$ and $l_2 \longrightarrow r_2$ and a position p in l_1. It will be denoted DEDUCE(ρ_1,ρ_2,p).

3 The proof object

In the proof as editing paradigm, proving is editing the proof object. Our first task is thus to define the structure of this proof object.

According to the definition of a completion procedure, a proof is a sequence

$$(E_0,R_0) \vdash \cdots \vdash (E_n,R_n)$$

Given (E_0,R_0), we can drop the intermediate terms (E_i,R_i) and define a proof as a sequence of applications of inference rules, for instance

$$\text{ORIENT}(e_n)\ ;\ \text{SIMPLIFY-IDENTITY}(e_m)\ ;\ \ldots$$

This linear structure is not very informative, in particular it doesn't emphasize the commutation properties of the deduction steps. We would like to consider a completion proof modulo commutation axioms such as

$$\text{ORIENT}(e_n)\ ;\ \text{ORIENT}(e_m) \quad = \quad \text{ORIENT}(e_m)\ ;\ \text{ORIENT}(e_n) \qquad \text{if } m \neq n$$

I will not try to give here all such axioms, but basically one would like to allow two deduction steps to commute as long as the second does not make use of a rule or equation that has been deleted or modified by the first.

The proof objects that we shall consider from now on are thus sequences of inference rules applications quotiented by these commutation axioms.

In order to represent these proof objects, I propose the following structure, based on the notion of acyclic *hypergraph*. A hypergraph has nodes and multiedges, that is arrows with an arbitrary number of sources and targets. Hypergraph rules describe transformations of hypergraph. See [6] for a formal definition.

The nodes of our hypergraph are rules and equations, possibly deleted (we are not allowed to discard them if we want to preserve the structure of the proof) and possibly marked or unmarked:

$\boxed{s = t}$	$\boxed{s \longrightarrow t}$	$\bbox[border:1px dashed]{s = t}$	$\bbox[border:1px dashed]{s \longrightarrow t}$
an equation	a rule	a deleted equation	a deleted rule

Hyperedges are labelled with the name of an inference rule. To each inference rule corresponds one of the following hypergraph rules:

$\boxed{\mathrm{s = t}}$	$\longrightarrow$	$\bbox[border:1px dashed]{s = t} \xleftarrow{\text{ORIENT}} \boxed{s \longrightarrow t}$
$\boxed{s = t}$	$\longrightarrow$	$\bbox[border:1px dashed]{s = t} \xleftarrow{\text{SIMPLIFY-IDENTITY}} \boxed{u = t}$
$\boxed{s = t}$	$\longrightarrow$	$\bbox[border:1px dashed]{s \longrightarrow t} \xleftarrow{\text{R-SIMPLIFY-RULE}} \boxed{s \longrightarrow u}$
$\boxed{s = t}$	$\longrightarrow$	$\bbox[border:1px dashed]{s \longrightarrow t} \xleftarrow{\text{L-SIMPLIFY-RULE}} \boxed{u = t}$
$\boxed{s = s}$	$\longrightarrow$	$\bbox[border:1px dashed]{s = s}$
$\boxed{l_1 \longrightarrow r_1}$ $\boxed{l_2 \longrightarrow r_2}$	$\longrightarrow$	$\boxed{l_1 \longrightarrow r_1}$ p $\xleftarrow{\text{DEDUCE}} \boxed{s = t}$ $\boxed{l_2 \longrightarrow r_2}$

In the first four cases (rules of type 1), a non deleted node is marked as deleted, and a new node is added with an edge labelled with the inference rule pointing to the old node. In the case of simplification rules, the reduced u is present as the left or right-hand side in the newly introduced node, and does not need to be made explicit as a parameter of the application.

In the fifth case (DELETE), the node is simply marked as deleted.

In the last case, an application of the inference rule DEDUCE depends on two rules and a position, hence the corresponding hyperedge has two targets, and is labelled with the position.

Without a formal proof, we will take for granted that the class of hypergraphs constructed by repeated applications of these hypergraph rules (plus the rule corresponding to the introduction of a new equation by the user) is isomorphic to the class of derivations quotiented by the appropriate commutation axioms.

We will say that a node n depends on a node m if there is a path from n to m in the hypergraph.

```
equations
  a1 = a2                          a1 = a2
  b1 = b2                          c1 -> c2
rules                              b1 = b2
  c1 -> c2                         d1 -> d2
  d1 -> d2
```

spacial separation   mixed representation

Figure 2: Two types of syntax

4 Editing operations

4.1 Textual representation

A basic principle of the proof as editing paradigm is that there should be *no state* (also called principle of visibility), ie. the visible text should be an exact representation of the proof object being manipulated.

The proof object can be shown in its entirety by displaying both deleted and non deleted rules and equations, and labelling them with the application that produced them (using rule numbers). Deleted rules and equations can be identified by a special marking and/or grayed text. For instance:

```
f(x)  =  x + x    [1,Intro,Deleted]
f(x) --> x + x    [2,Orient(1)]
```

In this example, `Intro` means a rule or equation introduced by the user. Although following the principles of proof as editing, such a display may not be suited for two reasons:

- The number of deleted rules and equations grows very quickly, and can become a nuisance. On the other hand, deleted rules and equations may provide valuable information about the paths that have already been explored and help make future choices. I suggest that it should be possible to switch between displaying or not deleted rules and equations.
- The information about edges is normally needed only at one point, when previewing the effect of dependence-based backtracking (see section 4.3). This information can be displayed by highlighting all rules and equations depending on the currently selected one, which avoids the need of a textual representation of edges and minimizes the amount of information displayed. It is however possible to provide a textual representation on demand.

  Obviously, a textual representation of edges is meaningless when the rules or equations referred to are deleted and not displayed.

Five out of six inference rules replace one given rule or equation with an other. In order to keep the user focused, it is important that these operations can be done in place, and thus choose a "mixed" syntax that does not rely on spacial separation to identify what is a rule and what is an equation (see figure 2).

The application of an inference rule of type 1 or 2 is now simple: point to the appropriate rule or equation, and select the rule to apply.

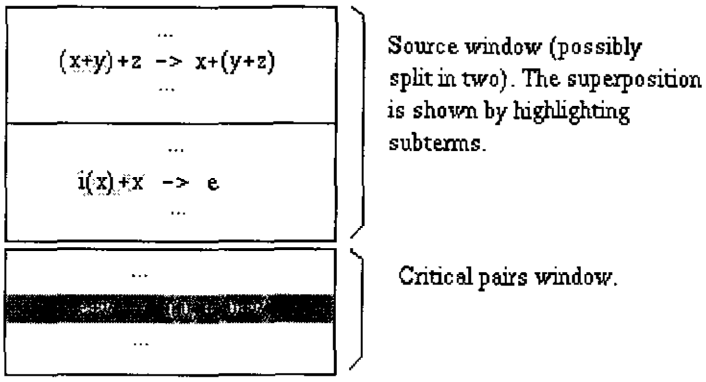

...
(x+y)+z -> x+(y+z)
...
...
i(x)+x -> e
...
...
...
Source window (possibly split in two). The superposition is shown by highlighting subterms.
Critical pairs window.

Figure 3: A sample display: selecting a critical pair for the application of DEDUCE.

The application of DEDUCE is defined by a critical pair. Because identifying critical pairs is a difficult and non intuitive task, it is not reasonable to leave this burden to the user. I propose to have a second window displaying all current critical pairs (possibly limited to non trivial ones). Applying DEDUCE would consist then of choosing a particular critical pair in this second window. A sample display may look like figure 3.

Finally, another editing operation should be provided, although not corresponding to any inference rule, namely the introduction of a new rule or equation. Not being able to complete a system may be a symptom of a bad design, for instance an incomplete function definition, and an introduction operation makes it possible to modify the original system without restarting the proof from scratch (it also makes explicit the initial input of the set of equations).

4.2 The order

In the definition of completion, a given order $>$ is used when orienting equations into rules (ORIENT). Finding the "right" order is part of the art of doing a completion (different orders may lead to failure or non termination of the completion procedure, or simply to different results), and it would be very convenient to be able to construct the order incrementally during completion (allowing backtracking also on the definition of the order). Such an incremental definition is actually possible for a large family of orderings (the recursive path ordering and its variations) and has been implemented for instance in REVE.

In order to apply ORIENT to an equation $s = t$, we should have either $s > t$ or $t > s$, but $>$ may be a partial order and s and t may not be comparable. When this is the case, one may try to extend the definition of the order according to the desired orientation (it may be the case that neither orientation is possible, this is a cause for failure of the completion procedure).

A nice feature of many kinds of orderings is that the definition of the order can be infered automatically from the chosen orientation. This leads to a nice user interface where one simply selects orientation of equations, without having to know about the underlying order. Otherwise, the application of ORIENT may start a dialog requesting the needed information.

4.3 Backtracking

Backtracking can take two forms:

- chronological: undo the last inference step
- dependence based: remove a rule or equation, undoing all operations that were dependent on it

Chronological ordering is easy to implement by numbering the edges of the hypergraph in the order in which they have been added. Dependence-based ordering can be implemented by following paths in the hypergraph, removing all nodes depending on a given node and all the edges between them.

This possibility of dependence-based backtracking, made possible by the structure we choosed for representing proof objects, provides a basis for proof reuse: all inference steps that were not dependent on the removed rule or equation are preserved. Proofs can be reused between sessions by saving the whole proof object.

As suggested in section 4.1, it is possible to preview the effect of dependence based backtracking by highlighting all rules and equations depending on the one to be deleted.

4.4 Other issues

Other aspects of this user interface should or could be addressed, but I will only give here a brief summary for lack of space:

- determining the set of critical pairs presented to the user is computationally intensive, it should be updated incrementally after each editing operation.
- the same goes for the rewrite engine: reduction in simplification rules or in the test of trivial critical pairs uses the current set of rules. Fast rewrite engines contruct automatas for fast matching, these should be updated incrementally.
- it should be possible to define tactics
- although this is not possible in general, some common cases of non termination can be detected by identifying so-called *forward closures* [9]. This information is highly valuable for the user in order to avoid running into dead ends and could be computed for instance by a background process.

5 Conclusion

The preliminary ideas presented here have yet to be implemented and tested against practical usage. I hope however to have made clear the whole architecture of the system and shown how it can be effectively realized.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1998. ISBN 0-521-45520-0.

[2] L. Bachmair. Proof by consistency in equational theories. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*, pages 228–233, 1988.

[3] Evelyne Contejean and Claude Marché. CiME: Completion Modulo *E*. 1996. System Description available from
`http://www.lri.fr/~demons/cime.html`.

[4] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment. In *First International Conference on Formal Engineering methods*, 1997. Available from the CafeOBJ home page at
`http://www.ldl.jaist.ac.jp/cafeobj/`.

[5] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *LNCS*, pages 258–263. Springer-Verlag, July 1987.

[6] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. *Fundamenta Informaticae*, XV:37–60, 1991.

[7] M. Hagiya. Boomborg-PC: A proof-checker of calculus of constructions running on a buffer of GNU emacs. Technical report, Department of Information Science, University of Tokyo, 1995. Available from `http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html`.

[8] M. Hagiya and Y. Takebe. A user interface for controlling term rewriting based on computing-as-editing paradigm. In *User Interfaces for Theorem Provers UITP'97*, 1997. Available from
`http://nicosia.is.s.u-tokyo.ac.jp/members/hagiya.html`.

[9] M. Hermann. Chain properties of rule closures. *Formal Aspects of Computing*, 2(3):207–225, 1990.

[10] J.-P. Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[11] C. Kirchner, Kirchner H., and Vittek M. Designing CLP using computational systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT press, 1995.

[12] P. Lescanne. REVE: a rewrite rule laboratory. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, LNCS, pages 696–697. Springer-Verlag, 1986.

[13] P. Lescanne. Orme, an implementation of completion procedures as sets of transitions rules. In M. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, pages 661–662. Springer-Verlag, 1990.

[14] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*. MIT Press, 1993.

[15] P.-E. Moreau and H. Kirchner. Compilation of associative-commutative normalisation with strategies, 1997.

[16] P. Viry. Rewriting modulo a rewrite system. Technical Report TR-95-20, Università di Pisa, 1995. Available from
`http://www.kurims.kyoto-u.ac.jp/~viry/`.

Aspects of the Proof-assistant Yarrow

Jan Zwanenburg
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
e-mail janz@win.tue.nl

May 29, 1998

Abstract

Yarrow is an interactive proof assistant based on the theory of Pure Type Systems, a family of typed lambda calculi. It offers both graphical and textual interfaces. It has been coded entirely in Haskell, making use of the Fudget library for the graphical interface. In this paper we concentrate on the software architecture of Yarrow, in particular the coupling of user interface and proof engine. We also treat the presentation of proofs in the flag-style format.

1 Introduction

In this paper we describe the system Yarrow, an interactive proof assistant based on the theory of Pure Type Systems, a family of typed lambda calculi. In typed lambda calculi, theorems and proofs can be represented as well-typed terms, proof checking amounts to type checking, and proof construction to the construction of a term of a given type. These properties make typed lambda calculi well suited as formal basis of systems that support interactive construction of proofs, so-called proof assistants. Some well-known proof assistants of this kind are Coq [Coq97], LEGO [LP92], and Alfa [Hal97].

The system Yarrow has been designed as a flexible environment for experimentation with PTSs, extended with a definition mechanism. It can handle a large class of PTSs, the so-called bijective PTSs [Pol93], which includes all systems of the lambda cube [Bar92]. A typical Yarrow session consists of: selecting a PTS and loading several modules of definitions and theorems based on this PTS, after which a number of proof tasks are carried out. Each proof task consists of the interactive construction of a term which is well typed with respect to the loaded context. Besides a conventional command line interface, Yarrow also offers a graphical interface with windows for global context and proof tasks, menu and mouse selection of tactics and subterms.

Yarrow has been coded entirely in the purely function language Haskell [Tho96, Pet97]. In this paper we discuss two interesting aspects of the Yarrow:

1. The architecture that supports for multiple interfaces: The Yarrow proof engine has been designed in such a way that it can cooperate with various user interfaces. Two such interfaces have been developed. First, a simple command line interface, which can be used on any platform supporting Haskell, because this interface is based on the standard IO monads [Tho96]. Second, a graphical interface based on the Fudget system [HC95], which is a Haskell library available for a limited number of platforms. The coupling between user interface and engine is very thin, consisting of just a single function.

2. The ability to print proofs in the so-called flag-style format.

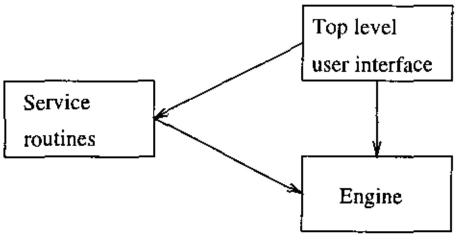


Figure 1: Global system architecture

In section 2 we describe the global architecture of Yarrow. Section 3 treats the top level user interface, and the communication with the engine. Section 4 discusses the flag-style format. We give the conclusion in section 5.

2 System architecture

Figure 1 shows the architecture of Yarrow. Each block is a module with a certain functionality, and each arrow indicates a dependency. The block labeled "Engine" is the part that defines the objects of our system (terms, contexts, specification of PTSs) and the functions that manipulate these objects, like the typing routine, the tactics, and the routines that extend the context.

The block labeled "Service routines" consists of all sorts of routines needed for the user interface without actually performing any IO. This includes printing and parsing routines, and the displaying of help texts. The service routines depend on the engine in a rather trivial way. They use only the representation of the objects and some elementary functions on these representations.

The block labeled "Top level user interface" (TLI) contains the main loop of the program. This handles input by the user, sends the appropriates messages to the engine, and presents the results of these messages on the screen. The TLI uses several service routines, but only one function of the engine. The combination of the TLI with the service routines forms the user interface. We have split the user interface in these two parts because of modularity; every pair of TLI and service routines can be combined into a user interface.

The following three sections each describe one of the blocks and the connection with other blocks.

3 Top level user interface

In section 3.1 we describe the communication between the TLI and the engine. Currently, there are two top level interfaces available. A command line interface (section 3.2) and a graphical user interface (section 3.3).

3.1 Communication

In this section we describe how the communication between the top level user interface and the Yarrow engine is implemented. From the viewpoint of the TLI, the engine is just a database to which queries can be sent, which will return a certain result. All possible queries are packed into a datatype called `Query`, and all possible results into `Result`. The only function from the engine available to the TLI is `doQuery`, which handles all queries. So the communication between the TLI and the engine can be visualized as in figure 2.

Since we work in a purely functional language, the engine does not own a state. How can the engine then change the context, for example? The answer is that `doQuery` is a state transformer. Concretely:

```
doQuery :: (Query,EngineState) -> (Result,EngineState)
```

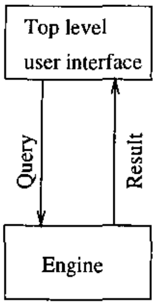


Figure 2: Communication

So a more precise way of viewing the information flow is that a pair `(query,engineState)` is sent to the engine, which responds with a pair `(result,engineState')`. It is the responsibility of the TLI that the next query is paired with this new `engineState'`.

Now we will consider the datatypes `Query` and `Result` in more detail. A stylized subset of the queries and results is depicted in figure 3, in total there are about 30 queries and 20 results. They are grouped around five subjects. The following list gives these subjects and one or two representative queries with their associated results.

1. Proof-tasks. The query `QProveVar (v,t)` starts a new proof-task, where `t` is the goal, and `v` is the name of the goal. The engine gives as result `RProofTask taskId toProve`. The variable `taskId` contains the identification of the proof-task (there may be several proof-tasks), and `toProve` contains all information about the proof-task, i.e. the proofterm, and all the subgoals. The query `QTactics taskId tacticTerm` performs the tactic associated with `tacticTerm` on proof-task `taskId`, with result `RTactic toProve`.
2. Loading and saving of modules (i.e. a group of related definitions). For example, the query `QLoadModule name contents` request the loading of module `name`. Since the engine cannot do any IO, the TLI also gives as parameter the contents of the file associated with `name`. Usually, this leads to the result `RModulesAre`, which indicates that loading is done, and gives the new list of currently loaded modules. If the module imports other modules, more communication is necessarily performed through queries and results. This is quite awkard.
3. The global context. The query `QDeclareVars` adds one or more declarations to the context, and `QGiveGlobContext` requests the current context. Both result in `RGlobContextIs`.
4. The parameters of the system. The query `QSetTypingSystem` asks for a change in PTS, and this is answered with `RTypingSystemOk`.
5. Calculation of normal forms or types. For example, `QGiveType t` requests calculation of the type of `t`, and `RTypeIs` returns `t` with its type and sort.

So each query is associated with a small number of results (usually one). Apart from that, the result `RError` is always allowed.

The big advantage of having only this narrow communication channel between the TLI and the kernel is modularity. This reveals itself in three ways:

- The connection between the engine and the TLI is very sharply defined.
- The TLI uses IO monads or fudgets (both with state) in order to perform IO. Since there is only one communication channel, the conversion between the ordinary functional types of the engine on one hand, and the IO monads or fudgets on the other hand, is limited to one place in the program.
- Since different queries can have the same sort of result, the output for these queries will be uniform.

```
data Query =                                 data Result =
  QProveVar (Vari,Term) |                      RProofTaskId TaskId ToProve |
  QTactic TaskId TacticTerm |                  RTactic ToProve |
  QLoadModule ModuleName String |              RModulesAre [ModuleInfo] |
  QDeclareVars ([Vari],Term) |                 RGlobContextIs GlobContext |
  QGiveGlobContext |
  QSetTypingSystem System |                    RTypingSystemOk |
  QGiveType Term |                             RTypeIs (Term,Term,Sort) |
  ...                                          RError ErrorMessage |
                                               ...
```

Figure 3: A subset of the queries and results

3.2 Command line interface

The actual implementation of the textual interface is not very interesting. Therefore we concentrate on how this top level user interface is used, which is quite similar to Coq and LEGO. We do this by a simple example.

Suppose the user is working on a proof of $\forall P, Q, R.\ (P \to Q \to R) \to (Q \to P \to R)$. After a few steps, the goal is to prove the following.

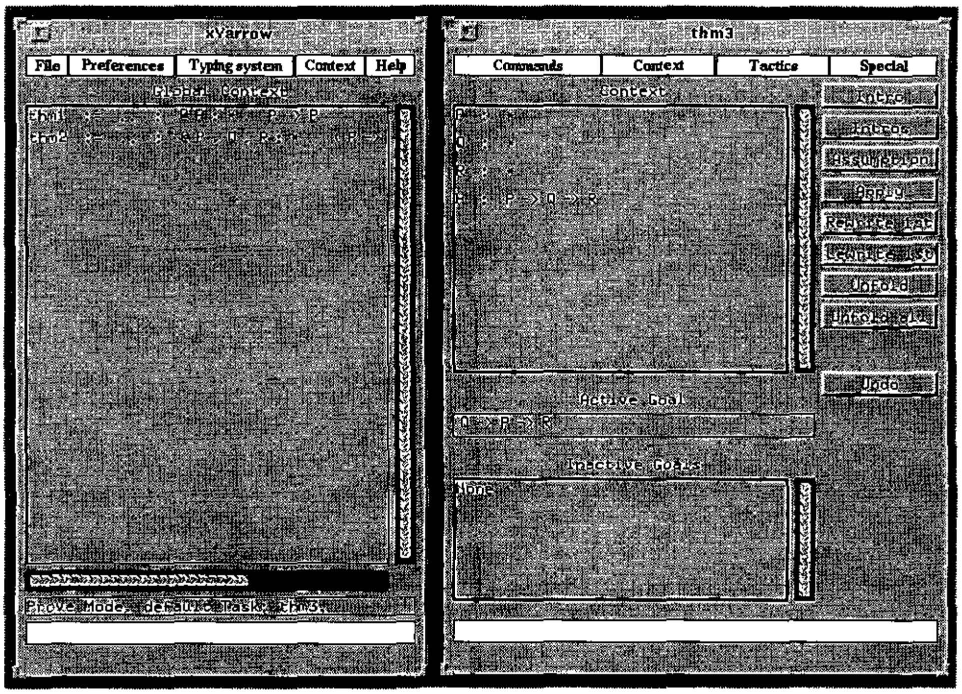
```
P : *
Q : *
R : *
H : P->Q->R
--------------------------------------------------
Q->P->R
```

Above the dashed line is the *local context* of the goal; these declarations and assumptions may be used to prove the proposition under the line. The user now types **intros** to perform the →-introduction tactic as often as possible.

```
P : *
Q : *
R : *
H : P->Q->R
H1 : Q
H2 : P
--------------------------------------------------
R
```

The →-elimination tactic on H is now invoked as follows.

```
$ apply H
2 goals

P : *
Q : *
R : *
H : P->Q->R
H1 : Q
H2 : P
--------------------------------------------------
P

2) Q
```



H : P -> Q -> R

Q -> R -> R

None

Prove Mode : default Task : thm3

Figure 4: A screen shot of the GUI

At this point there are two goals. The first goal is to prove P in the local context shown; the second goal is to prove Q, but its local context is not shown. (In this case it is the same context as for P). Here we abort the proof.

3.3 Graphical user interface

The Fudgets library [HC95] is used to implement the graphical user interface. Fudgets offers mechanisms to construct and combine windows, buttons, menus and other window gadgets. The implementation of the graphical user interface is described in detail in [Rai97].

We treat again the example given in section 3.2, but now in the graphical user interface. The starting point is shown in figure 4. There are two windows, one for the global context (on the left), and one for the proof we are working on (on the right). For this example, all actions take place in the latter. The main area of this window is divided into three parts. The top part shows the local context of the current goal, the middle part shows the current goal itself, and the bottom part shows the other goals. On the right-hand side of the window are several buttons for invoking commonly used tactics. A complete list of tactics can be found under the menu bar entries `Tactics` and `Special`. The far bottom of the window consists of a command line, where the user can type in commands, and a status line, which displays possible errors.

The user invokes the →-introduction tactic by clicking on the button labeled `Intros`. This results in an immediate change in the main area of the window. The user then selects the variable `H` in the local context by clicking on it, and clicks on the button labeled `Apply`. These actions form the →-elimination tactic. The same effect can be achieved in several other ways. First, by clicking `Apply` without having selected a term. This causes a pop-up window to appear, in which `H` can be typed in. Second, by selecting the apply tactic from the menu-bar entry `Tactics`. Third, by selecting `H`, summoning the pop-up menu, and selecting the `Apply H` option from this menu. Last, by typing `Apply H` in the command line on the bottom of the screen.

The graphical user interface has the usual advantages of GUIs over CLIs. One particular example for Yarrow is that unfolding one occurrence of a defined variable (e.g. in the goal) is achieved by clicking on this occurrence and clicking the `Unfold`-button. The user of Yarrow with the CLI has to count which occurrence he wants to unfold, and type in this number as parameter to the `unfold` tactic.

4 Flags

Yarrow has the ability to print proofs in the flag-style format [Ned90]. This is a formal notation for proofs which makes it clear which hypotheses are valid at each point in the proof. Every hypothesis is written in a box. Connected to this box is a "flagpole", which indicates the scope of this hypothesis. The justification for every proposition is written behind it; a justification typically consists of a logical construct (e.g. $\wedge$, $\forall$), a letter that indicates whether this construct is Introduced or Eliminated, and references to the lines or theorems which the current line depends on. For an example of this style, see figure 5, which proves in a certain context that the `insert` function keeps ordered lists ordered. We prefer this more formal notation to a textual presentation, because the flag-style format is clearer, more concise, and the propositions are not embedded within English "prose". A very similar notation is used in Jape [SB96], where this layout is used to build proofs interactively.

The algorithm that produces this presentation of a proof from a proof-object is quite similar to the ones described in [CKT95] and [Cos96], although they produce proofs in pseudo natural language. The basic algorithm is natural and simple: the presentation of a proof-term p is a composition of the presentations of the subterms of p. However, this produces quite lengthy proofs. A big improvement is the combination of similar steps into one step, e.g. in line 20 of figure 5, two steps are contracted into one ($\forall$E of line 9 with term b, and $\Longrightarrow$E of the result with line 19). Up to this improvement, the algorithm in [Cos96] and ours are similar. An important differnce is that their algorithm works for the Calculus of Inductive Constructions, whereas ours works for PTSs.

5 Conclusion

The main contribution of this paper is the design of Yarrow, that allows several user interfaces to be used with one engine. We decided to create a very narrow communication channel of queries and results, that are handled by one function of the engine. This approach is successful. It helps to separate the tasks of the total program, and allowed us to implement the user interfaces quite independently from the engine. Furthermore, it promotes uniformity, e.g. similar commands always give the same format of output. The strict adherence to this discipline made the implementation of a few commands with a lot of IO complicated, but most commands are well-suited to this approach.

Let us tell a bit more about our experiences in building and using the graphical interface. It was built using the Fudgets library, which allows easy construction of windows from components like buttons and text-fields. However, Fudgets offers only a very basic functionality; more fancy features like dragging are absent. But even without these features a GUI would be a big improvement over a CLI. However, when using the GUI, it turned out that the Fudgets library was rather slow and memory demanding, that it contained many bugs (i.e. resizing a window didn't work properly) and that certain essentail features were missing (i.e. saving a file was not implemented). To conclude our experiences, we think the concepts of the fudgets library made programming a GUI easy and enjoyable, but the library is not ripe enough to be used in a "real-world" application.

Another contribution of this paper is the presentation of proofs in the flag-style format, which is more formal than proofs in pseudo natural language. Future work could include interactive construction of proofs in this format, similar to Jape [SB96].

Yarrow with a command line interface is electronically available from the world wide web [Zwa97].

Figure 5: A long proof in flag-style

1	$\boxed{\mathtt{m} : \mathtt{Nat}}$	hyp
2	$\boxed{\mathit{Ordered}\ (\mathtt{nil\ Nat})}$	hyp
3	$\mathtt{insert\ m\ (nil\ Nat)} = \mathtt{singleton\ m}$	$\forall$E *insert_nil*
4	$\mathit{Ordered}\ (\mathtt{singleton\ m})$	$\forall$E *Ordered_singleton*
5	$\mathit{Ordered}\ (\mathtt{insert\ m\ (nil\ Nat)})$	$=\leftarrow$ 3,4
6	$\mathit{Ordered}\ (\mathtt{nil\ Nat}) \Longrightarrow \mathit{Ordered}\ (\mathtt{insert\ m\ (nil\ Nat)})$	$\Longrightarrow$I 2-5
7	$\mathtt{a} : \mathtt{Nat}$	hyp
8	$\mathtt{as} : \mathtt{List\ Nat}$	
9	$\mathit{Ordered}\ \mathtt{as} \Longrightarrow \mathit{Ordered}\ (\mathtt{insert\ m\ as})$	
10	$\mathit{Ordered}\ (\mathtt{a;as})$	
11	$(\forall \mathtt{b{:}Nat}.\ \mathit{Elem}\ \mathtt{b\ as} \Longrightarrow \mathtt{a} \leq \mathtt{b}) \wedge \mathit{Ordered}\ \mathtt{as}$	$\forall$E *Ordered_cons_*,10
12	$\forall \mathtt{b{:}Nat}.\ \mathit{Elem}\ \mathtt{b\ as} \Longrightarrow \mathtt{a} \leq \mathtt{b}$	$\wedge$EL 11
13	$\mathit{Ordered}\ \mathtt{as}$	$\wedge$ER 11
14	$\mathtt{m} \leq \mathtt{a} \vee \mathtt{a} < \mathtt{m}$	$\forall$E *Le_Or_Gt*
15	$\boxed{\mathtt{m} \leq \mathtt{a}}$	hyp
16	$\mathtt{insert\ m\ (a;as)} = \mathtt{m;a;as}$	$\forall$E *Le__insert*,15
17	$\mathtt{b} : \mathtt{Nat}$	hyp
18	$\mathit{Elem}\ \mathtt{b\ (a;as)}$	
19	$\mathtt{b} = \mathtt{a} \vee \mathit{Elem}\ \mathtt{b\ as}$	$\forall$E *Elem_cons_*,18
20	$\boxed{\mathtt{b} = \mathtt{a}}$	hyp
21	$\mathtt{a} \leq \mathtt{a}$	$\forall$E *Le_refl*
22	$\mathtt{a} \leq \mathtt{b}$	$=\leftarrow$ 20,21
23	$\boxed{\mathit{Elem}\ \mathtt{b\ as}}$	hyp
24	$\mathtt{a} \leq \mathtt{b}$	$\forall$E 12,23
25	$\mathtt{a} \leq \mathtt{b}$	$\vee$E 19,20-22,23-24
26	$\mathtt{m} \leq \mathtt{b}$	$\forall$E *Le_trans*,15,25
27	$\forall \mathtt{b{:}Nat}.\ \mathit{Elem}\ \mathtt{b\ (a;as)} \Longrightarrow \mathtt{m} \leq \mathtt{b}$	$\forall$I 17-26
28	$\mathit{Ordered}\ (\mathtt{m;a;as})$	$\forall$E *Ordered_cons*,10,27
29	$\mathit{Ordered}\ (\mathtt{insert\ m\ (a;as)})$	$=\leftarrow$ 16,28
30	$\boxed{\mathtt{a} < \mathtt{m}}$	hyp
31	$\mathtt{insert\ m\ (a;as)} = \mathtt{a;insert\ m\ as}$	$\forall$E *Gt__insert*,30
32	$\mathit{Ordered}\ (\mathtt{insert\ m\ as})$	$\Longrightarrow$E 9,13
33	$\mathtt{b} : \mathtt{Nat}$	hyp
34	$\mathit{Elem}\ \mathtt{b\ (insert\ m\ as)}$	
35	$\mathtt{b} = \mathtt{m} \vee \mathit{Elem}\ \mathtt{b\ as}$	$\forall$E *Elem_insert_*,34
36	$\boxed{\mathtt{b} = \mathtt{m}}$	hyp
37	$\mathtt{a} \leq \mathtt{m}$	$\forall$E *m_Lt_n__m_Le_n*,30
38	$\mathtt{a} \leq \mathtt{b}$	$=\leftarrow$ 36,37
39	$\boxed{\mathit{Elem}\ \mathtt{b\ as}}$	hyp
40	$\mathtt{a} \leq \mathtt{b}$	$\forall$E 12,39
41	$\mathtt{a} \leq \mathtt{b}$	$\vee$E 35,36-38,39-40
42	$\forall \mathtt{b{:}Nat}.\ \mathit{Elem}\ \mathtt{b\ (insert\ m\ as)} \Longrightarrow \mathtt{a} \leq \mathtt{b}$	$\forall$I 33-41
43	$\mathit{Ordered}\ (\mathtt{a;insert\ m\ as})$	$\forall$E *Ordered_cons*,32,42
44	$\mathit{Ordered}\ (\mathtt{insert\ m\ (a;as)})$	$=\leftarrow$ 31,43
45	$\mathit{Ordered}\ (\mathtt{insert\ m\ (a;as)})$	$\vee$E 14,15-29,30-44
46	$\forall \mathtt{a{:}Nat}.\ \forall \mathtt{as{:}List\ Nat}.\ (\mathit{Ordered}\ \mathtt{as} \Longrightarrow \mathit{Ordered}\ (\mathtt{insert\ m\ as})) \Longrightarrow$ $\mathit{Ordered}\ (\mathtt{a;as}) \Longrightarrow \mathit{Ordered}\ (\mathtt{insert\ m\ (a;as)})$	$\forall$I 7-45
47	$\forall \mathtt{l{:}List\ Nat}.\ \mathit{Ordered}\ \mathtt{l} \Longrightarrow \mathit{Ordered}\ (\mathtt{insert\ m\ l})$	$\forall$E *indlist*,6,46
48	$\forall \mathtt{m{:}Nat}.\ \forall \mathtt{l{:}List\ Nat}.\ \mathit{Ordered}\ \mathtt{l} \Longrightarrow \mathit{Ordered}\ (\mathtt{insert\ m\ l})$	$\forall$I 1-47

Acknowledgements

Many thanks to Kees Hemerik for his stimulation and help in writing this paper, and to Eric Raijmakers for implementing the graphical user interface. The author also thanks the anonymous referees for their useful suggestions.

Note

This is an extended abstract of an article submitted to the Journal of Functional Programming.

References

[Bar92] H. Barendregt. Lambda calculi with types. In D. M. Gabbai, S. Abramsky, and T. S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1992.

[CKT95] Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. In *Typed Lambda Calculi and Applications: Proceedings of the Second International Conference*, volume 902 of *LNCS*, pages 109–123. Springer-Verlag, Berlin, New York, 1995.

[Coq97] Coq. The Coq proof assistant. In *URL:* http://pauillac.inria.fr/coq, 1997.

[Cos96] Yann Coscoy. A natural language explantation for formal proofs. In *Logical Aspects of Computational Linguistics: Proceedings of the First International Conference*, volume 1328 of *LNAI*. Springer-Verlag, Berlin, New York, 1996.

[Hal97] Thomas Hallgren. Alfa home page. In *URL:* http://www.cs.chalmers.se/~hallgren/Alfa/, 1997.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In *Spring School on Advanced Functional Programming in Båstad.* Springer-Verlag, 1995. LNCS 925, see also [HC97].

[HC97] Thomas Hallgren and Magnus Carlsson. The Fudgets home page. In *URL:* http://www.cs.chalmers.se/Cs/Research/Functional/Fudgets/, 1997.

[LP92] Zhaohui Luo and Randy Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundation of Computer Science, 1992.

[Ned90] R.P. Nederpelt. Presentation of natural deduction. In *Symposium: Set theory, foundations of mathematics*, Recueil des travaux de l'Institut Mathématique, Nouv. série, tome 2 (10), Beograd, pages 115–125, 1990.

[Pet97] John Peterson. The Haskell home page. In *URL:* http://haskell.org/, 1997.

[Pol93] Erik Poll. A typechecker for bijective pure type systems. Technical Report 93-22, Eindhoven University of Technology, 1993.

[Rai97] Eric Raijmakers. A graphical user interface for the proof assistant Yarrow. Master's thesis, Eindhoven University of Technology, 1997.

[SB96] Bernard Sufrin and Richard Bornat. Jape - a framework for building interactive proof editors. In *URL:* http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/UNIXJAPEDOCHTML/jape.html, 1996.

[Tho96] Simon Thompson. *Haskell, The Craft of Functional Programming.* Addison-Wesley, 1996.

[Zwa97] Jan Zwanenburg. The Yarrow home page. In *URL:* http://www.win.tue.nl/cs/pa/janz/yarrow/, 1997.

Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61
96/14	S.H.J. Bos and M.A. Reniers	The $I^2 C$-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.
96/18	P.D.V. van der Stok	Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
96/19	M.A. Reniers	Static Semantics of Message Sequence Charts, p. 71
96/20	L. Feijs	Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
96/21	L. Bijlsma and R. Nederpelt	Predicate calculus: concepts and misconceptions, p. 26.
96/22	M.C.A. van de Graaf and G.J. Houben	Designing Effective Workflow Management Processes, p. 22.
96/23	W.M.P. van der Aalst	Structural Characterizations of sound workflow nets, p. 22.
96/24	M. Voorhoeve and W. van der Aalst	Conservative Adaption of Workflow, p.22
96/25	M. Vaccari and R.C. Backhouse	Deriving a systolic regular language recognizer, p. 28
97/01	B. Knaack and R. Gerth	A Discretisation Method for Asynchronous Timed Systems.
97/02	J. Hooman and O. v. Roosmalen	A Programming-Language Extension for Distributed Real-Time Systems, p. 50.
97/03	J. Blanco and A. v. Deursen	Basic Conditional Process Algebra, p. 20.
97/04	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time, p. 26.
97/05	J.C.M. Baeten and J.J. Vereijken	Discrete-Time Process Algebra with Empty Process, p. 51.
97/06	M. Franssen	Tools for the Construction of Correct Programs: an Overview, p. 33.
97/07	J.C.M. Baeten and J.A. Bergstra	Bounded Stacks, Bags and Queues, p. 15.

97/08	P. Hoogendijk and R.C. Backhouse	When do datatypes commute? p. 35.
97/09	Proceedings of the Second International Workshop on Communication Modeling, Veldhoven, The Netherlands, 9-10 June, 1997.	Communication Modeling- The Language/Action Perspective, p. 147.
97/10	P.C.N. v. Gorp, E.J. Luit, D.K. Hammer E.H.L. Aarts	Distributed real-time systems: a survey of applications and a general design model, p. 31.
97/11	A. Engels, S. Mauw and M.A. Reniers	A Hierarchy of Communication Models for Message Sequence Charts, p. 30.
97/12	D. Hauschildt, E. Verbeek and W. van der Aalst	WOFLAN: A Petri-net-based Workflow Analyzer, p. 30.
97/13	W.M.P. van der Aalst	Exploring the Process Dimension of Workflow Management, p. 56.
97/14	J.F. Groote, F. Monin and J. Springintveld	A computer checked algebraic verification of a distributed summation algorithm, p. 28
97/15	M. Franssen	λP-: A Pure Type System for First Order Loginc with Automated Theorem Proving, p.35.
97/16	W.M.P. van der Aalst	On the verification of Inter-organizational workflows, p. 23
97/17	M. Vaccari and R.C. Backhouse	Calculating a Round-Robin Scheduler, p. 23.
97/18	Werkgemeenschap Informatiewetenschap redactie: P.M.E. De Bra	Informatiewetenschap 1997 Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie Informatiewetenschap, p. 60.
98/01	W. Van der Aalst	Formalization and Verification of Event-driven Process Chains, p. 26.
98/02	M. Voorhoeve	State / Event Net Equivalence, p. 25
98/03	J.C.M. Baeten and J.A. Bergstra	Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.
98/04	R.C. Backhouse	Pair Algebras and Galois Connections, p. 14
98/05	D. Dams	Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.
98/06	G. v.d. Bergen, A. Kaldewaij V.J. Dielissen	Maintenance of the Union of Intervals on a Line Revisited, p. 10.
98/07	Proceedings of the workhop on Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98) June 22, 1998 Lisbon, Portugal	edited by W. v.d. Aalst, p. 209