Artificial Intelligence Project--RLE and MIT Computation Center

Memo 21--The Proofchecker

by

Paul Abrahams

January 25, 1961

ABSTRACT

The Proofchecker is a heuristically oriented computer program for checking mathematical proofs, with the checking of textbook proofs as its ultimate goal. It constructs, from each proof step given to it, a corresponding sequence of formal steps, if possible. It records the current state of the proof in the form of what it is sufficient to prove. There are two logical rules of inference: modus ponens and insertion (if it is sufficient to prove B, and A is a theorem, then it is sufficient to prove A implies B). The permissible formal steps include these rules of inference as well as provision for handling definitions, lemmas, calculations, and reversion to previous states. As of now, most of the formalisms are programmed and partially debugged, but the heuristic aspects have yet to be programmed.

TABLE OF CONTENTS

PURPOSE AND PHILOSOPHY

A proofchecker is a computer program which checks mathematical proofs to see if they are correct. A mathematical proof is a symbolic expression, and so can be used as input data for a computer program written in a symbol-manipulating language. Now if a proposed proof has been sufficiently formalized, it can be checked by a simple set of rules; but if the proof is of the sort that appears in most mathematics textbooks, the checking procedure must be extremely elaborate and even then cannot be guaranteed to work all the time. Checking textbook proofs is the ultimate goal of the Proofchecker project, even though it probably will not be reached. This goal will set the direction of these efforts; how far they go remains to be seen. Some programs have already been checked out; many remain to be written. As of this writing, two elementary formal proofs have been checked.

A number of the ideas presented in this paper arose from suggestions made by John McCarthy; however, the views expressed here are not necessarily in agreement with McCarthy's.

There are two difficulties that arise in devising a procedure for checking textbook proofs. First, the proofs are written in English; second, they are ambiguous, informal, and full of gaps. I am not at present planning to deal with the first problem; though it is interesting, it is irrelevant to the main purpose of the Proofchecker project. Therefore, I will assume that the proofs which I would like the Proofchecker to examine will be presented in a formalized and stylized language, but one which makes no attempts to clarify the logical defects of the English original from which they are taken. The Proofchecker will have to correct these defects and check the resulting formal proof at the same time.

Essentially, the Proofchecker works by taking each step that it is given and generating from it a sequence of formal steps. Thus the proof given to it really serves as a set of hints on how to construct a formal proof of the theorem at hand. The Proofchecker must be something of a proof inventor, for it must among other things be able to handle steps labelled as "obvious". As the capabilities of the Proofchecker increase, it will get better and better at proving new theorems. In fact, we may view the job of the Proofchecker to be to invent proofs, given a set of hints: in the two extreme cases, the hints are either nonexistent or are all the steps of a formalized proof.

We may consider the problem from the viewpoint of checking methods for generating proofs, i.e., seeing if a proposed method of proof generation generates a proof of a desired theorem. The two viewpoints are equivalent; for a technique which will utilize hints is simply a proof generator with an input. Further, the Proofchecker must be able to make use of information on that the members of a certain class of symbolic expressions are legitimate proofs. Such a piece of information is a metatheorem. In order for any proof-checking system to utilize metatheorems, they must be built into its structure. In the Proofchecker, metatheorems may be added without changing the overall pattern of operation; essentially, they are added to the list of permissible proof steps, so thay they act like additional rules of inference (which, indeed, they are).

The Heuristic Approach and Related Work

The Proofchecker will make extensive use of heuristic methods and techniques. This is necessary because a straightforward search through the space of connections between successive hints would involve an impossibly larg amount of computation. One scheme which I hope will be useful here will be the "General Problem Solving System" of Newell and Simon. This system has already been used in solving problems in trigonometry, elementary algebra, and logic. The chief feature of the system is its separation of problem content from problem-solving technique. It makes use of two principal heuristics: means-ends analysis, which involves knowing what methods of attack are appropriate to particular situations, and planning, which involves consideration of an abstraction of the problem at hand. Closely related to the planning heuristic is the heuristic of models as used by Gelernter. This heuristic, for the case of plane goemetry, involves drawing figures and checking them to see what relationships hold between various line segments, angles, etc.

There is a resemblance between the role of hints in the Proofchecker and the role of advice in McCarthy's Advice Taker scheme. In each case, we want a machine to make use of pieces of information given to it from an external source in order to solve a problem. In the case of the Proofchecker, the structure of the "advice" makes it much easier to determine how to use it.

Problem Domains

The Proofchecker will not be tied to any particular area of mathematics, although I expect to work mainly on abstract algebra at first. The Proofchecker, like the mathematician, is intended to be a general-purpose device. For a given problem domain, it will have relevant theorems, heuristics, and model-making mechanisms stored in its memory. Actually, it would be useful if the Proofchecker could be aware of all the theorems it had ever proved, though it might have less ready access to those from different domains. The difficulty here is that there simply isn't enough room in the computer, as things stand now, for the additional theorems.

One possible area of application, though admittedly somewhat futuristic, is to the checkout of large systems. The present way that one checks such a system to see if it works is to present it with a wide variety of inputs representing both typical and extreme cases, and see if the proper output results. However, it often happens that this method simply doesn't work; the system fails under some unanticipated combination of circumstances which never occurred in the test cases. One way to check out such a system, however, is to prove that it works. Such a proof, of course, would be impossible for a human to construct for a system of even mild complexity; but a machine might be able to construct such a proof, using as hints the description of the functions and intended mode of operations of various components and subcomponents of the system. And in a proof attempt, one might come up with a counterexample and possibly even suggest a remedy.

ORGANIZATION OF THE PROOFCHECKER

The Proofchecker is organized into two constellations of programs: Method and Verify. In addition to the data internal to these programs, there is a list of theorems, a list of definitions, and a list of legitimate proof steps. The theorem list and definition list may be augmented with time. Verify will accept a proposed formal proof step and determine if it is correct; if the step is correct, Verify will do the necessary updating of the state of the universe. Method will determine what the next step should be; it will make use of the hints given as input in finding this step. The steps which Verify will accept must be in standardized form; thus, all of the heuristics are in the province of Method, though Verify will handle updating in such a way as to make the use of heuristics easy.

The current state of the proof is recorded in the form of what it is sufficient to prove. We will call this expression the sufficiency, i.e., at each stage it is sufficient to prove the sufficiency. Then initially, the sufficiency is the theorem we wish to prove; and when the sufficiency has been reduced to T (truth), we know we have proved the theorem. We may record the sufficiency at various stages of the proof and refer back to it later, so that if we conduct an unsuccessful subproblem exploration, we can return to an earlier subproblem without having to repeat the initial steps of the proof.

This particular form of recording the current situation is convenient because of the wide variety of logical structures which may appear in a theorem or in its proof. To use a list of hypotheses and conclusions would be awkward, for instance, in handling an if-and-only-if theorem, unless it were to be broken down into two separate theorems. Yet such theorems are often proved by a sequence of biconditionals, and this natural sequence can be retained in the sufficiency formulation of the proof. Furthermore, this formulation makes backwards proofs quite easy to handle, and they are perhaps the most natural kind. At the same time, it is possible by using certain logical tautologies to make forward proofs; and the machinery for handling such proofs can be built into Method in such a way that the user need not be aware of the few complications that this entails.

The Theorem List

The Proofchecker is endowed with an initial set of theorems which are placed on a theorem list, and each newly proved theorem is added to this list. Lemmas proved during the course of proving a theorem are also added to this list. Whenever a theorem is referred to in a proof, it is brought to the head of the theorem list, so that frequently used theorems are easily accessible.

Each theorem consists of a name, a list of variables, and a form. Whenever the theorem is used in a proof, it is referred to by name; the variables designate those elements in the form which may be substituted for. We may think of the variables as designating universal quantifiers which implicitly precede the form so as to convert it to a theorem in the ordinary sense. The form is the actual statement of the theorem; no inherent restrictions are placed upon its format, though for any problem domain, the format will be specified.

The Definition List

The Proofchecker also has available to it a list of definitions. A proof may be interrupted at any point to make a new definition. A definition is simply an abbreviation, with provision for substituting for certain variables within it. Each definition consists of two names, a list of variables, and a form. The reference name is used when we refer to the definition in a proof step, and the internal name is the one actually recorded in the sufficiency and in any theorems on the theorem list in which the definition is used. The form and variables operate in the same way as the form and variables in a theorem. Definitions, like theorems, are arranged so that recently used definitions appear at the head the list.

The Need for Two Names

We need two names in a definition because otherwise we may accidentally assign to a definition a reference name which already has a meaning; and then by making use of the definition, we may make a false step which will pass undetected. An example will illustrate the danger. Suppose now that we have only one name associated with a definition. Let us define AND with variables X and Y as (NOT, (OR, X,Y)). Thus AND is the name, (X, Y) is the list of variables, and (NOT, (OR, X, Y)) is the form. Now suppose that we have as theorem (NOT, (OR, T, T)). Then the theorem is the original sufficiency. By using the definition, we may then get as sufficiency (AND, T, T); and by other means we may reduce this to T. Thus we would have somehow prevented from entering the AND into the sufficiency in the above example, we would not have been able to carry out the proof. If an internal name, say G0005, had been entered in place of AND, we would have had as sufficiency (G0005, T, T), which would be unprovable since G0005 has no known properties.

In general, we use the reference name when we refer to a definition in a proof, but the internal name is the one actually recorded in the sufficiency. The Verify program handles the substitution of the internal name for the reference name automatically. The internal name is a newly generated symbol which can be guaranteed not to have been used for anything before; the program which creates such symbols in LISP is called Gensym. The symbols it generates are of the form Gxxxx, where the x's are decimal digits.

Redefinition

It is possible to redefine a term even though it has previously been defined. When we do this, however, we lost the ability to refer to the previous definition. This is because when we search the definition list for the reference name, we will always come to the later definition first; and we can access a definition in a proof step only through its reference name. This is as one would like it to be; for it permits us to make temporary definitions of terms without precluding the later use of these terms for other purposes. The internal names of the successive definitions will remain unique, so there is no danger of confusion.

Preservation of Subproblems

Often, in a heuristic proof, we will make steps which, while correct, don't help to prove the theorem. Since such steps may actually lead us to a situation where the sufficiency is a falsity (and hence unprovable), we must have a way to go back to previous sufficiencies. At the same time, we don't want to keep all previous sufficiencies, because it may take up too much room. Therefore, there are two ways in which we may save past sufficiencies.

Reverting

When we make a step, we may request that the last sufficiency be preserved. We may gain access to it by reverting one or more times, depending on how far along we have moved subsequently. There is a parameter, set by Method, which tells Verify whether or not to preserve the most recent sufficiency.

The List of Preserved Subproblems

The reverting procedure is not adequate when we wish to back up several steps, move off in another direction, and then, after the new exploration proves feasible, return to where we started. It is inadequate because the starting point lies on a different branch of the exploration tree. In order to handle this, there is provision for naming the current sufficiency at any time and storing it on a list of subproblems to be preserved. At any point, we may reattack it if we wish. We may also remove any sufficiency from the list of preserved subproblems. The items on the list of preserved subproblems are then accessed by their names. When the theorem at hand has been proved, the list of preserved subproblems is automatically cleared.

Proof Steps and Their Programs

Each type of proof step is represented by a program which takes the parameters of the step as input, updates whatever needs to be updated, and gives as output T or F, depending on whether or not the proof step is correct. Metatheorems may be incorporated into the system if they are set up as proof steps; ordinarily, they will take the sufficiency as as input, test it or a subexpression of it for a particular property, and then transform it in some way. For instance, a logical simplification program would correspond to a metatheorem; such a program would reduce the sufficiency to a simpler but logically equivalent form. The actual metatheorem would be that the result of simplification is in every case logically equivalent to the original form.

The Verify Program

The Verify program, as we have pointed out, checks each successive step of the proof, and performs various kinds of updating. Before examining any steps, it assigns internal names to all the variables so as to avoid conflicts with previously assigned meanings. Thereafter, when variables are referred to by their names, Verify replaces these names with the internal names in the steps of the proof. Thus Method need not even be aware of the existence of these internal names. Verify also sets the sufficiency initially to be the theorem we wish to prove; when the theorem has been proved, Verify places it at the head of the theorem list. Verify repeatedly calls on Method to produce a proof step. Each proof step is indicated by the name of a program and a set of arguments for that program. Verify checks that the program name is on the list of legitimate proof steps, and, if so, operates the program. The result indicates whether or not the step was correct; the proof step program handles the updating that needs to be done.

The Method Program

Method is called upon repeatedly by Verify to furnish proof steps. While Verify will probably remain unchanged as the Proofchecker is modified, Method will be highly subject to change. Method uses heuristic methods, in general, to determine what the next proof step should be. At present, Method simply feeds the input directly to Verify, but this is only in order to check out Verify. As work progresses, Method will become more and more elaborate.

One possible way of elaborating Method is to do so hierarchically. Rather than rewrite Method to account for new improvements, a new Method would be written which uses the old one as a subprogram. This technique is well adapted to heuristic programs.

Role of Common Subexpressions

In successive sufficiencies, there will often be large subexpressions which are carried along intact and are therefore identical. The Proofchecker has been organized so as to have both sufficiencies refer to the same copy of such a subexpression. In general, copying is avoided as much as possible, saving storage and time.

This might appear to be merely a practical programming device, but it has an interesting counterpart in textbook mathematics. Essentially, common subexpressions act like pronouns; they are useful in the same way that it is useful in a book to write an equation, number it, and later on refer to it by number. If we did not use the numbering technique or its equivalent, we would have to write the equation out in full whenever we wanted to refer to it.

LOGICAL FOUNDATIONS OF THE PROOFCHECKER

Formal Description

We will start with a class of pairs (T_i, Θ_i) for $i = 1, 2, \ldots$, which we shall call *initial theorems*. Each T_i is a symbolic expression; each Θ_i is a set of atomic symbols $\alpha_{i1}, \alpha_{i2}, \ldots, \alpha_{in_i}$. We refer to T_i as the *form* of the theorem and to the α_{ij} as the *variables* of the theorem. If $\mu_1, \mu_2, \ldots, \mu_k$ are symbolic expressions, then the result of substituting μ_1 for α_{k1}, μ_2 for α_{k2}, $\ldots$, μ_k for α_{kn_k} in T_k is a *theorem instance*. Note that the T_i are theorem instances by the identity substitution.

A *propertyless symbol* is an atomic symbol which is used solely as a variable in a theorem, and for no other purpose. For instance, AND and PNAME would not be propertyless symbols. In LISP, the function GENSYM is used to create these symbols.

Let S_0 be a symbolic expression, and suppose that there exists a sequence of sentences $S_1, S_2, \ldots, S_n$ such that $S_{i-1} \leftarrow S_i$ ($i = 1, 2, \ldots, n$) and S_n is the symbol "T". (The meaning of "$\leftarrow$" will be explained shortly; T represents truth.) Let $\beta_1, \beta_2, \ldots, \beta_m$ be a set of propertyless symbols, and denote this set by η. Then the pair (S_0, η) is a theorem. Furthermore, the initial theorems are theorems. The class of theorems is thus formed recursively, starting with the initial theorems.

We denote "S is a theorem instance" by $\vdash S$, and "it is sufficient to prove S" by $\dashv S$. The logical system is based on two rules of inference:

1. If $\vdash T$ and $\dashv S$, then $\dashv T \supset S$. (Insertion rule)
2. If $\vdash S \supset S'$ and $\dashv S'$, then $\dashv S$. (Modus ponens)

Intuitively, $S \leftarrow S'$ means $\dashv S \supset \dashv S'$. Formally we say $S \leftarrow S'$ if and only if

1. S' is of the form $T \supset S$ and $\vdash T$, or
2. $\vdash S' \supset S$, or
3. S' is obtained from S by substituting for some subexpression of S the equivalent of that subexpression, provided that the subexpression is not part of a quoted subexpression.

Equivalent subexpressions are of two types:

1. A defined term is equivalent to that which it defines.
2. An expression which represents the result of a calculation is equivalent to the result itself. The calculation is performed by evaluating the expression and then replacing it by the quoted result of the evaluation.

For instance,

```
(SUBST, (QUOTE, (A, B)), (QUOTE, Q), (QUOTE, (CONS, P, Q)))
```

would be equivalent to

```
(QUOTE, (CONS, (P, (A, B))).
```

Here SUBST is a LISP function which substitutes its first argument for its second argument in its third argument.

The definition of $\leftarrow$ is changed when metatheorems are added to the system. Therefore, the above description corresponds to the system as it now stands, but is not permanent.

Role of Initial Theorems

The initial theorems differ from axioms as we think of them in that for most problem domains there is no effort to make them non-redundant. By using a system with a large initial endowment, interesting results are reached more quickly. It is true that all of mathematics may be constructed from lower predicate calculus with identity; but the working mathematician does not ordinarily make use of this fact, even though he is aware of it. This is as true for topologists and algebraicists as it is for hydrodynamicists and numerical analysts, even though the former may occasionally give

This page is missing from the original document.

The Dangers of the Calculation Rule

In permitting the calculation rule, we have let the wolf in the door. We would like the Proofchecker, when given a proof, to be guaranteed to give us a verdict. However, when we introduce the calculation rule, we can no longer guarantee this, except at great cost. For when we select some subexpression of the sufficiency and proceed to calculate its value, the calculation may not terminate; furthermore, it is in general impossible to protest the subexpression to see whether or not the calculation will run wild in some sense and destroy the Proofchecker program or modify its contents. (A fantastically clever learning program, for instance, moght learn to cheat by modifying the parameters of the Proofchecker so as to force it to accept false steps!) These problems may be solved by running all calculation steps on an interpretive basis, but this requires a private and quite elaborate interpreter for the Proofchecker; furthermore, it would be unbearably time-consuming because machine-language programs as well as higher-level-language programs would have to be interpreted.

This difficulty is handled by placing all calculations under the control of a program called Errorset. Errorset causes most detected errors and non-terminating calculations to return to control to the Proofchecker. It will not catch all such errors, so this logical loophole does remain. However, defective proofs which do get by Errorset will ordinarily cause the Proofchecker to stop rather than to give a false answer.

THE POSSIBLE STEPS IN THE PROOFCHECKER

When Verify receives a proof step from Method, it checks to see if the step is of one of the recognized types, and has the appropriate number of parameters with it. If not, the step is erroneous. If these criteria are met, the name of the step is taken as the name of a function, and the parameters of the step as arguments of that function. For each type of step, the corresponding function checks the step to see if it is correct and, if so, does the necessary updating. The function is called by Verify.

The types of proof steps have been selected for their usefulness as primitives or building-blocks in constructing the types of steps we are really interested in. They are not all useful by themselves. However, when used in groups and combined with certain specific theorems, they become quite powerful.

The following steps are available at present:

1. Insert

   This step has as parameters the name of a theorem and a list of substitutions. The corresponding function finds the theorem on the theorem list, performs the i indicated substitutions, and then applies the insertion rule to the resulting theorem instance and the current sufficiency. If the named theorem is not on the theorem list, or if the variables in the substitution list do not correspond to the variables of the theorem, the step is rejected.

2. Modus

   This step has as parameter the name of a theorem. The corresponding function finds the theorem on the theorem list, determines the proper substitutions for the variables of the theorem, and then applies the rule of reverse modus ponens to the resulting theorem and the current sufficiency. The step will be rejected if the named theorem is not on the theorem list, or if the theorem is not of the form (IMPLIES, antecedent, consequent), or if the sufficiency is not a substitution instance of the consequent with respect to the variables of the theorem.

3. Calculate

This step has as argument a list of A's and D's which indicate the subexpression of the sufficiency whose value is to be calculated. If the list has no elements, the subexpression is the entire expression. The step replaces the subexpression ℓ by list [QUOTE; eval[ℓ]]. The step will be rejected if the list reaches an atomic symbol or a quoted expression before it is exhausted.

4. Lemma

This step has as arguments a name, a list of variables, and a form. In this step, the Proofchecker is used recursively to check the proof of a lemma, while temporarily neglecting the main theorem. If the lemma is proven, it is added to the theorem list; in any case, control is returned to the proof of the main theorem when either the lemma is proven or the Proofchecker is told to give up on it.

5. Makedef

This step has as arguments a name, a list of variables, and a form. The corresponding function first assigns an internal name to be the equivalent of the reference name. It then places the list which contains both names, the variables, and the form at the beginning of the definition list.

6. Usedef

This step has as parameters a name and a list of A's and D's. The A's and D's are used to indicate a subexpression of the sufficiency; the name is the reference name of a definition. The corresponding function gets the definition from the definition list and checks the subexpression to see if it is a substitution instance of the form of the definition. If so, it replaces the subexpression by a list whose first element is the internal name of the definition. The remaining elements have the property that if they are matched sequentially with the variables of the definition and substituted for these variables in the form, the result is the original subexpression.

7. Undefine

This step has as parameter a list of A's and D's which indicate a subexpression of the sufficiency. The first element of this subexpression is taken as the internal name of a definition, and the definition list is searched for that name. A substitution list is formed by matching the successive elements of the subexpression with the variables of the definition, and then the indicated substitutions are performed on the form of the definition. Finally, the subexpression is replaced by the result of this substitution. The step is rejected if the named definition is not on the definition list, or if the number of elements following the internal name of the definition in the subexpression is not equal to the number of variables of the subexpression.

8. Nameprob

This step has as parameter a reference name. It causes a pair whose first element is the name and whose second element is the current sufficiency to be placed on the list of preserved subproblems. The current sufficiency will be able to be restored if necessary at a later time.

9. Reattack

This step has as parameter the name of a subproblem on the list of preserved subproblems. It causes the current sufficiency to be set to the sufficiency paired with the given name on the list of preserved subproblems. The step is rejected if the name does not correspond to a name on the list of preserved subproblems.

10. Killprob

This step has as parameter the name of a problem on the list of preserved subproblems. It causes that subproblem to be deleted from the list.

11. Donothing

This step has no parameters, and has no effect whatsoever.

12. Revert

This step has no parameters. It causes us to take the second most recent sufficiency as the current sufficiency. If the current sufficiency is the theorem we wish to prove, the step is rejected. Unless the current sufficiency is saved by a nameprob, a revert will cause it to be lost.

13. Qed

This step has no parameters. If the current sufficiency is "T", we place the theorem at the head of the theorem list, and Verify will return with an answer of T. If the sufficiency is not T, the step is rejected.

CURRENT STATUS AND FUTURE PLANS

As of this writing, the Verify program is in the process of debugging, and the Method program is in its most rudimentary form; Method merely takes a list of steps, peels them off one by one, and passes them to Verify, while printing out useful information. The program has been coded entirely in LISP, which seems to be well suited to the application. In particular, its handling of common subexpressions without unnecessary duplication will probably ease the storage difficulties considerably when the problems get more complicated and the theorem and definition lists get longer. I have already written all the functions which check the different steps, though some of them are in need of revision.

After Verify and its satellite functions have been checked out, I plan to work on improving Method, at the same time applying it to abstract algebra. My first step will be to incorporate macro-steps which will combine several individual steps along with specific theorems. Next I shall build in machinery which allows the steps to be somewhat ambiguously given--for instance, the substitutions might be omitted (as is often done in the citation of a theorem in a textbook proof). After that, I shall investigate various learning mechanisms which might be incorporated. However, it is difficult to anticipate at this point just what course the project will follow; the plans at each stage will be influenced strongly by the results of the preceding stage.