

marcelsequent:

A Sequent Calculus for Higher-Order Logic

M. Randall Holmes

Boise State University

November 29, 2005

Notes from a talk given to a logic seminar at the Free University of Brussels (ULB) on November 22, 2005. Some corrections made after the fact.

The author's e-mail is holmes@math.boisestate.edu. Inquiries are welcome. Thanks to the Math Department at Boise State University and the logic group at ULB for making this presentation possible.

I should eventually do something to make the demonstration component of the original talk publicly accessible; watch this space.

The software `marcelsequent` described here is a second try at implementation of foundations of mathematics using variants of Quine's "New Foundations" (1937). The first try resulted in the equational prover Watson (Holmes, 1990's). The major difficulty with Watson, in our view, is that the usual notions of first-order logic had to be implemented on the basis of a too economical set of primitive notions: such implementations are philosophically interesting but do not make for fluent implementation of natural reasoning. In `marcelsequent`, the basic notions of first-order logic are built in (it is an extension of a classical kind of sequent prover).

The system we chose to implement (with modifications) is the classical sequent calculus for a theory called SF (basically, “New Foundations” without any extensionality at all) which is described by Marcel Crabbé (thus `marcelsequent`) in his paper on a semantic proof of the Hauptsatz for SF . Crabbé proves there that his system (which is the core of ours) satisfies cut elimination.

We do not know whether the system we have implemented has such nice properties as cut elimination (though we guess that it probably does). We have added rules governing equality (which does not appear in SF) which enforce the weak extensionality found in Jensen's version NFU of "New Foundations" (NFU (with all the further extensions described here) is known to be consistent if the usual set theory is consistent: the consistency of NF remains an open question). Recall that SF has no extensionality at all. We have added a primitive type-level pair and projection operations, with appropriate sequent rules to govern them: this is effectively equivalent to adding the Axiom of Infinity to NFU .

Why *NFU*?

One might ask why one might want to use *NF* (or *NFU*) as the basis for a theorem proving project anyway. The answer is that these systems have always been seen as formally rather simple; they have been overshadowed by the dominant set theory following Zermelo, especially since *NF* itself is not known to be consistent (*NFU* is known to be consistent (relative to theories in which we mostly have confidence)). Formal simplicity does recommend itself in the logic of a theorem proving system; it may make use easier, and it certainly makes implementation easier!

It is also worth noting that I will as we go along explain just what *NFU* is!

Sequent Calculus

Our apologies to everyone who knows this already (perhaps everyone...)

A *sequent* is a pair of finite sequences of propositions, written $\Gamma \vdash \Delta$. A sequent is *valid* just in case it is the case, no matter what assignments of values are made to variables appearing in the sequent, that if *all* propositions in Γ are true, then *some* proposition in Δ must be true.

The *sequent calculus* is a technique of proof which relies on certain simple transformations which reduce the problem of showing that a sequent is valid to showing that one or two “simpler” sequents are valid.

Trivialities: beginning and ending a sequent proof

To prove that A is a theorem, prove that the sequent $\vdash A$ is valid. (the empty sequence is usually represented by a blank, and a length 1 sequence is notationally confused with its sole term).

Any sequent $B, \Gamma \vdash B, \Delta$ in which the same proposition appears on both sides of the “turnstile” is obviously valid. This is the commonest way to recognize that a sequent is valid.

Matching and pruning

An important way to prove that a sequent is valid is to show that it matches in structure a sequent already shown to be valid: this is legitimate since any assignment of more specific values to variables preserves validity. This means that saving valid sequents as “theorems” adds power to the prover as we work.

If a sequent is valid, adding propositions to it on either the left or the right will not change this. Conversely, omitting propositions that are not needed is a legitimate move (but not safe: it might turn out that the proposition you omit was needed after all!), and matching a sequent proved to be valid with a subsequent of a given sequent proves the given sequent valid.

Sequent rules for negation

$$\frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$$

$$\frac{\Gamma \vdash A, \Delta}{\neg A, \Gamma \vdash \Delta}$$

It is usual, as we see here, for there to be two rules to handle each proposition construction, a left rule and a right rule. Each rule we give has one or two sequents above the line and a sequent below the line: what the rule says is that if the sequents above the line are valid, so is the sequent below the line.

Sequent rules for conjunction

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash \Delta}$$

Note that here we have the first example of a sequent rule with two premises.

Sequent rules for disjunction

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

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

The duality here should be suspiciously familiar.

Sequent rules for implication

$$\frac{A, \Gamma \vdash B, \Delta}{\Gamma \vdash A \rightarrow B, \Delta}$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{A \rightarrow B, \Gamma \vdash \Delta}$$

Most seem to find the second rule harder to get their minds around than the first (which looks just like the deduction theorem). In any event, both follow easily from the rules for disjunction and negation if we define $A \rightarrow B$ as $\neg A \vee B$. The presence of negation can indicate to us why one of the letters crosses the turnstile.

Sequent rules for the other connectives could be written in the same style, but we handle them for the moment by definitional expansion and use of the rules given above.

Looking at the rule set for propositional logic which we have now completed suggests what the proof strategy for sequent calculus should be. In each case, the premises appropriate for a given conclusion are simpler than the conclusion and mechanically derivable from the conclusion. Further, validity of all premises is actually exactly equivalent to validity of the conclusion. So one can repeatedly generate premises appropriate to prove given conclusions until one has sequents consisting only of letters, and validity of such sequents is easily recognized: they are valid if the left and right sequences share a term, and invalid otherwise.

The strategy for proofs in first-order and higher order logic will be the same “goal directed” strategy.

Another interesting point about the proof strategy of sequent calculus is that it breaks proofs into smaller pieces which do not interact after they are separated.

We return to the introduction of rules, now for quantifiers.

Quantifier Rules which Introduce New Constants

In both rules, the constant a must be new in the sense that it cannot occur in Γ or Δ . $P(x)$ represents any formula; $P(a)$ is the result of replacing x with a in $P(x)$ (which may require care in the presence of bound variables, as we will discuss further below).

$$\frac{\Gamma \vdash P(a), \Delta}{\Gamma \vdash (\forall x.P(x)), \Delta}$$

$$\frac{\Gamma, P(a) \vdash \Delta}{\Gamma, (\exists x.P(x)) \vdash \Delta}$$

Quantifier Rules requiring User Input of a Witness

The relation between $P(x)$ and $P(a)$ is just as in the previous slide. t is any object notation at all (it would be chosen by the user in a proof).

$$\frac{P(t), (\forall x.P(x)), \Gamma \vdash \Delta}{(\forall x.P(x)), \Gamma \vdash \Delta}$$

$$\frac{\Gamma \vdash P(t), (\exists x.P(x)), \Delta}{\Gamma \vdash (\exists x.P(x)), \Delta}$$

In this rule, we leave the quantified sentence in the sequent when the rule is applied (which is not the usual practice). This has the advantage that there is never any need to copy a sequent in our system (sometimes this rule needs to be applied more than once to the same quantified sentence in a proof). It also preserves the equivalence of the validity of the premises with the validity of the conclusion.

First-order logic

This completes the basic rule set for first order logic. Though we will not prove it here, this system is complete (any valid sequent can in fact be proved using these rules). It is no longer the case that validity can be mechanically proved or disproved (though there are mechanical procedures which will eventually generate a proof if there is one, there is no way to distinguish in general between a proof search that will terminate in a proof and one which will continue futilely forever).

The goal directed proof style is still recommended, with the additional proviso that the user now needs to look for appropriate witnesses to use universal premises in sequents or verify existential conclusions. If it were not for our modification of the witness rules, we could clearly say that each rule application still makes the sequent simpler.

Higher order logic = set theory

This is a higher order logic prover, not just a first-order logic prover. Higher order logic is characterized by the use of quantifiers on predicates; here we represent predicates as sets.

Thus $P(x_1, \dots, x_n)$ will be represented by $\langle x_1, \dots, x_n \rangle \in P$, when P is a predicate and the x_i 's are its arguments. The advantage of this notation is that one does not need to introduce special variables or quantifiers for higher order quantification.

In a higher order logic, predicates come in various types indicated by the types of their arguments. An object which is not a predicate is said to be of order 0. A predicate whose highest order argument is of order n is said to be of order $n + 1$. If we admit that a pair (or tuple, but tuples can be implemented using pairs) is of the same order as its components, then it turns out that the orders can serve as the only types: the trick is to observe that any argument of a predicate of order n which is of order m lower than $n - 1$ can be raised in order by replacing it with the order $m + 1$ predicate true of that object alone (in terms of sets, a singleton) and this can be repeated until the argument is raised to type $n - 1$.

Orders could have been used ...

The prover could have been, but has not been, written as an implementation of the higher order logic with orders just outlined.

If this system had been implemented, every variable would be adorned with a numerical superscript indicating its order (some or most of these superscripts might be deduced by the system and not required in actual input and output). Certain sentences, such as $x \in x$, would be recognized as ill-formed (whatever is on the right side of a membership symbol must be one type lower than what is on the left side, and x cannot be one order lower than itself!)

The Hall of Mirrors

The difficulty with this is notational annoyance (at least some type superscripts would have to be used) combined with a well-grounded suspicion that the distinction represented by the type indices is somehow redundant: every theorem which can be proved remains a theorem if the orders of all variables in the theorem are raised by a fixed amount, and every object definable in one order has a precise analogue in each higher order obtained by raising the orders of all variables in its definition.

Quine's Solution

Quine proposed that these suspiciously similar orders are in fact *the same*. When type indices are raised, suppose for the sake of argument that truth value, not just provability, is preserved. It will then be seen that the analogous objects of different orders have the same properties exactly, and so may be supposed reasonably to be the same objects.

Notice, though, that when we do this, we do not create any new set definitions. In particular, we do not create the evil $\{x \mid x \notin x\}$. The sets/predicates remain *orderable (typable)* when we allow orders to be freely raised and lowered by uniform amounts, though they cease to be ordered (typed) objects.

We impose the constraint on set definitions that orders can be assigned to all variables in the definitions in a sensible way (summarized in the schemas $x^n \in y^{n+1}$, $x^n = y^n$). Notice that these orders are relative, not absolute (any assignment of course induces shifted assignments as well); it is convenient to allow the use of negative integers as orders as well as the usual natural numbers.

Set definitions with this property are said to be *stratified*.

Specker's Objection and Jensen's Rescue

Quine added to his system the apparently reasonable provision that predicates applying to the same objects are equal (sets with the same elements are the same).

No one has been able to prove the resulting set theory *NF* (New Foundations) consistent or inconsistent. Specker raised the best objection: he proved in 1953 that *NF* disproves the Axiom of Choice.

R. B. Jensen showed in 1969 that Quine's intuition does work. The suspect assumption is not the ability to form sets from all stratified predicates, but the assumption of extensionality. The system *NFU* in which extensionality is restricted to objects with elements is consistent (or, equivalently, extensionality can be restricted to *sets* and it can be admitted that there may be many non-sets with no elements; this preserves the distinction between \emptyset and the other elementless objects). Moreover, it is consistent with Infinity and Choice (and many stronger axioms).

Marcel Crabbé showed that *SF* (*NF* without any extensionality at all) interprets *NFU*.

Back to sequent rules: Membership

In both of these rules, the formula $P(t)$ must be stratified. It turns out that it is only necessary to assign types in set definitions to variables which are bound (in quantifications or set abstracts); free variables and constants may appear with more than one type.

$$\frac{\Gamma \vdash P(t), \Delta}{\Gamma \vdash t \in \{x \mid P(x)\}, \Delta}$$

$$\frac{P(t), \Gamma \vdash \Delta}{t \in \{x \mid P(x)\}, \Gamma \vdash \Delta}$$

The unstratified

In Crabbé's formulation of SF in his cut elimination paper, abstracts $\{x \mid P(x)\}$ must be stratified in order to be well-formed. This is not our approach. Thus, if an abstract happens not to be stratified, the following somewhat more complex rules may be invoked.

$$\frac{P(t), (\exists y. (\forall x. x \in y \leftrightarrow P(x))), \Gamma \vdash \Delta}{t \in \{x \mid P(x)\}, \Gamma \vdash \Delta}$$

$$\frac{\Gamma \vdash P(t), (\exists y. (\forall x. x \in y \leftrightarrow P(x))), \Delta}{\Gamma \vdash t \in \{x \mid P(x)\}, \Delta}$$

Membership of t in an arbitrary abstract $\{x \mid P(x)\}$ is equivalent to $P(t)$ plus the existence of some set with the extension expected of $\{x \mid P(x)\}$; the latter condition can be established, for example, by showing that the condition $P(x)$ is equivalent to a stratified condition.

The cut rule

An important labor-saving device, though it is not theoretically required for proofs in the system as described so far (leaving aside the novelty of the last slide) is the infamous *cut rule*:

$$\frac{\Gamma \vdash A, \Delta \qquad A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$$

This rule is clearly valid. We show that if we had A we could prove that (one of) Δ follows from Γ , and we show that A follows from Γ alone (or else one of Δ follows anyway); from this we see that Γ entails one of Δ .

This rule is not of the same kind as the rules above. The intermediate result A is “pulled out of the air”, as it were.

Equality

The equality predicate does not appear in Crabbé's cut elimination paper. Our basic approach is to *define* equality $x = y$ as $(\forall z. x \in z \leftrightarrow y \in z)$, then to assert in addition the weak extensionality axiom of *NFU*. This is manifest in the following rules.

$$\frac{(\forall z. x \in z \leftrightarrow y \in z), \Gamma \vdash \Delta}{x = y, \Gamma \vdash \Delta}$$

$$\frac{\Gamma \vdash (\forall z. x \in z \leftrightarrow y \in z), (\exists w. w \in x \vee w \in y) \wedge (\forall z. z \in x \leftrightarrow z \in y), \Delta}{\Gamma \vdash x = y, \Delta}$$

We do not know if introducing these rules perturbs Crabbé's cut elimination result; we are quite interested in this question.

Special Equality Rules

There are special equality rules associated with specific term constructions. For example, there is a rule which expands terms with defined operators on either side of an equation. Here is an important specific rule of this kind.

$$\frac{\Gamma \vdash \{x \mid P(x)\} = \{x \mid Q(x)\}, \Delta}{\Gamma \vdash (\forall x. P(x) = Q(x)), \Gamma \vdash \Delta}$$

This rule is unconditional in the default state of the prover, which has the curious effect that equality conditions on unstratified abstracts are the same as those on the proper classes they appear to represent, even if they are in fact urelements. This is demonstrably consistent, but if it is found distracting it can be turned off; it is then required that P and Q be stratified, and the real effect of this rule is to allow

the empty set its expected identity conditions and to save work with other sets.

There are also special equality rules for pairs and projections: equations with a pair can be expanded to the conjunction of equations with its projections, and there are similar if a little less obvious rules for projections.

Rewriting?

A kind of rule which is largely missing is the natural “rewriting” rule for equality. The official definition of equality requires one in effect to supply the context in which the rewriting is to take place as a parameter (we will see this in examples). There is a global rewriting rule built in which applies only to equations involving a free variable (and simply eliminates that variable everywhere).

More general rewriting tools are expected to appear; this was my specialty in Watson, which was oriented toward equational theorem proving (and implemented first-order reasoning only rather awkwardly).

Defining Predicates and Operators

The prover supports definitions of predicates, specific objects, and operations on objects. Expansion of a defined predicate in the lead position on the left or right of a sequent is in effect a sequent rule. Expansion of defined objects or operator terms on the right of the membership symbol takes place automatically (in the hope that it will expand into a set and the usual rule can then be applied); defined objects are expanded on the left of the membership symbol as well if it is clear that the right cannot be expanded. It is noted above that a special equality rule expands defined terms in equations.

The main technical point is that it is necessary to harmonize the ability to introduce defined notions with stratification. In a definition, one

has the option of supplying types to assign to the arguments of a predicate or the arguments and output of an operation; the stratification functions of the prover check that these types are correct in the definition, and if they are it is then possible to stratify set definitions containing these notions. It is also possible to define unstratified predicates and operators, but these can only appear in set definitions with arguments containing no bound variable.

We begin to discuss the implementation.

We describe the notation for objects and propositions under `marcelsequent`.

The notation for propositions follows. A and B are taken to be propositions. T and U are taken to be terms representing objects. n stands for an integer in all notations. x_n is a bound object variable, as we will see under objects.

Propositional (and predicate) variables: P_n is a proposition (propositional variable) and $P_n(T, \dots, U)$ is a proposition (with P_n a predicate variable). $T R_n U$ is syntactical sugar for $P_n(T, U)$.

Primitive Predicates: $T = U$ (equality) and $T \in U$ (membership) are propositions for any objects T and U .

Logical Connectives: $\neg A$ (negation), $A \ \& \ B$ (conjunction), $A \ \vee \ B$ (disjunction), $A \ \rightarrow \ B$ (implication), $A \ \leftarrow \ B$ (converse implication), $A \ \Leftrightarrow \ B$ (biconditional), $A \ \neq \ B$ (exclusive or) are propositions for any A and B . There is a sensible operator precedence so parentheses can often be omitted (and are omitted by display functions as well).

Quantifiers: $(\forall x_n. B)$ and $(\exists x_n. B)$ are universally and existentially quantified sentences for each variable x_n and proposition B . The parentheses here are mandatory.

Defined Predicates: If s is a string of lower case characters (with certain restrictions on the use of \vee and x), $\#s(T, \dots, U)$ is a proposition. $\#s$ here represents a defined predicate.

We now define notations for objects, with the same conventions about letters as above.

Bound Variables: x_n is a bound object variable (usually just called a bound variable).

Free Variables: a_n is a free object variable (usually just called a free variable).

Set Abstracts: $\{x_n | B\}$ is a set abstract, where B is any proposition.

Pairs and Projections: $\langle T, U \rangle$ is an object (an ordered pair), and $p_1(T)$ and $p_2(T)$ are objects (projections) for any objects T and U . Notations of the form $p_1(\langle T, U \rangle)$, $p_2(\langle T, U \rangle)$, and $\langle p_1(T), p_2(T) \rangle$ are automatically reduced to T , U , T , respectively: one can enter these but one will never see them displayed.

Defined Objects and Operations: If s is a string of lower-case characters as above, $*s$ is an object and $*s(T, \dots, U)$ is an object (in the latter case, $*s$ is a defined operator).

Notably lacking is infix notation for defined operations on both objects and predicates, which we fully expect to introduce shortly.

Substitution

The main technical problems at the level of propositions and object notations are the problems of substitution in the presence of variable binding constructions (universally problematic in theorem proving) and the problem of stratification.

Our basic approach to the substitution problem is to invariably rename the bound variable whenever substituting into a variable binding context. We use the notation $A[t/x]$ for the result of replacing the variable x with the term t in the (proposition or object) notation A .

For example, $(\forall x.P)[T/y]$ is defined as

$$(\forall x'.(P[x'/x])[T/y]),$$

where x and y are variables, T is a possibly complex term, and x' is a variable not otherwise found in the context. The overhead for

this kind of treatment is the necessity of keeping a new variable counter (all bound variables are of the form x_i) so that one can always get a new variable. Early versions of the substitution mechanism were quite robust but generated very large variable indices; lately I have been looking for ways to “update” the variable counter more often, but this sometimes leads to bugs (bug reports eagerly accepted if anyone wants to try out the software!)

This is a considerable change of philosophy from Watson, where I not only used a de-Bruijn scheme to handle bound variables, but expected the user to read it (I used “levels” rather than “indices”, counting from the top of the term rather than the bottom, so this is more like standard variable binding notation than the other de Bruijn scheme, but it is still peculiar).

Stratification

The current stratification algorithm is dynamic rather than static (it attempts to build a stratification on the fly while reading a set definition, with a certain amount of backtracking possible). The current algorithm is not complete, though I have yet to counter an example in practice that it cannot handle. A sufficient condition for the stratification algorithm currently in use to work is that every variable be “connected” in a suitable sense to the variable standing for an element of the set being defined.

What is needed from a stratification algorithm is an assignment of types to bound variables which preserves the appropriate displacements of type within atomic statements of membership and equality and between arguments of

defined predicates and between defined operation terms and their arguments. The approach taken is to read through the term making all forced assignments and making an arbitrary assignment to any variable encountered without any cues as to its type; if this fails, attempts are made to read the term in a different order. If a variable is connected by a chain of co-occurrences in atomic environments with the variable defining elements of a set, it will eventually be assigned a correct type.

I do know how to write a complete algorithm to check stratification, but so far this approach seems to work; I'm rather interested in whether any expression likely to appear in practice would make it fail. The advantage of the current algorithm is that I do not have to define as many operations on type assignment lists as data structures.

Sequent Presentation

The presentation of sequents is vertical: a numbered list of propositions is given, followed by a line with \vdash on it, followed by another numbered list of propositions. Above is the left side of the sequent, and below is the right side of the sequent.

One does not need to remember names or commands for all of the rules exhibited above. Most rules are subsumed under the two commands `LeftRule` and `RightRule`, which apply the appropriate rule to the first listed proposition on the left or right.

Rules which do not fall under this rubric are the rules for quantifiers which require the user to supply a witness and the cut rule, which requires the user to supply a proposition. The special equality rules are invoked by name (including the free variable rewriting rule).

Commands `GetLeft n` and `GetRight n` allow one to bring line on the left or right to the front for rule application; `PruneLeft n` and `PruneRight n` are provided if one wants to drop a line on the left or right. `Done` recognizes when the first lines on both sides are the same (so the sequent is valid). `UseThm "name" list1 list2` matches the named theorem against the sub-sequent of the current sequent consisting of listed lines.

All frequently used commands have brief mnemonics as well as their full names.

Proof Management

The management of proofs, which are trees of sequents to be proved, is largely invisible to the user. When application of a rule creates more than one sequent which requires proof, the prover presents one of them. When the proof of one sequent is complete, the prover presents the “first” (leftmost in a suitable order on the tree of sequents, rotating the tree if necessary to bring an unproved sequent to the fore) to the user. There are commands which allow the user to change the order in which sequents are proved, but I have never found them useful (and they are rather awkward at present).

There is a practical reason to sometimes view the sequents which have been proved, which is that quite often a sequent to be proved now may match a sequent already proved in structure. In such a case, one may review all sequents in the proof using a command `ShowAll` and save the sequent with index n as a “theorem” using the command `NameSequent n name`. (theorems are valid sequents here rather than logically valid propositions). A technical issue is that sequents often contain redundant propositions; one may have proved the current sequent in effect already, but the sequent on record as having been proved may have junk in it which obscures the structural match. However, there is a utility which allows one to eliminate all propositions not actually used in the proofs of all proved sequents in the current proof (`AutoPrune`) and this makes the re-use of sequents already proved feasible.

The automatic pruning is implemented by keeping a genealogy as part of the internal structure of the propositions in all sequents; when a proof is completed by finding matching propositions on both sides of the turnstile, only those propositions in the genealogy of the matching propositions turn out to have been needed (of course when a sequent's validity depends on the validity of two sequents, propositions useful for either of these sequents are preserved).

Proof Storage and Output

The prover will print out proofs to standard output or to a file. These outputs are written in a theoretically human-readable style, and they are somewhat structured. The sequents are displayed in a sensible traversal of the proof tree; conclusions appear before premises used to prove them, with numerical references to the “lines” (sequents) from which they are deduced.

The serious issues of structure come in when a sequent has been verified by using a theorem (saved sequent already proved). The default output mode of the prover is to print out the proof of the lemma the first time it is referenced (but not subsequently); it is possible to turn off lemma display. The default storage strategy of the prover is to store a copy of the lemma (tagged with the name of the

proved theorem) when a theorem is proved, to provide security: the lemma might be deleted or overwritten by another theorem with the same name. Theorems can be “locked” (protected from delete or overwrite) and in this case the theorem will be referenced directly without copying it. Within a theorem with nested uses of lemmas, the proliferation of copies of theorems is avoided by looking for a copy of a lemma at the same level as a lemma being copied before allowing it to make an internal copy of this lemma (when a lemma is needed, one looks for it inside the theorem whose proof called it, then in each environment above that, then at the top level if the lemma is “locked”).

It would be an interesting challenge to write a separate program that would read and check the output of this prover; for the moment this output is only for human consumption (it is useful to examine it locally but the global effect is wearying).

Current Research

I am currently proving theorems from Landau's little analysis book (also used as a target by the Automath group). A considerable detour into set theory was occasioned by my insistence on proving Axiom 4 (successors equal implies numbers equal) using the Frege definition of natural numbers; this required the development of some rather elaborate proofs in set theory.

I find that the proofs I have written earlier are extremely inefficient (for one thing, many of my early proofs were cut-free; I posted on the web a cut-free proof of Cantor's theorem!). With experience, and especially with practice in the intelligent use of definitions, proof of preliminary theorems, and recognition when the current sequent is probably a consequence of a sequent proved earlier in the

same proof, efficiency improves. This is still slow and sometimes boring: computer programming rather than conventional proof.

I have used the prover to teach first-order logic. It seems suitable for this purpose; the students who used it (there were only two in the class) said that it actually did help to give them some understanding of formal proof and of the informal strategies that we all learn to use in mathematical proofs, which are closely modelled by certain sequent rules (the quantifier rules, in particular).

Future Directions for Research

The prover requires some automation for greater ease of use. There are standard automatic strategies which are implemented in other similar systems, which I have made some preliminary attempts to simulate.

The ability to automatically search for matches in structure with the current sequent of sequents already proved would be very useful if it were reasonably fast. This would immediately mean that one did not have to supply the line numbers of the subsequent to be matched (and in the correct order!) to the `UseThm` command. This is also a well-understood issue in theorem proving; it just requires me to do more programming.

The addition of rewriting techniques (the general ability to apply equational theorems, and

also universally quantified equations to rewrite terms, as well as using biconditionals to rewrite propositions) would increase ease of use. There are some original ideas of mine for use of recursively chained rewrite rules to implement a kind of programming, which I will try to transfer from Watson to this environment. Another feature of Watson which should be emulated here is the direct implementation of the special properties of strongly Cantorian sets, which allow a more relaxed approach to stratification (but with more headaches for the implementor!)

An immediate need as I go into proofs in arithmetic is the introduction of infix defined predicates and operations.

Two general philosophical issues are or should be addressed by this kind of research:

the first is the serviceability of the particular approach to mathematical foundations that is used (here a set theory with stratified comprehension using classical logic).

the second, more general issue, is the possibility of bringing automated reasoning (and here we really mean user controlled automated reasoning: proof construction and checking managed by a mathematician with some automated tools to increase efficiency) to the point where it really is useful to a mathematician in his work.

I have nothing dramatic to claim about either of these as yet.