Interactive Theorem Proving: An Empirical Study of User Activity

J. S. Aitken, P. Gray, T. Melham and M. Thomas
Department of Computing Science, University of Glasgow, Scotland

21 November 1995

Abstract

In this paper the interaction between users and the interactive theorem prover HOL is investigated from a human-computer interaction perspective. First, we outline three possible views of interaction, and give a brief survey of some current interfaces and how they may be described in terms of these views. Second, we describe and present the results of an empirical study of intermediate and expert HOL users. The results are analysed for evidence in support of the proposed view of proof activity in HOL. We believe that this approach provides a principled basis for the assessment and design of interfaces to theorem provers.

1 Introduction

In the most general terms, mechanised theorem proving is about using computers to justify the elevation of a conjecture into a theorem by finding a formal proof—or at least by convincing oneself that such a proof exists. In the algorithmic tradition, this means employing a computer program to determine automatically the truth of a proposition by means of a mathematically-justified decision procedure or some more heuristic method. The main elements here are computation over symbolic data representing propositions and the automated search for a legitimate proof in some space of possible proofs.

In the more interactive tradition, the user interacts with a computer program in order to participate in proof discovery and construction. While the main elements still include computation over symbolic data representing propositions, as in the algorithmic tradition, the concept of *interaction* between user and system during the search for legitimate proof becomes central.

Interactive theorem provers and proof-assistants have been developed for a wide range of logics, styles of reasoning, and applications; some examples include HOL, Isabelle, LP, Mural, MERRILL and NuPrl. A commonly-cited difficulty with the use of interactive provers is associated with the key feature of any interactive system, namely the user interface. A poor interface obstructs the interaction between the user and the system. As a result, the central and distinctive concept of this approach to theorem proving, the interactive element, is not fully realised. In response to this, several projects have been undertaken to develop good, usually graphical, interfaces for specific systems. Many interfaces offer a bewildering variety of options such as menus, windows, mouse, click and drag, structure editors and proof trees. While the interface designers have clearly considered their interfaces carefully, the principles for design are seldom explicitly formulated; and even when they are, the evidence for those principles is lacking.

As a consequence, the resulting interfaces have met with mixed success and many interface problems remain unsolved. The user and the designer usually have no other criteria for evaluation of an interface than anecdotes and experience. More specifically, the interface designers have almost universally failed to draw upon the most relevant discipline—namely, *human-computer interaction*—in the analysis of the task structure and information flows between users and the proof system (Hewitt *et al.*, 1992).

Our long term goal is to rectify this lack of analysis and to produce design principles for interactive theorem provers based on the results of task-oriented analysis and empirical investigation of user activity. In this paper we present some preliminary results of our investigations. We present our thoughts on a framework for information flow and the results of an empirical study of intermediate and expert users of one particular interactive theorem prover, the HOL system (Gordon and Melham, 1993).

The HOL system is a LCF-style interactive theorem prover for constructing (or discovering) proofs in higher order logic. The LCF approach, devised in the late 1970s by Robin Milner, means that theorems are represented by the values of an abstract data type in a strongly-typed functional programming language. In the case of HOL, this is the language ML (Paulson, 1991). Theorem proving in HOL takes place by executing ML functions that operate on theorems. These functions may be primitive inference rules or more complex, user-defined ML functions. In all cases, however, the LCF approach ensures that functions can be constructed only so as to perform valid logical inferences.

The HOL system has been chosen for initial experiments because of our own familiarity with it, the system's large and active user community, and the fact that the interface problem has been signalled by the construction of several graphical interfaces for it. But this does not preclude the consideration of other interactive theorem provers in our future work.

The structure of this paper is as follows. We begin in Section 2 with some background material on LCF-style goal-directed proof and proof trees. In Section 3 we propose a three level model of interaction and within that model, discuss three styles, or 'views', of interaction between users and proof systems: proof as programming, proof by selection, and proof as structure editing. In Section 4 we give a brief overview of some current HOL interfaces, noting the views which are supported by each interface. In Section 5 the experimental method we use to evaluate user activity in theorem proving is presented; this is followed by the results of our experiments and an analysis of them. In the final section we draw some conclusions and outline our plans for future work.

2 Goal-directed Proof Search

The most primitive notion of formal proof is one in which rules of inference are simply applied in sequence to axioms and previously proved theorems until the desired theorem is obtained. Computations that achieve this so-called *forward proof* process are the ultimate basis for all logical deduction in LCF-style theorem provers, including HOL. In the LCF tradition, one performs forward proof by executing a program that invokes the appropriate sequence of inference rules. Thus, for example, a decision procedure is just an ML program that invokes the appropriate sequence of inference rules to prove any desired member of a well-defined and general class of conjectures.

Decision procedures operate without the need for user intervention, but forward proofs may also be performed *interactively*. The user specifies to the theorem prover which inference rule is to be applied at each step, obtaining feedback by observing the theorems that are generated by the rules. The machine acts as a proof-checker, but gives little support for proof discovery. This is often not a feasible way of finding a proof, since the exact sequence of inferences required—or even the first inference required—is rarely known in advance.

A more promising and natural approach is to set about discovering a proof by working backwards from the statement to be proved (called a *goal*) to previously proved theorems that imply it. This is the *backward proof* style, in which the search for a proof is the activity of exploring possible strategies for achieving a goal. For example, one possible approach to proving a conjunctive formula $P \wedge Q$ is to break this goal down into the two separate subgoals of proving P and proving Q. Likewise, one may seek to prove an implication $\forall x.\, P[x] \supset Q[x]$ by reducing this to the subgoal of proving $Q[x]$ under the assumption $P[x]$ for arbitrary x.

The HOL system, following LCF, supports this style of proof by means of ML functions called *tactics*. These are used to break goals down into increasingly simple subgoals, until the subgoals obtained follow immediately from theorems already derived. In addition to breaking a goal down into subgoals, a tactic also constructs a sequence of forward inference steps which can be used to prove the goal, once the subgoals have themselves been proved. This is necessary because all theorems must ultimately be obtained by forward proof.

In addition to tactics, ML allows one to implement functions (called *tacticals*) that combine elementary tactics together into more complex ones. This allows the user to build composite tactics that fully decompose a conjecture into immediately-provable subgoals, and hence can be executed to generate a complete proof of the conjecture. In practice, these monolithic, composite tactics are the main products of the theorem proving activity; the software deliverable for a theorem proving project using HOL largely consists of files of ML source text that execute composite tactics of this kind, generating a body of theorems related to the problem domain at hand.

The most primitive interface to HOL is just an ML interpreter offering only the facility to evaluate the application of tactic functions to goals. In practice, however, interactive goal-directed proof is supported by means of an interface that operates on a *proof state* which records the history of the decomposition of a conjecture into subgoals (including the behind-the-scenes forward proofs that justify the decompositions) together with the subgoals remaining to be proved. The most rudimentary interface is still a version of the ML interpreter, but one in which a suitable proof state has been implemented by an ML data structure and some associated functions for modifying and inspecting it.

The proof state can be viewed abstractly as a *goal decomposition tree*. This is often referred to as a *proof tree*—it being understood that the steps in the 'proof' are goal decompositions in a backward proof attempt. A proof tree represents successive stages in the decomposition into subgoals of a conjecture to be proved. The root is the original conjecture, and the leaves are the subgoals remaining to be proved. A goal-directed proof attempt is successful when all the leaves are immediately provable. The theorem prover provides a means for dismissing such trivial subgoals.

A simple example of a proof tree is shown in Figure 1. The initial goal is a conjunction. In the first decomposition, this is split into two subgoals consisting of the two conjuncts. The existential subgoal $\exists x.\, P[x]$ is then reduced to the subgoal of proving $P[w]$ for some witness value w. The other subgoal $\forall y.\, P[y] \supset Q[y]$ is reduced to the subgoal of proving that $Q[y]$ follows from the assumption $P[y]$ for arbitrary y.

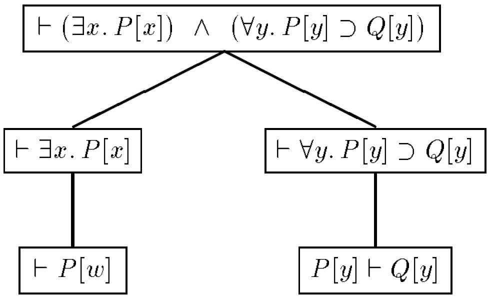

Figure 1: A simple proof tree

The notion of a proof state—the current state of a goal-directed backward proof attempt—lies at the core of interface support for goal-directed proof. The proof state is not a primitive HOL concept; the means by which proof states are interactively created, modified, and inspected are supplied by the implementation of an interface. Support for goal-directed proof in HOL is provided by the *subgoal package*, a collection of ML functions that operate on a data structure representing the proof state.

To complete a goal-directed proof, it is ultimately necessary to prove all the subgoals that arise. It is generally the case that subgoals are independent of each other and as a result the order in which subgoals are proven is not significant. This suggests that there is no real benefit to be gained by switching attention from branch to branch of the search tree. The search tree can simply be explored in a fixed depth-first order. Furthermore, it is not necessary to consider the proof as a whole; only the current subgoal need be considered. The current proof context can be indicated by a record of the proof steps applied so far. A record of past subgoal states is not necessarily the best indication of proof context. The overall shape of the proof *is* important, and a review of the progress of the proof may take it into consideration.

3 Describing Proof Behaviour

A number of theories and assumptions about user-proof system interaction underlie the design of existing interfaces. In many instances these ideas are not articulated explicitly. In descriptions of interface designs, the designer's views of the interaction are not always easily distinguished from other issues, such as system architecture, or from representational issues peculiar to a particular logical framework. In this section we aim to formulate precisely and explicitly certain views of user-proof system interaction which are relevant to the construction of effective user interfaces for interactive theorem-provers.

Cooperative, interactive theorem proving, like most human-computer (inter)activity, can be described at several different levels of abstraction. The very same user action, at a certain point in the proof, can be described as:

- – Choosing a tactic for application.
- – Selecting an item from a list.
- – Clicking the mouse button with the cursor at a particular location on the display.

All are 'correct' and complete descriptions of the same action, but each is framed in terms of a different model (set of objects and applicable operations) of the activity.

These descriptions of the activity are related to one another in at least two important ways. First, the relationships can form an *explanation* of user activity. That is, each level except the top level[1] can be viewed as an explanation of the level below it. Thus, the user clicks the mouse at a certain point in order to choose an item from a list, and s/he selects the item in order to choose a tactic for application. Second, from a design point of view, a designer must choose a representation (or choose not to offer a representation) in a lower level for the objects and operations in the next most abstract level. Thus, if selecting a tactic for application is a desired user action to support, then selecting an item from a list is one of the possible ways to effect this, and similarly, clicking the mouse button at a certain point is a way to effect the menu selection.

Such a multi-level view of interaction has been used as the basis for previous explanations of human-computer interaction, albeit never applied specifically to the domain of interactive theorem-proving. Nielson (1986) presents a linguistic account of interaction by asserting that the levels correspond to the lexical, syntactic and semantic levels of linguistic activity. Norman (1988) asserts that translations from one level to another form the basis for important potential interaction problems (the so-called gulfs of execution and evaluation) which arise when a user is not able to perform the transformation from one level to another. This characterisation of interaction forms the basis for an explanation of the potential advantages of direct manipulation (Hutchins, Holland and Norman, 1986) (viz., it reduces the complexity of the inter-level mappings) and metaphorical representations (viz., mappings can be inferred based on a similarity relation to a separate known representation).

How many levels of abstraction are necessary to characterise an interaction? This will depend upon the purpose and nature of the explanation or design rationals which the characterisation is used to support (Pylyshn, 1986). We have found that for a design-oriented description of user interfaces to theorem-provers it is sufficient to use three levels:

A logical level. This is a description solely in terms of logical concepts.

An abstract interaction level. At this level are the shared objects and operations in terms of which information is communicated from user to system. Typically, these are visual(isable) objects and the operations upon them, but abstracted away from details of their physical form. Examples include diagrams, structured text, visualised lists.

A concrete interaction level. At this level are actions on input devices and the perceptual characteristics of display objects.

Each level of abstraction is self-contained, in the sense that a full description of the activity can be framed within a level. One can instruct a user to carry out a full proof entirely in terms of articulations of devices. Or, assuming that the user knows, can guess, or learn the representation, the user can be instructed in terms of the higher level abstractions.

But a full description, even at a given level, must include operations which extend beyond those in which there is a flow of information between the parties to the interaction. That is, there are operations which belong solely to one party or the other and are needed to form

[1] The top level description forms the motivational basis of the activity. That is, goals at this level are not open to further explanation. To move beyond this level is to look for the motivation of the user to engage in the activity.

a complete picture of the activity. In our example, the user must have determined that the tactic to be chosen is the appropriate one at the current state of the proof. This determination may be an entirely private, cognitive activity. Similarly, once the tactic has been specified to the computer system, its application is an internal computer action requiring no intervention on the part of the user.

There is a class of user cognitive activity which, although not strictly part of the user-system interaction as described so far, is an important part of our analysis. User goals and plans determine the form of the interaction and the outcomes of interaction fed back into subsequent plan formulation. We take no theoretical stance on the form of such cognitive activity or the mental representation of cognitive structures (mental models), but we are concerned to discover user plans and beliefs which influence and are influenced by the interaction.

It is our contention that developments in user interface design for interactive theorem provers have been driven largely by considerations at the level of the concrete interaction domain and its relationship to the abstract interaction domain. These are often called styles or techniques of interaction and include such components as particular visual representations of tree structures, drag-and-drop interaction techniques, pop-up menus, and so on.

Relatively unexamined is the relationship between the abstract interaction domain and the logical domain. Our work to date has focused on this relationship. In the same way that representation choices at the lower level are grouped as styles, one can categorise the ways that the top level, logical, domain is related to its representation in the abstract interaction domain. We call these categories *views*. Views determine what in the logical domain will be represented, how it will be represented (at least in part), and place constraints on the nature of the relationship.

It should be emphasised that neither the notion of views, nor our multi-level account of proof construction activity, imply that the user is aware of the mediating relationship. A user might think of the proof activity, and develop action plans, entirely in terms of a description in the logical and/or abstract interaction domain. We follow Nardi and Zarmer (1993) in believing that the abstract interaction domain—the shared representation of user and system—forms a structure or *framework* in terms of which to solve problems in the logical domain. Views provide the organising principles by which such a framework can be generated. The view determines:

1. The aspects of the logical domain which should be made salient to the user.
2. The relevant operations in the logical domain which should be performed in the abstract interaction domain.
3. Constraints on action and properties of the logical domain which either need not or should not be represented.

In examining current user interfaces to theorem provers, we have identified three such views, although others are possible. They are *proof as programming*, *proof by pointing*, and *proof as structure editing*. We now turn to an examination of these views.

3.1 Proof as programming

The *proof as programming* view stems from two fundamental elements of the LCF approach to theorem-proving: the system's command language is a strongly-typed programming language, and an abstract data type of formal *theorems* in this language is used to distinguish formulas that have been proved from arbitrary propositions. The merit of this scheme is that the type discipline ensures security; theorems can be generated only by using the functions that implement primitive inference rules and programs that call them. A consequence is that, whatever code a user may write intending to implement a particular proof strategy, the system can never perform an invalid logical inference.

In LCF-style systems, backward proof is implemented using tactics—higher order functions that map goals into subgoals together with proof procedures that justify this decomposition. Historically, backward proof using tactics is the chief method of proof in LCF-style systems; indeed, for many users it is virtually the only method of proof. But tactics are not the only way of generating theorems; with the security of the LCF approach comes the freedom to write arbitrary code intended to compute a desired theorem. One could, for example, code a decision procedure that operates purely by forward proof (e.g. by equational rewriting, or first-order resolution) and can be invoked to generate any element of a whole class of theorems. In this case, the user is clearly engaged in an activity having the nature of programming—writing programs that compute theorems.

The view that proof is programming also regards tactic proofs as programs to compute theorems. The user develops tactic proofs piecemeal, by applying individual tactics that break down the goal into ever simpler subgoals. But once the proof has been found, the tactics are composed into a monolithic and complete proof strategy—a functional program that can be executed to prove the desired theorem from scratch. The final product of the activity is a program; and thus goal-directed theorem proving using tactics is a specialised kind of programming activity.

Thus, proof as programming recasts the proof problems, as defined in the logical domain, as programming problems. Figure 2 sketches the levels of abstraction making up the proof as programming view. As can be seen, construction and execution of program *texts* is a fundamental organising concept of this view. Tactics and the functions that combine them are the medium through which the user interactively explores possibilities and constructs a proof. But they also constitute program texts, which are executed to extend the proof state and are kept as a permanent record of the proof.

It does not follow from the above that users think of proof construction as programming. Rather, one would expect that, given a successful user interface, users would be relatively unaware of writing and executing parts of a program. This is an important point. *Progamming is not a metaphor for proof construction; instead, programming is the medium through which proofs are constructed and expressed.* One would expect that a descent from thinking at the logical level to planning and evaluating at the abstract interaction level would occur if there is a breakdown in the interaction (i.e., some required information is not available or an operation not known). An advantage of proof as programming is the fact that when breakdown occurs because of the lack of a representation in the abstract interaction domain (e.g. there is no tactic which represents some desired goal decomposition), one can 'bridge the representational gap' without switching abstract interaction domains.

It is not, of course, clear that the idea of programming is to be extended to the goal-directed process of proof discovery itself. Nonetheless, the analogy is one possible explanation of the activity that relates the user's notion of the proof strategy and the abstract interaction domain concepts of proof states and tactics. Hence this view has implications for interface evaluation and design—how well does proof as programming explain what really goes on, and does designing an interface consonant with this view increase productivity?

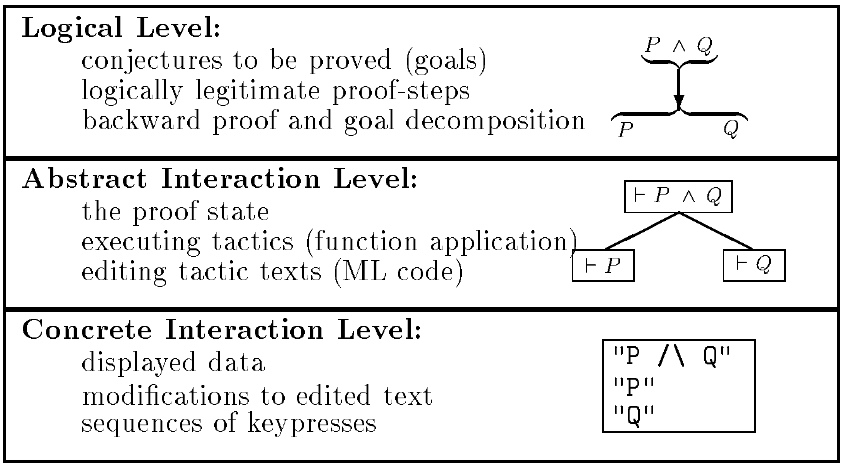

Figure 2: Proof as Programming

3.2 Proof by pointing

Proof by pointing (or, perhaps more accurately, proof by selection) has been proposed by Berot, Kahn and Théry (1993) as a means of synthesizing commands to the theorem prover by selecting a subexpression of the current goal (typically, using a mouse). It has been shown that in the Sequent Calculus, an appropriate sequence of rules drawn from the rules for elimination and introduction of logical connectives can be selected by identifying a particular subexpression on the left or right hand side of the turnstile symbol $\vdash$. For example, pointing at '$P(a)$' in the disjunctive hypothesis of the goal $P(a) \vee Q(a) \vdash R$ is interpreted as a desire to do a case analysis, and consequently two new subgoals are introduced, one for each of the two disjuncts. Selecting a subterm not governed by the outermost connective is interpreted as a desire to bring that term to the outside. In doing so, the prover may apply several inference rules and may be able to solve any simple subgoals that arise.

In tactic-based provers, proof by pointing is intended to free the user from having to edit commands during goal-directed proofs (Bertot, 1994). The user may then concentrate directly on the goals and theorems of the proof. The proof by pointing tool in the Coq theorem prover deals with the logical connectives $\wedge$, $\vee$, $\supset$, $\neg$, $\forall$ and $\exists$. If a subexpression of a premise is selected and the premise is existentially quantified, then a command which produces a witness for this formula will be generated. If the premise is universally quantified then a command which generates an instance of the formula will be produced. In a similar fashion, commands that break up conjunctions and introduce case splits will be generated according to a convention.

For simple logics, proof by pointing provides a complete proof technique that can be executed by purely mouse-based input operations. For the logics of most tactic-based provers, however, pointing can replace only some typed commands. Certain sequences of tactics can be synthesised from pointing actions, but for others some different kind of input may be required—perhaps a selection from a list of several possible actions, or even some text. This is because there may be no clear one-one correspondence between logical-level maneuvers and the expressions at which one may possibly point.

The proof by pointing idea can, however, be extended by connecting the application of a tactic with the pointing action. This yields the so-called 'point and shoot' technique. Suppose that `Elim` is the induction proof command. Then the actions 'point' and 'select `Elim`' introduce an induction scheme that would validate the propositional expression pointed to.

In proof by pointing, the logical level appears to be identical to that in proof as programming. But at the abstract interaction level, there are no program texts and no program execution. There is only a proof state, which is transformed by acts of selection—choosing an operation to perform on the current goal.

3.3 Proof as structure editing

Some logical formalisms include an object-language notion of proof. For example, in type theory (Martin-Löf, 1984) one has the 'propositions-as-types' reading, in which one views a type A as a proposition and a well-typed term $a :: A$ as a proof of A. Here, the logical term a is a formal proof object—a syntactic object, with an underlying syntactic structure related to the structure of the proposition A. This makes possible the view that proof construction and exploration consist in editing proof objects using a structure editor.

One theorem-proving system that exemplifies the proof as structure editing view is the ALF proof editor (Nordström, 1993). In ALF, the process of proving a conjecture A consists in building a proof object for A. The proof state has two components: the goal A, together with an incomplete proof object that represents the current state in the process of proving A. The intent is that a proof object is built up by direct manipulation—i.e. some form of structure editing. An incomplete proof object is a structure with one or more placeholders indicating positions in the structure for those parts of the object yet to be created. A typical editing operation extends an incomplete proof object by filling in a placeholder with a proof object, perhaps itself incomplete.

Proof as structure editing can also take place when the notion of proof is meta-linguistic, where proofs are not objects in the term language of the logic. For example, the Mural system (Jones *et. al.*, 1991) supports essentially the same 'natural deduction' style of proof as HOL, but using a structure editing interface for proof construction and display. This is in keeping with one of the general principles adopted by the designers of Mural, namely to allow 'direct manipulation' of objects wherever possible. This leads to extensive use of structure editors, not just for proof construction, but for the construction and editing of many other objects as well.

In proof as structure editing, the abstract interaction level is organised around a proof state which includes proof objects. The proof proceeds by structure editing of these proof objects. We suspect that this view leads to a different perspective at the logical level, but we have not yet investigated the issue in depth.

4 Some Interfaces to HOL

In this section we describe three interfaces to HOL, with reference to the views described above. All three adopt the proof as programming view to a greater or lesser extent. Two are X-windows based interfaces, XHOL and CHOL, and extend the basic emacs interface to HOL by providing a proof tree display and a simple structure editing facility respectively.

We begin by describing a proposal to extend support for proof as programming without necessarily modifying the visual presentation of HOL.

4.1 Extending support for proof as programming

Slind and Prehofer (1994) advocate that users of verification systems be given support in three areas: formalisation, proof and interface. These cover the entire range of verification activities, from the formalisation in logic of the domain of interest, through to the proof procedure and storing the product of the proof attempt. They do not aim to provide a distinct new interface, but to extend the ML environment in useful ways. Such extensions would provide the user with a more supportive programming environment, and hence this work can be classified under the proof as programming heading.

When discussing the interface, Slind and Prehofer note the problems of the presentation of long and complex expressions. Their recommendations for interface design include modifying certain technical aspects of the ML interface to HOL, for example the overloading of constants. They also discuss the use of prettyprinting facilities and the possibility of using full scale document preparation systems for the presentation of expressions. They endorse the argument that the visualisation of theories (and their relationships) increases modularity, which must aid the development of large proofs.

Slind (1994) describes a proof manager whose role is to handle the complexities of a large proof attempt which involves a collection of proofs. This is to be achieved by introducing *notes*, a form of documentation originally due to Kalvala (1991, 1994), into the ML data structures. The basic approach to discovering a proof using the proof manager is depth-first search, although forward proof is also permitted. While a proof attempt is being made, the user makes textual notes that comment the proof, perhaps describing the reasons for particular decisions. In developing a collection of proofs, the user makes notes on entire proof attempts.

The interface should be capable of presenting information in multiple forms, appropriate to the context. The interface should, presumably, support the informal documentation of proofs and maintain the appropriate links between notes.

4.2 XHOL: An interface with a proof tree display

The XHOL interface to HOL has been developed by Schubert and Biggs (1994). This provides a four-panelled display, including a window where a standard HOL session runs, an emacs-style window which displays the proof script, and a window displaying a proof tree. The user can type tactics into the editor window or select tactics from a menu and these appear in the editor window. The highlighted region of the proof script can be sent to the HOL session by clicking on a 'send' button. This replaces the typical cut and paste method of transferring the tactic text from the editor to HOL. XHOL provides a means of automatically constructing a new tactic as the concatenation of a sequence of tactics. The operation is termed 'tactic extraction'. Once extracted, a tactic sequence can easily be re-applied to a new problem. This facility aids the user at the abstract interaction level, and is clearly within the proof as programming view. This interface modifies many features at the concrete interaction level, while remaining similar to the emacs interface at the surface level.

XHOL displays the subgoal structure of the current proof as a tree. A tree is a natural representation of the decomposition of a goal into subgoals. In backwards proof, a proof tree shows what has been proven and what subgoals remain to be proven. It has been claimed (Schubert and Biggs, 1994) that such a display provides clues about the techniques and tactics that may be useful in completing the proof attempt.

Schubert and Biggs are concerned with the needs of large proof development and with speeding up the user-proof system interaction. In order to cope with the former, their HOL interface provides a means to view only part of the proof tree (which may become very large) and to control how much information is displayed at each node of the tree. Speeding up the interaction is addressed by providing a menu of tactics which the user may select from, with a view to saving the time required to type complex commands.

The rationale for the proof tree presentation appears to be that it is useful, during the proof attempt, to view the proof as a whole. It must also be assumed that a display of goal states is the most appropriate way to display the proof. Evidence for the usefulness of examining the proof as a whole has not been presented. In addition, no justification has been given for the decision to present the goal states and not to present the tactics which transform one state into another, or to label the branches of the proof tree in some appropriate way, e.g. with the case argument (P and $\neg P$ in a case split) or with the name of a witness in an existence proof.

The idea of presenting the proof attempt as a tree is intuitively appealing, but requires scrutiny. There are practical problems in the use of proof trees as in long proofs many nodes are generated and their contents may be large expressions. This appears to work against the aim of providing support for large scale proof development.

4.3 CHOL: A Centaur interface to HOL

CHOL (Théry, 1993) is also an X interface to HOL. The proof script window of the standard interface is retained, but its functionality is altered so that tactics may be entered by editing or by menu selection and placeholders may be left in the tactic script. For example, after introducing `INDUCT_TAC` (which would usually be followed by two arguments) it is possible to leave the first argument blank, and to fill in the second argument. This is equivalent to solving the step case of the induction first, and then proceeding to solve the base case. The view here is no longer proof as programming, where the same effect could be achieved by rotating the subgoals (i.e. using an ML command to swap the subgoal order). A structure editing view is required to explain this style of interaction.

CHOL does not present the user with a window to the HOL session; instead, the current goal is displayed in a 'goal window'. If the proof appears to require a rewriting step, CHOL has the capability to calculate all possible rewrites of a selected expression and to present these as a menu to the user (Théry, Bertot and Kahn, 1992). The user can then select the required rewrite and CHOL will update the goal state. The user can select the theories which are used to calculate the possible rewrites, and hence, exercise control over this process. The user is given visual feedback as to the tactic which effected the particular rewrite s/he selected, as the tactic, including the theorem name, appears in the script. This treatment of the rewrite operation has much in common with the proof by pointing view. However, as the user is also informed about which tactic was used, the proof as programming view is maintained.[2]

At the concrete interaction level, there is little similarity to the standard HOL interface, hence a learning overhead for all users is inevitable. CHOL also provides assistance to the user in searching the contents of the theory libraries for theorems and definitions. This activity can be very time consuming for new users and also for those unfamiliar with a particular library.

[2]The automatic insertion of the tactic in the script could also be consistent with the structure editing view: the insertion being automatic instead of manual.

4.4 Changing views

We see from the discussion above, that the CHOL interface does not exclusively adhere to the proof as programming view. This raises the interesting question as to whether users can easily move from the dominant view, which we believe for HOL is proof as programming, to one or both of the other views utilised in CHOL.

However, prior to being able to investigate whether users can make use of different views of their activity, we should validate the proposed view of proof as programming. In the following section, we present an empirical investigation into the use of the standard HOL interface.

5 An Empirical Investigation of Interactive Proof in HOL

This investigation aims to discover the degree to which proof as programming matches what users actually do in practice. The prover under consideration is the HOL prover and the organising view of the standard interface to this prover is proof as programming. We are certain of our model of the prover's behaviour and our objective is validate our model of its use.

In common with Polson (1987) we do not try to model fundamental cognitive processes such as retrieval of knowledge from long term memory. Rather, we are testing our claims regarding the logical and abstract interaction levels of the proposed view. In this study, we are not interested in evaluating the interaction model at the concrete interaction level; this would only confirm properties of the chosen editor at the keystroke level, which we do not believe to be critical in the proof as programming view. In other views, the evaluation would necessarily consider the concrete interaction level. For example, in proof by pointing the point and click action is central to the user-system interaction.

In the following sections we describe the behaviour we expect to observe, indicate how common we expect it to be and state how we shall quantify it. We then present the results of an experimental trial and our assessment of the proposed view of interaction. We also examine features of the user-HOL interaction about which we can make no predictions, but which are important to interface design. These include the granularity at which users enter commands and the errors made when eliminating logical connectives from formulae. The latter is a task which might be aided by a graphical interface.

Experimental data on the granularity of user input is relevant to interface design as designers often modify the way in which users are able to input commands. Interface designers must make decisions as to what information will be displayed and by what means. Experimental evidence about the visual cues that users commonly utilise can be used to rank the importance of the many sources of information that might be displayed by a graphical interface. Therefore, the investigation we describe is not restricted to testing the proof as programming view, it also aims to provide valuable information about how users perform proof using the current HOL interface. This information can be used to inform interface design and can serve as a reference against which new designs can be evaluated.

5.1 Experimental method

Subjects were asked to prove a theorem using HOL88 or HOL90, the two most widely used HOL implementations. They were asked to start up a HOL session in an emacs window and to set up an editor window for the ML file in the manner in which they would usually do so.

Proof Step	Grouping	Visual Cues			Was the outcome: Successful, Partially Successful or Unsuccessful
		Current Goal	Current Assumptions	Past Subgoals	
REPEAT STRIP_TAC	prepare for induction	Yes	/	/	Unsuccessful
GEN_TAC THEN STRIP_TAC		Yes	/	No	Successful
...					

Figure 3: A partially completed questionnaire

The interface then consisted of two windows and the activities in both windows were recorded. Subjects were given the theorem to be proved and asked to think-aloud while performing the proof but not to give detailed explanations. Their speech was recorded on audio tape.

After the proof was completed subjects were asked to explain the central ideas of the proof and to complete a questionnaire. For each proof step, i.e. for each sequence of tactics input to HOL, the subject was asked the questions listed in Figure 3.[3] The subject was also asked to group proof steps together to indicate the strategic thinking behind the phases of the proof attempt.

The experimental method records all proof steps entered into HOL and the results of their application. Hence the sequence of user inputs can be assessed at the abstract interaction level. The subjective importance of visual cues is documented. This provides valuable evidence about the visual cues which users make use of during the proof attempt. Evidence about the structure of the proof at the logical level can also be inferred from the post-proof questionnaire. A detailed analysis of the think-aloud protocols yields more knowledge of the logical level thinking of the subjects and its relation to the abstract interaction level, but this is beyond the scope of this paper. We can make predictions about the way in which proofs are developed and about the relative importance of visual cues from the proof as programming view, as we shall now show.

5.1.1 Backwards proof

We predict that proofs will be developed in a backwards manner. As described above, this means of developing a proof is supported by the HOL subgoal package and is an essential feature of the abstract interaction level. We also expect users to explore unfruitful parts of the search space and to use the backup command to return them to an earlier proof state. In order to judge the extent to which this happens, we decided to quantify the number of backing up steps and the size of the proof space explored. Proofs may also be developed in a forwards manner and we might expect users to perform short proofs of lemmas in this way.

We hypothesise that proof steps are selected on local context information and the experimental trial was designed to test this hypothesis. The local context is defined by the current goal and the current assumptions and is a component of the abstract interaction level as illustrated in Figure 2. It is certainly the case that task structure (Norman, 1988), or logical, considerations also play a very important role in decision making. However, there is no means

[3] In fact, the questionnaire also asked other questions, but these are of no relevance here.

other than protocol analysis to assess the influence of the subject's current logical strategy on tactic choice and we shall not investigate this question here.

It is possible that the user may modify the proof script at a place other than the end of the script. The proof script is an object at the abstract interaction level and the activity of editing the script is consistent with the proof as programming view. However, this activity does require the user to understand the proof script as a program. As stated above, we do not expect users to view their activity as programming and so we do not expect the refinement of the proof script to be common. The frequency of this behaviour was measured by counting the number of times a tactic is inserted into the proof script (at a position other than the end of the current script). A second measure of this activity is to count the number of times that expressions occurring in commands are repeated. If we observe that a subject repeatedly modifies a command and replays it, we can conclude that command refinement is taking place. If the subject follows the backwards proof model, then it is unlikely that an expression used in a previous tactic would occur again. Hence, we do not expect command refinement to be common.

5.1.2 Command granularity and the usefulness of visual cues

It was observed in an earlier study of HOL users that one interaction, that is one proof step, may combine several tactics. The granularity at which users might enter tactics is not restricted by the standard interface. We determined the granularity at which users actually do enter tactics from the trace of the proof session.

Users base their choice of tactic on many sources of information; the possibilities include the current goal, the current assumptions and past subgoal states. Past subgoal states are sometimes displayed in the proof tree presentation and evidence that users examine these states would support the use of proof trees as an appropriate display for HOL proofs. The relative importance of these information sources was assessed by the questionnaire of Figure 3. The questionnaire asks for the subject's opinion on whether a particular cue was used and is intended to collect *subjective* evidence for the use of a cue. These questions are not intended to document whether or not a particular tactic modifies the goals or modifies the assumptions in a purely logical sense. However, a strong correlation between these is to be expected.

5.1.3 Investigating the logical level

It is known that several tactics are often required to perform a particular operation at the logical level. For example, the tactic for a key proof idea (e.g. induction) may be precedeed by several tactics that put the goal into the right form first. Likewise, some tidying up steps may be needed afterwards.

The number of tactic groups, or *contexts*, in a proof attempt therefore reflects the logical level structure of the proof better than the number of proof steps or the number of individual tactics. To further investigate the relation between tactics, proof steps, and contexts, we decided to ask subjects to group proof steps into sequences of related steps.

5.1.4 The utility of a pointing mechanism for HOL

Selecting a particular subexpression e.g. with the mouse, is one potential application of proof by pointing in HOL. This arises in the elimination of quantifiers and truth functional con-

A

```
1# GEN_TAC THEN STRIP_TAC THEN LIST_INDUCT_TAC
         THEN ASM_REWRITE_TAC[];;
```

B

```
2# GEN_TAC THEN FIRST_ASSUM(UNDISCH_TAC o
                      assert is_forall o concl);;
```

C

```
3# DISCH_THEN(MP_TAC o SPEC "h*:");;

4# DISCH_THEN(CONJUNCTS_THEN2 ASSUME_TAC MP_TAC);;

5# DISCH_THEN(MP_TAC o SPECL ["[h:*]"; "l:(*)list"]);;
```

D

```
6# ASM_REWRITE_TAC[APPEND];;
```

Figure 4: An example of a tactic proof

nectives from the goal and from assumptions. In the command line interface to HOL, the user must expand tactics which successively eliminate connectives to the desired depth. This may result in errors such as eliminating too many quantifiers by a choosing a tactic such as `REPEAT STRIP_TAC`, which repeatedly removes connectives, and subsequently having to back up and apply `STRIP_TAC` until the desired connective is reached. We analysed the traces of the proof sessions for this type of error. The observation of such errors alone does not provide evidence for proof by pointing as a framework for interaction with HOL. However, if the error rate for this task is high, this would indicate the utility of a selection tool in a HOL interface.

5.2 Results

All subjects were asked to prove the following induction theorem:

$$\vdash \forall P.(P\,[\,] \wedge (\forall x.P[x] \wedge \forall l_1\,l_2.\,(P\,l_1 \wedge P\,l_2) \supset P(\mathsf{APPEND}\,l_1\,l_2)) \quad\supset\quad \forall l.\,P\,l$$

This states that if P holds of the empty list and all singleton lists, and if P holds for the append of any two lists for which P holds, then P holds of all lists. Figure 4 shows the tactic proof of this theorem discovered by subject 1. This figure illustrates the relationship between proof steps, tactics and contexts. The first two of the six proof steps contain multiple tactics (connected by `THEN`). The four contexts of the proof are indicated by boxes. The first context, A, contains one proof step, made up of four tactics. This context includes the preparation for list induction, the induction step itself and the solution of the base case. Other subjects discovered this part of the proof in several proof steps. They also indicated more contexts; for example, the solution of the base case was often distinguished from the earlier steps.

Each subject estimated their total experience, in months, with the HOL system and quantified their current usage. Current usage was estimated over the past six months and categorised as very frequent (VF), frequent (F), occasional (O) or nil (NIL). The tables of results are ordered according to current usage and total HOL experience. Consequently, subject 1 is a very frequent user with the greatest total experience with HOL and subject 7 is the least frequent user. The data is presented in Table 1. It should be noted that subject 7 did not complete the proof and hence the figure of 8 in the *In Proof* column means that 8 proof steps remained in the proof script at the end of the trial. It is also notable that three proof steps executed by subject 4 were carried out as forwards proof.

Table 1: Basic parameters

Subject	HOL Experience (months)	Current HOL Usage	Number of Interactions	Proof Steps		Time (mins.sec)
				Explored	In Proof	
1	54	VF	13	6	6	7.36
2	12	VF	37	14	7	10.44
3	7	VF	50	16	10	19.25
4	120	F	30	12	10	10.32
5	108	F	38	13	6	33.15
6	96	F	40	14	9	30.32
7	24	NIL	72	20	8*	61.25

The total number of interactions with the HOL system, the number of proof steps explored and the number of proof steps in final proof are also given in Table 1. These figures show the efficiency of the interactive proof, both in terms of the steps explored which were unsuccessful, and the number of interactions required for each proof step. The time taken for each proof attempt is also listed in Table 1.

The traces of the interaction with HOL were analysed to yield the data in Table 2. The data was obtained from an assessment of the entire proof attempt. The number of times the backup command was used is indicated in column 2 of Table 2. This table also shows the numbers of steps inserted into a proof script. The number of repeated expressions is the number of proof steps which occur multiple times. These figures indicate whether a subject has backed up and edited the proof script (as opposed to backing up and exploring an alternative proof) and how many expressions were repeatedly entered into HOL.

Table 2 also shows the number of errors in decomposing the goal, i.e. in eliminating logical connectives and in instantiating universally quantified variables. Whether a decomposition step was judged to be unsuccessful was determined by the subjects' response to the questionnaire.

Table 3 contrasts the number of proof steps (the number of lines of inputs to HOL) with the number of tactics and the number of contexts. A proof step may be composed of several tactics and a number of proof steps may be grouped together as performing a specific task. The number of proof steps and tactics were read from the log of the proof session, while the grouping of tactics was obtained from the questionnaire. Each column of Table 3 lists the

Table 2: Data on backing up and refinement

Subject	Backing-Up Steps	Inserted Proof Steps	Repeated Expressions	Decomposition Errors
1	0/6	0	0	0/4
2	5/14	1	5	1/10
3	2/16	1	2	1/8
4	0/12	0	1	0/4
5	0/13	1	2	0/5
6	2/14	0	1	1/6
7	2/20	0	1	0/3

Table 3: Data on proof steps, tactics and contexts

Subject	**Proof Steps**		**Tactics**		**Contexts**	
1	6	(6)	10	(10)	4	(4)
2	7	(14)	8	(18)	3	(5)
3	10	(16)	13	(19)	6	(6)
4	10	(12)	9	(10)	3	(3)
5	6	(13)	10	(19)	3	(7)
6	9	(14)	9	(14)	3	(7)
7	8*	(20)	10*	(24)	3*	(7)

number of proof steps (tactics and contexts respectively) in the final proof and also lists the number of proof steps explored during the entire proof attempt. The latter is the figure in brackets.

Table 4 shows the frequency with which subjects stated that their cues for selecting a proof step included the current goal, the current assumptions or a past subgoal. This data is simply that obtained from the questionnaire and refers to the entire proof attempt. The figures for the percentage of proof steps where the current assumptions were stated to be a cue are expressed as a percentage of all proof steps where there were assumptions, as opposed to a percentage of all explored proof steps.

5.3 Analysis

There was a large variation in the time taken to prove the goal, from 7 minutes 36 seconds to over 33 minutes—a factor of 4.4. There is no simple relationship between the time taken to find the proof and the number of interactions or the number of proof steps explored. More frequent users can interact with the theorem prover more rapidly, both in terms of recalling and entering tactics and theorems and interpreting the results. This implies that more frequent users do not, in general, try to minimise the number of interactions with the prover but use the prover as a means to formally check their conjectures. This conclusion is supported by the results of Table 1 which shows that subjects 2 and 3 explored more proof steps than subjects 4, 5 and 6 and interacted more with the HOL prover, but took less time to solve the problem (on average).

Less frequent users used the HOL prover less fluently. They did not explore proof steps in

Table 4: Data on cues

Subject	**Current Goal (%)**	**Current Assumptions (%)**	**Past Subgoals (%)**
1	83	40	0
2	71	82	0
3	88	82	0
4	92	90	0
5	85	64	8
6	36	50	50
7	100	69	23
Average	79	68	12

proportion to the time they took to find the proof. Instead, they spent time discovering how to construct syntactically correct tactics and discovering if a particular theorem was loaded into the system or whether it existed at all.

5.3.1 Evidence for backwards proof

All proof attempts were developed backwards from the goal. Three of the seven proof attempts contain no backing up steps (Table 2). This means that the subject simply applied proof step after proof step. Three subjects backed up twice. Two of these proof attempts (subjects 6 and 7) did not include an inserted proof step which indicates that backing up was done to explore a new proof. Consequently, in five out of the seven proof attempts studied the activity can best be described as backwards proof. It should be noted that subject 4 actually performed a short forwards proof in the midst of the backwards proof.

In the case of subject 3, backing up was connected with inserting a new proof step. This indicates that the proof script was refined once. In the case of subject 2, the proof script was refined once and the number of backing up steps was more than twice that of any other subject. The number of repeated expressions was also more than twice that of any other subject and this indicates that commands were also being refined during this proof attempt. In common with all other subjects, subject 2 developed a backwards proof; but the manner in which this was done was significantly different.

The current goal was stated to be a cue for 79% of proof step selections, averaged over the seven trials (Table 4). The current assumptions were stated to be cues for 68% of proof step selections, while past subgoals were stated to be cues on an average of 12% of proof step selections (averaged over the seven trials).[4] These results indicate that for the majority of subjects the local proof context was the main visual cue in choosing the next tactic. This result is in line with the proof as programming view. It is notable that subjects 2 and 3 stated that past subgoals were never a cue and, in this regard, their approach to proof discovery corresponds to the proof as programming view.

The dominant method of applying proof steps in six out of seven trials was the depth-first approach. This was not the exclusive mode of interaction as there is evidence that this approach was combined with proof script refinement to a greater extent than predicted.

5.3.2 Evidence on granularity and visual cues

On average a proof step was composed of 1.2 tactics and there were 2.44 proof steps in each context. Correspondingly, there were 2.92 tactics in each context (averages derived from Table 3). These results clearly show that the granularity of the user interaction with HOL is greater than the level of single tactics. Subjects made use of the command line interface to enter proof steps composed of several tactics on a significant number of occasions.

The data of Table 4 provide evidence for the utility of continuous display of the current goal and assumptions. There is much less evidence for the usefulness of the past subgoals of the current proof tree as a cue. In some cases, the subjects who stated that past subgoals were a cue were referring to the subgoal states reached by the *failure* of past proof steps and these states do not remain on the proof tree once a backup command has been given.

[4] Subjects were permitted to name any number of cues, hence the percentages for each subject do not add up to one hundred.

5.3.3 The logical level

The proof context is interpreted as reflecting the structure of the proof at the logical level. Each context is explored by more than two interactions and involves expanding almost three tactics on average. This is strong evidence for the idea that users organise the proof attempt at a significantly larger granularity than the tactic level. There is the possibility that an interface could represent these contexts explicitly and we discuss this further below.

5.3.4 Decomposition by selection

While all subjects tackled the same problem, the solutions varied in approach as well as in length and in the proportion of decomposition steps. The highest proportion of decomposition steps was 66% and the lowest 33%, see Tables 1 and 2 (excluding subject 7 who did not complete the proof). Table 2 shows that no subject made more than one error. This indicates that the HOL users who took part in this trial did not have significant problems in selecting tactics to eliminate logical connectives or to instantiate variables.

There was evidence from informal discussions and from the think-aloud protocols that subjects found the identification of assumptions and the specialisation of variables to be difficult, or to require significant thought. The results of Table 2 indicate that they could solve such problems correctly. However, an aid to decomposition such as a selection mechanism might decrease the time required to do so.

5.4 Discussion

The dominant mode of interaction with HOL, which we observed, can be described as proof as programming. It was observed that some HOL users have developed a style of interaction which differs from that which is typically taught to new users and which appears to be equally effective in terms of time spent on the proof. The results of the study suggest that our original view of proof as programming placed insufficient emphasis on the proof script as an interaction object. The results show that users find it useful to switch between simple backwards proof and proof script refinement as a means of developing a proof. This possibility did not feature clearly in our original view of the interaction.

There is some evidence for the potential utility of a proof by selection mechanism. There is less evidence in favour of a proof tree representation of the proof state, at least for problems of similar complexity to the example proof we studied.

The results concerning the logical level structure of proofs suggest a number of ways that proof context could be incorporated into interface design. Designers might provide displays of the current context, and hide all other contexts. Contexts could be labelled with notes to document the proof or the proof may be displayed as a tree of contexts, which would be a more manageable prospect than a tree of subgoal states. We consider that representing the proof structure at the logical level to be an important means by which users abstract from the detail of the concrete proof and reflect on proof structure. Consequently, we believe that an interface should make salient those features which we observed to be central in user planning and proof construction.

6 Conclusions

HCI-based studies typically need both to model the phenomena and to collect empirical data. Approaches to interactive proof differ such that they cannot be characterised by a single model; we need to specialise a rather general framework in order to capture important distinctions. Specifying models of the activity is an important part of improving our understanding of this particular form of human-computer interaction. However, we have a further objective: to formulate design principles and guidelines based on the aforementioned theoretical and empirical results.

In this paper we have proposed a three layer model which accounts for interaction with a theorem prover: at the concrete level of keystrokes, at the abstract interaction level of tactic texts and proof states, and at the logical level where the explanation is in terms of conjectures and legitimate proof steps. We have suggested that the relationship between the abstract and logical levels should be examined more closely, and identified three distinct views which determine the nature of the relationship and how it is represented: proof as programming, proof by pointing, and structure editing. We used these views to propose an analysis of both the standard, and other interfaces, to the HOL theorem prover.

An empirical study of HOL users was carried out and the results were interpreted with respect to what we argued was the dominant view of interaction, proof as programming. We conclude that this view is indeed the dominant style of interaction with HOL. We therefore suggest that interfaces to HOL should make salient those features which are supported by this view, though there may also be a role for interface features which are derived from other views of proof. We currently have no evidence for or against the mixing of views in a single interface and believe that this is a question which needs to be addressed. On answering this question we shall be in a position to recommend how views should be embodied in interface design and how, if at all, they can be combined. In this way we shall arrive at design principles.

We note that the scope of the investigation did not include any substantial development or formalization of new mathematical theories. We concentrated on proof development within the reasonably familiar theory of lists. Clearly, it is important to see how our models scale up, and so interaction in the development of larger-scale proofs, including substantial theory developments, will be the subject of future investigations.

Our experience in conducting this work confirms the value of an HCI-oriented investigation of theorem proving. Such an investigation offers a source of insights to those interested in improving the quality of interaction with theorem provers, as well as enlarging our understanding of the activity of computer-assisted reasoning. Careful analysis of empirical studies such as the one reported here provide valuable information about what users of theorem provers actually do. HCI research can offer important organizing concepts for thinking about interface design—for example the ideas about views and levels of description explained above. Finally, HCI research can provide the designers of theorem proving interfaces (who are seldom HCI experts themselves) with a specialised vocabulary for discussing the merits of specific designs and features.

Acknowledgements We acknowledge the assistance of the Automated Reasoning group of the University of Cambridge Computer Laboratory with the empirical investigation. This work was supported by EPSRC grant number GR/K25038.

References

Bertot, Y. (1994) User guide to the Coq+Centaur proof environment. Revision 1.40 July 1994. Available with the CtCoq distribution at WWW site http://www.inria.fr/croap/ctcoq/ctcoq-eng.html.

Bertot, Y., Kahn, G. and Théry, L. (1994) Proof by pointing. *International Symposium on Theoretical Aspects of Computer Science*, Sendai, Japan, Lecture Notes in Computer Science, Volume 789, Springer-Verlag, 1994.

Coquand, T., Nordström, B., Smith, J. M. and von Sydow, B. (1994) Type theory and programming. *EATCS Bulletin (52)* February 1994.

Gordon, M. J. C., Melham, T. F., eds. (1993). *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press 1993.

Green, T. R. G. (1980) Programming as a cognitive activity. in *Human Interaction with Computers.* eds. Smith, H. T. and Green, T. R. G. Academic Press 1980 pp. 271–320.

Green, T. R. G. (1991) Describing information artifacts with cognitive dimensions and structure maps. in *People and Computers IV.* eds. Diaper, D. and Hammond, N., Cambridge University Press 1991 pp. 297–315.

Hewitt, T. Baecker, R., Card, S., Carey, T., Garsen, J., Mantei, M., Perlman, G., Strong, G. and Verplank, W. (1992) *ACM SIGCHI Curricula for Human-Computer Interaction,* Report of the ACM SIGCHI Curriculum Development Group. p. 5.

Hutchins, E., Holland, J. and Norman, D. (1986) Direct Manipulation Interfaces. in *User Centred System Design.* eds. Norman, D. and Draper, S., Erlbaum, 1986.

Jones, C. B., Jones, K. D., Lindsay, P. A., and Moore, R. E. (1991) *mural: A formal development support system.* Springer Verlag 1991.

Kalvala, S. (1991) Developing an interface for HOL. *1991 International Tutorial and Workshop on the HOL Theorem Proving System and its Applications.* eds. Archer, M., Joyce, J. J., Levitt, K. N., and Windley, P. J., IEEE Computer Society Press 1991.

Kalvala, S. (1994) Annotations in formal specifications and proofs. *Formal Methods in System Design*, 5, 1994, pp. 119–144.

Martin-Löf, P. (1984) *Intuitionistic Type Theory.* Naples: Bibliopolis.

Nardi, B. A. and Zarmer, C. L. (1993) Beyond Models and Metaphors: Visual Formalisms in User Interface Design. *Journal of Visual Languages and Computing* 4, 5 1993 pp. 5–33.

Neilsen, J. (1986) A virtual protocol model for human-computer interaction. *IJMMS* vol 24. 1986 pp. 301–312.

Nordström, B. (1993) The ALF proof editor. *Informal proceedings of the 1993 workshop on types for proofs and programs*, University of Nijmegen, May 1993, ed. Geuvers, H.

Norman, D. A. (1988) *The Psychology of Everyday Things.* Basic Books 1988.

Paulson, L. C. (1991). *ML for the Working Programmer.* Cambridge: Cambridge University Press 1991.

Polson, P. G. (1987) A quantitative theory of human-computer interaction. in *Interfacing Thought: Cognitive aspects of human-computer interaction.* ed. Carroll, J. M. MIT Press 1987 pp. 184–235.

Pylyshn, Z. (1986) *Computation and Cognition.* MIT Press, 1986.

Schubert, T. and Biggs, J. (1994) A tree-based, graphical interface for large proof development. *Supplementary Proceedings of the 7th International Workshop on Higher Order Logic Theorem Proving and its Applications*, University of Malta, Valletta, September 1994. eds. Melham, T. F. and Camilleri, J.

Slind, K. (1994) A parameterised proof manager. *Higher Order Logic Theorem Proving and Its Applications: 7th International Workshop*, Valletta, Malta, September 1994, eds. Melham, T. F., and Camilleri, J., Lecture Notes in Computer Science, Volume 859, Springer-Verlag, 1994, pp. 407–423.

Slind, K. and Prehofer, C. (1994) Desiderata for Interactive Verification Systems, Internal Report for the German 'Deduktion' Project, February 1994, Available on WWW at ftp://ftp.informatik.tu-muenchen.de/local/lehrstuhl/nipkow/slind/papers/rqt.html

Théry, L. (1993) A proof development system for HOL. *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop*, HUG'93, Vancouver, B.C., August 11–13 1993, eds. Joyce, J. J. and Seger, C. H., Lecture Notes in Computer Science, Volume 780, Springer-Verlag, 1994, pp. 115–128.

Théry, L., Bertot, Y., and Kahn, G. (1992) Real theorem provers deserve real user interfaces. *Proceedings of the 5th Symposium on Software Development Environments*, Tyson's Corner, Virginia, Software Engineering Notes Vol. 17, No. 5 1992.