CBJones

Department of Computer Science
University of Manchester
Manchester M13 9PL, England

Technical Report Series
UMCS-87-12-8

Cliff B. Jones and R. Moore

An Experimental User Interface for a Theorem Proving Assistant

An Experimental User Interface for a Theorem Proving Assistant

C.B. Jones and R. Moore
Department of Computer Science,
The University,
Manchester M13 9PL

December 1987

Abstract

A Theorem Proving Assistant is taken to be a computer program which can be used interactively to keep track of, and automate some steps of, attempts to prove theorems. This paper describes an experiment in the design of user interfaces to such programs. The approach is introduced using a formal description from which the program has been implemented in Smalltalk-80.

Contents

1 **Introduction** **1**

2 **Basic Issues** **2**
- 2.1 Surface UI 2
- 2.2 Deep UI 3
- 2.3 Beyond Trees 4

3 **Muffin** **4**
- 3.1 Muffin's State 4
- 3.2 Muffin's UI 9
- 3.3 A Sample Session 12

4 **Evaluation** **14**

5 **Beyond Muffin** **15**

A **Formal Specification** **17**
- A.1 Muffin 17
- A.2 Expressions 22
- A.3 Subsequents 24
- A.4 Nodes 25
- A.5 Problems 26
- A.6 Proofs 26
- A.7 Names 28
- A.8 Solved and Unsolved Problems and Rules of Inference 29

B **The Proof of** ***and-or-dist*** **31**

1 Introduction

The authors of this paper are involved in a major collaborative Alvey/SERC project to build an "Integrated Project Support Environment" (IPSE). The aims of the project place it between the second and third generation objectives of the Alvey Software Engineering Strategy (see [20]). For this reason the project was christened "IPSE 2.5". One major innovative aspect of the project is the planned support for formal methods. (For a wider view of the project see [5], [19], [18].)

Work on the project is divided into five themes. That concerned with formal reasoning is being conducted by researchers at Manchester University and SERC's Rutherford Appleton Laboratory. The aim is to create a "FRIPSE" (i.e. Formal Reasoning IPSE) which is generic in that different logics and/or proof styles are supported. Such a FRIPSE would, for example, help with the construction of formal proofs to discharge the sort of proof obligations which arise in the development of programs using VDM (cf. [6]).

Early in the FRIPSE sub-project a list of basic ideas was recorded in a, so-called, concept paper; this document [7] is now somewhat dated but it does provide useful background information. The authors had experimented with several different Theorem Provers before writing [7] but a more detailed survey has also been undertaken subsequently: the published version of this is available as [11]. Nearly all of the systems surveyed had been created with far more emphasis on underlying theory than on user interface issues: many of them use the "glass teletype" as the mode of interaction. Perhaps as a consequence of this, most users find such systems of little help when trying to discover proofs. The typical approach is for the user to carefully plan a proof before trying to persuade the system to accept it.

The overriding objective of the FRIPSE project is to design a Theorem Proving Assistant which provides enough support for the activity in hand that a user would prefer to use the system rather than pencil and paper. Some computer scientists doubt the wisdom of this objective; many more question its feasibility. It is, therefore, worth trying to give some indication of why the FRIPSE group believe the goal to be worthy of pursuit. Nearly all of the proof obligations which arise in, for instance, the examples discussed in [6] are provable in rather simple steps. They can be characterized as being rather shallow results: their purpose is not to extend mathematics but rather to provide a cross-check between a specification and its putative design. Experience in constructing large numbers of such proofs by hand suggests that a system providing powerful symbol manipulation facilities could help remove the tedious aspects of the process.

The aim of FRIPSE is to build a system which leaves the user free to provide the insight which steers the proof in as natural a way as possible. The next section explores some of the user interface questions prior to the specification being developed in Section 3.

Before embarking on this, it is necessary to clarify the (rather limited) objectives of the current experiment. Several members of the FRIPSE team were involved in earlier projects to write systems which supported formal methods. In particular, the "Mule" project (cf. [2], [17], [3], [4]) built a "Structure Editor" which went significantly beyond the then current "Syntax Directed Editors". This work helped us identify the following objectives for the experiment described in this paper:

1. we believe that formal specifications help fix many aspects of such projects – this belief should be tested;
2. in particular, the underlying data structures of the implementation are crucial – could these be fixed ahead of implementation;
3. it was appreciated that we do not yet know how to (formally) specify good properties of user interfaces (and develop designs which provably fulfil them) – to what extent do these underlying data structures dictate what is referred to below as "deep UI" and leave the "surface UI" open to experimentation;
4. specifically, we wanted to find ways of letting a user focus on the essential details of one part of a proof at a time;
5. one possible way of limiting the length of proofs is to use derived rules – we wanted to experiment with their efficacy;

6. it was *not* an objective to handle any specific logic – the experiment could be undertaken with something as simple as propositional calculus.

It was therefore decided that some experimentation was necessary in the User Interface area. The internals of the system were described in VDM specification style (see [14]) before any code was written. Furthermore, the implementation language chosen to provide the underlying data structures was Smalltalk-80. A lack of familiarity with this language provided another argument for an experimental system.

In order to simplify the task, a trivial logic was chosen as the basis. Our experimental system (known as "Muffin") supports proofs in Propositional Calculus only—we could clearly have written a decision procedure but the experiment in interactive use has been revealing even in such a simple problem domain. Actually, Muffin has to be seeded with a set of axioms and we have used the Logic of Partial Functions of [1] as well as classical Propositional Logic. In fact, a range of logics could be supported. It would be interesting to characterize exactly what assumptions on this class of logics have been made in the design of Muffin: this task is left for the work on the (full) FRIPSE (cf. [10]). Furthermore, in the final system, it will be necessary to construct a proof that a specification like that given in Appendix A has operations which are consistent with a formal notion of proof: this has not been attempted for Muffin. This limitation made it possible to postpone questions about, for example, bound variables and substitution. None of the limitations is such that it invalidates the use of the interaction style with other logics. In fact, it is hoped that the proof style will be readily extendible to other domains.

2 Basic Issues

A distinction can be made between "surface UI" questions, which concern the appearance of the screen of the workstation, and "deep UI" issues, which govern the sort of interactions which are possible. These two points are first enlarged, then a final section discusses a particular problem with the representation of large formal objects.

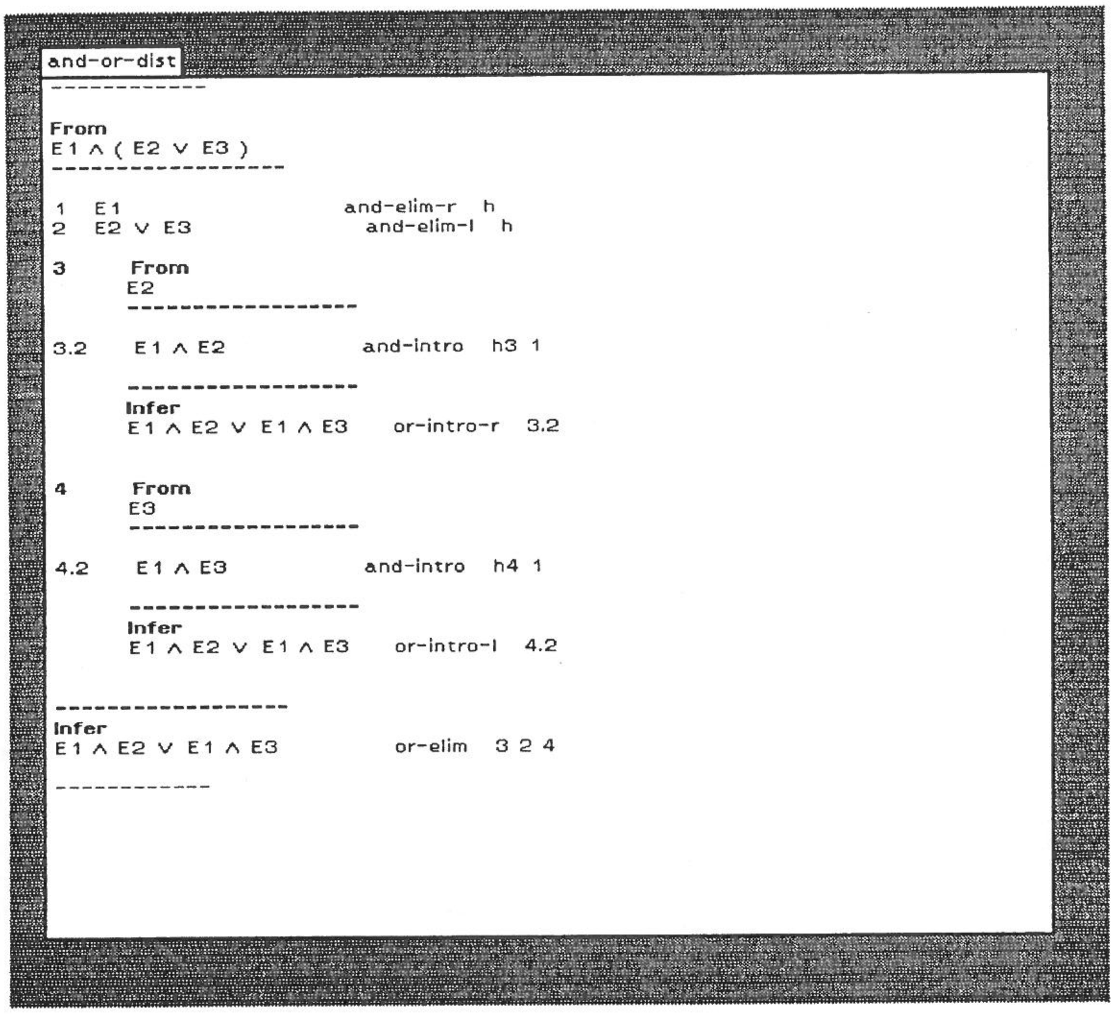
2.1 Surface UI

The evolution of workstation technology has brought to the user both significant computation speed and screens larger than the "24 by 80 glass teletype". One of the reasons for believing that "proof at the workstation" might be possible is the change to the surface UI brought about by the workstations now available.

Ideally, a screen as large as the desk on which one might do proofs by hand should be available. Apart from specifications and designs which give rise to a proof obligation, one might want to look both at results from the theories of data types in use and at derived inference rules in the logic itself. To some extent, multi-windowing systems, with their ability to overlay and collapse windows to icons, ameliorate the constraint to "A3 screens", and the speed of the processor means that even complex rearrangements of screen layout can be carried out extremely rapidly. Unfortunately, the need to rearrange the screen can be distracting to a user, so the limitations are still annoying and the size of the screen is still of concern.

Very early in the FRIPSE project it was realised that different users would want to be able to project the information held in the system in different ways. It has subsequently become clear that a single user might also wish to view the same information in different ways. That the design of Muffin was based on an underlying abstract state was a great advantage here as it meant that this facility could be provided simply by designing alternative "projection functions".

In [3] it was envisaged that a user would interact with a theorem proving assistant via the natural deduction proof style used in [6], an example of which is shown in Figure 1. Experimentation with other longer proofs has shown, however, that, for such proofs, the amount of information presented to the user in this natural deduction style display is more that can comfortably be assimilated. This prompted the design of an alternative projection in Muffin (Muffin's "prover" – see Section 3.2 for more details), which focuses on one "box" of such a proof at a time and which attempts to reduce the information presented to the user to essentials. The natural deduction style projection is also available, and it is sometimes useful to look at both views side by side (though not one on top of the other!). Modification of the proof can, however, only be made via the single box view (the prover).



```
and-or-dist
------------

From
E1 ∧ ( E2 ∨ E3 )
-------------------

1   E1                      and-elim-r   h
2   E2 ∨ E3                   and-elim-l   h

3        From
         E2
         -------------------

3.2      E1 ∧ E2            and-intro    h3  1

         -------------------
         Infer
         E1 ∧ E2 ∨ E1 ∧ E3     or-intro-r   3.2

4        From
         E3
         -------------------

4.2      E1 ∧ E3            and-intro    h4  1

         -------------------
         Infer
         E1 ∧ E2 ∨ E1 ∧ E3     or-intro-l   4.2

-------------------
Infer
E1 ∧ E2 ∨ E1 ∧ E3            or-elim   3 2 4

------------
```

Figure 1: Proof of *and-or-dist*

2.2 Deep UI

A bad surface UI can mar a good design; it cannot redeem a poor one. The deep UI questions are the real concern of the Muffin experiment. Experience with many other, in themselves excellent, theorem proving systems gives rise to a deep sense of frustration. One has the feeling that, with a huge body of code embodying many clever algorithms, the user is limited to a very restricted, pre-planned, menu of options: much of the functionality of the system is hidden from the user. The ground rule of our UI design has been to expose the whole state of the system both to view and to modification.

The idea of projection functions is mentioned above. This must be extended to allow the user to interact with the state through these projections, thus invoking changes to the state. For example, in Section 3, either forward or backward proof steps are catered for, and in a more powerful system, one would be able to unify two arbitrary expressions or to instantiate any definition with chosen arguments. Of course, the alterations that can be made to the state must be such that the logical soundness and consistency of the system, achieved in LCF-like systems by careful use of the type structure of ML, is preserved.

2.3 Beyond Trees

It is almost axiomatic that (abstract) syntaxes of things like *Expressions* describe them as "Trees". It would be possible to extend this to cope with *Proofs* etc. but a difficulty arises which makes it worth considering an alternative. In the specification developed in Section 3.1 and collected in Appendix A, references are introduced in a way which permits (acyclic) graph-like structures to be created. This section motivates that development.

A first guess at the underlying state of Muffin might lead one to view Theorems and Axioms as subsets of *Problems*. Thus the database might be:

$$Db_1 = \text{set of } Problem_1$$

$$\begin{array}{lrl} Problem_1 :: & hyp & : \text{set of } Exp_1 \\ & con & : Exp_1 \\ & proofs & : [\text{set of } Proof_1] \end{array}$$

Axioms would have <u>*nil*</u> proofs; unproven theorems would have empty sets; whilst proven theorems would have one, or more, proofs in the set.

When attention is turned to *Proofs* it is clear that individual steps could be justified by appeal to other *Problems* (be they axioms or proven theorems). This means that a series of *Problems/Proofs* constitute a graph-like structure which is more general than can be represented by a tree. This sort of "sharing" is inevitable and it is normal in formal descriptions to model it by the use of some "reference" object. (The archetypal example being *Locations* in the denotational semantics of imperative programming languages.) Introducing *ProblemRef* one might define the proof database as:

$$\begin{array}{lrl} Db_2 :: & pm & : \text{map } ProblemRef \text{ to } Problem_2 \\ & jm & : \text{map } ProblemRef \text{ to } (\text{set of } Proof_2) \end{array}$$

$$\begin{array}{lrl} Problem_2 :: & hyp & : \text{set of } Exp_2 \\ & con & : Exp_2 \end{array}$$

Here, references to axioms are simply omitted from the domain of the justification map *jm*. The reader should be able to picture how graph-like structures can be built; acyclicity would be defined by an invariant. The next section shows how expressions can also be treated as graphs.

One other detail is worth mentioning. Some inference rules permit a sequent, rather than just an expression, to be present in the *hyp* component of a *Problem*. This is handled by bringing in the notion of a *Subsequent* to restrict the nesting to one level (other reasons for using *Subsequents* are given below).

3 Muffin

3.1 Muffin's State

This section contains a description of the state underlying the actual Muffin system, though concrete formulae are only given here for the abstract syntax of the state and for the abstract syntax and the invariants on the basic objects comprising the state. The interested reader can find full details of the formal specification of Muffin, including the invariant on the state as well as the operations for changing the state, in Appendix A.

First, the full description of the Muffin state. The first four components are object stores for each of the basic types of object in Muffin, that is, *expressions*, *subsequents*, *problems* and *proofs*. Their domains are infinite sets of structureless tokens and are all disjoint. Next come name stores for each type of object. The *Proofmap* associates problems with complete proofs, and some solved problems are designated as rules of inference via the *Rulemap*. The *Incomplete-proofmap* associates problems with incomplete proofs. As has been mentioned above, Muffin supports both forward and backward inferencing, which, in terms of the state, means that an incomplete proof effectively consists of two parts, a forward proof and a backward proof. Elements can be added either to the tail of the forward proof or to the head of the backward proof, as explained in more detail later in this section. The remaining component of the state, the *Indexmap*, therefore records the last element of the forward proof for each incomplete proof, that is the point in the sequence at which new elements can be inserted.

$$
\begin{array}{lrcl}
\mathit{Muffin} :: & es & : & \mathit{Expstore} \checkmark \\
 & ss & : & \mathit{Subseqstore} \checkmark \\
 & ps & : & \mathit{Problemstore} \checkmark \\
 & fs & : & \mathit{Proofstore} \checkmark \\
 & en & : & \mathit{ExpNames} \\
 & sn & : & \mathit{SubseqNames} \\
 & pn & : & \mathit{ProblemNames} \\
 & fn & : & \mathit{ProofNames} \\
 & jm & : & \mathit{Proofmap} \checkmark \\
 & rm & : & \mathit{Rulemap} \times \\
 & im & : & \mathit{Incomplete\text{-}proofmap} \\
 & xm & : & \mathit{Indexmap}
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{inv\text{-}Muffin}(\mathit{mk\text{-}Muffin}(es, ss, ps, fs, en, sn, pn, fn, jm, rm, im, xm)) \quad \triangleq \\
\quad \mathit{is\text{-}valid\text{-}subseqstore}(ss, es) \wedge \\
\quad \mathit{is\text{-}valid\text{-}problemstore}(ps, ss, es) \wedge \\
\quad \mathit{is\text{-}valid\text{-}proofstore}(fs, ps, ss, es) \wedge \\
\quad \mathit{is\text{-}valid\text{-}expnames}(en, es) \wedge \\
\quad \mathit{is\text{-}valid\text{-}subseqnames}(sn, ss) \wedge \\
\quad \mathit{is\text{-}valid\text{-}problemnames}(pn, ps) \wedge \\
\quad \mathit{is\text{-}valid\text{-}proofnames}(fn, fs) \wedge \\
\quad \mathit{is\text{-}valid\text{-}proofmap}(jm, fs, ps, ss, es) \wedge \\
\quad \mathit{is\text{-}valid\text{-}rulemap}(rm, jm, fs) \wedge \\
\quad \mathit{is\text{-}valid\text{-}incomplete\text{-}proofmap}(im, jm, fs, ps, ss, es) \wedge \\
\quad \mathit{is\text{-}valid\text{-}indexmap}(xm, im, jm, fs, ps, ss, es)
\end{array}
$$

In the remainder of this section, each component of Muffin, together with the appropriate validity conditions that it has to satisfy, will be dealt with in turn.

The fundamental entities in Muffin are *expressions*. These are either meta-variables (*Atoms* in the specification) or are built up of some logical connective having other expressions for its operand(s). In order to make the specification and the description clearer, a particular set of logical connectives has been chosen and is built into Muffin, though the particular choice made is essentially unimportant to the specification.

A particular expression may, of course, be a subexpression of many different expressions (though not of itself!) and it is therefore appropriate to make use of the acyclic graphs described above and make the arguments of expressions *references* to other expressions. The *Expstore* records the relationship between expressions and their references. For ease of testing of equality on expressions, we make the *Expstore* 1–1; this means that any two expressions are the same if and only if their references are the same.

Other conditions are imposed on the *Expstore* for consistency. First, all subexpressions of any expression in the *Expstore* must also be in the *Expstore* (*closedness*), and second, the graphical structure representing the *Expstore* must be acyclic (*finiteness*). This latter condition is equivalent to saying that no expression can be a subexpression of itself. The full syntax for expressions is therefore given by:

$$\mathit{Exp} = \mathit{Not} \mid \mathit{And} \mid \mathit{Or} \mid \mathit{Impl} \mid \mathit{Equiv} \mid \mathit{Delta} \mid \mathit{Atom}$$

$$\mathit{Not} :: \mathit{not} : \mathit{Expref}$$

$$
\begin{array}{lrcl}
\mathit{And} :: & \mathit{andl} & : & \mathit{Expref} \\
 & \mathit{andr} & : & \mathit{Expref}
\end{array}
$$

$$
\begin{array}{lrcl}
\mathit{Or} :: & \mathit{orl} & : & \mathit{Expref} \\
 & \mathit{orr} & : & \mathit{Expref}
\end{array}
$$

$$
\begin{array}{lrcl}
\mathit{Impl} :: & \mathit{ant} & : & \mathit{Expref} \\
 & \mathit{con} & : & \mathit{Expref}
\end{array}
$$

$$
\begin{array}{lrcl}
\mathit{Equiv} :: & \mathit{eql} & : & \mathit{Expref} \\
 & \mathit{eqr} & : & \mathit{Expref}
\end{array}
$$

$Delta :: del : Expref$

$Expstore =$ map $Expref$ into Exp

where

$inv\text{-}Expstore(es) \quad \triangleq \quad is\text{-}closed(es) \wedge is\text{-}finite(es)$

Note that *Atom* and *Expref* are (disjoint) infinite sets of structureless tokens.

Concrete examples of how expressions are stored in Muffin according to this syntax, as well as examples of the other types of object in Muffin which are described in this section, can be found in Section 3.3, where the data structures behind the problem $\{E1 \wedge (E2 \vee E3)\} \vdash E1 \wedge E2 \vee E1 \wedge E3$, the proof of which is shown in Figure 1, are explained.

In addition to the standard logical expressions, Muffin also contains the *subsequent*, written $a \leadsto b$. This is introduced largely to avoid arbitrary nesting of turnstiles, as has already been mentioned above. A subsequent has a *left-hand side*, which is a set of expressions, and a *right-hand side*, which is a single expression. Again, references to the expressions rather than the expressions themselves are used for the arguments of subsequents.

Expressions and subsequents, collectively referred to as *nodes*, form the building blocks for *problems*, which represent many similar entities, such as *axioms*, *(derived) rules of inference*, *lemmas*, *theorems* and *conjectures*, in Muffin. Problems have a set of *hypotheses* to the left of their turnstile, and a single *conclusion* to the right. The arbitrary nesting of turnstiles is avoided by restricting the hypotheses to be nodes and the conclusion to be an expression. In addition, a subsequent with an empty left-hand side can be identified with the expression on its right-hand side. This gives rise to an invariant on subsequents:

$$\begin{aligned} Subseq :: \; & lhs : \text{set of } Expref \\ & rhs : Expref \end{aligned}$$

where

$inv\text{-}Subseq(mk\text{-}Subseq(z, y)) \quad \triangleq \quad z \neq \{\}$

Subsequents and problems are stored respectively in the *Subseqstore* and the *Problemstore*, with both stores being taken to be 1–1 mappings as for the *Expstore*. The validity condition on the *Subseqstore* states that any expression which forms part of some subsequent in the *Subseqstore* must be in the *Expstore*, that on the *Problemstore* that any expression or subsequent forming part of some problem in the *Problemstore* must be in, respectively, the *Expstore* or the *Subseqstore*.

$Subseqstore =$ map $Subseqref$ into $Subseq$

$Node = Expref \mid Subseqref$

$$\begin{aligned} Problem :: \; & hyp : \text{set of } Node \\ & con : Expref \end{aligned}$$

$Problemstore =$ map $Problemref$ into $Problem$

The *Subseqrefs* and the *Problemrefs* are both infinite sets of structureless tokens, assumed disjoint from each other and from both *Exprefs* and *Atoms*.

At this basic level, the notions of *substitution* and *matching* can also be introduced. Thus, one is allowed to build an *instance* of, for example, some expression e by substituting expressions for any meta-variables occurring in e. In the formal specification a map *Atom* to *Expref* represents such a substitution. If the result of making some substitution m in an expression A yields the expression B then the expression A is said to *match* the expression B. Alternatively, B is an instance of A, with the particular instance being given by the substitution m. The notions of matching and substitution extend trivially to both subsequents and problems.

Problems divide naturally into several different categories. They may be assumed true without proof (corresponding to the axioms of the system), proved (corresponding to derived inference rules, lemmas and theorems) or unproven (corresponding to conjectures). The first two categories together will be referred to as *solved* problems, the third as *unsolved* problems. Some subset of the solved problems is designated as the *rules of inference* of the system.

Given some existing set of solved problems, there are essentially two ways of creating a new solved problem. First, one can simply build an instance of some existing rule of inference by replacing some or all of the variables appearing in the statement of the rule with expressions. The new solved problem so created then has a proof which is simply an *instantiation*, consisting of a reference to the problem which is the statement of the rule together with a map recording the necessary variable substitution. Thus:

$$\begin{array}{lll} \textit{Instantiation} :: & \textit{of} & : \textit{Problemref} \\ & \textit{by} & : \mathsf{map}\ \textit{Atom}\ \mathsf{to}\ \textit{Expref} \end{array}$$

where

$$\textit{inv-Instantiation}(\textit{mk-Instantiation}(o, m)) \quad \triangle \quad m \neq \{\}$$

Note that the case in which the substitution map is the empty map is excluded – building an instance of something with a null substitution does nothing.

The second way of creating new solved problems is by combining existing solved problems together in some order as a *composite proof.* This is represented in Muffin simply as a sequence of solved problems:

$$\textit{Composite-proof} = \mathsf{seq\ of}\ \textit{Problemref}$$

Proofs as a whole are then just the union of *Instantiations* and *Composite-proofs*:

$$\textit{Proof} = \textit{Instantiation} \mid \textit{Composite-proof}$$

For a composite proof c, the *knowns* of some problem p with respect to c are the set of things deducible from the hypotheses of p via the elements of c taken in order. Thus, if c is empty, the knowns of p are the same as the hypotheses of p. Otherwise, the $i + 1$ *th* element v of c contributes an expression or a subsequent to the knowns as follows: if the hypotheses of v are all contained in the knowns collected with the first i elements of c, then v contributes its conclusion to the knowns; if not, but if there is some subsequent s in the *Subseqstore* whose right-hand side is the same as the conclusion of v and whose left-hand side added to the hypotheses of p gives the hypotheses of v, then v contributes s to the knowns; if neither of these two conditions is satisfied, v contributes nothing to the knowns. A composite proof c is therefore a *complete proof* of a problem p if the knowns of p with respect to c contains the conclusion of p; otherwise c is an *incomplete proof* of p.

The method by which the knowns of the problem p with respect to the composite proof c are generated also forms the basis for the full natural deduction style display of Figure 1. Taking the elements of the proof c in turn, if a particular element e adds an expression to the knowns, that element adds a line like line 1 to the display, with the line showing the new known generated and the justification of the new known. If the element e adds a subsequent to the knowns, this generates a *from-infer* box in the display, the *from* line listing the elements of the left-hand side of the subsequent, the *infer* line the right-hand side. The lines of the proof internal to that box or subproof are generated similarly by displaying the proof of the problem e between the *from* and the *infer* lines.

Each proof is assigned a reference via the *Proofstore*, with the *Proofrefs* being yet another infinite set of structureless tokens, disjoint from all the others:

$$\textit{Proofstore} = \mathsf{map}\ \textit{Proofref}\ \mathsf{to}\ \textit{Proof}$$

where

$$\textit{inv-Proofstore}(\textit{fs}) \quad \triangle \quad \forall p, q \in \mathsf{dom}\,\textit{fs} \cdot \textit{fs}(p) = \textit{fs}(q) \wedge \textit{is-Instantiation}(\textit{fs}(p)) \Rightarrow p = q$$

The invariant states that each *Instantiation* is assigned a unique reference via the *Proofstore*. This restriction turns out to be impractical for composite proofs in general, however – new complete proofs are going to be built by editing existing incomplete proofs, so sometimes different references to the same proof might be required (for example, it may not be clear to a user how exactly to proceed from the current state of some proof in order to complete the proof, and he might wish to try several different possibilities, thus necessitating duplicating the current state of the proof). In Muffin, this restriction is in fact taken to the other extreme and *no* composite proof is shared. The difficulties only really occur in the case of incomplete proofs, however, so it would in fact be possible to extend the invariant here so that all complete proofs were assigned unique references.

The validity condition on the *Proofstore* states simply that any component of any proof in the *Proofstore* has to be in the *Expstore*, the *Subseqstore* or the *Problemstore* as appropriate.

Any of the objects introduced so far can be given a name, so that the user of Muffin can more readily identify those objects of particular interest. The names are stored in a name store for each of the basic types of object, though no two objects of the same type may have the same name, and the empty string is not a valid name. It is possible, however, for objects of different types to have identical names. Thus:

$$ExpNames = \mathsf{map}\ String\ \mathsf{into}\ Expref$$

$$SubseqNames = \mathsf{map}\ String\ \mathsf{into}\ Subseqref$$

$$ProblemNames = \mathsf{map}\ String\ \mathsf{into}\ Problemref$$

$$ProofNames = \mathsf{map}\ String\ \mathsf{into}\ Proofref$$

Note that *String* here represents non-empty sequences of characters.

The validity constraints ensure that only existing objects, that is objects in the relevant object stores, can be given a name.

In order to associate proofs with problems, the *Proofmap* and the *Incomplete-proofmap* are introduced. The first of these associates problems with complete proofs, the second with incomplete proofs.

$$Proofmap = \mathsf{map}\ Problemref\ \mathsf{to\ set\ of}\ Proofref$$

$$Incomplete\text{-}proofmap = \mathsf{map}\ Problemref\ \mathsf{to\ set\ of}\ Proofref$$

where

$$inv\text{-}Incomplete\text{-}proofmap(im) \quad \triangleq \quad \{\,\} \notin \mathsf{rng}\, im \wedge \forall k, m \in \mathsf{dom}\, im \cdot im(k) \cap im(m) \neq \{\,\} \;\Rightarrow\; k = m$$

The first part of the invariant on the *Incomplete-proofmap* says that it doesn't bother to record the fact that a given problem has no incomplete proofs as this can be inferred from the absence of the problem in question from the domain of the *Incomplete-proofmap*. The second part of the invariant occurs as a result of the restriction that no composite proof is shared (note that only composite proofs can be incomplete – instantiations are by definition always complete).

Those problems occurring in the domain of the *Proofmap* are the solved problems, those not, the unsolved problems. The axioms of Muffin are made to conform to this definition by mapping them to the empty set under the *Proofmap*. In addition, some of the solved problems are designated as the *rules of inference* of the system, and these are given names via the *Rulemap*:

$$Rulemap = \mathsf{map}\ String\ \mathsf{into}\ Problemref$$

Again no two rules may have the same name, and the empty string is not a valid rule name.

The validity condition on the *Rulemap* states that all rules are solved problems, all axioms are rules, and all *Instantiations* are instantiations of rules. The condition on the *Proofmap* is, however, somewhat more complicated. First, all the solved problems must be in the *Problemstore* and all the complete proofs in the *Proofstore*. Second, any proof attached to some problem via the *Proofmap* must not only *be* a complete proof of that problem but may itself only contain solved problems. Thirdly, it must be possible to associate exactly one of its complete proofs with each solved problem other than the axioms in such a way that there are no circularities of reasoning amongst this set of associations, that is the system must be logically sound. Note however, that circularities *can* exist due to a problem being allowed to have multiple complete proofs. Thus, for instance, a user might prove result A from the basic axioms, then prove B making use of A as a derived rule, then prove A again using B as a derived rule, without destroying the logical soundness of the system. Finally, no two solved problems share a complete composite proof.

The validity condition on the *Incomplete-proofmap* is built up similarly. First, any problem in its domain must be in the *Problemstore* and any incomplete proof attached to that problem in the *Proofstore*. In fact, this condition is extended to state that the *Proofstore* contains only those proofs which are attached to some problem via either the *Proofmap* or the *Incomplete-proofmap*. Here, however, the proof must *not* be a complete proof of the problem, though it must still consist only of solved problems. There should be no *Instantiations* occurring in any element in the range of the *Incomplete-proofmap*, and no proof is both an incomplete proof of some problem and a complete proof of some other.

Incomplete proofs consist effectively of two parts, the *forward proof* and the *backward proof*, with the proof as a whole being the concatenation of the backward proof onto the forward proof. When attempting to convert an incomplete proof of some problem into a complete proof thereof, new elements can be added either to the tail of the forward proof or to the head of the backward proof, corresponding respectively to *forward inferencing* and *backward inferencing.* In order to be able to insert new elements into a proof at the correct point, it is therefore necessary to record for each incomplete proof the position in the sequence which marks the division between the forward and backward proofs. The last component of Muffin, the *Indexmap*, does this by recording the index of the last element of the forward proof:

$Indexmap = \mathsf{map}\ Proofref\ \mathsf{to}\ \mathbb{N}$

The elements of the forward proof give rise to all the knowns, with part of the validity condition on the *Indexmap* being that each element of the forward proof should actually contribute to the knowns. Thus, by adding a new element to the tail of the forward proof which satisfies one of the conditions for adding to the knowns it is possible to increase the current knowns (forward inferencing).

The elements of the backward proof, on the other hand, provide a proof of the conclusion of the relevant problem from some set of subgoals. Proving all these subgoals would be sufficient to complete the proof. In this case, a new element can be added to the head of the backward proof if the conclusion of that element is amongst the current subgoals. (This condition also forms part of the validity constraint.) The conclusion of the element is then removed from the subgoals and all its hypotheses are added to get the new subgoals, that is, one of the existing subgoals is itself reduced to subgoals. This is backward inferencing.

Another part of the validity condition on the *Indexmap* states that the backward proof should contain no element all of whose hypotheses are among the current knowns – such an element would correctly contribute its conclusion to the knowns and should therefore be positioned at the tail of the forward proof. After every forward inferencing step, therefore, any such elements in the backward proof must be removed therefrom and transferred to the tail of the forward proof, then any spurious elements in what remains of the backward proof must be discarded (this latter is necessary because the new forward element might have added something to the knowns which had previously been reduced to subgoals by some backward step). If the new knowns contains the conclusion of the problem, the new forward proof forms a complete proof of it and so the problem–proof association can be removed from the *Incomplete-proofmap* and added to the *Proofmap*, at the same time removing the proof–index association from the *Indexmap*. Otherwise, the new incomplete proof becomes the new forward proof concatenated with the new backward proof, with the appropriate change to the *Indexmap*. Note that this reorganisation is unnecessary for backward inferencing as this does not alter the knowns.

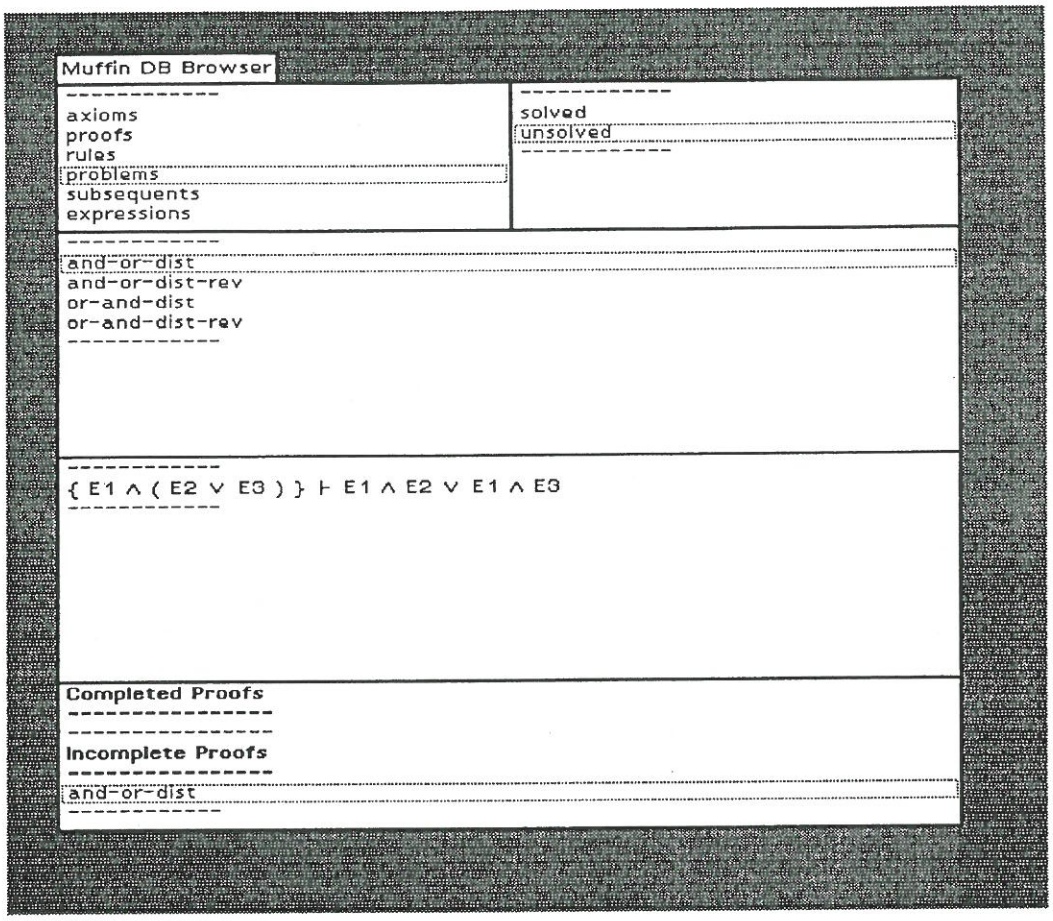
The remaining parts of the validity condition on the *Indexmap* say that it records an index for each incomplete proof but for no complete proof, and that the value of the index lies somewhere between zero and the number of elements in the proof.

3.2 Muffin's UI

The various components of the surface UI can be conveniently divided into three categories, the *browser*, the *builder* and the *prover.*

The *browser* essentially allows the user to inspect the current state of Muffin. The user selects the type of object he wishes to inspect from the list *axioms*, *proofs*, *rules*, *problems*, *subsequents* and *expressions*. The browser will then show all objects of the selected type. Where the particular type selected has multiple subtypes, e.g. *complete* and *incomplete* for proofs, *and*, *or*, etc. for expressions, the user can additionally select one of these subtypes and the browser will then show only those objects of the selected subtype. Objects can be accessed via their names or via some textual representation of the objects themselves. When the particular object selected is a problem, the browser shows additionally either the status of any existing proofs of that problem or that the selected problem is an axiom. In the latter case, the axiom name is also shown. Figure 2 shows the browser where the selection is the unsolved problem named *and-or-dist* and its incomplete proof of the same name, the completed version of which is shown in Figure 1.

In addition, the browser allows a few simple changes to be made to the state of Muffin, such as naming and renaming of objects, conversion of an unsolved problem to an axiom, conversion of a solved problem to a (derived) rule, and addition of a new empty composite proof to the set of incomplete proofs of some problem. The interested reader can find the specification of these actions amongst the operations on the Muffin state given in Appendix A.


```
Muffin DB Browser
------------
axioms
proofs
rules
problems
subsequents
expressions

------------
solved
unsolved
------------

------------
and-or-dist
and-or-dist-rev
or-and-dist
or-and-dist-rev
------------

------------
{ E1 ∧ ( E2 ∨ E3 ) } ⊢ E1 ∧ E2 ∨ E1 ∧ E3
------------

Completed Proofs
----------------
----------------
Incomplete Proofs
----------------
and-or-dist
------------
```

Figure 2: Muffin's Browser

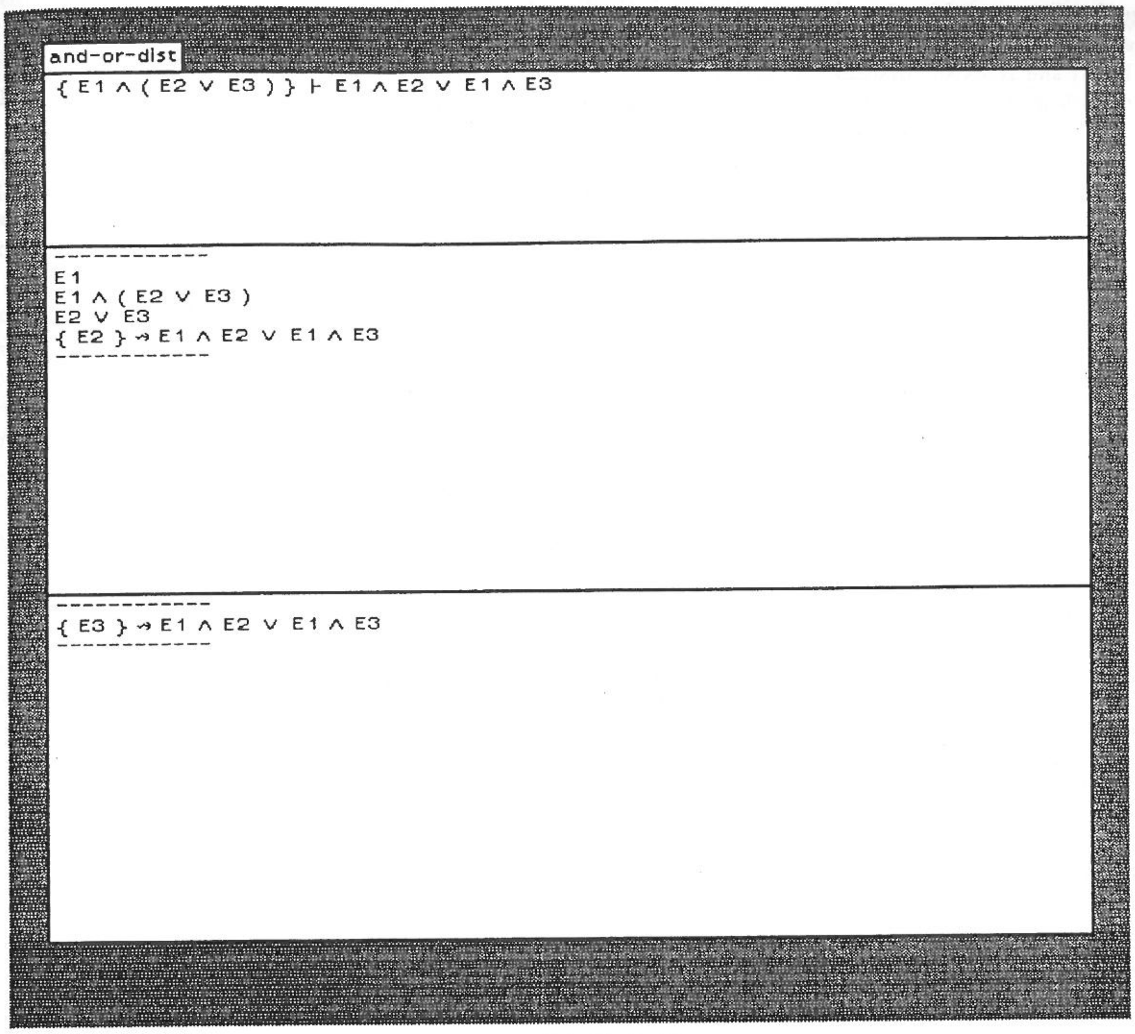
Finally, the browser acts as a controller for the other components of Muffin. Thus, for instance, it allows the user to start up either a builder or a prover, to inspect the current status of some existing proof, to remove incomplete proofs and unsolved problems from Muffin's stores, and to restart some abandoned proof at the point at which it was abandoned.

The *builder*, of which there are several different forms, allows the user to create new expressions, subsequents and problems and add them to the relevant object stores. The operations for updating the object stores can also be found in Appendix A.

Lastly, the *prover* allows the user to edit an incomplete proof with a view to converting it into a complete proof. Both forward and backward inferencing are supported, together with the ability to undo proof steps (by removing them either from the tail of the forward proof or from the head of the backward proof).

Proofs in the style of [6], an example of which was given in Figure 1, contain much information which is in general of little help to someone actually in the process of constructing such a proof. Muffin's prover therefore attempts to reduce the information presented to the user to essentials in two ways. First, proof steps internal to some subproof or "box" in such a proof are only meaningful within that subproof and not within the containing proof. Muffin therefore restricts attention to one box at a time, with any subproof of that box being represented as a subsequent. A set containing all the expressions appearing on the *from* line of the box forms the left-hand side of the subsequent, whilst the right-hand side is what appears on the *infer* line of the box.

Second, the prover hides all the details of ordering of lines and justification of proofsteps within a box by using the knowns and the goals as described above as the basis for its display. The prover thus displays the problem the user is attempting to solve, the current knowns, and the *visible goals*, where the visible goals are those current goals (as obtained via the elements of the backward proof) not amongst the current knowns. Figure 3 shows a prover at that point during the construction of the proof of Figure 1 at which the proof is complete apart from the subproof at box 4.



```
and-or-dist
{ E1 ∧ ( E2 ∨ E3 ) } ⊢ E1 ∧ E2 ∨ E1 ∧ E3

------------
E1
E1 ∧ ( E2 ∨ E3 )
E2 ∨ E3
{ E2 } ⇝ E1 ∧ E2 ∨ E1 ∧ E3
------------

------------
{ E3 } ⇝ E1 ∧ E2 ∨ E1 ∧ E3
------------
```

Figure 3: Muffin's Prover

The full natural deduction style display of Figure 1 is also available, however, and the user can additionally display any complete or incomplete proof in this style if so desired.

In addition, the user can choose to further reduce the amount of information displayed by the prover by making use of the facility of *elision* of knowns. Thus, if the user decides that a particular element of the knowns is not going to be of any use in the remainder of the proof, it can be designated as *hidden* and it will be removed from the display. When a prover has hidden knowns, Muffin reminds the user of this by displaying ellipsis points at the foot of the list of displayed knowns. Any hidden known remains a known of the proof, of course, and the reverse operation of redisplaying hidden knowns is available to the user at any time.

The prover offers the user three ways of adding a new element to an incomplete proof. First, if it is possible to build some instance i of some rule of inference such that all hypotheses of i are amongst the current knowns, then i can be added to the tail of the forward proof. If, on the other hand, the user can build such an i such that the conclusion of i is amongst the current goals, then i can be added to the head of the backward proof. Finally, if there is a subsequent $a \rightsquigarrow b$ amongst the current goals and if the problem p is solved and has hypotheses $h \cup a$ and conclusion b, where h represents the hypotheses of the problem the user is currently trying to solve, then p can be added to the tail of the forward proof (the conditions on p are exactly those that must be satisfied in order for it to contribute the subsequent $a \rightsquigarrow b$ to the knowns).

Muffin offers assistance with each of these three processes. In the first two cases, the user can select an expression from either the knowns or the goals and ask Muffin to provide a menu of rules matching the selection. When matching to a goal, Muffin returns the list of rules whose conclusions match the selected goal; when matching to a known, it returns the list of rules which have amongst their hypotheses something which matches the selected known. Selecting a rule from the list returned then causes Muffin to try to build the relevant instance of the selected rule and add this to the proof.

The variable substitution deduced from the initial matching process is not always complete, however. For instance, more than one element of the hypotheses of the selected rule might match the selected known, or the rule might contain more variables than the expressions which were matched do. In such circumstances, Muffin prompts the user to complete the parts of the variable substitution it was unable to deduce for itself (this is done via the *substitution editor*—see Section 3.3). Then, when the instantiation is complete, it adds the new element to the proof.

The assistance offered with the third method of adding a new element to a proof is less sophisticated. Here, the user can select a subsequent from the current goals and ask Muffin to search through the solved problems to see whether the problem p which would have to be added to the tail of the forward proof in order for the selected subsequent to be added to the knowns is amongst them. If it is, Muffin automatically adds p to the forward proof. If not, the user is given the opportunity to open a new prover in order to attempt to solve the problem p.

The user may have as many provers, browsers and builders as he wishes active and displayed on the screen at once, and can at any time change which one he is currently working in. In particular, he may have provers in which he is attempting to solve different problems as well as provers showing different attempted proofs of the same problem. Thus, for example, if, while working on some proof, he decides that the proof would be more straightforward if he were to prove some new derived rule first, he can leave the current proof, build the problem stating that derived rule in a builder, prove it in some other prover, then designate it as a derived rule, maybe in a browser. On returning to the original proof, the new rule will now be available and it can be used there as desired.

Finally in this section, it is worth noting that any window appearing on the screen is simply a *view* of the current underlying Muffin state. Closing a window simply removes a particular view from the display; it *never* changes the state. Thus, if the user is unable to complete a particular proof, the prover showing that proof can be safely deleted from the display; it can later be retrieved via the browser in exactly the same form as it had when deleted.

3.3 A Sample Session

This section explains in some detail the sequence of actions a user of Muffin would typically perform in order to state and prove the theorem *and-or-dist* of Figure 1. For the purposes of this exercise, it is assumed that all the rules of inference used to justify the steps of the proof as shown in Figure 1 are already rules of inference in Muffin.

First, the user has to build the problem $\{E1 \wedge (E2 \vee E3)\} \vdash E1 \wedge E2 \vee E1 \wedge E3$, which is the statement of the theorem he wishes to prove. Using the bottom-up approach, which is the easier to explain, he begins by creating the three *Atoms* $E1$, $E2$ and $E3$. Assuming, for sake of argument, that $E1$ is already in the *Expstore*, that $E2$ and $E3$ are not, and that $r1$ is the existing reference of $E1$. Two new *Exprefs* must be created and the associations $r2 \mapsto E2$ and $r3 \mapsto E3$ added to the *Expstore*. The next step is to build $E2 \vee E3$, $E1 \wedge E2$ and $E1 \wedge E3$. In the correct internal syntax, these are represented respectively as $\mathit{mk\text{-}Or}(r2, r3)$, $\mathit{mk\text{-}And}(r1, r2)$ and $\mathit{mk\text{-}And}(r1, r3)$. Since it was assumed that neither $E2$ nor $E3$ were existing expressions, the closedness condition on the *Expstore* implies that none of these expressions can already exist, so all must be assigned new references, say $r4$,$r5$ and $r6$, and the associations $r4 \mapsto \mathit{mk\text{-}Or}(r2, r3)$, $r5 \mapsto \mathit{mk\text{-}And}(r1, r2)$ and $r6 \mapsto \mathit{mk\text{-}And}(r1, r3)$ must be added to the *Expstore*. The last step is entirely analogous and results in the addition of two new associations to the *Expstore*, $r7 \mapsto \mathit{mk\text{-}And}(r1, r4)$ and $r8 \mapsto \mathit{mk\text{-}Or}(r5, r6)$. Of course, all the r's are simply internal references and are not seen by the user: Muffin always simply displays the formulae in concrete form (e.g. $r8$ will always be displayed as $E1 \wedge E2 \vee E1 \wedge E3$).

If p is some new *Problemref*, the statement of the theorem is completed by adding the association $p \mapsto \mathit{mk\text{-}Problem}(\{r7\}, r8)$ to the *Problemstore*.

The new problem will now appear in the browser as an unsolved problem (cf. Figure 2). By selecting it there, the user can give it a name (*and-or-dist* in Figure 2), associate a new empty incomplete proof with it, and give the new proof a name (also *and-or-dist* in Figure 2). The effect of these actions on the Muffin state is as follows. The first adds the association *and-or-dist* $\mapsto p$ to *ProblemNames*. The second begins by adding the association $f \mapsto []$ to the *Proofstore*, where f is some new *Proofref*, then adds $p \mapsto \{f\}$ to the *Incomplete-proofmap* and $f \mapsto 0$ to the *Indexmap*. The third adds *and-or-dist* $\mapsto f$ to the *ProofNames*.

A side effect of adding the new proof f is that Muffin automatically opens a prover on f. Initially, this shows the problem p in its top pane, and there is a single known $E1 \wedge (E2 \vee E3)$ (i.e. the hypothesis of p), and a single goal, $E1 \wedge E2 \vee E1 \wedge E3$ (i.e. the conclusion of p).

As a first step, the user might select the known $E1 \wedge (E2 \vee E3)$ and ask Muffin to display the matching rules. One of these will be the rule called *and-elim-r*, which corresponds to the problem $\{X \wedge Y\} \vdash X$. Suppose that $q1$ is the reference of this problem in the *Problemstore*. Selection of the rule causes several changes to be made to the state. First, Muffin builds the problem $\{E1 \wedge (E2 \vee E3)\} \vdash E1$ and adds it to the *Problemstore* by adding the association $p1 \mapsto$ *mk-Problem*$(\{r7\}, r1)$ thereto. This problem is, of course, just an instance of the *and-elim-r* rule. It is therefore solved, and its proof is simply the relevant instantiation. The next step is therefore to build the instantiation and add the new proof to the *Proofstore*. This is done by adding the association $f1 \mapsto$ *mk-Instantiation*$(q1, \{X \mapsto r1, Y \mapsto r4\})$ thereto. The fact that the problem $p1$ is solved is then recorded by adding the association $p1 \mapsto \{f1\}$ to the *Proofmap*. Finally, the problem $p1$ is added to the forward proof of the proof f. Thus, in the *Indexmap*, the association $f \mapsto 0$ is replaced by $f \mapsto 1$, and in the *Proofstore* $f \mapsto []$ is replaced by $f \mapsto [p1]$. Since the backward proof is empty, no reorganisation of the backward proof is necessary.

All this behind the scenes activity is, of course, hidden from the user, and the only visible effect is that $E1$ is added to the knowns.

To proceed from here, the user again selects $E1 \wedge (E2 \vee E3)$ from the knowns and asks for the menu of matching rules. This time he selects *and-elim-l*, which corresponds to the problem $\{X \wedge Y\} \vdash Y$. This step proceeds entirely analogously to that just described, so will not be dealt with in detail here. It results in the element $p2$, corresponding to the problem $\{E1 \wedge (E2 \vee E3)\} \vdash E2 \vee E3$, being added to the tail of the forward proof of f and the index of f being incremented by 1. The expression $E2 \vee E3$ now additionally appears in the knowns.

Next, the user selects the goal $E1 \wedge E2 \vee E1 \wedge E3$ in the goals pane and asks for the matching rules. One of these will be the *or-elim* rule, which has the form $\{X \vee Y, \{X\} \leadsto Z, \{Y\} \leadsto Z\} \vdash Z$. The user selects this rule. From the match so far performed, Muffin can deduce that it must instantiate Z to $E1 \wedge E2 \vee E1 \wedge E3$, but not what substitutions to make for the X and Y appearing in the rule. It therefore prompts the user for the required substitutions by opening a substitution editor. He chooses that X should be replaced by $E2$ and Y by $E3$. Muffin then builds the instance of the *or-elim* rule defined by this substitution, namely $\{E2 \vee E3, \{E2\} \leadsto E1 \wedge E2 \vee E1 \wedge E3, \{E3\} \leadsto E1 \wedge E2 \vee E1 \wedge E3\} \vdash E1 \wedge E2 \vee E1 \wedge E3$.

The two subsequents appearing in the hypotheses of this problem are new, so the first step is to add them to the *Subseqstore*. This is achieved by adding the associations $s1 \mapsto$ *mk-Subseq*$(\{r2\}, r8)$ and $s2 \mapsto$ *mk-Subseq*$(\{r3\}, r8)$, where $s1$ and $s2$ are new *Subseqrefs*.

Now the new problem can be included in the *Problemstore* by adding to it the association $p5 \mapsto$ *mk-Problem*$(\{r4, s1, s2\}, r8)$, with $p5$ some new *Proofref*. Next, the new instantiation is added to the *Proofstore* via $f5 \mapsto$ *mk-Instantiation*$(q2, \{X \mapsto r2, Y \mapsto r3, Z \mapsto r8\})$, where $q2$ is the reference of the *or-elim* rule in the *Problemstore*. The association $p5 \mapsto \{f5\}$ is now added to the *Proofmap*, thus marking $p5$ as a solved problem, and the value of f under the *Proofstore* becomes $[p1, p2, p5]$. The value of the index remains unaltered in this case since the step was a backward inference. The goals pane of the prover now shows the two subsequents $\{E2\} \leadsto E1 \wedge E2 \vee E1 \wedge E3$ and $\{E3\} \leadsto E1 \wedge E2 \vee E1 \wedge E3$. Note that the subgoal $E2 \vee E3$ does not appear as it is amongst the current knowns (that is, it is a current goal but not a visible goal).

Next, the user selects the first of these two subsequents in the goals pane and asks Muffin if the problem which would contribute the selected subsequent to the knowns if it were added to the tail of the forward proof has already been solved. Since the variable $E2$ was only recently introduced, it should come as no surprise to him to be told that it is unsolved. He is then offered the chance to open a new prover in order to attempt to solve the problem in question, namely $\{E2, E1 \wedge (E2 \vee E3)\} \vdash E1 \wedge E2 \vee E1 \wedge E3$. Accepting the offer causes the new problem to be added to the *Problemstore* via $p3 \mapsto$ *mk-Problem*$(\{r2, r7\}, r8)$, adds a new empty proof to the *Proofstore* via $f3 \mapsto []$, and designates $f3$ as an incomplete proof of $p3$ by adding $p3 \mapsto \{f3\}$ to the *Incomplete-proofmap* and $f3 \mapsto 0$ to the *Indexmap*.

The user now switches attention to the new problem $p3$ and tries to solve it. By using forward and/or backward inferencing as detailed above, he should eventually arrive at the point at which $f3$ is a composite proof containing (references to) the three elements $\{E1 \wedge (E2 \vee E3)\} \vdash E1$, $\{E1, E2\} \vdash E1 \wedge E2$ and $\{E1 \wedge E2\} \vdash E1 \wedge E2 \vee E1 \wedge E3$, in that order. It is worth noting that the first of these three elements is identical to the first element of the main proof f. It is, however, *not* redundant here, but is, as can readily be seen, a vital step in the proof of $p3$ as it contributes $E1$ to its knowns. Some redundancy does occur, however, when this proof is considered as a subproof of the proof f. In that case, this step contributes $E1$ to the knowns just as the identical step in f does. Muffin realises this when displaying the proof of f in the natural deduction style of Figure 1 and simply doesn't bother to display the known $E1$ when it generates it for the second time as part of the subproof at box 3.

Having completed the proof of $p3$, the user can now return to the proof f of the problem p and ask Muffin again to search for the solved problem it failed to find earlier. Now, of course, the problem is solved, and Muffin duly finds it. The problem $p3$ is added to the tail of the forward proof of f and the index of f becomes 3. The subsequent $\{E2\} \rightsquigarrow E1 \wedge E2 \vee E1 \wedge E3$ vanishes from the goals pane and appears in the knowns pane.

The user completes the proof f by dealing with the remaining subsequent in an entirely analogous way. The goals pane finally shows *"Q.E.D."* to indicate that the proof is complete. The association $f \mapsto 4$ is removed rom the *Indexmap*, and $p \mapsto \{f\}$ is removed from the *Incomplete-proofmap* and added to the *Proofmap*.

The interested reader can find details (in concrete form) of the full proof structure behind the proof *and-or-dist* in Appendix B.

4 Evaluation

As stated earlier, one of the main aims of the Muffin exercise was to develop both a proof style and an interaction style which would encourage a user to use the system to actually discover formal proofs rather than just as a proof checker. Ideas on what form this proof style would take were initially somewhat nebulous, of course, so the first stage of the process consisted simply of designing and specifying a theorem store for simple propositional calculus. The resulting specification [14] then formed the basis for a prototype working system possessing a rudimentary user interface (see [15]).

This prototype system was implemented in Smalltalk–80, a language with which we were totally unfamiliar prior to our using it as the implementation language for Muffin. The second of the main aims of the exercise was therefore to assess the suitability or otherwise of Smalltalk–80 as an implementation language for formal reasoning systems.

One of the main reasons for choosing Smalltalk–80 as opposed to, say, standard ML, was that a lot of basic UI components are already built into the basic Smalltalk system. It is therefore very easy to construct a user interface simply by combining these existing primitives as desired. Moreover, it is also very easy to experiment with different user interface designs by simply combining these primitives in different ways. In fact, it turned out that the choice of language was even more fortuitous, due to the strong natural parallel between on the one hand the abstract data types and the operations and functions defined thereon of the formal specification and on the other hand the object classes and the messages acting thereon of the language. Exploiting this parallel made it possible to progress from the formal specification of the theorem store to a working prototype of Muffin with an expenditure of only two man-months of effort. Such rapid progress was no doubt partly due to the fact that much of the design of the system had been carried out at the specification level, which probably accounted for approximately three quarters of the total effort expended on Muffin. Nevertheless, we do not believe that such rapid progress would have been made had, for instance, standard ML been chosen as the implementation language, because it would not have offered the pre-packaged user interface components. The availability of Smalltalk–80 classes corresponding to VDM's type formation operators (e.g. sets, maps, etc.) made it very easy to implement the specification in this language. We therefore conclude that Smalltalk–80 is eminently suited not only to the implementation of formal reasoning systems but also to the implementation of formally specified systems. For a fuller assessment of these points see [8].

Following the completion of the prototype system, an attempt was made to evaluate the rudimentary interface it possessed by inviting people to come and try to prove theorems using Muffin. Those who took part in this evaluation exercise covered the whole spectrum, from rank novice to complete expert, in terms of familiarity with each of workstation technology and propositional logic, and came from a wide variety of backgrounds. The most important factor influencing the ease with which a given person managed to complete his proof seemed to be his familiarity with the workstation, with the depth of knowledge of propositional calculus being far less significant. Indeed, those expert in workstation technology but having little or no knowledge of propositional logic generally managed to complete their proofs successfully simply by making extensive use of Muffin's pattern matching facilities. Familiarity with Smalltalk–80 was, of course, an extra bonus as the functionality of the mouse parallels that in the standard Smalltalk system closely. Most participants felt that the proof style of Muffin's prover was easier to understand than the natural deduction style of Figure 1.

The evaluation exercise also provided a large and varied range of opinions as to how the prototype Muffin might be modified to make it easier to use. Some of these suggested modifications (e.g. the provision of a parser) went beyond Muffin's stated aim of investigating novel proof and interaction styles for formal reasoning systems, though some of these will undoubtedly be relevant to the design of the general FRIPSE user interface. Those suggested modifications lying within Muffin's scope, along with opinions similarly elicited from colleagues within the FRIPSE project, provided extremely useful input, however, and some of these have since been incorporated into a new version of Muffin.

It was observed above that it is not possible to formally specify (and reason about) properties like "user-friendliness". In keeping with this observation, our development path can be viewed as one of evolution of the user-interface aspects. It should, however, be noted that the changes made during this process were mainly at the level of the surface UI and that the (formally specified) deep UI aspects remained largely constant.

Whilst this evaluation exercise was underway, one of us (RM) was invited to give a talk ([13]) on and a demonstration of the prototype Muffin at the Workshop on Programming for Logic Teaching, held at the University of Leeds, 6-8 July 1987. The participants at this workshop were in the main academics drawn from philosophy, computer science and pure mathematics (with particular emphasis on mathematical logic) departments. Their reactions, as well as the other talks and demonstrations presented at the workshop, also provided much useful input to the considerations of how to modify the prototype. In addition, many of the participants expressed an interest in acquiring a copy of Muffin for their own experimentation and use. This caused us to reconsider our original conception of Muffin as an essentially throwaway experiment within the wider FRIPSE project, and led to our decision to make the new version generally available for research purposes[1].

Up to this point, only the theorem store of Muffin had been formally specified, and all the higher level interactive mechanism had simply been coded on top of that, due mainly to the uncertainty as to the exact form it would take. Having fixed on the version of the system for general release, it was decided that, for the purpose of this exercise of documenting Muffin, the formal specification should be extended to cover the whole of the system as it then stood (apart, of course, from the actual surface UI which we still don't know how to describe formally, but see [12]). It is interesting to note that this additional specification exercise led to the discovery of a couple of bugs in the code!

The importance of the "derived rule" feature cannot be too strongly emphasised. Even in the very simple domain of propositional calculus to which Muffin restricts attention, it soon becomes extremely tedious if a user has to prove everything from the basic axioms. The ability to store a proved result and make use of it in later proofs, thus augmenting the set of rules, makes what might be a very long proof when proved from first principles much shorter, and hence much easier to understand. In fact, it is obvious from the Muffin experiment that a theorem proving system would be unusable without concepts such as derived rules which bring the level of interaction closer to that of normal mathematical reasoning.

If we ourselves look at the evaluation exercise, we cannot help but think Muffin's proof style(s) are a significant step towards the objective of proof discovery at the terminal. People who have never written a proof in a formal logic before have proved theorems using Muffin; what's more they have enjoyed the experience!

5 Beyond Muffin

Within the FRIPSE project as a whole, the Muffin exercise has always been looked upon as a simple experiment aimed at providing input to the eventual design of the general FRIPSE user interface, with the result that it was assigned only limited resources and a fixed duration. These limitations have meant

[1]Contact M.K.Tordoff, STL NW, Copthall House, Nelson Place, Newcastle-under-Lyme, Staffs ST5 1EZ

that some issues, which will obviously be important in FRIPSE, remain unexplored, though after the Muffin exercise some aspects of these are perhaps clearer.

One possible extension to Muffin, namely the generalisation of the syntax of expressions to allow for an essentially user-defined set of logical operators as opposed to the specific set built into Muffin, has, in fact, already been explored. The modifications which have to be made to the specification of Muffin as given in Appendix A in order to achieve this generalisation can be found in [16].

The same document also mentions a second important way in which the syntax of expressions could be extended, namely by introducing the notions of commutativity and associativity of binary operators. This would then permit further generalisation of the syntax of expressions so that expressions built from such operators could be stored in, say, some normal form, with the different equivalent forms being simply different projections of that normal form. This would have the additional implication that certain proofsteps, for example the commuting of the operands of a commutative binary operator, could be made automatic.

A second area in which Muffin could be developed further is that of the rules of inference. One possibility here would be an extension of the mechanism by which matching rules are determined to allow the user to match to more than one expression, for instance to a goal and a known. It is believed that this could go a long way towards reducing the number of rules presented to the user, thus making the choice of rule that much easier. Unfortunately, given enough stamina, a persistent user will eventually be able to create so many derived rules that even this more sophisticated matching process will produce a menu of matching rules containing more elements than the user can comfortably assimilate ("lemma explosion"). Some structuring of the menu of matching rules will therefore also be required. One possibility here would be to order the matching rules according to how closely they match the current selection(s) then present them a few at a time according to this ordering.

Another way in which the task of proving some theorem might be simplified is via the use of tactics. Some thought has been given to how tactics might be incorporated into Muffin, with the conclusion that it would be relatively straightforward to include a simple tactic language consisting of a set of combinators for derived rules, with a tactic being applied in much the same way as a derived rule, but that it was unclear how to deal with anything more sophisticated.

This document marks the end of the "official" work on Muffin, and attention is now being switched to the formal specification of FRIPSE as a whole, with work on outstanding issues, such as the general treatment of tactics, genericity over logics and proof styles, etc., proceeding in parallel with this. Some details of this work can be found in [9] and [10]. Muffin, of course, remains available as a test bed on which we can easily try out any new ideas arising from this more general FRIPSE work.

Acknowledgements

Our thanks to all our colleagues in the FRIPSE project at Manchester and RAL for many useful discussions and comments as the work on Muffin progressed, and to those members of the wider IPSE 2.5 project who provided similar contributions. Thanks also to both the Alvey Directorate and the SERC for financial support, and to John Derrick of Leeds University for inviting us to present our work at the Workshop on Programming for Logic Teaching held there.

A Formal Specification

A.1 Muffin

$$\begin{aligned}
\textit{Muffin} :: \; & es : \textit{Expstore} \\
& ss : \textit{Subseqstore} \\
& ps : \textit{Problemstore} \\
& fs : \textit{Proofstore} \\
& en : \textit{ExpNames} \\
& sn : \textit{SubseqNames} \\
& pn : \textit{ProblemNames} \\
& fn : \textit{ProofNames} \\
& jm : \textit{Proofmap} \\
& rm : \textit{Rulemap} \\
& im : \textit{Incomplete-proofmap} \\
& xm : \textit{Indexmap}
\end{aligned}$$

where

$\textit{inv-Muffin}(\textit{mk-Muffin}(es, ss, ps, fs, en, sn, pn, fn, jm, rm, im, xm)) \quad \triangleq$

$$\begin{aligned}
& \textit{is-valid-subseqstore}(ss, es) \wedge \\
& \textit{is-valid-problemstore}(ps, ss, es) \wedge \\
& \textit{is-valid-proofstore}(fs, ps, ss, es) \wedge \\
& \textit{is-valid-expnames}(en, es) \wedge \\
& \textit{is-valid-subseqnames}(sn, ss) \wedge \\
& \textit{is-valid-problemnames}(pn, ps) \wedge \\
& \textit{is-valid-proofnames}(fn, fs) \wedge \\
& \textit{is-valid-proofmap}(jm, fs, ps, ss, es) \wedge \\
& \textit{is-valid-rulemap}(rm, jm, fs) \wedge \\
& \textit{is-valid-incomplete-proofmap}(im, jm, fs, ps, ss, es) \wedge \\
& \textit{is-valid-indexmap}(xm, im, jm, fs, ps, ss, es)
\end{aligned}$$

$\textit{add-exp} \;\; (x\colon \textit{Exp}) \; y\colon \textit{Expref}$

ext wr es : $\textit{Expstore}$

pre $\textit{args}(x) \subseteq \operatorname{dom} es$

post $x \in \operatorname{rng} \overleftarrow{es} \wedge y \in \operatorname{dom} \overleftarrow{es} \wedge \overleftarrow{es}(y) = x \wedge es = \overleftarrow{es} \vee$
$\quad x \notin \operatorname{rng} \overleftarrow{es} \wedge y \notin \operatorname{dom} \overleftarrow{es} \wedge es = \overleftarrow{es} \cup \{y \mapsto x\}$

$\textit{add-subseq} \;\; (z\colon \text{set of } \textit{Expref}, y\colon \textit{Expref}) \; g\colon \textit{Subseqref}$

ext rd es : $\textit{Expstore}$
 wr ss : $\textit{Subseqstore}$

pre $z \cup \{y\} \subseteq \operatorname{dom} es \wedge y \notin z \wedge z \neq \{\,\}$

post let $t = \textit{mk-Subseq}(z, y)$ in
$\quad t \in \operatorname{rng} \overleftarrow{ss} \wedge g \in \operatorname{dom} \overleftarrow{ss} \wedge \overleftarrow{ss}(g) = t \wedge ss = \overleftarrow{ss} \vee$
$\quad t \notin \operatorname{rng} \overleftarrow{ss} \wedge g \notin \operatorname{dom} \overleftarrow{ss} \wedge ss = \overleftarrow{ss} \cup \{g \mapsto t\}$

$\textit{add-problem} \;\; (n\colon \text{set of } \textit{Node}, y\colon \textit{Expref}) \; u\colon \textit{Problemref}$

ext rd es : $\textit{Expstore}$
  ss : $\textit{Subseqstore}$
 wr ps : $\textit{Problemstore}$

pre $y \in \operatorname{dom} es \wedge n \subseteq \operatorname{dom} es \cup \operatorname{dom} ss \wedge y \notin n$

```
post let $t = mk\text{-}Problem(n, y)$ in
    $t \in \mathsf{rng}\,\overleftarrow{ps} \land u \in \mathsf{dom}\,\overleftarrow{ps} \land \overleftarrow{ps}(u) = t \land ps = \overleftarrow{ps}\ \lor$
    $t \notin \mathsf{rng}\,\overleftarrow{ps} \land u \notin \mathsf{dom}\,\overleftarrow{ps} \land ps = \overleftarrow{ps} \cup \{u \mapsto t\}$
```

```
$instantiate\text{-}exp\ \ (y{:}\,Expref, m{:}\,\mathsf{map}\ Atom\ \mathsf{to}\ Expref)\ r{:}\,Expref$
ext wr $es\ :\ Expstore$
pre $y \in \mathsf{dom}\, es \land is\text{-}substitution(\{y\}, m, \{\,\}, es)$
post $\overleftarrow{es} \subseteq es \land r \in \mathsf{dom}\, es \land is\text{-}exp\text{-}match(y, r, m, es)\ \land$
    $\mathsf{dom}\, es = \mathsf{dom}\,\overleftarrow{es} \cup descendents(\{r\}, es)$
```

```
$instantiate\text{-}exp\text{-}set\ \ (y{:}\,\mathsf{set\ of}\ Expref, m{:}\,\mathsf{map}\ Atom\ \mathsf{to}\ Expref)\ r{:}\,\mathsf{set\ of}\ Expref$
ext wr $es\ :\ Expstore$
pre $y \subseteq \mathsf{dom}\, es \land is\text{-}substitution(y, m, \{\,\}, es)$
post $\overleftarrow{es} \subseteq es \land r \subseteq \mathsf{dom}\, es \land is\text{-}exp\text{-}set\text{-}match(y, r, m, es)\ \land$
    $\mathsf{dom}\, es = \mathsf{dom}\,\overleftarrow{es} \cup descendents(r, es)$
```

```
$instantiate\text{-}subseq\ \ (y{:}\,Subseqref, m{:}\,\mathsf{map}\ Atom\ \mathsf{to}\ Expref)\ r{:}\,Subseqref$
ext wr $es\ :\ Expstore$
       $ss\ :\ Subseqstore$
pre $y \in \mathsf{dom}\, ss \land is\text{-}substitution(\{y\}, m, ss, es)$
post $\overleftarrow{ss} \subseteq ss \land \mathsf{dom}\, ss = \mathsf{dom}\,\overleftarrow{ss} \cup \{r\}\ \land$
    $post\text{-}instantiate\text{-}exp\text{-}set(exps(ss(y)), m, \overleftarrow{es}, exps(ss(r)), es)$
    $\land\ is\text{-}subseq\text{-}match(y, r, m, ss, es)$
```

```
$instantiate\text{-}node\ \ (n{:}\,Node, m{:}\,\mathsf{map}\ Atom\ \mathsf{to}\ Expref)\ r{:}\,Node$
ext wr $es\ :\ Expstore$
       $ss\ :\ Subseqstore$
pre $n \in \mathsf{dom}\, es \cup \mathsf{dom}\, ss \land is\text{-}substitution(\{n\}, m, ss, es)$
post $n \in \mathsf{dom}\, ss \land post\text{-}instantiate\text{-}subseq(n, m, \overleftarrow{es}, \overleftarrow{ss}, r, es, ss)\ \lor$
    $n \in \mathsf{dom}\, es \land post\text{-}instantiate\text{-}exp(n, m, \overleftarrow{es}, r, es) \land \overleftarrow{ss} = ss$
```

```
$instantiate\text{-}node\text{-}set\ \ (n{:}\,\mathsf{set\ of}\ Node, m{:}\,\mathsf{map}\ Atom\ \mathsf{to}\ Expref)\ r{:}\,\mathsf{set\ of}\ Node$
ext wr $es\ :\ Expstore$
       $ss\ :\ Subseqstore$
pre $n \subseteq \mathsf{dom}\, es \cup \mathsf{dom}\, ss \land is\text{-}substitution(n, m, ss, es)$
post $\overleftarrow{ss} \subseteq ss \land \overleftarrow{es} \subseteq es \land r \subseteq \mathsf{dom}\, es \cup \mathsf{dom}\, ss \land is\text{-}node\text{-}set\text{-}match(n, r, m, ss, es)\ \land$
    $\mathsf{dom}\, es = \mathsf{dom}\,\overleftarrow{es} \cup descendents(components(r, ss, es), es)\ \land$
    $\mathsf{dom}\, ss = \mathsf{dom}\,\overleftarrow{ss} \cup (r \cap \mathsf{dom}\, ss)$
```

```
$instantiate\text{-}problem\ \ (p{:}\,Problemref, m{:}\,\mathsf{map}\ Atom\ \mathsf{to}\ Expref)\ r{:}\,Problemref$
ext wr $es\ :\ Expstore$
       $ss\ :\ Subseqstore$
       $ps\ :\ Problemstore$
pre $p \in \mathsf{dom}\, ps \land is\text{-}substitution(nodes(ps(p)), m, ss, es)$
post $\overleftarrow{ps} \subseteq ps \land \mathsf{dom}\, ps = \mathsf{dom}\,\overleftarrow{ps} \cup \{r\} \land is\text{-}problem\text{-}match(p, r, m, ps, ss, es)\ \land$
    $post\text{-}instantiate\text{-}node\text{-}set(nodes(ps(p)), m, \overleftarrow{es}, \overleftarrow{ss}, nodes(ps(r)), es, ss)$
```

name-exp $(n\colon String,\ e\colon Expref)$

ext wr en : $ExpNames$
    rd es : $Expstore$

pre $e \in \mathrm{dom}\, es \wedge (n \in \mathrm{dom}\, en \wedge en(n) = e \vee n \notin \mathrm{dom}\, en)$

post $n \in \mathrm{dom}\, \overleftarrow{en} \wedge en = \overleftarrow{en} \vee n = [\,] \wedge en = \{e\} \rhd \overleftarrow{en} \vee$
    $n \notin \mathrm{dom}\, \overleftarrow{en} \wedge n \neq [\,] \wedge en = (\{e\} \rhd \overleftarrow{en}) \cup \{n \mapsto e\}$

name-subseq $(n\colon String,\ s\colon Subseqref)$

ext wr sn : $SubseqNames$
    rd ss : $Subseqstore$

pre $s \in \mathrm{dom}\, ss \wedge (n \in \mathrm{dom}\, sn \wedge sn(n) = s \vee n \notin \mathrm{dom}\, sn)$

post $n \in \mathrm{dom}\, \overleftarrow{sn} \wedge sn = \overleftarrow{sn} \vee n = [\,] \wedge sn = \{s\} \rhd \overleftarrow{sn} \vee$
    $n \notin \mathrm{dom}\, \overleftarrow{sn} \wedge n \neq [\,] \wedge sn = (\{s\} \rhd \overleftarrow{sn}) \cup \{n \mapsto s\}$

name-problem $(n\colon String,\ p\colon Problemref)$

ext wr pn : $ProblemNames$
    rd ps : $Problemstore$

pre $p \in \mathrm{dom}\, ps \wedge (n \in \mathrm{dom}\, pn \wedge pn(n) = p \vee n \notin \mathrm{dom}\, pn)$

post $n \in \mathrm{dom}\, \overleftarrow{pn} \wedge pn = \overleftarrow{pn} \vee n = [\,] \wedge pn = \{p\} \rhd \overleftarrow{pn} \vee$
    $n \notin \mathrm{dom}\, \overleftarrow{pn} \wedge n \neq [\,] \wedge pn = (\{p\} \rhd \overleftarrow{pn}) \cup \{n \mapsto p\}$

name-proof $(n\colon String,\ f\colon Proofref)$

ext wr fn : $ProofNames$
    rd fs : $Proofstore$

pre $f \in \mathrm{dom}\, fs \wedge (n \in \mathrm{dom}\, fn \wedge fn(n) = f \vee n \notin \mathrm{dom}\, fn)$

post $n \in \mathrm{dom}\, \overleftarrow{fn} \wedge fn = \overleftarrow{fn} \vee n = [\,] \wedge fn = \{f\} \rhd \overleftarrow{fn} \vee$
    $n \notin \mathrm{dom}\, \overleftarrow{fn} \wedge n \neq [\,] \wedge fn = (\{f\} \rhd \overleftarrow{fn}) \cup \{n \mapsto f\}$

remove-problem $(p\colon Problemref)$

ext wr ps : $Problemstore$
        fs : $Proofstore$
        im : $Incomplete\text{-}proofmap$
        xm : $Indexmap$
        pn : $ProblemNames$
        fn : $ProofNames$
    rd jm : $Proofmap$

pre $p \in \mathrm{dom}\, ps \wedge p \notin \mathrm{dom}\, jm$

post $ps = \{p\} \lhd \overleftarrow{ps} \wedge pn = \{p\} \rhd \overleftarrow{pn} \wedge$
    $(p \in \mathrm{dom}\, \overleftarrow{im} \wedge im = \{p\} \lhd \overleftarrow{im} \wedge fs = \overleftarrow{im}(p) \lhd \overleftarrow{fs} \wedge xm = \overleftarrow{im}(p) \lhd \overleftarrow{xm} \wedge fn = \overleftarrow{im}(p) \rhd \overleftarrow{fn}$
    $\vee\ p \notin \mathrm{dom}\, \overleftarrow{im} \wedge im = \overleftarrow{im} \wedge fs = \overleftarrow{fs} \wedge xm = \overleftarrow{xm} \wedge fn = \overleftarrow{fn})$

remove-proof $(f\colon Proofref)$

ext wr fs : $Proofstore$
        im : $Incomplete\text{-}proofmap$
        xm : $Indexmap$
        fn : $Proofnames$

pre $f \in incomplete\text{-}proofs(im)$

post $fs = \{f\} ⩤ \overleftarrow{fs} \wedge xm = \{f\} ⩤ \overleftarrow{xm} \wedge fn = \{f\} ⩥ \overleftarrow{fn} \wedge p \in \mathsf{dom}\, \overleftarrow{im} \wedge f \in \overleftarrow{im}(p) \wedge$
$\quad (\overleftarrow{im}(p) = \{f\} \wedge im = \{p\} ⩤ \overleftarrow{im} \vee \overleftarrow{im}(p) \neq \{f\} \wedge im = \overleftarrow{im} \dagger \{p \mapsto \overleftarrow{im}(p) - \{f\}\})$

name-rule $(n\colon String, p\colon Problemref)$
ext wr rm : *Rulemap*
$\quad$ rd jm : *Proofmap*
pre $n \neq [\,] \wedge p \in \mathsf{dom}\, jm \wedge (n \in \mathsf{dom}\, rm \;\Rightarrow\; rm(n) = p)$
post $rm = (\{p\} ⩥ \overleftarrow{rm}) \cup \{n \mapsto p\}$

make-axiom $(p\colon Problemref, n\colon String)$
ext wr im : *Incomplete-proofmap*
$\quad jm$: *Proofmap*
$\quad rm$: *Rulemap*
$\quad xm$: *Indexmap*
$\quad fs$: *Proofstore*
$\quad fn$: *ProofNames*
pre $n \neq [\,] \wedge [p \in axioms(jm) \wedge (n \in \mathsf{dom}\, rm \;\Rightarrow\; rm(n) = p) \vee p \notin \mathsf{dom}\, jm]$
post $p \notin \mathsf{dom}\, \overleftarrow{jm} \wedge jm = \overleftarrow{jm} \cup \{p \mapsto \{\,\}\} \wedge rm = \overleftarrow{rm} \cup \{n \mapsto p\} \wedge im = \{p\} ⩤ \overleftarrow{im} \wedge$
$\quad (p \in \mathsf{dom}\, \overleftarrow{im} \wedge xm = \overleftarrow{im}(p) ⩤ \overleftarrow{xm} \wedge fs = \overleftarrow{im}(p) ⩤ \overleftarrow{fs} \wedge fn = \overleftarrow{im}(p) ⩤ \overleftarrow{fn} \vee$
$\quad p \notin \mathsf{dom}\, \overleftarrow{im} \wedge xm = \overleftarrow{xm} \wedge fs = \overleftarrow{fs} \wedge fn = \overleftarrow{fn}) \vee$
$\quad p \in \mathsf{dom}\, \overleftarrow{jm} \wedge im = \overleftarrow{im} \wedge xm = \overleftarrow{xm} \wedge jm = \overleftarrow{jm} \wedge rm = (\{p\} ⩥ \overleftarrow{rm}) \cup \{n \mapsto p\}$
$\quad \wedge fs = \overleftarrow{fs} \wedge fn = \overleftarrow{fn}$

add-instantiation $(p\colon Problemref, m\colon$ map $Atom$ to $Expref, q\colon Problemref)$
ext wr fs : *Proofstore*
$\quad jm$: *Proofmap*
$\quad$ rd rm : *Rulemap*
$\quad ps$: *Problemstore*
$\quad ss$: *Subseqstore*
$\quad es$: *Expstore*
pre $p \in \mathsf{rng}\, rm \wedge m \neq \{\,\} \wedge \mathsf{dom}\, m \subseteq vars(nodes(ps(p)), ss, es) \wedge$
$\quad is\text{-}substitution(nodes(ps(p)), m, ss, es) \wedge is\text{-}problem\text{-}match(p, q, m, ps, ss, es) \wedge$
$\quad q \notin axioms(jm)$
post let $i = mk\text{-}Instantiation(p, m)$ in
$\quad [i \in \mathsf{rng}\, \overleftarrow{fs} \wedge f \in \mathsf{dom}\, \overleftarrow{fs} \wedge \overleftarrow{fs}(f) = i \wedge \overleftarrow{fs} = fs \vee$
$\quad i \notin \mathsf{rng}\, \overleftarrow{fs} \wedge f \notin \mathsf{dom}\, \overleftarrow{fs} \wedge fs = \overleftarrow{fs} \cup \{f \mapsto i\}] \wedge [q \in \mathsf{dom}\, \overleftarrow{jm} \wedge s = \overleftarrow{jm}(q) \cup \{f\} \vee$
$\quad q \notin \mathsf{dom}\, \overleftarrow{jm} \wedge s = \{f\}] \wedge jm = \overleftarrow{jm} \dagger \{q \mapsto s\}$

add-assumption $(p\colon Problemref)$
ext wr fs : *Proofstore*
$\quad jm$: *Proofmap*
$\quad$ rd ps : *Problemstore*
pre $p \in \mathsf{dom}\, ps \wedge p \notin axioms(jm) \wedge con(ps(p)) \in hyp(ps(p))$
post $f \notin \mathsf{dom}\, \overleftarrow{fs} \wedge fs = \overleftarrow{fs} \cup \{f \mapsto [\,]\} \wedge$
$\quad (p \in \mathsf{dom}\, \overleftarrow{jm} \wedge jm = \overleftarrow{jm} \dagger \{p \mapsto \overleftarrow{jm}(p) \cup \{f\}\} \vee p \notin \mathsf{dom}\, \overleftarrow{jm} \wedge jm = \overleftarrow{jm} \cup \{p \mapsto \{f\}\})$

```
$add\text{-}empty\text{-}proof\ \ (p\colon Problemref)$
ext wr $im$ : $Incomplete\text{-}proofmap$
       $xm$ : $Indexmap$
       $fs$ : $Proofstore$
    rd $jm$ : $Proofmap$
       $ps$ : $Problemstore$
pre $p \notin axioms(jm) \wedge p \in \mathrm{dom}\, ps \wedge con(ps(p)) \notin hyp(ps(p))$
post $f \notin \mathrm{dom}\, \overleftarrow{fs} \wedge fs = \overleftarrow{fs} \cup \{f \mapsto [\,]\} \wedge xm = \overleftarrow{xm} \cup \{f \mapsto 0\} \wedge$
    $(p \in \mathrm{dom}\, \overleftarrow{im} \wedge im = \overleftarrow{im} \dagger \{p \mapsto \overleftarrow{im}(p) \cup \{f\}\} \vee p \notin \mathrm{dom}\, \overleftarrow{im} \wedge im = \overleftarrow{im} \cup \{p \mapsto \{f\}\})$
```

```
$spawn\text{-}proof\ \ (p\colon Problemref, f\colon Proofref)$
ext wr $im$ : $Incomplete\text{-}proofmap$
       $xm$ : $Indexmap$
       $fs$ : $Proofstore$
pre $p \in \mathrm{dom}\, im \wedge f \in im(p)$
post $g \notin \mathrm{dom}\, \overleftarrow{fs} \wedge fs = \overleftarrow{fs} \cup \{g \mapsto \overleftarrow{fs}(f)\} \wedge xm = \overleftarrow{xm} \cup \{g \mapsto \overleftarrow{xm}(f)\} \wedge$
    $im = \overleftarrow{im} \dagger \{p \mapsto \overleftarrow{im}(p) \cup \{g\}\}$
```

```
$add\text{-}fwd\text{-}step\ \ (p\colon Problemref, f\colon Proofref, s\colon Problemref)$
ext wr $fs$ : $Proofstore$
       $im$ : $Incomplete\text{-}proofmap$
       $xm$ : $Indexmap$
       $jm$ : $Proofmap$
    rd $ps$ : $Problemstore$
       $ss$ : $Subseqstore$
       $es$ : $Expstore$
pre let $k = knowns(p, hyp(ps(p)), forward\text{-}proof(f, fs, xm), ps, ss)$ in
    $p \in \mathrm{dom}\, im \wedge f \in im(p) \wedge s \in \mathrm{dom}\, jm \wedge adds\text{-}known(p, k, s, ps, ss)$
post let $y = forward\text{-}proof(f, \overleftarrow{fs}, \overleftarrow{xm}) \curvearrowright [s]$,
         $z = backward\text{-}proof(f, \overleftarrow{fs}, \overleftarrow{xm})$,
         $k = knowns(p, hyp(ps(p)), y, ps, ss)$,
         $l = new\text{-}fwd\text{-}steps(k, z, ps)$,
         $bwd = new\text{-}bwd\text{-}steps(\{con(ps(p))\}, reverse(\mathrm{rng}\, l \rhd z), ps)$,
         $fwd = y \curvearrowright l$,
         $new\text{-}proof = fwd \curvearrowright bwd$
         in
    $\neg(is\text{-}complete\text{-}proof(fwd, p, ps, ss, es)) \wedge fs = \overleftarrow{fs} \dagger \{f \mapsto new\text{-}proof\} \wedge$
    $jm = \overleftarrow{jm} \wedge im = \overleftarrow{im} \wedge xm = \overleftarrow{xm} \dagger \{f \mapsto \overleftarrow{xm}(f) + \mathrm{len}\, l + 1\} \vee$
    $is\text{-}complete\text{-}proof(fwd, p, ps, ss, es) \wedge xm = \{f\} \ntriangleleft \overleftarrow{xm} \wedge$
    $(\overleftarrow{im}(p) = \{f\} \wedge im = \{p\} \ntriangleleft \overleftarrow{im} \vee \overleftarrow{im}(p) \neq \{f\} \wedge im = \overleftarrow{im} \dagger \{p \mapsto \overleftarrow{im}(p) - \{f\}\})$
```

```
$add\text{-}bwd\text{-}step\ \ (p\colon Problemref, f\colon Proofref, s\colon Problemref)$
ext wr $fs$ : $Proofstore$
    rd $im$ : $Incomplete\text{-}proofmap$
       $xm$ : $Indexmap$
       $jm$ : $Proofmap$
       $ps$ : $Problemstore$
       $ss$ : $Subseqstore$
```

pre let $k = knowns(p, hyp(ps(p)), forward\text{-}proof(f, fs, xm), ps, ss)$,
$\quad g = goals(\{con(ps(p))\}, reverse(backward\text{-}proof(f, fs, xm)), ps)$
in
$\quad p \in \mathrm{dom}\, im \land f \in im(p) \land s \in \mathrm{dom}\, jm \land \neg(hyp(ps(s)) \subseteq k) \land con(ps(s)) \in g - k$

post let $new\text{-}proof = forward\text{-}proof(f, \overleftarrow{fs}, xm) \curvearrowright [s] \curvearrowright backward\text{-}proof(f, \overleftarrow{fs}, xm)$ in
$\quad fs = \overleftarrow{fs} \dagger \{f \mapsto new\text{-}proof\}$

undo-fwd-step $(p\colon Problemref, f\colon Proofref)$
ext wr $fs : Proofstore$
$\quad xm : Indexmap$
rd $im : Incomplete\text{-}proofmap$
pre $p \in \mathrm{dom}\, im \land f \in im(p) \land xm(f) \neq 0$
post $xm = \overleftarrow{xm} \dagger \{f \mapsto \overleftarrow{xm}(f)\text{-}1\} \land$
$\quad fs = \overleftarrow{fs} \dagger \{f \mapsto \overleftarrow{xm}(f) \lhd \overleftarrow{fs}(f)\}$

undo-bwd-step $(p\colon Problemref, f\colon Proofref)$
ext wr $fs : Proofstore$
$\quad xm : Indexmap$
rd $im : Incomplete\text{-}proofmap$
pre $p \in \mathrm{dom}\, im \land f \in im(p) \land xm(f) \neq \mathrm{len}\, fs(f)$
post $xm = \overleftarrow{xm} \land fs = \overleftarrow{fs} \dagger \{f \mapsto (xm(f) + 1) \lhd \overleftarrow{fs}(f)\}$

A.2 Expressions

$Texp = Tnot \mid Tand \mid Tor \mid Timpl \mid Tequiv \mid Tdelta \mid Atom$

$Tnot :: tn : Texp$

$Tand :: tandl : Texp$
$\quad tandr : Texp$

$Tor :: torl : Texp$
$\quad torr : Texp$

$Timpl :: tant : Texp$
$\quad tcon : Texp$

$Tequiv :: teql : Texp$
$\quad teqr : Texp$

$Tdelta :: td : Texp$

$Exp = Not \mid And \mid Or \mid Impl \mid Equiv \mid Delta \mid Atom$

$Not :: not : Expref$

$And :: andl : Expref$
$\quad andr : Expref$

$Or :: orl : Expref$
$\quad orr : Expref$

$Impl :: ant : Expref$
$\quad con : Expref$

```
$Equiv$ :: $eql$ : $Expref$
           $eqr$ : $Expref$

$Delta$ :: $del$ : $Expref$

$Expstore$ = map $Expref$ into $Exp$
where

$inv\text{-}Expstore(es) \quad \triangleq \quad is\text{-}closed(es) \wedge is\text{-}finite(es)$


$args : Exp \rightarrow$ set of $Expref$

$args(x) \quad \triangleq$  cases $x$ of
                 $mk\text{-}And(l, r) \rightarrow \{l, r\}$
                 $mk\text{-}Or(l, r) \rightarrow \{l, r\}$
                 $mk\text{-}Impl(l, r) \rightarrow \{l, r\}$
                 $mk\text{-}Equiv(l, r) \rightarrow \{l, r\}$
                 $mk\text{-}Not(l) \rightarrow \{l\}$
                 $mk\text{-}Delta(l) \rightarrow \{l\}$
                 $Atom \rightarrow \{\,\}$
                 end


$is\text{-}closed$ : map $Expref$ to $Exp \rightarrow \mathbb{B}$

$is\text{-}closed(m) \quad \triangleq \quad \forall x \in$ rng $m \cdot args(x) \subseteq$ dom $m$


$offspring$ : set of $Expref \times$ map $Expref$ to $Exp \rightarrow$ set of $Expref$

$offspring(z, m) \quad \triangleq \quad \bigcup\{args(m(y)) \mid y \in$ dom $m \cap z\} \cup z$


$descendents$ : set of $Expref \times$ map $Expref$ to $Exp \rightarrow$ set of $Expref$

$descendents(z, m) \quad \triangleq$
    let $l = offspring(z, m)$ in
    if $l = z$
    then $z$
    else $descendents(l, m)$


$trace$ : set of $Expref \times$ map $Expref$ to $Exp \rightarrow$ map $Expref$ to $Exp$

$trace(z, m) \quad \triangleq \quad descendents(z, m) \lhd m$


$is\text{-}finite$ : map $Expref$ to $Exp \rightarrow \mathbb{B}$

$is\text{-}finite(m) \quad \triangleq \quad \forall y \in$ dom $m \cdot \nexists x \in$ rng $trace(\{y\}, m) \cdot y \in args(x)$


$leaves$ ($z$: set of $Expref$, $es$: $Expstore$) $r$: set of $Atom$
pre $z \subseteq$ dom $es$
post $r = \{x \mid x \in$ rng $trace(z, es) \wedge is\text{-}Atom(x)\}$
```

```
expand (y: Expref, es: Expstore) t: Texp
pre $y \in \mathrm{dom}\ es$
post $t =$ cases $es(y)$ of
        $mk\text{-}Not(l) \rightarrow mk\text{-}Tnot(expand(l, es))$
        $mk\text{-}And(l, r) \rightarrow mk\text{-}Tand(expand(l, es), expand(r, es))$
        $mk\text{-}Or(l, r) \rightarrow mk\text{-}Tor(expand(l, es), expand(r, es))$
        $mk\text{-}Impl(l, r) \rightarrow mk\text{-}Timpl(expand(l, es), expand(r, es))$
        $mk\text{-}Equiv(l, r) \rightarrow mk\text{-}Tequiv(expand(l, es), expand(r, es))$
        $mk\text{-}Delta(l) \rightarrow mk\text{-}Tdelta(expand(l, es))$
        $Atom \rightarrow es(y)$
        end
```

```
expand-inst (y: Expref, m: map Atom to Expref, es: Expstore) t: Texp
pre $y \in \mathrm{dom}\ es \wedge is\text{-}substitution(\{y\}, m, \{\,\}, es)$
post let $x = es(y)$ in
    $t =$ if $x \in \mathrm{dom}\ m$
        then $expand(m(x), es)$
        else cases $x$ of
            $mk\text{-}Not(l) \rightarrow mk\text{-}Tnot(expand\text{-}inst(l, m, es))$
            $mk\text{-}And(l, r) \rightarrow mk\text{-}Tand(expand\text{-}inst(l, m, es), expand\text{-}inst(r, m, es))$
            $mk\text{-}Or(l, r) \rightarrow mk\text{-}Tor(expand\text{-}inst(l, m, es), expand\text{-}inst(r, m, es))$
            $mk\text{-}Impl(l, r) \rightarrow mk\text{-}Timpl(expand\text{-}inst(l, m, es), expand\text{-}inst(r, m, es))$
            $mk\text{-}Equiv(l, r) \rightarrow mk\text{-}Tequiv(expand\text{-}inst(l, m, es), expand\text{-}inst(r, m, es))$
            $mk\text{-}Delta(l) \rightarrow mk\text{-}Tdelta(expand\text{-}inst(l, m, es))$
            $Atom \rightarrow x$
            end
```

```
is-exp-match (x: Expref, y: Expref, m: map Atom to Expref, es: Expstore) r: $\mathbb{B}$
pre $x, y \in \mathrm{dom}\ es \wedge is\text{-}substitution(\{x\}, m, \{\,\}, es)$
post $r \Leftrightarrow expand\text{-}inst(x, m, es) = expand(y, es)$
```

```
is-exp-set-match (z: set of Expref, a: set of Expref, m: map Atom to Expref, es: Expstore) r: $\mathbb{B}$
pre $(z \cup a) \subseteq \mathrm{dom}\ es \wedge is\text{-}substitution(z, m, \{\,\}, es)$
post $r \Leftrightarrow \{expand\text{-}inst(x, m, es) \mid x \in z\} = \{expand(y, es) \mid y \in a\}$
```

A.3 Subsequents

```
Tsubseq :: tlhs : set of Texp
           trhs : Texp
where
```

$inv\text{-}Tsubseq(mk\text{-}Tsubseq(l, r)) \quad \triangleq \quad l \neq \{\,\}$

```
Subseq :: lhs : set of Expref
          rhs : Expref
where
```

$inv\text{-}Subseq(mk\text{-}Subseq(z, y)) \quad \triangleq \quad z \neq \{\,\}$

$Subseqstore = \text{map } Subseqref \text{ into } Subseq$

$exps : Subseq \rightarrow \text{set of } Expref$

$exps(q) \quad \triangleq \quad lhs(q) \cup \{rhs(q)\}$

$is\text{-}valid\text{-}subseqstore : Subseqstore \times Expstore \rightarrow \mathbb{B}$

$is\text{-}valid\text{-}subseqstore(ss, es) \quad \triangleq \quad \forall q \in \text{rng } ss \cdot exps(q) \subseteq \text{dom } es$

$expand\text{-}subseq \;\; (q\colon Subseqref, m\colon \text{map } Atom \text{ to } Expref, ss\colon Subseqstore, es\colon Expstore) \; t\colon Tsubseq$
pre $q \in \text{dom } ss \wedge is\text{-}substitution(\{q\}, m, ss, es) \wedge is\text{-}valid\text{-}subseqstore(ss, es)$
post let $r = expand\text{-}inst(rhs(ss(q)), m, es),$
$\quad\quad l = \{expand\text{-}inst(y, m, es) \mid y \in lhs(ss(q))\}$ in
$\quad t = mk\text{-}Tsubseq(l, r)$

$is\text{-}subseq\text{-}match \;\; (g\colon Subseqref, q\colon Subseqref, m\colon \text{map } Atom$
$\quad\quad \text{to } Expref, ss\colon Subseqstore, es\colon Expstore) \; r\colon \mathbb{B}$
pre $g, q \in \text{dom } ss \wedge is\text{-}valid\text{-}subseqstore(ss, es) \wedge is\text{-}substitution(\{g\}, m, ss, es)$
post $r \;\Leftrightarrow\; expand\text{-}subseq(g, m, ss, es) = expand\text{-}subseq(q, \{\,\}, ss, es)$

A.4 Nodes

$Tnode = Texp \mid Tsubseq$

$Node = Expref \mid Subseqref$

$parts \;\; (n\colon Node, ss\colon Subseqstore, es\colon Expstore) \; r\colon \text{set of } Expref$
pre $n \in \text{dom } ss \cup \text{dom } es$
post $(n \in \text{dom } es \wedge r = \{n\}) \vee (n \in \text{dom } ss \wedge r = exps(ss(n)))$

$components \;\; (n\colon \text{set of } Node, ss\colon Subseqstore, es\colon Expstore) \; r\colon \text{set of } Expref$
pre $n \subseteq \text{dom } ss \cup \text{dom } es$
post $r = \bigcup\{parts(k, ss, es) \mid k \in n\}$

$vars \;\; (n\colon \text{set of } Node, ss\colon Subseqstore, es\colon Expstore) \; r\colon \text{set of } Atom$
pre $is\text{-}valid\text{-}subseqstore(ss, es) \wedge n \subseteq \text{dom } es \cup \text{dom } ss$
post $r = leaves(components(n, ss, es), es)$

$is\text{-}substitution \;\; (n\colon \text{set of } Node, m\colon \text{map } Atom \text{ to } Expref, ss\colon Subseqstore, es\colon Expstore) \; r\colon \mathbb{B}$
pre $is\text{-}valid\text{-}subseqstore(ss, es) \wedge n \subseteq \text{dom } es \cup \text{dom } ss$
post $r \;\Leftrightarrow\; \forall x \in \text{dom } m \cdot x \in vars(n, ss, es) \;\Rightarrow\; m(x) \in \text{dom } es \wedge es(m(x)) \neq x$

$expand\text{-}node \;\; (n\colon Node, m\colon \text{map } Atom \text{ to } Expref, ss\colon Subseqstore, es\colon Expstore) \; t\colon Tnode$
pre $n \in \text{dom } ss \cup \text{dom } es \wedge is\text{-}valid\text{-}subseqstore(ss, es) \wedge is\text{-}substitution(\{n\}, m, ss, es)$
post $n \in \text{dom } ss \wedge t = expand\text{-}subseq(n, m, ss, es) \vee$
$\quad n \in \text{dom } es \wedge t = expand\text{-}inst(n, m, es)$

$is\text{-}node\text{-}match\ \ (n\colon Node, k\colon Node, m\colon \text{map } Atom \text{ to } Expref, ss\colon Subseqstore, es\colon Expstore)\ r\colon \mathbb{B}$
pre $\{n, k\} \subseteq \text{dom } es \cup \text{dom } ss \wedge is\text{-}valid\text{-}subseqstore(ss, es) \wedge is\text{-}substitution(\{n\}, m, ss, es)$
post $r \Leftrightarrow expand\text{-}node(n, m, ss, es) = expand\text{-}node(k, \{\,\}, ss, es)$

$is\text{-}node\text{-}set\text{-}match\ \ (n\colon \text{set of } Node, k\colon \text{set of } Node, m\colon \text{map } Atom$
$\quad \text{to } Expref, ss\colon Subseqstore, es\colon Expstore)\ t\colon \mathbb{B}$
pre $n \cup k \subseteq \text{dom } ss \cup \text{dom } es \wedge is\text{-}valid\text{-}subseqstore(ss, es) \wedge is\text{-}substitution(n, m, ss, es)$
post $t \Leftrightarrow \{expand\text{-}node(l, m, ss, es) \mid l \in n\} = \{expand\text{-}node(r, \{\,\}, ss, es) \mid r \in k\}$

A.5 Problems

$Problem :: hyp : \text{set of } Node$
$\qquad\qquad con : Expref$

$Problemstore = \text{map } Problemref \text{ into } Problem$

$nodes : Problem \rightarrow \text{set of } Node$

$nodes(o) \ \triangleq\ hyp(o) \cup \{con(o)\}$

$is\text{-}valid\text{-}problemstore : Problemstore \times Subseqstore \times Expstore \rightarrow \mathbb{B}$

$is\text{-}valid\text{-}problemstore(ps, ss, es)\ \triangleq\ \forall o \in \text{rng } ps \cdot nodes(o) \subseteq \text{dom } ss \cup \text{dom } es$

$is\text{-}problem\text{-}match\ \ (o\colon Problemref, u\colon Problemref, m\colon \text{map } Atom$
$\quad \text{to } Expref, ps\colon Problemstore, ss\colon Subseqstore, es\colon Expstore)\ r\colon \mathbb{B}$
pre $o, u \in \text{dom } ps \wedge is\text{-}valid\text{-}subseqstore(ss, es) \wedge$
$\quad is\text{-}valid\text{-}problemstore(ps, ss, es) \wedge is\text{-}substitution(nodes(ps(o)), m, ss, es)$
post let $l = ps(o), t = ps(u)$ in
$\quad r \Leftrightarrow$
$\quad is\text{-}exp\text{-}match(con(l), con(t), m, es) \wedge$
$\quad is\text{-}node\text{-}set\text{-}match(hyp(l), hyp(t), m, ss, es)$

A.6 Proofs

$Instantiation :: of : Problemref$
$\qquad\qquad\qquad by : \text{map } Atom \text{ to } Expref$

where

$inv\text{-}Instantiation(mk\text{-}Instantiation(o, m))\ \triangleq\ m \neq \{\,\}$

$Composite\text{-}proof = \text{seq of } Problemref$

$Proof = Instantiation \mid Composite\text{-}proof$

$Proofstore = \text{map } Proofref \text{ to } Proof$

where

$inv\text{-}Proofstore(fs)\ \triangleq\ \forall p, q \in \text{dom } fs \cdot fs(p) = fs(q) \wedge is\text{-}Instantiation(fs(p)) \Rightarrow p = q$

```
$\mathit{new\text{-}known}\ \ (u\colon \mathit{Problemref},\ k\colon \text{set of } \mathit{Node},\ q\colon \mathit{Problemref},$
              $ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore})\ r\colon \text{set of } \mathit{Node}$
pre $\{u, q\} \subseteq \text{dom } ps$
post $r = $ if $\mathit{hyp}(ps(q)) \subseteq k$
        then $\{\mathit{con}(ps(q))\}$
        else if $\exists g \in \text{dom } ss \cdot \mathit{lhs}(ss(g)) \cup \mathit{hyp}(ps(u)) = \mathit{hyp}(ps(q)) \wedge \mathit{rhs}(ss(g)) = \mathit{con}(ps(q))$
            then $\{g\}$
            else $\{\,\}$
```

```
$\mathit{adds\text{-}known}\ \ (u\colon \mathit{Problemref},\ k\colon \text{set of } \mathit{Node},\ q\colon \mathit{Problemref},$
              $ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore})\ r\colon \mathbb{B}$
pre $\{u, q\} \subseteq \text{dom } ps$
post $r \iff \mathit{new\text{-}known}(u, k, q, ps, ss) \neq \{\,\}$
```

```
$\mathit{knowns}\ \ (u\colon \mathit{Problemref},\ n\colon \text{set of } \mathit{Node},\ c\colon \text{seq of } \mathit{Problemref},$
         $ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore})\ r\colon \text{set of } \mathit{Node}$
pre $\{u\} \cup \text{rng } c \subseteq \text{dom } ps$
post $r = $ if $c = [\,]$
        then $n$
        else let $y = \mathit{new\text{-}known}(u, n, \text{hd } c, ps, ss)$ in
            $\mathit{knowns}(u, n \cup y, \text{tl } c, ps, ss)$
```

$\mathit{problems} : \mathit{Proof} \rightarrow \text{set of } \mathit{Problemref}$

```
$\mathit{problems}(v)\ \ \triangle\ \ $ cases $v$ of
                     $\mathit{mk\text{-}Instantiation}(o, m) \rightarrow \{o\}$
                     otherwise $\text{rng } v$
                     end
```

```
$\mathit{is\text{-}valid\text{-}instantiation}\ \ (i\colon \mathit{Instantiation},\ ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore},$
                             $es\colon \mathit{Expstore})\ r\colon \mathbb{B}$
pre $\mathit{is\text{-}valid\text{-}subseqstore}(ss, es) \wedge \mathit{is\text{-}valid\text{-}problemstore}(ps, ss, es)$
post let $\mathit{mk\text{-}Instantiation}(o, m) = i$ in
     let $n = \mathit{nodes}(ps(o))$ in
     $r \iff \mathit{problems}(i) \subseteq \text{dom } ps \wedge \text{dom } m \subseteq \mathit{vars}(n, ss, es) \wedge \mathit{is\text{-}substitution}(n, m, ss, es)$
```

$\mathit{is\text{-}valid\text{-}composite} : \mathit{Composite\text{-}proof} \times \mathit{Problemstore} \rightarrow \mathbb{B}$

$\mathit{is\text{-}valid\text{-}composite}(c, ps)\ \ \triangle\ \ \mathit{problems}(c) \subseteq \text{dom } ps$

```
$\mathit{is\text{-}valid\text{-}proofstore}\ \ (fs\colon \mathit{Proofstore},\ ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore},\ es\colon \mathit{Expstore})\ r\colon \mathbb{B}$
pre $\mathit{is\text{-}valid\text{-}subseqstore}(ss, es) \wedge \mathit{is\text{-}valid\text{-}problemstore}(ps, ss, es)$
post $r \iff \forall v \in \text{rng } fs \cdot (\mathit{is\text{-}Instantiation}(v) \Rightarrow \mathit{is\text{-}valid\text{-}instantiation}(v, ps, ss, es))$
     $\wedge\, (\mathit{is\text{-}Composite\text{-}proof}(v) \Rightarrow \mathit{is\text{-}valid\text{-}composite}(v, ps))$
```

```
$\mathit{new\text{-}fwd\text{-}steps}\ \ (n\colon \text{set of } \mathit{Node},\ c\colon \mathit{Composite\text{-}proof},$
                  $ps\colon \mathit{Problemstore})\ v\colon \mathit{Composite\text{-}proof}$
pre $\text{rng } c \subseteq \text{dom } ps$
post $v = $ if $\exists g \in \text{rng } c \cdot \mathit{hyp}(ps(g)) \subseteq n$
        then $[g] \curvearrowright \mathit{new\text{-}fwd\text{-}steps}(n \cup \mathit{con}(ps(g)), \{g\} \rhd c, ps)$
        else $[\,]$
```

```
new-bwd-steps (n: set of Node, c: Composite-proof,
               ps: Problemstore) v: Composite-proof
pre rng c ⊆ dom ps
post v = if c = []
         then c
         else let y = con(ps(hd c)), z = hyp(ps(hd c)) in
             if y ∈ n
             then new-bwd-steps((n − {y}) ∪ z, tl c, ps) ⌢ hd c
             else new-bwd-steps(n, tl c, ps)
```

A.7 Names

ExpNames = map *String* into *Expref*

where

inv-ExpNames(*en*) $\triangleq$ [] ∉ dom *en*

SubseqNames = map *String* into *Subseqref*

where

inv-SubseqNames(*sn*) $\triangleq$ [] ∉ dom *sn*

ProblemNames = map *String* into *Problemref*

where

inv-ProblemNames(*pn*) $\triangleq$ [] ∉ dom *pn*

ProofNames = map *String* into *Proofref*

where

inv-ProofNames(*fn*) $\triangleq$ [] ∉ dom *fn*

String = seq of *Character*

is-valid-expnames : *ExpNames* × *Expstore* → $\mathbb{B}$

is-valid-expnames(*en*, *es*) $\triangleq$ rng *en* ⊆ dom *es*

is-valid-subseqnames : *SubseqNames* × *Subseqstore* → $\mathbb{B}$

is-valid-subseqnames(*sn*, *ss*) $\triangleq$ rng *sn* ⊆ dom *ss*

is-valid-problemnames : *ProblemNames* × *Problemstore* → $\mathbb{B}$

is-valid-problemnames(*pn*, *ps*) $\triangleq$ rng *pn* ⊆ dom *ps*

is-valid-proofnames : *ProofNames* × *Proofstore* → $\mathbb{B}$

is-valid-proofnames(*fn*, *fs*) $\triangleq$ rng *fn* ⊆ dom *fs*

A.8 Solved and Unsolved Problems and Rules of Inference

$\mathit{Proofmap} = \mathsf{map}\ \mathit{Problemref}\ \mathsf{to\ set\ of}\ \mathit{Proofref}$

$\mathit{Incomplete\text{-}proofmap} = \mathsf{map}\ \mathit{Problemref}\ \mathsf{to\ set\ of}\ \mathit{Proofref}$

where

$\mathit{inv\text{-}Incomplete\text{-}proofmap}(im) \quad \triangleq \quad \{\,\} \notin \mathsf{rng}\, im \land \forall k, m \in \mathsf{dom}\, im \cdot im(k) \cap im(m) \neq \{\,\} \;\Rightarrow\; k = m$

$\mathit{Rulemap} = \mathsf{map}\ \mathit{String}\ \mathsf{into}\ \mathit{Problemref}$

where

$\mathit{inv\text{-}Rulemap}(rm) \quad \triangleq \quad [\,] \notin \mathsf{dom}\, rm$

$\mathit{solved\text{-}problems} : \mathit{Proofmap} \to \mathsf{set\ of}\ \mathit{Problemref}$

$\mathit{solved\text{-}problems}(jm) \quad \triangleq \quad \mathsf{dom}\, jm$

$\mathit{rules} : \mathit{Rulemap} \to \mathsf{set\ of}\ \mathit{Problemref}$

$\mathit{rules}(rm) \quad \triangleq \quad \mathsf{rng}\, rm$

$\mathit{axioms} : \mathit{Proofmap} \to \mathsf{set\ of}\ \mathit{Problemref}$

$\mathit{axioms}(jm) \quad \triangleq \quad \{u \mid u \in \mathit{solved\text{-}problems}(jm) \land jm(u) = \{\,\}\}$

$\mathit{complete\text{-}proofs} : \mathit{Proofmap} \to \mathsf{set\ of}\ \mathit{Proofref}$

$\mathit{complete\text{-}proofs}(jm) \quad \triangleq \quad \bigcup \mathsf{rng}\, jm$

$\mathit{is\text{-}valid\text{-}rulemap} : \mathit{Rulemap} \times \mathit{Proofmap} \times \mathit{Proofstore} \to \mathbb{B}$

$\mathit{is\text{-}valid\text{-}rulemap}(rm, jm, fs) \quad \triangleq \quad \mathit{axioms}(jm) \subseteq \mathit{rules}(rm) \land \mathit{rules}(rm) \subseteq \mathit{solved\text{-}problems}(jm) \land$
$\quad \forall p \in \mathit{complete\text{-}proofs}(jm) \cdot p \in \mathsf{dom}\, fs \;\Rightarrow\; (\mathit{is\text{-}Instantiation}(fs(p)) \;\Rightarrow\; \mathit{of}(fs(p)) \in \mathit{rules}(rm))$

```
derivable-results (jm: Proofmap, fs: Proofstore,
                   w: set of Problemref) r: set of Problemref
pre $\mathit{complete\text{-}proofs}(jm) \subseteq \mathsf{dom}\, fs \land w \subseteq \mathit{solved\text{-}problems}(jm)$
post let $l = \{b \mid b \in \mathit{axioms}(jm) \lor b \in \mathit{solved\text{-}problems}(jm) \land$
          $\exists v \in jm(b) \cdot \mathit{problems}(fs(v)) \subseteq w)\} \cup w$ in
     $r =$ if $l = w$
          then $w$
          else $\mathit{derivable\text{-}results}(jm, fs, l)$
```

```
is-self-consistent (jm: Proofmap, fs: Proofstore) r: $\mathbb{B}$
pre $\mathit{complete\text{-}proofs}(jm) \subseteq \mathsf{dom}\, fs$
post $r \Leftrightarrow \mathit{solved\text{-}problems}(jm) = \mathit{derivable\text{-}results}(jm, fs, \mathit{axioms}(jm))$
```

```
$\mathit{is\text{-}complete\text{-}proof}\ \ (v\colon \mathit{Proof},\ u\colon \mathit{Problemref},\ ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore},$
                        $es\colon \mathit{Expstore})\ r\colon \mathbb{B}$
pre $u \in \mathrm{dom}\, ps \wedge \mathit{is\text{-}valid\text{-}subseqstore}(ss, es) \wedge \mathit{is\text{-}valid\text{-}problemstore}(ps, ss, es) \wedge$
    $(\mathit{is\text{-}Instantiation}(v) \ \Rightarrow\ \mathit{is\text{-}valid\text{-}instantiation}(v, ps, ss, es)) \wedge$
    $(\mathit{is\text{-}Composite\text{-}proof}(v) \ \Rightarrow\ \mathit{is\text{-}valid\text{-}composite}(v, ps))$
post let $t$ = cases $v$ of
            $\mathit{mk\text{-}Instantiation}(o, m) \rightarrow \mathit{is\text{-}problem\text{-}match}(o, u, m, ps, ss, es)$
            otherwise $\mathit{con}(ps(u)) \in \mathit{knowns}(u, \mathit{hyp}(ps(u)), v, ps, ss)$
            end
       in
    $r \ \Leftrightarrow\ t$
```

```
$\mathit{is\text{-}valid\text{-}proofmap}\ \ (jm\colon \mathit{Proofmap},\ fs\colon \mathit{Proofstore},\ ps\colon \mathit{Problemstore},$
                      $ss\colon \mathit{Subseqstore},\ es\colon \mathit{Expstore})\ r\colon \mathbb{B}$
pre $\mathit{is\text{-}valid\text{-}subseqstore}(ss, es) \wedge \mathit{is\text{-}valid\text{-}problemstore}(ps, ss, es) \wedge$
    $\mathit{is\text{-}valid\text{-}proofstore}(fs, ps, ss, es)$
post $r \ \Leftrightarrow\ \mathit{solved\text{-}problems}(jm) \subseteq \mathrm{dom}\, ps \wedge \mathit{complete\text{-}proofs}(jm) \subseteq \mathrm{dom}\, fs \wedge$
       $\mathit{is\text{-}self\text{-}consistent}(jm, fs) \wedge$
       $\forall u \in \mathit{solved\text{-}problems}(jm) \cdot \forall v \in jm(u) \cdot$
           $\mathit{problems}(fs(v)) \subseteq \mathrm{dom}\, jm \wedge \mathit{is\text{-}complete\text{-}proof}(fs(v), u, ps, ss, es)$
       $\wedge\ \forall k, m \in \mathrm{dom}\, jm \cdot (\exists v \in jm(k) \cap jm(m) \cdot \mathit{is\text{-}composite\text{-}proof}(fs(v))) \ \Rightarrow\ k = m$
```

$\mathit{incomplete\text{-}proofs} : \mathit{Incomplete\text{-}proofmap} \rightarrow \text{set of } \mathit{Proofref}$

$\mathit{incomplete\text{-}proofs}(im) \ \ \underline{\triangle}\ \ \bigcup \mathrm{rng}\, im$

```
$\mathit{is\text{-}valid\text{-}incomplete\text{-}proofmap}\ \ (im\colon \mathit{Incomplete\text{-}proofmap},\ jm\colon \mathit{Proofmap},\ fs\colon \mathit{Proofstore},$
                                    $ps\colon \mathit{Problemstore},\ ss\colon \mathit{Subseqstore},\ es\colon \mathit{Expstore})\ r\colon \mathbb{B}$
pre $\mathit{is\text{-}valid\text{-}subseqstore}(ss, es) \wedge \mathit{is\text{-}valid\text{-}problemstore}(ps, ss, es) \wedge$
    $\mathit{is\text{-}valid\text{-}proofstore}(fs, ps, ss, es) \wedge \mathit{is\text{-}valid\text{-}proofmap}(jm, fs, ps, ss, es)$
post $r \ \Leftrightarrow\ \mathrm{dom}\, im \subseteq \mathrm{dom}\, ps \wedge \mathit{axioms}(jm) \cap \mathrm{dom}\, im = \{\,\} \wedge$
       $\mathit{complete\text{-}proofs}(jm) \cup \mathit{incomplete\text{-}proofs}(im) = \mathrm{dom}\, fs \wedge$
       $\mathit{complete\text{-}proofs}(jm) \cap \mathit{incomplete\text{-}proofs}(im) = \{\,\} \wedge$
       $\forall u \in \mathrm{dom}\, im \cdot \forall v \in im(u) \cdot \mathit{problems}(fs(v)) \subseteq \mathit{solved\text{-}problems}(jm) \wedge$
       $\mathit{is\text{-}Composite\text{-}proof}(fs(v)) \wedge \neg(\mathit{is\text{-}complete\text{-}proof}(fs(v), u, ps, ss, es))$
```

$\mathit{Indexmap} = \text{map } \mathit{Proofref} \text{ to } \mathbb{N}$

```
$\mathit{forward\text{-}proof}\ \ (h\colon \mathit{Proofref},\ fs\colon \mathit{Proofstore},\ xm\colon \mathit{Indexmap})\ v\colon \mathit{Composite\text{-}proof}$
pre $h \in \mathrm{dom}\, fs \cap \mathrm{dom}\, xm \wedge \mathit{is\text{-}Composite\text{-}proof}(fs(h)) \wedge 0 \le xm(h) \le \mathrm{len}\, fs(h)$
post $v = \{n \in \mathbb{N} \mid 1 \le n \le xm(h)\} \lhd fs(h)$
```

```
$\mathit{backward\text{-}proof}\ \ (h\colon \mathit{Proofref},\ fs\colon \mathit{Proofstore},\ xm\colon \mathit{Indexmap})\ v\colon \mathit{Composite\text{-}proof}$
pre $h \in \mathrm{dom}\, fs \cap \mathrm{dom}\, xm \wedge \mathit{is\text{-}Composite\text{-}proof}(fs(h)) \wedge 0 \le xm(h) \le \mathrm{len}\, fs(h)$
post $fs(h) = \mathit{forward\text{-}proof}(h, fs, xm) \curvearrowright v$
```

```
goals (n: set of Node, c: seq of Problemref, ps: Problemstore) r: set of Node
pre rng $c \subseteq$ dom $ps$
post $r =$ if $c = []$
          then $n$
          else let $k = hyp(ps(\text{hd } c)), y = con(ps(\text{hd } c))$ in
              if $y \in n$
              then $goals((n - \{y\}) \cup k, \text{tl } c, ps)$
              else $goals(n, \text{tl } c, ps)$
```

```
is-valid-indexmap (xm: Indexmap, im: Incomplete-proofmap, jm: Proofmap, fs: Proofstore,
                   ps: Problemstore, ss: Subseqstore, es: Expstore) r: B
pre is-valid-subseqstore(ss, es) $\wedge$ is-valid-problemstore(ps, ss, es) $\wedge$
    is-valid-proofstore(fs, ps, ss, es) $\wedge$ is-valid-proofmap(jm, fs, ps, ss, es) $\wedge$
    is-valid-incomplete-proofmap(im, jm, fs, ps, ss, es)
post dom $xm$ = incomplete-proofs(im) $\wedge \forall u \in$ dom $im \cdot \forall v \in im(u) \cdot$
        let $fp$ = forward-proof(v, fs, xm),
            $bp$ = backward-proof(v, fs, xm),
            $gp$ = reverse(bp) in
        $0 \le xm(v) \le$ len $fs(v) \wedge$
        $\nexists z \in$ rng $bp \cdot hyp(ps(z)) \subseteq knowns(u, hyp(ps(u)), fp, ps, ss) \wedge$
        $\forall g \in$ dom $gp \cdot con(ps(gp(g))) \in goals(\{con(ps(u))\}, \{g, \ldots, \text{len } gp\} \lhd gp, ps) \wedge$
        $\forall b \in$ dom $fp \cdot$
            adds-known(u, knowns(u, hyp(ps(u)), $\{b, \ldots, \text{len } fp\} \lhd fp$, ps, ss), fp(b), ps, ss)
```

B The Proof of *and-or-dist*

This appendix gives the full proof structure arising from the proof of $\{E1 \wedge (E2 \vee E3)\} \vdash E1 \wedge E2 \vee E1 \wedge E3$ as dealt with in Section 3.3. For clarity, objects are written here in concrete (dereferenced) form, though it should be borne in mind that all are stored internally as references to objects.

Following the session described in Section 3.3, the complete proof of $\{E1 \wedge (E2 \vee E3)\} \vdash E1 \wedge E2 \vee E1 \wedge E3$ will be a composite proof consisting of five elements, $[p1, p2, p3, p4, p5]$, where

$p1 \equiv \{E1 \wedge (E2 \vee E3)\} \vdash E1$
$p2 \equiv \{E1 \wedge (E2 \vee E3)\} \vdash E2 \vee E3$
$p3 \equiv \{E1 \wedge (E2 \vee E3), E2\} \vdash E1 \wedge E2 \vee E1 \wedge E3$
$p4 \equiv \{E1 \wedge (E2 \vee E3), E3\} \vdash E1 \wedge E2 \vee E1 \wedge E3$
$p5 \equiv \{E2 \vee E3, \{E2\} \leadsto E1 \wedge E2 \vee E1 \wedge E3, \{E3\} \leadsto E1 \wedge E2 \vee E1 \wedge E3\} \vdash E1 \wedge E2 \vee E1 \wedge E3.$

Each of $p1$ to $p5$ is a solved problem, and therefore itself has a proof. The problems $p1$, $p2$ and $p5$ are simply instances of rules. Their proofs, $f1$, $f2$ and $f5$ are thus instantiations:

$f1 \equiv$ *Instantiation* $\underline{of}$ $\{X \wedge Y\} \vdash X$ $\underline{by}$ $\{X \mapsto E1, Y \mapsto E2 \vee E3\}$
$f2 \equiv$ *Instantiation* $\underline{of}$ $\{X \wedge Y\} \vdash Y$ $\underline{by}$ $\{X \mapsto E1, Y \mapsto E2 \vee E3\}$
$f5 \equiv$ *Instantiation* $\underline{of}$ $\{X \vee Y, \{X\} \leadsto Z, \{Y\} \leadsto Z\} \vdash Z$ $\underline{by}$ $\{X \mapsto E2, Y \mapsto E3, Z \mapsto E1 \wedge E2 \vee E1 \wedge E3\}$

The other two elements, $p3$ and $p4$, themselves have composite proofs $f3 \equiv [p1, p6, p7]$ and $f4 \equiv [p1, p8, p9]$, where

$p6 \equiv \{E1, E2\} \vdash E1 \wedge E2$
$p7 \equiv \{E1 \wedge E2\} \vdash E1 \wedge E2 \vee E1 \wedge E3$
$p8 \equiv \{E1, E3\} \vdash E1 \wedge E3$
$p7 \equiv \{E1 \wedge E3\} \vdash E1 \wedge E2 \vee E1 \wedge E3$

Again, each of $p6$ to $p9$ is a solved problem and has a proof. Their proofs are simply the instantiations $f6$ to $f9$:

$f6 \equiv$ *Instantiation* $\underline{of}$ $\{X, Y\} \vdash X \wedge Y$ $\underline{by}$ $\{X \mapsto E1, Y \mapsto E2\}$
$f7 \equiv$ *Instantiation* $\underline{of}$ $\{X\} \vdash X \vee Y$ $\underline{by}$ $\{X \mapsto E1 \wedge E2, Y \mapsto E1 \wedge E3\}$
$f8 \equiv$ *Instantiation* $\underline{of}$ $\{X, Y\} \vdash X \wedge Y$ $\underline{by}$ $\{X \mapsto E1, Y \mapsto E3\}$
$f9 \equiv$ *Instantiation* $\underline{of}$ $\{Y\} \vdash X \vee Y$ $\underline{by}$ $\{X \mapsto E1 \wedge E2, Y \mapsto E1 \wedge E3\}$

References

[1] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[2] I.D. Cottam, C.B. Jones, T. Nipkow, and A.C. Wills. Mule: a support system for formal specification and rigorous software development. March 1983. BCS-FACS/SERC Conference on Program Specification and Verification, University of York, Proceedings not published.

[3] I.D. Cottam, C.B. Jones, T. Nipkow, A.C. Wills, M.I. Wolczko, and A. Yaghi. Project support environments for formal methods. In J. McDermid, editor, *Integrated Project Support Environments*, chapter 3, Peter Peregrinus Ltd., 1985.

[4] I.D. Cottam, C.B. Jones, T.N. Nipkow, A.C. Wills, M. Wolczko, and A. Yaghi. *Mule — An Environment for Rigorous Software Development (Final Report to SERC on Grant Number GR/C/05762).* Technical Report, Department of Computer Science, University of Manchester, 1986.

[5] B.T. Denvir, V.A. Downes, C.B. Jones, R.A. Snowdon, and M.K. Tordoff. *IPSE 2.5 Project Proposal.* Technical Report, ICL/STC-IDEC/STL/University of Manchester, 7th February 1985.

[6] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, Englewood Cliffs, NJ, 1986. 300 pages.

[7] C.B. Jones, P. Lindsay, and C. Wadsworth. *IPSE 2.5 Theorem Proving Concepts Paper 060/00021/1.5.* Technical Report, Manchester University and Rutherford Appleton Laboratory, 18th June 1986.

[8] Kevin D. Jones. *The Muffin Prototype: Experiences with Smalltalk-80.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/00065.

[9] Kevin D. Jones. *The Preliminary Specification of FRIPSE: FRIPSE$_{pc}$.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/00114.

[10] P. A. Lindsay. *FSIP2: Son of FSIP.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/pal014.

[11] Peter A. Lindsay, Richard Moore, and Brian Ritchie. *Review of Existing Theorem Provers.* Dept. of Computer Science, University of Manchester. Technical Report UMCS-87-8-2.

[12] L.S. Marshall. *A Formal Description Method for User Interfaces.* PhD thesis, University of Manchester, October 1986.

[13] Richard Moore. *Muffin - The Design of a User Interface to a Formal Reasoning System.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/rm002.

[14] Richard Moore. *The Muffin Database.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/00060/1.3.

[15] Richard Moore. *The Muffin Prototype.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/00066.

[16] Richard Moore. *Towards a Generic Muffin.* Dept. of Computer Science, University of Manchester. IPSE Project Document 060/00140.

[17] T. Nipkow. Mule: persistence and types in an IPSE. In *Persistance and Data Types*, pages 1–25, 1986.

[18] R.A. Snowdon. *IPSE 2.5 Technical Overview.* STL NW. IPSE Project Document 060/00055/1.2.

[19] R.A. Snowdon. *Scope of the IPSE 2.5 Project.* STL NW. IPSE Project Document 060/00002/4.1.

[20] D. Talbot and R.W. Witty. Alvey programme for software engineering. November 1983. Published by the Alvey Directorate.