Animating Formal Proof at the Surface: The Jape Proof Calculator

RICHARD BORNAT[1] AND BERNARD SUFRIN[2]

[1]*Department of Computer Science, Queen Mary and Westfield College, University of London, UK*
[2]*Programming Research Group, University of Oxford, 8-11 Keble Road, Oxford, UK*
Email: richard@dcs.qmw.ac.uk and sufrin@comlab.ox.ac.uk

Jape is a program which supports the step-by-step interactive development of proofs in formal logics, in the style of proofs-on-paper. It is uncommitted to any particular logic and is customized by a description of a collection of inference rules and the syntax of judgements. It works purely at the surface syntactic level, as a person working on paper might. In that spirit it makes use of explicit visible provisos rather than a conventional encoding of logical properties. Its principal mechanism is unification, employed as a search tool to defer decisions about how to proceed at difficult points in a proof. The design aim is to produce a tool which makes step-by-step proof calculations so straightforward that novices can learn by exploring the use of a pre-encoded logic. Examples of proof development are given in several small logics.

Received November 18, 1994; revised April 28, 1999

1. INTRODUCTION

Software engineers are often required to construct formal specifications of the programs they construct, but rarely expected to prove a formal relationship between specification and program. Partly this is because formal proof is intrinsically difficult, but partly because software engineers—and others—find it more difficult than they should. They lack practice, they lack effective support tools and they are not expert logicians. Even experienced program-provers will typically be logically skilful only within some favourite logic.

If this situation is to be remedied—and we take it for granted that it should be—then it will be necessary to build proof support tools which practitioners find helpful. To design such a tool we must consider every part of the activity: the way that a tool is customized to fit its users' preferred logic; the way it presents information to its users; the way that its users can control its operation.

In this paper we report on the development, over the last six years, of an interactive proof support tool with a high-quality graphical interface. We have attempted in our design to allow our users a light touch: the encoder does not have to specify more than is necessary and the prover does not see more than they might wish to. We have attempted to keep it transparently literal in its working: that is, we try to make the screen display look just like a proof that might be printed in a book, from the symbols used down to the literal presentation of provisos as visible constraints on progress. To do so we have had to do much more than simply bolt an interface onto a theorem prover: it has been necessary to construct a system in which every feature is used to support interaction and to expend ingenuity and computer power to preserve an illusion of simplicity.

This paper describes the way that we have used Jape to encode logics and construct proofs. It touches only briefly on the internal design and construction of the program itself.

2. AN INTERACTIVE, UNCOMMITTED, LITERAL, LIGHTWEIGHT CALCULATOR

We call Jape a *proof calculator* to make an analogy with a numerical calculator—a device which acts passively under the user's control—and to distinguish it from a *theorem prover* such as HOL [1], Isabelle [2], 2OBJ [3], Boyer–Moore [4], LWB [5] or PVS [6]—a device which acts under program control to search for the solution of a problem. Systems designed with similar aims to our own include Mural [7], EUODHILOS [8] and MacLogic [9]. Like a numerical calculator, Jape acts with perfect precision when it can act at all, but offers no help or advice. In that respect it is distinguished from many educational efforts such as MacLogic and the CMU proof tutor [10].

In designing and constructing the program, we were greatly influenced by our experience as novice users of several other formal proof tools. We had some educational experience of MacLogic [9]: that program inspired us a great deal in the early stages of our design, especially because we felt that it would be improved if its user interface were made more declarative and less history-based. We had some experience with the B-tool [11] which we found very difficult to use: at the time its user interface was restrictive and intricate, and proof-search decisions had to be made much earlier than we wished. We were influenced by reading about Isabelle [2], ICLE [12] and Mural [7].

We envisaged a proof assistant with a declarative, non-historical interface [13]—that is, one which displays at each stage the state of a calculation rather than a sequence of commands which led to that state—and which used the same obedient style of interaction as a calculator or word-processor. We wanted it to be configured by a description of a logic and we wanted that description to be derived transparently from the kind of definition which you might find in a textbook. We wanted the displayed proofs to look like those which might be found in the same textbook and the intermediate stages of proof to be the sort of thing that a prover might write on paper or on a blackboard whilst developing a proof. We are conscious that these are large aims and that we shall always fall short of them; nevertheless they were valuable in setting the direction of our endeavours.

We wanted our program to be lightweight enough to be used on machinery available to undergraduates, so that it could be used in practice for teaching logic. The program has grown, but advances in computer power have been spectacular over the same period and Jape is still relatively lightweight when compared to much of the other application software that our users employ. We also intended our program to be useful to experts who were developing small logical systems for specification and proof in particular problem areas. In both respects we have had some success: the program is being used for teaching purposes on courses at QMW, Oxford, Glasgow and Eindhoven. It has been used at Oxford [14] and elsewhere to work on serious proofs in application-oriented logics.

Our approach is in contrast to the tradition of user interface work in theorem proving, best typified by [15], in which a user interface is bolted on to an otherwise unmodified theorem prover. That tradition starts with a proof engine and fits the user interface to it: we feel that the user interface should be at the centre of the design, supporting only as much proof machinery as it will bear. The proof display in Jape is constantly available and not, as in [16], extracted from a proof term once the proof is complete. Our treatment of user gestures differs from that of [17] in that the interpretation of gestures is relative to a proof context and any other gestures which might have been made, as well as being programmable by the user.

We have applied Jape in earnest to natural deduction, to several variations of the sequent calculus, to reasoning about functional programs, to proof of hardware circuits, to protocol-authentication logic, temporal logic and to the Hindley–Milner type-assignment algorithm. We have also experimented in several other areas including Hoare logic, group theory, category theory, operational semantics and the π-calculus.

3. HOW IT WORKS

In operation Jape is two programs: one a proof engine, building a Gentzen tree of sequents using inference rules; the other a graphical user interface, communicating with the engine via a UNIX pipe or MacOS AppleEvents. The proof engine is about 24,000 lines of Standard ML and when compiled by SML-NJ is about 1.5 Mbytes of code. There are currently three implementations of the user interface: one for X written in Tcl/Tk, one in Java and one for MacOS written in C.

An important feature of Jape is that it has no type-theoretic meta-logic of proof in the sense of the meta-logic of Isabelle or ELF. It is uncommitted to any particular model of what a logic should be, other than that the logic must be expressed as inference rules in a two-sided sequent calculus and that the problem statements must be in some recursively-definable syntax. We claim that as a result it is easy to transcribe logics into the Jape notation, though the burden of interpretation makes the proof mechanism less efficient. Only when manipulating provisos and substitution forms does Jape use any built-in rules—all of them to do with binding and substitution forms—and make simplifications on its own initiative.

Jape proceeds by matching rules to problem statements, working all the time in the user's logic, always at the surface level. It acts as a slow Prolog interpreter might: the consequent of an inference rule is treated as if it were the left-hand side of a Prolog clause and its antecedents as if they were the right-hand sides; once the rule matches the antecedents are added to the proof tree. A step fails if the rule does not match or if any of the provisos applicable to the proof, including any added by the rule itself, is violated. This makes Jape a backward-reasoning engine, but we show below how it can be harnessed to give an illusion of forward proof.

Jape uses unification as its rule-matching mechanism, in order to enable the user to defer decisions about the identity of formulae in a proof. When a rule does not have the subformula property—that is, when an antecedent contains a formula which is not derived from anything in the consequent—Jape will invent an unknown as a placeholder. When a partial proof contains unknowns, Jape can defer the checking of provisos and the simplification of formulae. It augments its basic unification algorithm with mechanisms designed to support α-conversion, substitution forms and both list and bag unification at the level of sequents (associative and/or commutative matching).

Although the basic internal data structure of Jape is a tree of sequents, in many cases it is inconvenient or confusing to show all of the tree or to show it as a tree. Jape has mechanisms to filter and prune a tree before it is displayed. It is capable of displaying a proof in a box-and-line style and uses special mechanisms to hide parts of that kind of display.

4. THE TACTIC LANGUAGE

Following LCF [18], every proof editor and theorem prover has a programming notation for directing the course of a proof. Although Jape has mechanisms which support search for proofs, its tactic language is used as much to control the display and to interpret user gestures as to control the course of proof. It would be inappropriate to explain the details of the language in this paper, but we give below some examples of how it can be used. A fuller description is provided in the manual available from the Jape web sites [19].

The primitive operations of the tactic language are the application of rules, application of previously proved theorems. If a Jape tactic succeeds, it is because a successful sequence of primitive steps has been found and therefore the validity of steps made by the application of tactics is implicitly guaranteed. Communication with external decision procedures is also supported.

5. EXAMPLES

The main feature of Jape is that it shows a proof, or partial proof, at every stage of development. All interaction is with that proof: clicking or double-clicking on formulae within the proof; choosing actions from menus to apply to the selected components of the proof. The aim is to assist reflection by the user: for the expert at important decision points; for the novice throughout the process. The notation of the displayed logic, the content and behaviour of menus and on-screen panels are all under the logic encoder's control, as is the interpretation of a user's mouse gestures. Some of the techniques used are illustrated below.

5.1. Reasoning in the single-conclusion sequent calculus

Our first example is an encoding of the rules of the intuitionistic fragment of the sequent calculus shown in Figure 1a. The syntax of the encoding is given in Figure 1b, and the encoded rules in Figure 1c.[1] Formulae are written in a language which includes juxtaposition ($F1\ F2$), substitution ($F[x1, \ldots, xn \backslash F1, \ldots, Fn]$), tupling ($(F1, \ldots, Fn)$), prefix, postfix and infix operators ($\bullet F$, $F1 \bullet F2$ and $F\bullet$) and user-defined bracketed constructs (for example, **if** $F1$ **then** $F2$ **else** $F3$). The logic encoder must define the range of names that can be used in theorems and rules, the binary and unary operators with their binding powers, any additional bracketed constructs which are needed and the syntax of a sequent.

The first part of Figure 1b describes bracketing forms for the binding operators and gives them a position in a hierarchy of operations—the numbers are binding powers and are set lower than those that follow, so that quantification in this logic implicitly brackets all other kinds of operator. The second part gives the priority of the binary connectives of the logic and their associativity (only $\rightarrow$ is right-associative); the third gives the relative priorities of juxtaposition and substitution forms. The various declarations of names allow a relatively quiet description of rules and theorems: for example, every rule is schematic in names prefixed by $A, B, C, \ldots, u, v, \ldots, \Gamma$. In the fourth part, binding structures are defined by pattern. The syntax of a sequent is described: it is a comma-separated bag (multiset) of hypotheses, followed by a turnstile and a single conclusion; the turnstile can be omitted if the hypothesis list is empty. Finally, Jape is instructed to interpret predicate notation (e.g. $P(m)$) as shorthand for substitution forms.

The rules in Figure 1c are named (e.g. *hyp*, $\vdash\wedge, \ldots$); they may have parameters (e.g. A, OBJECT m) and provisos (WHERE ...); the antecedents, if any, are introduced by FROM and separated by AND; the consequent is introduced by INFER. Because Jape cannot, at present, accommodate definitional equivalence, $\neg P$ is converted into $P \rightarrow \perp$ with two explicit rules ($\neg\vdash$ and $\vdash\neg$ in Figure 1). Otherwise we have a straightforward transcription of a conventional collection of rules.

Two of the rules in Figure 1c—$\exists\vdash$ and $\vdash\forall$—include a proviso FRESH m. In each case this means that the name m must not appear free in the problem sequent which matches the consequent of the rule. If a proviso is satisfied then it disappears from the record; if it is violated then the proof step cannot be made; if it cannot be decided immediately then it is visibly carried forward.

The same rules each have a parameter OBJECT m: this directs Jape that, when instantiating the rule, it should use a newly-minted identifier in place of m rather than an unknown. The effect is that a new identifier is automatically introduced into the proof whenever the rule is used without an argument: this is usually enough to satisfy the proviso.

The quantifier rules of Figure 1a are expressed using predicate notation and this is reflected in Figure 1c. Because of the setting of the *interpretpredicates* parameter in Figure 1b, those rules will be converted to use substitution forms. For example the $\forall\vdash$ rule is read as 'FROM Γ, P[x\B] $\vdash$ C INFER Γ, $\forall$ x.P $\vdash$ C'.

The structural roles of three of the rules—hyp, cut and weaken—are described to Jape following the declaration of the rules. Uses of IDENTITY and CUT rules are hidden in the box display of a proof, as illustrated later. The presence of a WEAKEN rule allows the application of a theorem with fewer hypotheses than the problem sequent (because we could have used the WEAKEN rule to eliminate extra hypotheses from the problem). The presence of a CUT rule allows the application of a theorem whose hypotheses do not match those in the problem sequent (we can present them as additional problems, because we could have done so by a number of uses of the CUT rule).

The rules of this particular logic are *additive* in the sense of linear logic—that is, unmatched hypotheses are copied from the consequent to every antecedent—and the closing hyp rule implicitly includes weakening by ignoring unmatched hypotheses. This is a choice of the logic encoder: a more austere presentation using multiplicative rules and no unmatched hypotheses in the hyp rule is possible in Jape (see the discussion of multiple-conclusion calculi below) but, as is well known, such austerity makes a proof search more difficult.

The AUTOMATCH directive in Figure 1c causes Jape to try the hyp rule on every open problem sequent at the end of each proof step. The rule will only be applied 'by matching'—that is, when to do so does not change any unknowns in the proof.

We organize the rules into a menu for presentation to the user, as shown in Figure 1d. In this example there is one entry per rule. Conjectures can be presented in a menu or in a floating panel which includes a scrolling list and buttons to manipulate entries in the list. Figure 1e shows some of

[1]In this and subsequent illustrations, words written in upper-case are built-in elements of the logic description notation.

$$\frac{}{\Gamma, A \vdash A}\ hyp$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \vdash\wedge \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \vdash\rightarrow \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vdash\vee_L \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vdash\vee_R$$

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash B \rightarrow A}{\Gamma \vdash A \equiv B}\ \vdash\equiv \qquad \frac{\Gamma \vdash A \rightarrow \bot}{\Gamma \vdash \neg A}\ \vdash\neg \qquad \frac{\Gamma \vdash P(m)}{\Gamma \vdash \forall x.P(x)}\ (\text{FRESH } m)\ \vdash\forall \qquad \frac{\Gamma \vdash P(B)}{\Gamma \vdash \exists x.P(x)}\ \vdash\exists$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\ \wedge\vdash \qquad \frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C}\ \rightarrow\vdash \qquad \frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ \vee\vdash \qquad \frac{\Gamma, A \rightarrow B, B \rightarrow A \vdash C}{\Gamma, A \equiv B \vdash C}\ \equiv\vdash$$

$$\frac{\Gamma, A \rightarrow \bot \vdash B}{\Gamma, \neg A \vdash B}\ \neg\vdash \qquad \frac{}{\Gamma, \bot \vdash A}\ \bot\vdash \qquad \frac{\Gamma, P(B) \vdash C}{\Gamma, \forall x.P(x) \vdash C}\ \forall\vdash \qquad \frac{\Gamma, P(m) \vdash C}{\Gamma, \exists x.P(x) \vdash C}\ (\text{FRESH } m)\ \exists\vdash$$

$$\frac{\Gamma \vdash B \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ cut \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ weaken \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ contract$$

FIGURE 1a. Rules of a single-conclusion sequent calculus.

```
LEFTFIX      20     ∀.
LEFTFIX      20     ∃.

INFIX        100L   ≡
INFIX        110R   →
INFIX        150L   ∧
INFIX        160L   ∨
PREFIX       200    ¬
JUXTFIX      300
SUBSTFIX     400

CLASS BAG Γ
CLASS FORMULA A B C D P Q R S
CLASS VARIABLE u v w x y z m n
CONSTANT ⊥

BIND x SCOPE P IN ∃x . P
BIND x SCOPE P IN ∀x . P

SEQUENT IS BAG ⊢ FORMULA

INITIALISE interpretpredicates true
```

FIGURE 1b. Syntax of formulae and sequents.

```
RULE          hyp(A)                                    INFER Γ,A ⊢ A
RULE          "⊢∧"         FROM Γ ⊢ A AND Γ ⊢ B         INFER Γ ⊢ A∧B
RULE          "∧⊢"         FROM Γ, A, B ⊢ C             INFER Γ, A∧B ⊢ C
RULE          "⊢∨(L)"      FROM Γ ⊢ A                   INFER Γ ⊢ A∨B
RULE          "⊢∨(R)"      FROM Γ ⊢ B                   INFER Γ ⊢ A∨B
RULE          "∨⊢"         FROM Γ, A ⊢ C AND Γ, B ⊢ C   INFER Γ, A∨B ⊢ C
RULE          "⊢¬"         FROM Γ ⊢ A→ ⊥                INFER Γ ⊢ ¬A
RULE          "¬⊢"         FROM Γ, A→ ⊥ ⊢ B             INFER Γ, ¬A ⊢ B
RULE          "⊢→"         FROM Γ, A ⊢ B                INFER Γ ⊢ A→B
RULE          "→⊢"         FROM Γ ⊢ A AND Γ, B ⊢ C      INFER Γ, A→B ⊢ C
RULE          "⊢≡"         FROM Γ ⊢ A→B AND Γ ⊢ B→A     INFER Γ ⊢ A≡B
RULE          "≡⊢"         FROM Γ, A→B, B→A ⊢ C         INFER Γ, A≡B ⊢ C
RULE          "⊥⊢"         INFER Γ, ⊥ ⊢ A
RULE          "⊢∀"(OBJECT m) WHERE FRESH m
                           FROM Γ ⊢ P(m)                INFER Γ ⊢ ∀x.P(x)
RULE          "∀⊢"(B)      FROM Γ, P(B) ⊢ C             INFER Γ, ∀x.P(x) ⊢ C
RULE          "⊢∃"(B)      FROM Γ ⊢ P(B)                INFER Γ ⊢ ∃x.P(x)
RULE          "∃⊢"(OBJECT m) WHERE FRESH m
                           FROM Γ, P(m) ⊢ C             INFER Γ, ∃x.P(x) ⊢ C
RULE          cut(A)       FROM Γ ⊢ A AND Γ, A ⊢ C      INFER Γ ⊢ C
RULE          weaken(A)    FROM Γ ⊢ B                   INFER Γ, A ⊢ B
RULE          contract(A)  FROM Γ, A, A ⊢ B             INFER Γ, A ⊢ B

IDENTITY            hyp
CUT                 cut
WEAKEN              weaken

AUTOMATCH           hyp
```

FIGURE 1c. Encoded rules.

```
MENU Rules IS
          ENTRY hyp
          ENTRY cut
          SEPARATOR
          ENTRY "∧⊢"
          ENTRY "∨⊢"
          ENTRY "→⊢"
          ...
          SEPARATOR
          ENTRY "⊢∧"
          ENTRY "⊢∨(L)"
          ENTRY "⊢∨(R)"
          ENTRY "⊢→"
          ...
END
```

FIGURE 1d. Menu of rules.

```
CONJECTUREPANEL "Conjectures"
 THEOREM INFER P→(Q→R) ⊢ (P→Q)→(P→R)
 THEOREM INFER P→(Q→R), Q ⊢ P→R
 THEOREM INFER R→S ⊢ (P→R) → (P→S)
 THEOREM INFER P→(P→Q) ⊢ P→Q
 THEOREM INFER P ⊢ Q→(P ∧ Q)
 THEOREM INFER P∧(Q∧R) ⊢ (P∧Q)∧R
 ...
 THEOREM INFER ∀x.¬Q(x), P→(∀x.Q(x)) ⊢ ¬P
 THEOREM WHERE x NOTIN P INFER P∨¬P, ∀x.P→Q(x), ∀x. ¬P→Q(x) ⊢ ∀x.Q(x)
 THEOREM INFER R∨¬R, ∀x.R→S(x), ∀x. ¬R→S(x) ⊢ ∀x.S(x)
 THEOREM INFER ∀x.P(x)→Q(x), ∀x.Q(x)→R(x) ⊢ ∀x.P(x)→R(x)
 THEOREM INFER ∀x.P(x)→R(x), ∀x.Q(x)→ ¬R(x)  ⊢ ∀x.(P(x)→¬Q(x)) ∧ (Q(x)→¬P(x))
...
END
```

FIGURE 1e. Panel of conjectures.

```
PREFIX          10   var
POSTFIX         10   inscope

RULE            "⊢∀"(OBJECT m) WHERE FRESH m
                         FROM Γ, var m ⊢ P(m)                INFER Γ ⊢ ∀x.P(x)
RULE            "∀⊢"(m)  FROM Γ, P(m) ⊢ C AND Γ ⊢ m inscope  INFER Γ, ∀x.P(x) ⊢ C
RULE            "⊢∃"(m)  FROM Γ ⊢ P(m) AND Γ ⊢ m inscope     INFER Γ ⊢ ∃x.P(x)
RULE            "∃⊢"(OBJECT m) WHERE FRESH m
                         FROM Γ, var m, P(m) ⊢ C             INFER Γ, ∃x.P(x) ⊢ C

RULE            “inscope”                                    INFER Γ, var m ⊢ m inscope
```

FIGURE 1f. Rules for variable scoping.

the conjectures we present with this encoding.[2] The panel automatically includes a button which allows the user to define additional conjectures.

Jape's screen display includes a ‘proof pane’ and a ‘proviso pane’. The illustrations which follow, taken directly from the screen display of the MacOS implementation, include the content of the proviso pane whenever it is not empty. We begin with a proof of the first conjecture from Figure 1e.[3]

A rule or a tactic can be applied most simply by selecting it from the menu. In this case the correct step is to apply $\vdash\rightarrow$ once, giving the tree

$$\dfrac{P\rightarrow(Q\rightarrow R),\ (P\rightarrow Q)\ \vdash\ (P\rightarrow R)}{P\rightarrow(Q\rightarrow R)\ \vdash\ (P\rightarrow Q)\rightarrow(P\rightarrow R)}\ {\scriptstyle\vdash\rightarrow}$$

and then again, expanding it to

[2]Some of the bracketing in these conjectures is redundant, for example because $\rightarrow$ is defined to be right-associative in Figure 1c. Jape preserves the user's original bracketing as far as possible, even when it is redundant.

[3]This, like the other examples in this paper, is a trivial logical problem and we are well aware that any self-respecting theorem prover would solve it instantaneously. We use examples not to illustrate the power of our system, but the principles of our design.

$$\dfrac{\dfrac{P\to(Q\to R),\ P\to Q,\ P\vdash R}{P\to(Q\to R),\ (P\to Q)\vdash(P\to R)}\ {\scriptstyle\vdash\to}}{P\to(Q\to R)\vdash(P\to Q)\to(P\to R)}\ {\scriptstyle\vdash\to}$$

Next we need to apply $\to\vdash$, but the rule matches ambiguously. The rule is

$$\dfrac{\Gamma\vdash A \quad \Gamma, B\vdash C}{\Gamma, A\to B\vdash C}\to\vdash$$

To match the consequent of the rule we have to find a match for $A \to B$, its principal formula. There are two hypothesis formulae which would match: $P \to (Q \to R)$ and $P \to Q$. In this example it does not matter which of the two is chosen, since either will lead eventually to a successful conclusion. But Jape requires that we arbitrate and we can do so most conveniently by formula-selecting one of the two with a mouse click:

$$\dfrac{\dfrac{\boxed{P\to(Q\to R)},\ P\to Q,\ P\vdash R}{P\to(Q\to R),\ (P\to Q)\vdash(P\to R)}\ {\scriptstyle\vdash\to}}{P\to(Q\to R)\vdash(P\to Q)\to(P\to R)}\ {\scriptstyle\vdash\to}$$

The effect is to focus attention on the selected formula, to force the tool to unify it with a principal formula of the rule that is applied.[4]

The effect of the $\to\vdash$ rule is to decompose the selected formula; at the same time, because of the AUTOMATCH directive of Figure 1c, Jape tries to apply *hyp* to each of the antecedents and succeeds in one case. The result is the tree

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P\to Q,\ P\vdash P}\ {\scriptstyle\text{hyp}} \qquad P\to Q,\ P,\ (Q\to R)\vdash R}{P\to(Q\to R),\ P\to Q,\ P\vdash R}\ {\scriptstyle\to\vdash}}{P\to(Q\to R),\ (P\to Q)\vdash(P\to R)}\ {\scriptstyle\vdash\to}}{P\to(Q\to R)\vdash(P\to Q)\to(P\to R)}\ {\scriptstyle\vdash\to}$$

Two further applications of $\to\vdash$, each with AUTOMATCH assistance, complete the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P\to Q,\ P\vdash P}\ {\scriptstyle\text{hyp}} \qquad \dfrac{\dfrac{}{P,\ Q\to R\vdash P}\ {\scriptstyle\text{hyp}} \qquad \dfrac{\dfrac{}{P,\ Q\vdash Q}\ {\scriptstyle\text{hyp}} \quad \dfrac{}{P,\ Q,\ R\vdash R}\ {\scriptstyle\text{hyp}}}{P,\ Q\to R,\ Q\vdash R}\ {\scriptstyle\to\vdash}}{P\to Q,\ P,\ (Q\to R)\vdash R}\ {\scriptstyle\to\vdash}}{P\to(Q\to R),\ P\to Q,\ P\vdash R}\ {\scriptstyle\to\vdash}}{P\to(Q\to R),\ (P\to Q)\vdash(P\to R)}\ {\scriptstyle\vdash\to}}{P\to(Q\to R)\vdash(P\to Q)\to(P\to R)}\ {\scriptstyle\vdash\to}$$

[4]Without pre-selection Jape would conduct a dialogue requiring us to choose between the two possible matches. Such dialogues are unnecessarily confusing and pre-selection helps to avoid them whenever possible. The slogan we use to explain the gesture to our users is 'click on the formula you want to work with'.

5.2. Proof by pointing

Rule selection in this calculus, as in many others, is a matter of choosing a formula and applying the rule determined by the principal connective or binding operator of that formula. Once a formula is chosen, rule selection can be automatic: Jape supports automatic rule selection by associating rules with 'hits' (double-clicks) on formulae in either hypothesis or conclusion positions. The HYPHIT and CONCHIT directives give a pattern to describe what must be selected and to differentiate between which side of the sequent has been 'hit'. This is proof by pointing at level 1, in the sense of [17]:

```
HYPHIT      A⊢A       IS hyp
HYPHIT      A∧B⊢C     IS "∧⊢"
HYPHIT      A→B⊢C     IS "→⊢"
...
CONCHIT     B∧C       IS "⊢∧"
CONCHIT     B→C       IS "⊢→"
...
```

Given these directives, almost every proof in the calculus can be carried out just by double-clicking formulae—proof as video game! Such automation is optional and we intentionally omit it from many encodings—in particular when we want our novice users to learn how to parse formulae and select a rule for themselves.

5.3. Variables and provisos

We go to a great deal of trouble to ensure that proofs of theorems in Jape are schematic in the names used in the theorem itself: that is, that every substitution instance of the theorem corresponds to a valid instance of a proof tree. This can expose some subtle distinctions and non-distinctions between substitution instances.

For example, consider the theorem schema $\lambda x.\lambda y.x: T1 \to T2 \to T1$ of the polymorphically-typed λ-calculus. Given object types int and real and object variables a and b, it is quite clear that $\lambda a.\lambda b.a$: int→real→ int is an instance of the theorem and that $\lambda a.\lambda a.a$: int→real→int is not, simply because the first obeys the binding structure of the theorem and the second violates it. Variables assigned to x and y in an instance of the theorem must be distinct and Jape enforces the distinction with an internally generated proviso. When the theorem is stated, Jape will deduce from its binding structure that it must be augmented with x NOTIN y. It will not, without special instruction, show that proviso to the user, but it is exploited during the proof of the theorem and it is enforced on instantiation.

One might suppose that schematic names which are distinct in the statement of a theorem should always require distinct assignments on instantiation, but that is not so. For example, $\lambda a.\lambda a.a$: int→real→real is an instance of $\lambda x.\lambda y.y: T1 \to T2 \to T2$. One would expect to be able to prove the theorem without having to appeal to a distinction between x and y. Jape's analysis of the binding does not show a need for a proviso, so the distinction is not enforced during proof or instantiation.

It is surprising, though, to find that there is a proof of

$\exists x.(\forall y.Q)\vdash\forall y.(\exists x.Q)$ in the logic of Figure 1a which does not at any stage appeal to a distinction between schematic variables x and y:[5]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Q \vdash Q}\;\text{hyp}}{Q \vdash \exists x.Q}\;\vdash\exists}{\forall y.Q \vdash \exists x.Q}\;\forall\vdash}{\forall y.Q \vdash \forall y.(\exists x.Q)}\;\vdash\forall}{\exists x.(\forall y.Q) \vdash \forall y.(\exists x.Q)}\;\exists\vdash$$

Because the proof does not require that x is distinct from y, we can substitute z for both, and $\exists z.(\forall z.P)\vdash\forall z.(\exists z.P)$ is a substitution instance of this theorem in the logic of Figure 1a—a theorem which appears to have quite a different binding structure to the original.

There is nothing wrong with Jape: the instance is a theorem of the logic and the tree above with both x and y replaced by z, Q by P is a proof of that. There is nothing necessarily wrong with the logic: the proof does not suppose that there must be an occurrence of x or of y in Q and the instance exploits that. The proof above does depend on the fact that $\forall v.R\vdash\exists v.R$ is a theorem of the logic of Figure 1a: that is so because the rules of Figure 1a implicitly assume that the universe of quantification is non-empty.

Jape does not have a built-in meta theory of universes and instances, but it can accept a syntactic description of such a theory. To accommodate a theory which does not assume a non-empty universe, it is sufficient to include the syntax and rules of Figure 1f. Here the assumption *var c* says that the universe includes an individual *c*; the goal *d inscope* demands that the universe includes an individual *d* and is proved only by a matching assumption *var d*.

The scoping rules prevent an attempt to prove $\forall v.P(v)\vdash\exists v.P(v)$:

$$\dfrac{\dfrac{\dfrac{}{P(m) \vdash P(m)}\;\text{hyp} \qquad P(m) \vdash m\ \text{inscope}}{P(m) \vdash \exists v.P(v)}\;\vdash\exists \qquad m\ \text{inscope}}{\forall v.P(v) \vdash \exists v.P(v)}\;\forall\vdash$$

Neither of the *m inscope* goals is provable, so the attempt fails.

The additional assumptions introduced by the $\exists\vdash$ and $\vdash\forall$ rules of Figure 1f prevent a proof like that above of $\exists x.\forall y.Q\vdash\forall y.\exists x.Q$. At an early stage in any attempt the FRESH proviso in either one of those rules will demand a distinction between x and y and the offending instance will be prohibited:

$$\dfrac{\dfrac{\text{var}\ x\ ,\ \forall y.Q,\ \text{var}\ y \vdash \exists x.Q}{\text{var}\ x\ ,\ \forall y.Q \vdash \forall y.(\exists x.Q)}\;\vdash\forall}{\exists x.(\forall y.Q) \vdash \forall y.(\exists x.Q)}\;\exists\vdash$$

$y\ \text{NOTIN}\ x$

Jape's treatment of schematic variables is subtle but extremely powerful and is much more than simple α-conversion. Where distinctions can be deduced from binding structure, they are automatically and invisibly enforced upon instances and exploited during proofs. Where they are unnecessary, they are not applied. Most users will not even notice what is going on, since a simple use of predicate notation will automatically insert an invisible proviso. For example, a distinction between x and y is automatically imposed if the theorem above is restated as $\exists x.(\forall y.R(x,y))\vdash\forall y.(\exists x.R(x,y))$. Conversely, the possibility of a distinction between x and w in an instance of $\exists x.(\forall y.P(x,y))\vdash\forall v.(\exists w.P(w,v))$ is exploited during a proof.

The intention of this treatment is to preserve soundness in the proof of schematic rules, so that every theorem becomes a derived rule without antecedents[6] and it is never necessary to re-run the proof of that theorem when establishing an instance of it.

5.4. Alternative display styles: box or tree

The examples shown so far have been compact, but it is obvious that Jape's commitment to interaction with a full proof display will restrict its range to relatively small proofs—or at best, with heavy use of lemmas, to sections of proofs small enough for convenient display.

The tree presentation corresponds directly to the inference rules of the calculus but is wasteful of space. In single-conclusion calculi it is easy to be more economical. The box display, adapted from the notation of [20], writes out hypothesis formulae no more often than is necessary and hides applications of IDENTITY rules. The result is a more nearly linear presentation that is much easier for the user to search and much more compact on the screen. For example, the partial proof shown in Figure 2a is already becoming rather wide and if the subproofs were developed further it would become completely unwieldy. The equivalent box display of Figure 2b is far more economical and it is much easier to search.

In constructing the box version Jape has done no significant translation. The tree is printed in postfix order—subtrees first, then root—and a box corresponds to a subtree whose root has a hypothesis formula that is not in its parent sequent. The reason printed next to each line quotes the line (or box) of each antecedent subtree, together with a reference to any principal hypothesis formulae in the consequent.[7]

[5]To construct this proof in Jape it is necessary to provide, by text selection, an argument to each rule application: x in the first step, y in the second and so on. Without those arguments, and without helpful provisos, it is difficult to find a proof.

[6]Jape has for some time been capable of proofs of derived rules with antecedents, but the GUI mechanisms required are still under development at the time of writing.

[7]Left rules of this logic have principal hypothesis formulae; for example,

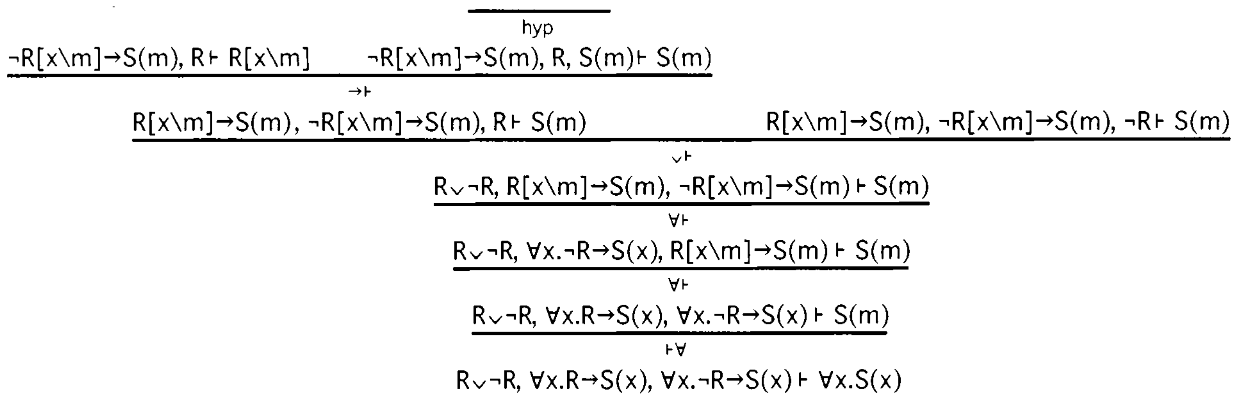

hyp
¬R[x\m]→S(m), R⊢ R[x\m]
¬R[x\m]→S(m), R, S(m)⊢ S(m)
→⊢
R[x\m]→S(m), ¬R[x\m]→S(m), R⊢ S(m)
R[x\m]→S(m), ¬R[x\m]→S(m), ¬R⊢ S(m)
∨⊢
R∨¬R, R[x\m]→S(m), ¬R[x\m]→S(m) ⊢ S(m)
∀⊢
R∨¬R, ∀x.¬R→S(x), R[x\m]→S(m) ⊢ S(m)
∀⊢
R∨¬R, ∀x.R→S(x), ∀x.¬R→S(x) ⊢ S(m)
⊢∀
R∨¬R, ∀x.R→S(x), ∀x.¬R→S(x) ⊢ ∀x.S(x)

FIGURE 2a. Wide tree proof.

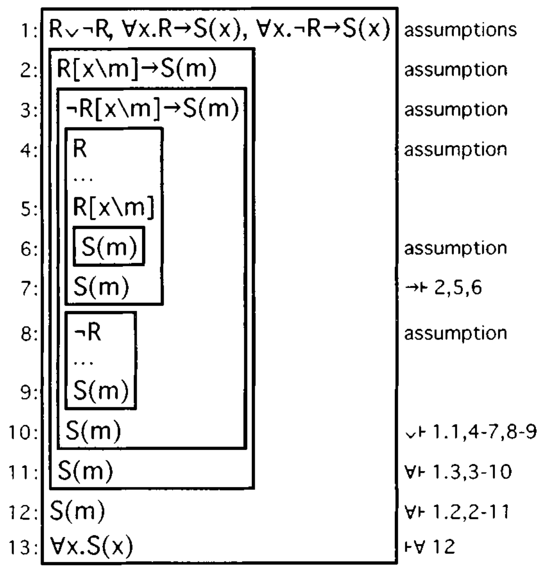


FIGURE 2b. Narrower box proof.

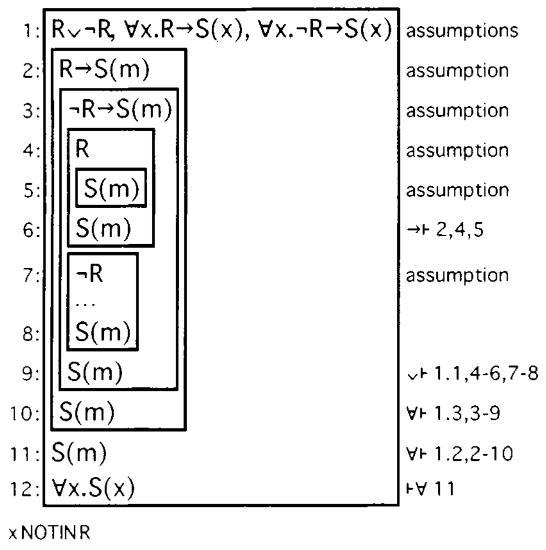


FIGURE 2c. Manual hyp generates proviso.

The box style of display is best suited to natural deduction and when used with the single-conclusion sequent calculus it exhibits some oddities. For example, the justification of line 6 in Figure 2c cites three formulae: $R \to S(m)$ on line 2 (the principal hypothesis formula in the consequent of the rule); R on line 4; and $S(m)$ on line 5. 'Line' 5 is a box, in which the conclusion $S(m)$ is proved by hyp from the hypothesis $S(m)$—Jape hides the use of hyp and shows a single line.

5.5. Unification of substitution forms

In Figures 2a and 2b the lack of an explicit proviso x NOTIN R has forced a substitution form to surface and the display contains a mixture of substitution and predicate formulae. It is evident that in Figure 2b we would like lines 4 and 5 to be unified by the hyp rule, but one is a substitution form and the other is not: AUTOMATCH *hyp* will not solve the problem. When directed to use hyp by the user, Jape easily resolves the problem to produce Figure 2c, which includes an inferred proviso. The inference is that since the two forms are to be the same then there cannot be any occurrences of x in R (the only alternative would be that x and m were always the same, which is impossible since x is schematic and m is internal). Completion of the proof is straightforward.

When the consequent of a rule includes a substitution form, unification is by no means so simple. To illustrate the problem we consider an encoding of equational reasoning. Most of the details of the encoding will not be discussed here: it is based on a straightforward treatment of equality, using the rules given in Figure 3a. The encoding takes advantage of an optional facility in Jape to leave out the hypothesis part of rules specified in logics in which every rule is purely additive—a minor textual convenience.

We use set braces rather than square brackets in substitution forms, because square brackets are conventionally used to enclose lists in functional programming. The necessary syntactic directive is

```
SUBSTFIX  2000{x\A}
```

The rewrite rule which is the basis of our treatment of equational reasoning is

see line 10 of Figure 2b which uses the left ∨ rule.

```
RULE "= reflexive"                                             INFER X = X
RULE "= transitive"(Y)     FROM X = Y AND Y = Z                INFER X = Z
RULE "= symmetric"         FROM X = Y                          INFER Y = X
RULE "(,)="                FROM X0=X1 AND Y0=Y1                INFER (X0, Y0) = (X1, Y1)

RULE rewrite (X,OBJECT v) FROM X=Y AND A{v\Y} INFER A{v\X}
```

FIGURE 3a. Equality rules.

```
RULE id            INFER id X = X
RULE "•"           INFER (F • G) X = F(G X)
RULES rev ARE
      INFER rev [ ] = [ ] AND
      INFER rev [X]= [X] AND
      INFER rev (Xs++Ys) = rev Ys ++ rev Xs
END
```

FIGURE 3b. Function definitions as rewrite rules.

```
RULE ext (OBJECT x) WHERE FRESH x FROM F x = G x INFER F = G

RULE listinduction (B, OBJECT x, OBJECT xs, OBJECT ys, ABSTRACTION A) WHERE FRESH x, xs, ys
     FROM A[] AND A[x] AND A xs, A ys ⊢ A(xs++ys) INFER A(B)
```

FIGURE 3c. Extensionality and induction rules.

$$\frac{\Gamma \vdash X = Y \quad \Gamma \vdash A\{v\backslash Y\}}{\Gamma \vdash A\{v\backslash X\}}$$

In the particular realm of functional programming, we have represented definitions of functions with collections of equality rules. So, for example, we have the definitions shown in Figure 3b, using the notation of [21].

In this encoding we treat functions over lists symmetrically in terms of their effect on the empty list [ ], a singleton list $[x]$ and a concatenation of lists $xs{+}{+}ys$. To support these definitions we have rules of extensionality and list induction, shown in Figure 3c. Each of these rules makes a step of generalization and so each requires one or more FRESH provisos—although in practice that proviso is almost always rendered redundant by the fact that, unless otherwise directed, Jape will use a freshly-minted internal variable in place of each of the OBJECT parameters.

To prove the theorem rev•rev=id we begin with extensionality:

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

At this point we use $(rev{\bullet}rev)x = rev(rev\ x)$, an instance of the definition $(F{\bullet}G)x = F(G\ x)$, to replace the left-hand side of line 1. This gives the display

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

Not everything in the tree is displayed, because a tactic has been employed that applies the rewrite rule, then hides the left antecedent of the application, which in this case is the equality $(rev{\bullet}rev)x = rev(rev\ x)$. The detail of the proof step can be uncovered by double-clicking on the justification of line 2, to see

```
1: (rev•rev)x=rev(rev x) •
  ...
2: rev(rev x)=id x
3: (rev•rev)x=id x        [rewrite] 1,2
4: rev•rev=id             ext 3
```

The particular instance of the rewrite rule which was used in this step was

$$\frac{\vdash (rev{\bullet}rev)x = rev(rev\,x) \quad \vdash (x1 = id\,x)\{x1\backslash rev(rev\,x)\}}{\vdash (x1 = id\,x)\{x1\backslash (rev{\bullet}rev)x\}}$$

and it is necessary to describe how this instance was selected.

Jape can tackle the problem using one of two mechanisms. The first, which we call *abstraction*, is given a subformula and constructs a substitution by searching for all occurrences of that subformula. For example, given the problem of unifying the consequent $\_A\{x1\backslash(rev{\bullet}rev)x\}$ with the problem formula $(rev{\bullet}rev)x = id\,x$, it converts the problem formula into the substitution $(x1 = id\,x)\{x1\backslash(rev{\bullet}rev)x\}$ and then unifies $\_A$ with $x1 = id\,x$. The algorithm is a simplified form of higher-order unification.

The alternative mechanism is called *substitution selection*. The user describes the particular substitution which is to be used by text-selecting one or more instances of a subformula within it. In this case the user would make the single text-selection

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

That describes the substitution $(\_x1 = id\,x)\{\_x1\backslash (rev{\bullet}rev)x\}$ which is then unified with the form $\_A\{x2\backslash\_Y\}$ from the consequent of the rule to produce the same effect as the abstraction mechanism. It would be inappropriate to give full details here: we note that interpretation of the user's gesture as a substitution description is by means of a tactic and that the unification mechanism treats the resulting substitution carefully, avoiding simplifying it and unifying by structure if possible.

The induction rule of this encoding, given above, uses a substitution formula in its consequent and in each of its antecedents. By placing the correct tactic in the relevant menu, we ensure that list induction is always applied using the substitution-selection mechanism. We therefore choose each of the subformula instances over which we wish to perform the induction

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

and then apply the induction rule:

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

In this encoding, automation has been carried to the point that the function-definitions panel includes a button which first invokes a search for a subformula that will unify with the left-hand side of some function definition, then invokes the rewrite rule, using the abstraction mechanism with that subformula as argument, then finally applies the definition to the left antecedent of the rewrite rule. Applied repeatedly, this mechanism solves each of the remaining problems, giving the proof shown in Figure 4.

5.6. Simulating forward reasoning

Jape is a backward-reasoning engine and its fundamental data structure is a tree of sequents. We have, nevertheless, found it possible to support a partial simulation of forward reasoning, based on a mechanism which hides instances of cut rules when a proof is shown in box style. The effect is to make natural deduction [22], which in tree form is difficult to explain [23, 24] let alone to use, sufficiently natural that it can be introduced to first-year undergraduates without difficulty.

We illustrate the mechanism using natural-deduction elimination and introduction rules for $\to$, normally written as

$$\dfrac{\begin{matrix}\vdots & \vdots\\ A & A\to B\end{matrix}}{B}\to -E \text{ and } \dfrac{\begin{matrix}[A]\\ \vdots\\ B\end{matrix}}{A\to B}\to -I$$

but encoded in Jape as

$$\dfrac{\Gamma\vdash A \quad \Gamma\vdash A\to B}{\Gamma\vdash B} \text{ and } \dfrac{\Gamma, A\vdash B}{\Gamma\vdash A\to B}$$

We consider first a proof of the conjecture $P\to Q, Q\to R, P\vdash R$. As a backward-reasoning problem this requires two steps with the elimination rule and at each step the user must either provide an argument to correspond to A in the rule or must force a unification afterwards; we choose here to provide an argument in each case in order to cut down the number of steps.

The first step uses $\to$-E from Q and $Q\to R$ to conclude R; it is sufficient in Jape to text-select Q as an argument and apply the rule

$$P\to Q, Q\to R, P\vdash R$$

The second antecedent of that step is proved automatically with hyp. The next step is similar, to prove Q from P and $P\to R$; as before we text-select an instance of P to serve as argument to the $\to$-E rule

$$\dfrac{P\to Q, Q\to R, P\vdash Q \qquad \dfrac{}{P\to Q, Q\to R, P\vdash Q\to R}\text{hyp}}{P\to Q, Q\to R, P\vdash R}\to\text{-E}$$

and the proof is completed with two automatic applications of hyp

$$\dfrac{\dfrac{\dfrac{}{P\to Q, Q\to R, P\vdash P}\text{hyp} \qquad \dfrac{}{P\to Q, Q\to R, P\vdash P\to Q}\text{hyp}}{P\to Q, Q\to R, P\vdash Q}\to\text{-E} \qquad \dfrac{}{P\to Q, Q\to R, P\vdash Q\to R}\text{hyp}}{P\to Q, Q\to R, P\vdash R}\to\text{-E}$$

This proof is hardly *natural* deduction: we have both P and $P\to Q$ to begin with and what could be more natural than to use the $\to$-E rule to produce Q? Yet in reasoning backwards we have had to start with the last step of the proof. Jape can simulate the more natural forward calculation in its box display style. The starting position is

```
1: | P→Q, Q→R, P |  premises
   | ...         |
2: | R           |
```

We select $P\to Q$ as the formula we want to work with (Jape does not demand that we select a conclusion because only one is open)

```
1: | [P→Q], Q→R, P |  premises
   | ...           |
2: | R             |
```

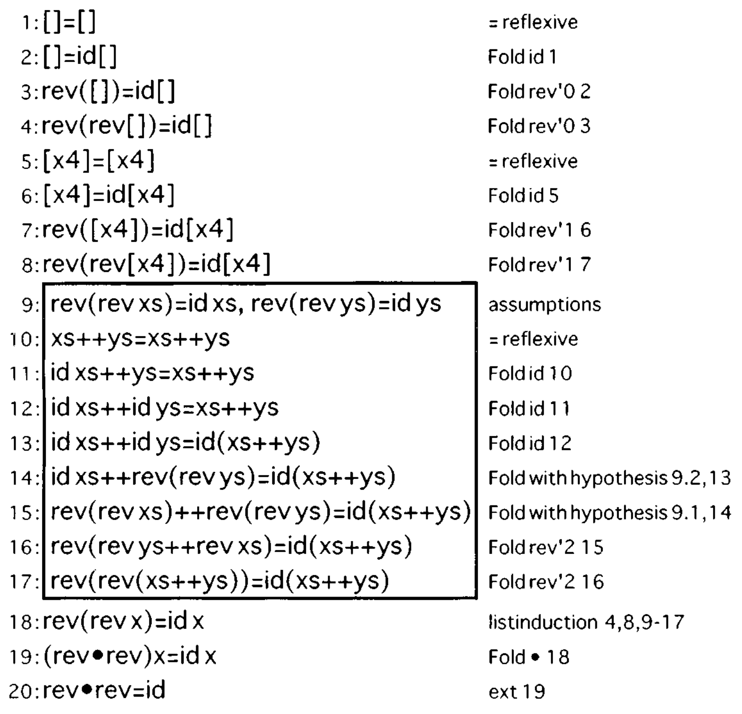
and apply the rule

```
1: | P→Q, Q→R, P |  premises
2: | Q           |  →-E 1.3,1.1
   | ...         |
3: | R           |
```

The effect is as if a forward step has been taken: from the hypotheses we have deduced Q and we can use it to establish R:

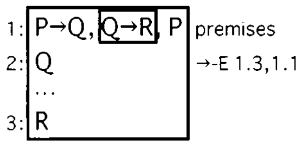
$$\begin{matrix}\dfrac{P \quad P\to Q}{Q}\\ \vdots\\ R\end{matrix}$$

We continue to bridge the gap between Q and R to complete the proof. First we select the other implication hypothesis



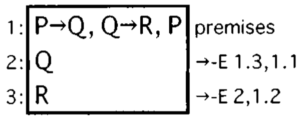
```
 1: []=[]                                      = reflexive
 2: []=id[]                                    Fold id 1
 3: rev([])=id[]                               Fold rev'0 2
 4: rev(rev[])=id[]                            Fold rev'0 3
 5: [x4]=[x4]                                  = reflexive
 6: [x4]=id[x4]                                Fold id 5
 7: rev([x4])=id[x4]                           Fold rev'1 6
 8: rev(rev[x4])=id[x4]                        Fold rev'1 7
 9: rev(rev xs)=id xs, rev(rev ys)=id ys       assumptions
10: xs++ys=xs++ys                              = reflexive
11: id xs++ys=xs++ys                           Fold id 10
12: id xs++id ys=xs++ys                        Fold id 11
13: id xs++id ys=id(xs++ys)                    Fold id 12
14: id xs++rev(rev ys)=id(xs++ys)              Fold with hypothesis 9.2,13
15: rev(rev xs)++rev(rev ys)=id(xs++ys)        Fold with hypothesis 9.1,14
16: rev(rev ys++rev xs)=id(xs++ys)             Fold rev'2 15
17: rev(rev(xs++ys))=id(xs++ys)                Fold rev'2 16
18: rev(rev x)=id x                            listinduction 4,8,9-17
19: (rev•rev)x=id x                            Fold • 18
20: rev•rev=id                                 ext 19
```

FIGURE 4. A completed proof in equational reasoning.



```
1: P→Q, Q→R, P     premises
2: Q               →-E 1.3,1.1
   ...
3: R
```

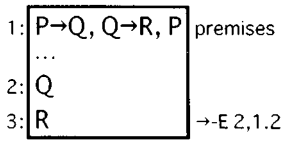
and then we apply the →-*E* rule once again



```
1: P→Q, Q→R, P     premises
2: Q               →-E 1.3,1.1
3: R               →-E 2,1.2
```

This proof is far more convenient to construct than the backward version, and not only for those users who prefer to think in terms of forward reasoning steps.

Jape is not committed to making the proof in just the order shown above: if the assumptions are worked on in the other order, the effect is apparently to produce the backward proof. After the first step we have

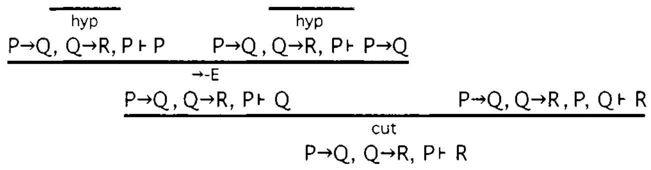


```
1: P→Q, Q→R, P     premises
   ...
2: Q
3: R               →-E 2,1.2
```

and the second step produces the same final proof as before.

No artificial intelligence has been used to produce an apparent discrimination between forward and backward proof steps. The effects are the consequence of a simple mechanism. Inside Jape each step has been based on an application of cut followed by an application of hyp.

After the first step in the first example, the inference tree behind the scenes is



$$\dfrac{\dfrac{\dfrac{}{P\to Q,\ Q\to R,\ P\vdash P}\ \text{hyp}\qquad \dfrac{}{P\to Q,\ Q\to R,\ P\vdash P\to Q}\ \text{hyp}}{P\to Q,\ Q\to R,\ P\vdash Q}\ \to\text{-E}\qquad P\to Q,\ Q\to R,\ P,\ Q\vdash R}{P\to Q,\ Q\to R,\ P\vdash R}\ \text{cut}$$

Each of the hyp applications has served to move a hypothesis from left to right and the cut has moved conclusion *Q* from right to left. The box and line display simply hides these structural steps.

To make a simulated forward step the user selects a hypothesis and applies an elimination rule. To make a backward step, a conclusion is selected before the rule is applied. The mechanism we employ detects whether or not a hypothesis has been selected and if so runs a tactic to apply that rule surrounded with the necessary cut and hyp steps.

We associate a tactic with the menu entry of each elimination rule, as shown in Figures 5a and 5b. The ForwardOrBackward tactic has to decide which kind of step to make: it does so by using the construct (LETHYP _P(F n Rule)) to test whether there is a selected hypothesis formula which can be unified with _P; if so it runs the tactic (F n Rule). If not, it runs the tactic (WITHSELECTIONS Rule), which applies the rule as if a tactic had not been involved.

The argument F to ForwardOrBackward is always either ForwardCut or ForwardUncut; ForwardCut applies a cut step, then the rule (WITHARGSEL Rule), then chooses an antecedent (SUBGOAL n, which in this example always selects antecedent number 0 or number 1) and applies hyp to the selected hypothesis formula (WITHHYPSEL hyp). ForwardUncut does the same without the cut step.

If nothing is selected when a rule is applied, Jape makes

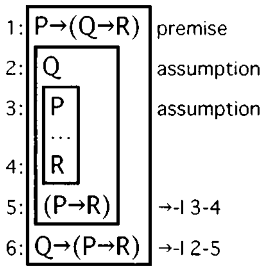
```
MENU Rules IS
    ENTRY "→-I"
    ENTRY "∧-I"
    ENTRY "∨-I(L)"    IS ForwardOrBackward ForwardCut 0 "∨-I(L)"
    ENTRY "∨-I(R)"    IS ForwardOrBackward ForwardCut 0 "∨-I(R)"
    ENTRY "¬-I"
    ENTRY "∀-I"
    ENTRY "∃-I"
    SEPARATOR
    ENTRY "→-E"       IS ForwardOrBackward ForwardCut 1 "→-E"
    ENTRY "∧-E(L)"    IS ForwardOrBackward ForwardCut 0 "∧-E(L)"
    ENTRY "∧-E(R)"    IS ForwardOrBackward ForwardCut 0 "∧-E(R)"
    ENTRY "∨-E"       IS ForwardOrBackward ForwardUncut 0 "∨-E"
    ENTRY "¬-E"       IS ForwardOrBackward ForwardCut 0 "¬-E"
    ENTRY "∀-E"       IS ForwardOrBackward ForwardCut 0 "∀-E"
    ENTRY "∃-E"       IS ForwardOrBackward ForwardUncut 0 "∃-E"
    SEPARATOR
    ENTRY hyp
END
```

FIGURE 5a. Menu entries for natural deduction.

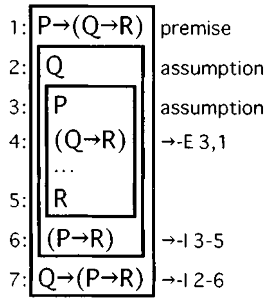
```
TACTIC ForwardOrBackward (F, n, Rule)
  WHEN  (LETHYP _P (F n Rule))
        (WITHSELECTIONS Rule)

TACTIC ForwardCut (n,Rule)
  SEQ cut (ForwardUncut (n,Rule))

TACTIC ForwardUncut (n,Rule)
  SEQ   (WITHARGSEL Rule)
        (SUBGOAL n)
        (WITHHYPSEL hyp)
```

FIGURE 5b. Tactics supporting forward reasoning.

a backward step if it can. Most introduction rules are best applied backwards in any case. We make no attempt to support the introduction of arbitrary assumptions: they are rationally introduced when a conclusion is reduced or a theorem is invoked. For example, the first two steps in a proof of $P \to Q \to R \vdash Q \to P \to R$ are backward-reasoning steps using an introduction rule:



followed by a forward step:



5.7. Multiple-conclusion calculi—additive, intuitionistic, multiplicative

Sequent formulae are treated symmetrically in Jape and it is as straightforward to treat a calculus with multiple conclusions as with multiple hypotheses. The most obvious example is the classical sequent calculus.

Figure 6a is a proof of Peirce's law in the classical sequent calculus. The rules used in this example are 'additive' in the sense of linear logic: non-principal assumptions and conclusions are copied from consequent to antecedents:

```
RULE "⊢→"     FROM Γ,A ⊢ B,Δ INFER Γ ⊢ A→B,Δ
RULE "→⊢"     FROM Γ ⊢ A,Δ AND Γ,B ⊢ Δ
              INFER Γ,A→B ⊢ Δ
```

Additive copying is by no means essential: we have encoded intuitionistic versions of the same rules and with these versions of the rules attempts to prove Peirce's law fail, as illustrated by Figure 6b.

```
RULE "⊢→"     FROM Γ,A ⊢ B INFER Γ ⊢ A→B,Δ
RULE "→⊢"     FROM Γ ⊢ A AND Γ,B ⊢ Δ
              INFER Γ,A→B ⊢ Δ
```

We have also encoded 'multiplicative'—context-splitting—versions which, in association with a version of axiom that permits only a single conclusion on each side and the correspondingly necessary weakening and contraction rules, permits a proof of Peirce's law as shown in Figure 6c.

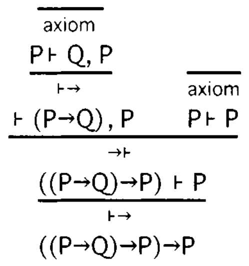


FIGURE 6a. Peirce's law in the classical sequent calculus.

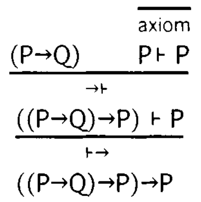


FIGURE 6b. Doomed attempt to prove Peirce's law in intuitionistic multiple-conclusion sequent calculus.

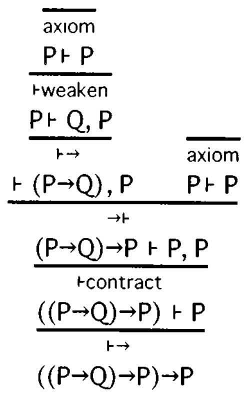


FIGURE 6c. Peirce's law in a calculus with multiplicative rules and non-weakening *axiom*.

```
RULE "⊢→"     FROM Γ,A ⊢ B,Δ INFER Γ ⊢ A→B,Δ
RULE "→⊢"     FROM Γ ⊢ A,Δ AND Γ',B ⊢ Δ'
              INFER Γ,Γ',A→B ⊢ Δ,Δ'
RULE axiom    INFER P ⊢ P
```

Jape defers context-splitting problems by using the UNIFIESWITH proviso and the user can employ a drag-and-drop gesture to resolve any uncertainty. In an intermediate stage of the proof using the multiplicative rules, shown in Figure 6d, the split context is described and the uncertainty registered with a proviso. We can drag one of the copies of P into $\Delta 1$ and the other into $\Delta 2$ to produce Figure 6e, in which the uncertainty is resolved.

5.8. Proofs using unknowns

In the examples so far unknowns have appeared rarely and have been avoided by text-selection of arguments. But unknowns are often necessary markers of deferred choice during a proof and can remain visible for several proof steps. An example of this is found in our encoding of the Hindley–Milner type-assignment algorithm. Unknowns are used to stand for so-far-undiscovered type formulae, to be replaced

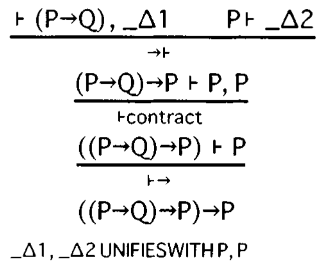


FIGURE 6d. Intermediate stage of proof with multiplicative rule.

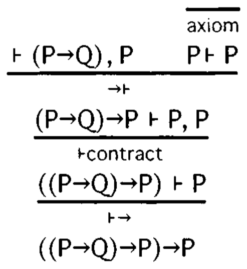


FIGURE 6e. Uncertainty of Figure 6d resolved by drag-and-drop gestures.

by type variables once the structure of those formulae is known.

The rules encoded in Jape, shown in Figure 7, are faithful to the algorithm as presented in [25]. C and C′ are contexts; the left-hand side of a sequent is a list rather than a bag; we have omitted rule names for simplicity. We mark a monotype with a # symbol; we use $\gg$ for specialization from type scheme to type and $\ll$ for generalization from type to type scheme.

An example problem is shown in Figure 8a. After the first step (Figure 8b) the type of the λ formula is undetermined and to calculate the most general type it is necessary either to guess correctly the type which corresponds to $\_T3$ or to complete the deduction on the left-hand antecedent before starting the generalization step. That deduction produces the tree shown in Figure 8c. Note the proviso that has been generated as a result of the fact that x must not appear in the context $y \Rightarrow \_T3$; that proviso will disappear after the generalization step, once $\_T3$ is replaced by a type variable.[8] Note that there is no proviso x NOTIN y: Jape has silently deduced from the binding structure of the formula $\lambda x.\lambda y.x$ that x and y must be distinct.

The basis of the generalization step is a structural induction which computes the list of type variables which are to replace unknowns in the monotype. Each step of the calculation uses a formula $type \bullet scheme_{in} \triangleleft scheme_{out}$: the rules used are shown, in the order in which they are considered, in Figure 9a; the operators $\bullet$ and $\triangleleft$ are arbitrary punctuation dividing the components. Note the *ad hoc* treatment of an indexed family of rules 'new t$\bullet$...' used to generate a range of possible bindings. Using a tactic which

[8]With a better description of problem syntax, a more capable tool could deduce that, in this logic, program variable names cannot appear in type formulae.

```
FROM C⊢F:T1→T2 AND C⊢G:T1        INFER C⊢F G:T2
FROM C⊢E:T1 AND C⊢F:T2           INFER C⊢(E,F):T1×T2

FROM C⊢E:T1 AND C⊢#T1«S1 AND C,x⇒S1 ⊢F:T        INFER C⊢let x=E in F end:T

FROM C⊢x⇒S AND S»T               INFER C⊢x:T
WHERE x NOTIN C'                 INFER C,x⇒S,C' ⊢ x⇒S
                                 INFER #T » T
                                 INFER ∀tt1,tt2,tt3,tt4.TT » TT{T1,T2,T3,T4/tt1,tt2,tt3,tt4}
```

FIGURE 7. Some of the rules of the polymorphically-typed λ-calculus.

```
C ⊢ let f=λx.λy.x in(f f 1,f 1 f)end:(T1→T2→T1)×num
```

FIGURE 8a. A problem in the polymorphically-typed λ-calculus.

```
C ⊢ λx.λy.x:_T3      C ⊢ _T3«_S1      C, f⇒_S1 ⊢ (f f 1,f 1 f):(T1→T2→T1)×num
─────────────────────────────────────────────────────────────────────────────── letrules'0
               C ⊢ let f=λx.λy.x in(f f 1,f 1 f)end:(T1→T2→T1)×num
```

FIGURE 8b. After the first step of the Hindley–Milner type-assignment algorithm.

```
    ──────────────────── C(x)⇒S; S»T
  C, x⇒#_T6, y⇒#_T3 ⊢ x:_T6
  ────────────────────────── λx.E:T1→T2
  C, x⇒#_T6 ⊢ λy.x:_T3→_T6
  ────────────────────────── λx.E:T1→T2
  C ⊢ λx.λy.x:_T6→_T3→_T6      C ⊢ _T6→_T3→_T6«_S1      C, f⇒_S1 ⊢ (f f 1,f 1 f):(T1→T2→T1)×num
  ───────────────────────────────────────────────────────────────────────────────────────────── letrules'0
                     C ⊢ let f=λx.λy.x in(f f 1,f 1 f)end:(T1→T2→T1)×num

  x NOTIN _T3
```

FIGURE 8c. After the type of the λ formula has been calculated.

```
RULES "new t•..." (OBJECT t1) WHERE t1 NOTIN C ARE
         C⊢t1 • #T◁∀t1.T
AND      C⊢t1 • ∀tt1.T◁∀tt1,t1.T
AND      C⊢t1 • ∀tt1,tt2.T◁∀tt1,tt2,t1.T
AND      C⊢t1 • ∀tt1,tt2,tt3.T◁∀tt1,tt2,tt3,t1.T
END

RULE "T1→T2•..."    FROM C⊢T1 • Sin◁Smid AND C⊢T2 • Smid◁Sout        INFER C⊢T1→T2 • Sin◁Sout
RULE "T1×T2•..."    FROM C⊢T1 • Sin◁Smid AND C⊢T2 • Smid◁Sout        INFER C⊢T1×T2 • Sin◁Sout
RULE "[T]•..."      FROM C⊢T • Sin◁Sout                        INFER C⊢[T] • Sin◁Sout
RULE "same T•..."                                              INFER C⊢T • S◁S
```

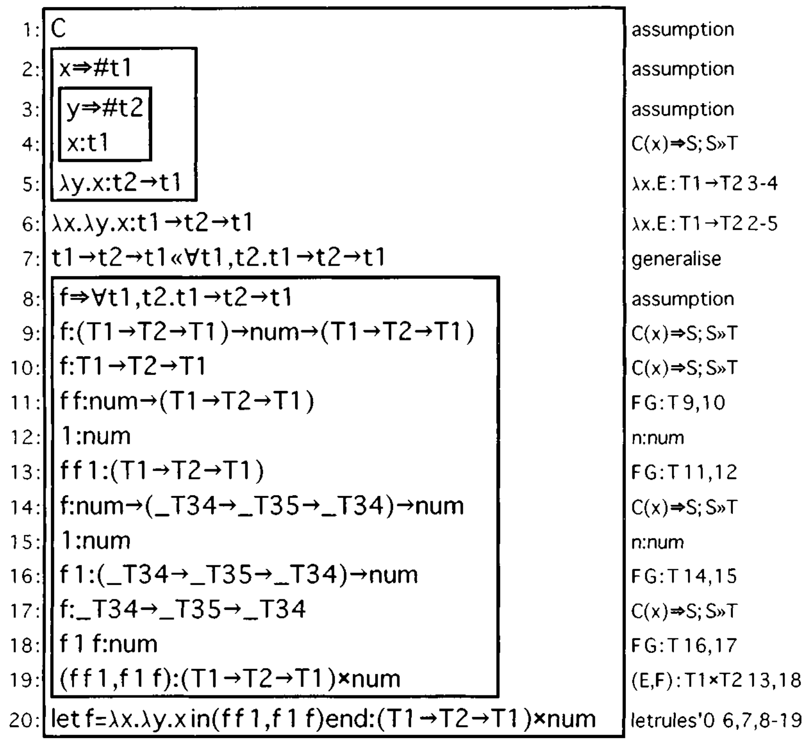
FIGURE 9a. Rules used in the generalization step.

```
   ──────────────── C(x)⇒S; S»T
C, x⇒#t1, y⇒#t2 ⊢ x:t1
─────────────────────── λx.E:T1→T2
C, x⇒#t1 ⊢ λy.x:t2→t1
─────────────────────── λx.E:T1→T2          ──────────────────── generalise
C ⊢ λx.λy.x:t1→t2→t1      C ⊢ t1→t2→t1«∀t1,t2.t1→t2→t1      C, f⇒∀t1,t2.t1→t2→t1 ⊢ (f f 1,f 1 f):(T1→T2→T1)×num
──────────────────────────────────────────────────────────────────────────────────────────────────────── letrules'0
                         C⊢ let f=λx.λy.x in(f f 1,f 1 f)end:(T1→T2→T1)×num
```

FIGURE 9b. After the generalization step.

employs these rules and hides the resulting subtree, Jape carries out generalization as a single action. The effect is shown in Figure 9b.

The rest of the proof is completed in Figure 10, in which there are several distinct specializations of the type scheme bound to f. The remaining unknowns in the proof tree, on



```
 1: C                                              assumption
 2: x⇒#t1                                          assumption
 3: y⇒#t2                                          assumption
 4: x:t1                                           C(x)⇒S; S»T
 5: λy.x:t2→t1                                     λx.E:T1→T2 3-4
 6: λx.λy.x:t1→t2→t1                               λx.E:T1→T2 2-5
 7: t1→t2→t1«∀t1,t2.t1→t2→t1                       generalise
 8: f⇒∀t1,t2.t1→t2→t1                              assumption
 9: f:(T1→T2→T1)→num→(T1→T2→T1)                    C(x)⇒S; S»T
10: f:T1→T2→T1                                     C(x)⇒S; S»T
11: f f:num→(T1→T2→T1)                             FG:T 9,10
12: 1:num                                          n:num
13: f f 1:(T1→T2→T1)                               FG:T 11,12
14: f:num→(_T34→_T35→_T34)→num                     C(x)⇒S; S»T
15: 1:num                                          n:num
16: f 1:(_T34→_T35→_T34)→num                       FG:T 14,15
17: f:_T34→_T35→_T34                               C(x)⇒S; S»T
18: f 1 f:num                                      FG:T 16,17
19: (f f 1,f 1 f):(T1→T2→T1)×num                   (E,F):T1×T2 13,18
20: let f=λx.λy.x in(f f 1,f 1 f)end:(T1→T2→T1)×num letrules'0 6,7,8-19
```

FIGURE 10. The completed type calculation.

lines 14, 16 and 17, can safely be ignored: they could be instantiated to any type formula whatsoever and the proof would still stand.

6. CONCLUSIONS AND FUTURE WORK

As it stands Jape is evidence that it is possible to build a system which can aid step-by-step interactive reasoning over a range of logical systems by animation at the surface level. It is actively in use in teaching at Oxford, QMW and at other locations in the UK.

The aim of the design is to encourage reflective exploration of logical proofs and to operate at the surface level so that users work on the proof task, rather than the tool-direction task. We believe that both novices and experts can benefit from such a mechanism, albeit in different ways. We hope to be able to adapt it to the needs of users, not necessarily experts, who must make industrial-strength proofs.

Our immediate ambition is to extend Jape's interactive behaviour to cover more of the range of two-sided sequent calculi, more conveniently than it does at present. We hope thereby to apply the surface polish of Jape to the kind of logics which industrial users must deal with. If it could be done, that would make Jape a useful tool for conducting experiments with industrial-strength logics.

Dealing with industrial-strength problems is a different matter. It seems unlikely that we could strengthen Jape's automatic proof mechanisms so as to challenge the power of existing industrial-strength theorem provers and we do not intend to try. Instead, we have begun to experiment with using Jape as a front-end to a more conventional mechanism, in the expectation that its high-quality interaction style will make it easier for an expert user to intervene in a proof at those points—such as choosing an induction schema or a rewrite—where search mechanisms are inefficient or ineffective.

REFERENCES

[1] Gordon, M. and Melham, T. (eds) (1993) *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge.

[2] Paulson, L. C. (1990) The foundation of a generic theorem prover. *J. Automated Reasoning*, **5**, 363–397.

[3] Goguen, J. and Malcolm, G. (1997) *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, MA.

[4] Boyer, R. S. and Moore, J. S. (1990) A theorem prover for a computational logic. *Automated Deduction—CADE-10*, Lecture Notes in Computer Science 449, pp. 1–15. Springer, Berlin.

[5] Heuerding, A., Jäger, G., Schwendimann, S. and Seyfried, M. (1995) Propositional logics on the computer. In *Theorem Proving with Analytic Tableaux and Related Methods, 4th Int. Workshop, TABLEAUX '95*. Lecture Notes in Computer Science 918, pp. 310–323. Springer, Berlin.

[6] Owre, S., Rushby, J. M. and Shankar, N. (1992) PVS: a prototype verification system. *11th International Conf. on Automated Deduction*. Lecture Notes in Computer Science 607, pp. 748–752. Springer, Berlin.

[7] Jones, C. B., Jones, K. D., Lindsay, P. A. and Moore, R. (1991) *mural: A Formal Development Support System*. Springer, Berlin.

[8] Sawamura, H., Minami, T. and Ohashi, K. (1992) EUODHILOS: A general reasoning system for a variety of logics. In *Logic Programming and Automated Reasoning, Int. Conf. LPAR '92*. Lecture Notes in Computer Science 624, pp. 501–503, Springer, Berlin.

[9] Dyckhoff, R. (1987) Implementing a simple proof assistant. In *Workshop on Programming for Logic Teaching, Leeds, July 1987* (program available from Machine Assisted Logic Teaching Project, Computer Science Division, University of St Andrews).

[10] Scheines, R. and Sieg, W. (1993) The Carnegie Mellon proof tutor. In Boettcher, J. V. (ed.), *101 Success Stories of Information Technology in Higher Education: The Joe Wyatt Challenge.* McGraw-Hill, New York.

[11] EPC Ltd (1992) *The B Tool User Manual*. EPC Ltd, Edinburgh.

[12] Dawson, W. M. G. (1990) *A Generic Logic Environment*. PhD thesis, Imperial College, University of London.

[13] Thimbleby, H. W. (1990) *User Interface Design*. ACM Press, New York.

[14] Pace, G. J. (1994) *From Real-Time Specification to Timed Circuit Generalization of Proofs*. PRG internal memorandum, Oxford University.

[15] Théry, L., Bertot, Y. and Kahn, G. (1992) Real theorem provers deserve real user-interfaces. *5th ACM Symp. on Software Development Environments*, pp. 120–129.

[16] Coscoy, Y., Kahn, G. and Théry, L. (1995) Extracting text from proof. In *2nd Int. Conf. on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 902, pp. 109–123. Springer, Berlin.

[17] Bertot, Y., Kahn, G. and Théry, L. (1994) Proof by pointing. In *Symp. on Theoretical Aspects of Computer Software (STACS)*, Sendai (Japan), Lecture Notes in Computer Science 789, pp. 141–160, Springer, Berlin.

[18] Gordon, M., Milner, R. and Wadsworth, C. (1979) *Edinburgh LCF*. Lecture Notes in Computer Science 78. Springer, Berlin.

[19] Jape http://www.dcs.qmw.ac.uk/˜richard/jape and http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin (visited: 23rd April 1999).

[20] Fitch, F. B. (1952) *Symbolic Logic*. Ronald Press, New York.

[21] Bird, R. J. and Wadler, P. (1991) *An Introduction to Functional Programming* (3rd edn). Prentice-Hall International.

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

[23] Reeves, S. and Clarke, M. (1990) *Logic for Computer Science.* Addison-Wesley.

[24] Woodcock, J. and Davies, J. (1996) *Using Z: Specification, Refinement and Proof*. Prentice-Hall International.

[25] Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML*. MIT Press.