Automatic Generation of Epsilon-Delta Proofs of Continuity

Michael Beeson

Department of Mathematics and Computer Science
San Jose State University
San Jose, California 95192, USA
beeson@mathcs.sjsu.edu

Abstract. As part of a project on automatic generation of proofs involving both logic and computation, we have automated the production of some proofs involving epsilon-delta arguments. These proofs involve two or three quantifiers on the logical side, and on the computational side, they involve algebra, trigonometry, and some calculus. At the border of logic and computation, they involve several types of arguments involving inequalities, including transitivity chaining and several types of bounding arguments, in which bounds are sought that do not depend on certain variables. Control mechanisms have been developed for intermixing logical deduction steps with computational steps and with inequality reasoning. Problems discussed here as examples involve the continuity and uniform continuity of various specific functions.

1 Context of this Research

Mathematics consists of logic and computation, interwoven in tapestries of proofs. "Logic" is represented by the manipulation of phrases (or symbols) such as *for all x*, *there exists an x*, *implies*, etc. "Computation" refers to chains of formulas progressing towards an "answer", such as one makes when evaluating an integral or solving an equation. Typically computational steps move "forwards" (from the known facts further facts are derived) and logical steps move "backwards" (from the goal towards the hypothesis, as in *it would suffice to prove*. The mixture of logic and computation gives mathematics a rich structure that has not yet been captured, either in the formal systems of logic, or in computer programs. The research reported on here is part of a larger research program to do just that: capture and computerize mathematics.

At present, there exist computer programs that can do mathematical computations, such as *Mathematica*, *Maple*, and *Macsyma*. These programs, however, do not keep track of the logical conditions needed to make computations legal, and can easily be made to produce incorrect results.[1]

[1] Just to give one example: Start with the equation $a = 0$. Divide both sides by a. In all the three systems mentioned, you can get $1 = 0$ since the system thinks $a/a = 1$ and $0/a = 0$. Many other examples have been given in the literature [1],[15].

On the other hand, there are theorem-proving programs such as *Otter* [13](and others too numerous to mention) which perform logical reasoning. These programs are quite limited in their computational abilities, although some of them can perform rewrites using a specified set of equations. The input consists of a file containing axioms, and a goal, usually expressed in clausal form. The program contains no mathematical knowledge except that supplied in the axioms; it only "knows" the laws of logic. Proof-search, such as these programs perform, is not what is meant here by "computation"; although of course in some sense the execution of any algorithm is computation, what we call computation here is more like what an ordinary mathematician means by the word, a sequence of more or less purposeful-appearing steps, with little or no trial-and-error involved.

This paper will present a framework for integrating logic and computation, and report on experiments with the implementation of that framework. The implementation is contained in two computer programs, *Mathpert* and *Weierstrass*. The former has been reported on elsewhere in detail [1–3]: it contains implementations of over two thousand mathematical operations, together with logical apparatus to keep track of assumptions that may be required or generated by those operations. *Mathpert* (as in "Math Expert") uses these operations to provide a computerized environment for learning algebra, trigonometry, and calculus. It is the second program, *Weierstrass*, which is used in the research reported here. *Weierstrass* began life as a C-language implementation of a "backwards Gentzen" theorem prover, whose Prolog progenitor was described in [4]. To this backbone has been added a set of control structures, or if you like, implementations of special inference rules, to allow the proper meshing of logical and computational steps. These control structures operate at the top level of *Weierstrass*, but the computational steps themselves can use, in principle, anything that has been implemented in *Mathpert*, which is all of high-school algebra, trigonometry, and one-variable calculus including limits, differentiation, and integration, as well as a good many techniques for rewriting inequalities, and a few advanced algorithms, such as the Coste-Roy algorithm [10], based on Sturm's theorem, for determining whether polynomials have roots in given intervals. The implementations of these operations in *Mathpert* are logically correct, so that they can be used in *Weierstrass* without the risk of inconsistency that would accompany the similar use of *Mathematica*, *Maple*, or *Macsyma*.[2]

The plan of the paper is to describe the control structures used in *Weierstrass*, and then to illustrate their use by giving several examples of proofs produced by *Weierstrass*.

No program has ever before produced an epsilon-delta proof of the continuity of any specific non-linear function. For example, Bledsoe and his student Hines have used the prover STRIVE [6] to prove that the sum of continuous functions is continuous, and that linear functions are continuous, but it lacks the

[2] *Weierstrass* is not an interactive program for producing a proof step-by-step. The user supplies axioms and a goal, and *Weierstrass* finds a proof if it can. However, the techniques discussed in this paper could easily be used to build an interactive program.

computational ability to carry out the proofs given here. *Analytica* [9] is linked to the computational facilities of *Mathematica*, but essentially deals only with quantifier-free proofs. The Boyer-Moore prover [7] has proved some impressive theorems of number theory, including the law of quadratic reciprocity, but like *Analytica*, works best with free-variable proofs, and cannot find epsilon-delta proofs. Other directly relevant work includes [11], [16], [8]. A lengthier discussion of these and other projects is precluded by the length limit on papers in this volume.

2 Nature of the Proofs Produced by *Weierstrass*

To avoid confusion, some discussion of nature and purpose of computer-generated proofs is necessary. *Weierstrass* produces (internally) a proof-object, which can be displayed or saved in more than one form. The intention is, to produce a proof that can be read and checked for correctness by a human mathematician; the standard to be met is "peer review", just as for journal publication. By contrast: the purpose of *Weierstrass* is *not* to produce formal proofs in a specified formal system.

Nevertheless, the program does produce a formal proof object. This object can be regarded as a proof in a formal system, but some of the steps in the proof involve more computation than is normal in formal systems. In traditional logical systems, checking that an inference step is correct according to the system is a simple syntactic comparison to the rule used at that inference. In *Weierstrass*, an inference step might involve the use of a mathematical algorithm, even for example a complicated algorithm based on Sturm sequences, so that the correctness of the step might not be obvious by inspection. If the algorithms used have been correctly implemented, then the proofs are formally correct.[3] But certainly we do not have proofs (formal or informal) of the correctness of the specific programs implementing more than two thousand mathematical operations available in *Weierstrass*, so we must rely on human verifications that the actual output of *Weierstrass* is an acceptable proof.

There is a different and more interesting reason why the proofs produced by *Weierstrass* should be judged by the "peer review" standard. Namely, the algorithms it uses represent theorems of all different "levels". For instance, one might protest that the "right way" to prove continuity theorems such as are considered here, is to prove general theorems about the continuity of compositions of functions, etc., and quote them. Indeed, that is the way *Mathpert* proceeds internally, e.g. when it has to verify the continuity of an integrand in evaluating a definite integral. But when trying to prove the continuity of x^3, we don't want a one-line proof based on the continuity of polynomials. To state the point another

[3] This is really no different than in a purely logical theorem prover: one does not demand that one should prove the correctness of the theorem-prover before accepting its output as a proof. Otherwise, one would be involved either in an infinite regress or a reflexive situation where a prover would prove its own correctness. And would we believe it then?

way: People interested in foundations of mathematics try to order the concepts and theorems of mathematics so that each one depends only on earlier results, with everything resting on a few self-evident axioms. The body of mathematics available to *Weierstrass* has *not* been so ordered; rather, there is simply a "web" of known facts and algorithms, any of which can be used as required. We see one example of this in the third example presented in this paper: *Weierstrass* doesn't mind using the mean value theorem to prove the continuity of $f(x) = \sqrt{x}$, even though the result seems "simpler" than the tool.

Some of the descendants of this program may become the "mathematician's assistant" of the future, a tool to which a practicing mathematician may turn when a stubborn inequality needs to be proved. The standard of "peer review" is the appropriate one for this type of program. Other descendants of this program may be used in a project to construct a database of formalized mathematics, similar to the Mizar project of today. In that case the questions of formal correctness proofs for the computational steps, and of ordering the results and deriving them from axioms, will eventually arise, but this will be a difficult enterprise.

3 The Logical Backbone of *Weierstrass*

The core of the logical apparatus in *Weierstrass* is a "backwards Gentzen prover". I shall now explain what is meant by that phrase. A *Gentzen sequent*, or just *sequent*, is an expression of the form $A_1, \ldots, A_n \Rightarrow B$, where the A_i and B are logical formulae (in some language).[4] The right side of the sequent symbol $\Rightarrow$ is called the *succedent* and the left side is the *antecedent*. The semantic interpretation of a sequent is that the conjunction of the A_i implies B. We allow **true** and **false** as atomic propositions. The *sequent calculus* is a set of inference rules for deducing one sequent from another. One standard reference for these rules is Kleene's book [12]. In that reference, an empty list can appear in the succedent; we use **false** for this purpose, so that a formula always stands in the succedent.

When the sequent calculus is implemented in *Weierstrass*, most of the antecedent is kept in a list of *assumptions*, which could in principle be quite long. If axioms of induction are used, for example, new instances of the axioms can be generated as required; axioms belong in the antecedent, since the sequent calculus is for producing purely logical proofs. The only parts of the antecedent that will be "passed" as function parameters are assumptions that are made temporarily during the argument. For example, to prove an implication $A \rightarrow B$, we "assume" A and then try to derive B. This is the implementation of the Gentzen rule

$$\frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \rightarrow B}$$

[4] These are sometimes called *intuitionistic sequents*, by contrast with *classical sequents* which allow $B_1, \ldots, B_n$ on the right. *Weierstrass* uses intuitionistic logic, but that is for efficiency and convenience only, and is not essential. Indeed some of the computational steps may not be intuitionistically valid.

In addition to the ordinary variables of logic, we also make use of "metavariables", whose values are expressions (terms or sometimes formulae), rather than numbers or other mathematical objects which are the values of ordinary (object) variables. *Weierstrass* introduces metavariables when it uses the Gentzen rules

$$\frac{\Gamma \Rightarrow A[t/x]}{\Gamma \Rightarrow \exists x A} \qquad \frac{A[t/x], \Gamma \Rightarrow B}{\forall x A, \Gamma \Rightarrow B}$$

For example, to prove $\exists\delta\forall x, y(|x-y| < \delta - |f(x)-f(y)| < \epsilon)$, *Weierstrass* will change δ to a metavariable, and try to prove $\forall x, y(|x-y| < \delta - |f(x)-f(y)| < \epsilon)$. Eventually, δ will be given a value (if the proof is successful), and when that value is put in for the metavariable δ, the result will be a (part of a) proof tree which is legal according to the Gentzen rules. In the meantime, that is before the final unifications take place, what is being constructed is something slightly more general than a Gentzen proof tree; it is a Gentzen proof tree in which metavariables are allowed, and the metavariables may have values, and the values may be expressions involving other metavariables. (That is, the metavariables can be "partially instantiated".) A formal definition of an *extended derivation*, and some related theorems, can be found in [4].

In general, *Weierstrass* starts by loading an axiom file, which contains (zero or more) axioms and a goal. The goal is placed in the succudent, and the axioms in the assumption list, which implements the antecedent. Weierstrass then attempts to construct an extended derivation of this sequent. Unification is used in order to instantiate metavariables introduced as described above. Unification is also used to control the selection of the next rule to be applied. Some of the logical rules are broken into subcases, and the order in which they are tried is important. These matters are discussed in [4]. For the present work, it suffices to note that the logical apparatus, functioning on its own, is a decent theorem-prover. While it makes no attempt to compete with Otter, the logic required for ordinary mathematics is comparatively simple, and in no example has the logical apparatus required revision beyond what was described in [4] seven years ago.

The implementation of a metavariable X includes a data structure designed to keep track of a list of variables that are "forbidden" to X; this means that X cannot have a value that contains variable forbidden to X. We make this part of the definition (and implementation) of unification; see [5] for theoretical reasons. This method provides an efficient way to keep track of the conditions on variables that accompany the quantifier rules that introduce metavariables. Instead of giving a metavariable a forbidden value, and carrying out another long subproof, only to discard the result because the conditions on the quantifier rule are violated, the unification will fail instead. I credit Natarajan Shankar for first telling me something similar to this. The idea has been fruitful beyond this improvement in efficiency, since we can use it in connection with computational steps where we want to bound a certain quantity in terms of some bound that does not depend on certain variables; this will be discussed below.

4 Computational Methods in a Quantifier-free Prover

The theorem prover, or logical apparatus, in *Mathpert* is responsible for maintaining the correctness of computations; it must block incorrect steps and ensure that the assumptions do support the steps that are taken. The program *Weierstrass* began by combining the methods of [4] for handling first-order logic, with the methods of *Mathpert* for handling computations in a quantifier-free setting. More precisely, the logical apparatus in *Mathpert* deals with sequents composed of formulae which involve no implication or negation, but only disjunctions and conjunctions of equalities and inequalities. When we say "quantifier-free" below, this is what we mean. We describe those methods, as implemented in *Weierstrass*, in this section.

4.1 Logical and Mathematical Simplification

Computational methods can be applied either to a mathematical expression (term), or to a logical expression (proposition). That is, we can treat rewriting propositions according to the laws of Boolean algebra on the same footing as rewriting algebraic expressions according to the laws of eighth-grade algebra. We apply the term "simplification" to describe the application of both algebraic and logical operations.

4.2 Operations More General than Rewrite Rules

The term *operation* is here used to mean an algorithm that transforms an expression of a certain form into another expression, which is equal (algebraically or logically) to the input, possibly under certain "side conditions". For example, $\sqrt{x^2} = x$ represents an operation that transforms an expression of the form $\sqrt{A^2}$, where A is any expression, into A; but it has the side condition $0 \leq A$. Operations may be rewrite rules, but they may well be more general than rewrite rules. For example, the operation named *collect powers* can be used to rewrite x^2x^3 as x^5, which is not a rewrite rule since arithmetic on the exponents is involved. The same operation can be used to collect powers separated by other terms, as in x^2yx^3, and to collect any number of powers, as in $x^2yx^3x^4$. But, like rewrite rules, operations can be applied to any subterm.

4.3 Operations and Side Conditions

Weierstrass keeps the assumption list (the antecedent) in simplified form, so it is not necessary to look for operations to apply to the antecedent.[5]

[5] When an operation is applied to the succedent, any occurrences of the same formula that was changed in the succedent are also changed in the antecedent, after which some simplifications in the antecedent may be performed to keep the antecedent in simplified form.

When the succedent is quantifier-free, *Weierstrass* will try to simplify it, trying a large number of logical and algebraic operations. These operations, for example, may simplify Boolean combinations of inequalities, or simplify certain inequalities to **true** or **false**. Purely algebraic simplification will also be performed, but not, for example, factoring or common denominators; except that greatest common divisors may be cancelled out of fractions.

This last example brings up the interesting question of the relationship between the antecedent (assumptions) and the simplifications performed in the succudent. Consider, for example, the proposition

$$\frac{x^4 - 1}{x^2 - 1} > 0$$

If we cancel $x^2 - 1$ from numerator and denominator, we arrive at the proposition $x^2 + 1 > 0$, which will simplify to **true**. But, at some point we must assume $x^2 - 1 \neq 0$; otherwise the result is incorrect. The points to consider here are two: (1) the original expression is not defined for all values of x, and (2) the domain changes as a result of the application of an operation, which *on the common domain* preserves equivalence. The problem of "partial terms" (terms which can be undefined) is thus closely related to the problem of "side conditions" of symbolic operations.

4.4 Partial Terms, Domains, and Side Conditions

There are two natural ways to make the assumption $x^2 - 1 \neq 0$ in the above example: either at the outset, or when the cancellation is performed. Plan A would be to analyze the domain of the goal when the problem is set up, and put the domain conditions into the antecedent. According to plan A, the condition $x^2 - 1$ would have been assumed at the outset, and hence could have been inferred when required as the side condition for cancelling the common factor of numerator and denominator. Plan B would be to allow potentially undefined terms in the partially constructed proof, and only assume $x^2 - 1 \neq 0$ when it is required as a side condition for an operation. Plan B was mentioned in earlier publications such as [4], but in the practical implementation of *Mathpert*, Plan A was found to be more efficient; after all, we certainly need to assume that the goal is defined to prove anything sensible at all. Therefore, Plan A has been adopted in *Weierstrass* as well.

However, Plan B is not thereby consigned to the dustbin of history. There remain situations in which a symbolic operation may have a side condition that is not necessarily implied by the domain.[6] In such situations, Plan B will still be used. An example would be the application of the operation $\sqrt{x^2} = x$, whose

[6] We use the word *domain* to mean a proposition giving the conditions under which a term is defined; thus the domain of $\sqrt{x}$ is $0 \leq x$. If propositions are thought of as Boolean-valued functions, and sets are also thought of as Boolean-valued functions, as Church suggested, then this coincides with the usual usage that the domain of $\sqrt{x}$ is $\{x | 0 \leq x\}$.

side condition $0 \leq x$ is not implied by the domain.[7] In the proof tree formalism of logic, Plan B would entail copying the new assumption to the antecedents all the way down the tree from the place where the operation is applied to the conclusion. In the implementation, however, most of the antecedent is kept in the assumption list, rather than duplicated at every line of the proof tree, and so adding the new assumption once suffices.

4.5 Infer, Refute, Assume

When an operation has a side condition, there are two choices: either the operation can try to *infer* the side condition, and fail if the inference fails, or it can try to *check* the side condition, which means that it will try to infer it, and if that fails, it will try to refute it, and if that fails it will simply assume it. Thus, if we try to simplify an expression of the form $\sqrt{A^2}$ to A, the side condition $0 \leq A$ will be checked. If, for example, A is 3, the condition $0 \leq 3$ will be successfully inferred, so the simplification takes place without an assumption. If, on the other hand, A is -3, then the condition will be refuted, and the simplification will not take place. If, however, A is an expression such that $0 \leq A$ can neither be inferred nor refuted, then it will be assumed.

Consider the example mentioned in the introduction, of dividing both sides of $a = 0$ by a. The side condition for dividing both sides of an equation by a is that $a \neq 0$. Can we infer this? No. Can we refute it? Not officially, since $a = 0$ is in the succedent, rather than the antecedent. Then, we will assume it, and obtain the logically correct but useless proof

$$\frac{a \neq 0 \Rightarrow 1 = 0}{a \neq 0 \Rightarrow a = 0}$$

To prevent this sort of thing, *refute* is also allowed to use the antecedent as a temporary assumption. That way, if the side condition is inconsistent with the goal, we will avoid making a contradictory assumption. When this is done, the attempt to divide $a = 0$ by a will result in an error message to the effect that you can't divide by zero. This can be seen working in *Mathpert*.

Note that the choice whether *infer* or *check* is used is specified in the operation itself. That is, there will be two different operations represented loosely by the equation $\sqrt{x^2} = x$. One of them will *infer* the side condition (or fail) and the other will *check* the side condition. In practice, it seems to work best to avoid using *check* in the elementary simplification that are automatically applied in *Weierstrass*; after all, if we fail to prove the desired theorem because we failed to list all the assumptions, we can run *Weierstrass* again after adding the omitted assumption in the axiom file. But, the method is used to good advantage in *Mathpert*, and may prove of value in future applications of systems like *Weierstrass*.

To make this scheme work, *infer* and *refute* must be guaranteed to terminate, and hence must be incomplete; that is, sometimes a true side condition will not be

[7] In practice, few such operations are applied automatically in *Weierstrass*, but an interactive prover based on these principles would certainly use Plan B extensively.

inferred, or a false one not refuted. We may wind up making a false assumption. For example, if $p(x)$ is an expression which is really identically zero, but can't be simplified to zero by the means of simplification used by *infer*, then we might be led to make the assumption $p(x) \neq 0$, e.g. to divide both sides of an equation by $p(x)$. This could lead to logically correct but senseless results. This is, however, unavoidable, as the problem of determining whether mathematical expressions $p(x)$ are identically zero is recursively unsolvable [14].

A related situation arises in solving equations. For example, consider the equation $x^2 - x = 0$. If we divide both sides by x, we make the assumption $x \neq 0$ and find the solution $x = 1$. This is logically correct, but we didn't achieve the goal of finding all solutions of the original equation. This may not be a logical error, but it is a mathematical error, and hence has been blocked in *Mathpert*, but in an interactive system based on *Weierstrass*, it would not necessarily be blocked.

4.6 Using the Assumptions in a Computation

Suppose we try to simplify $0 \leq x$, while $0 < x$ is in the assumption list. Then it is efficient to allow $0 \leq x$ to simplify to **true**. For example, if $0 \leq x$ occurs inside a disjunction in the succedent, the whole succedent may simplify to **true**, completing (that branch of) a proof. Similarly, a side condition involving $x \neq 0$ should be reduced to $0 < x$ if $0 \leq x$ is in the assumption list.

4.7 Computation within the Scope of a Bound Variable

Even though *Weierstrass* applies simplification only to quantifier-free formulae, sometimes it is still necessary to compute inside the scope of a bound variable, since variables can be bound by definite integrals or indexed sums. For example, we want to conclude that $\sum_{k=1} \rightarrow 5x^k$ is everywhere defined, even though the condition for x^k to be defined (for an integer k) requires $k > 0 \vee x \neq 0$. But $k > 0$ holds because the lower limit of the sum is positive. In the case of definite integrals and indexed sums, this is handled by making temporary assumptions out of the limits of the sum or integral, while the focus of computation is in the scope of the sum or integral. Limit terms are handled similarly, but the assumptions to made involve infinitesimals and the use of non-standard analysis; this much more complicated algorithm is discussed in detail in [3]. Computation within the scope of bound variables will not be discussed further in this paper.

4.8 What Formal System has been Implemented?

It is an interesting question to formulate precisely a language and rules of inference that could be said to be implemented by *Weierstrass*. One such language has been specified in [3]; it essentially allows variables for integers and real numbers, equality and inequality, and symbols for all the elementary functions used in calculus. A complete and precise grammar for such a language can be found

in [3]. This language also allows the formation of integrals, derivatives, indexed sums, and limit terms; definite integrals, indexed sums, and limit terms can bind variables. *Weierstrass* also allows the formation of λ-terms which are not specified in [3].

We turn now to rules of inference. A single additional rule schema describes the simplest way to add computation to a quantifier-free prover:

$$\frac{\Gamma \Rightarrow B \qquad B, \Gamma \Rightarrow A\sigma}{\Gamma \Rightarrow A}$$

where $A\sigma$ denotes the result of applying some mathematical or logical operation to A, or to a subterm of A, replacing the subterm by the result of the operation. In the rule, B is the side condition of the operation, if any; if the operation has no side condition, the premise $\Gamma \Rightarrow B$ does not occur. In principle an operation could also have more than one side condition, in which case there might be more than two premises.

The control strategy for applying this rule is this: whenever A contains no quantifiers or implications or negations, try this rule, with a certain selection of operations in a certain pre-specified order. But, the second premise $\Gamma \Rightarrow B$ representing the side condition is not passed recursively to the main theorem-prover, but must be derived by very limited means. This is to prevent long delays or even infinite regresses attempting to verify the side conditions of mathematical operations; in other words, a practical rather than a theoretical consideration.

This rule of inference does not, however, adequately describe the technique of using the antecedent in simplification as described above. One way to do so, although it is admittedly not very elegant, is to generalize the rule to this:

$$\frac{C, \Gamma \Rightarrow (C \rightarrow A)\sigma \qquad C, \Gamma \Rightarrow B}{C, \Gamma \Rightarrow A}$$

Here C is one assumption, and σ is an operation that can work on an implication (usually of inequalities). For example, σ might simplify $a < c \rightarrow a \leq c$ to **true**. In both *Mathpert* and *Weierstrass*, we never use more than one assumption at once in the simplification process.

The above rules still don't adequately describe *Weierstrass* or even *Mathpert*, because they do not account for keeping the assumption list in simplified form. To describe this we need to add the rule

$$\frac{\Gamma\sigma \Rightarrow A}{\Gamma \Rightarrow A}$$

where $\Gamma\sigma$ represents the result of simplifying the assumption list Γ. Since simplification generally can use formulas in the assumption list, *Weierstrass* has to be careful when simplifying assumptions, or each assumption would simplify to **true**! Each assumption is temporarily removed from the assumption list, then simplified (possibly using the other assumptions), and the result replaces the original assumption. This process is continued until nothing changes. The result of these simplifications is $\Gamma\sigma$.

5 Combining Computation with First-Order Logic

In previous sections, we have considered the backwards-Gentzen framework for a theorem-prover, and the means of adding computation (simplification) to the quantifier-free fragment of such a prover. We now take up the additional features which were added to *Weierstrass* to allow it to handle epsilon-delta proofs.

The first point is that we must, under certain circumstances, allow *Weierstrass* to factor, or even use trig factor identities. This is a question of control, and not of something new in principle: since factoring preserves mathematical equality, it can be treated exactly like the other computation rules discussed above. It is just a question of factoring when it is useful, and not factoring when it is not useful. To achieve this, we simply put it at the bottom of the list of things to try; that is, below all the things that have been discussed above. It will thus not be tried unless without it, the proof would fail. That will dispose of the problem of factoring when it is not useful.

The other new features can be represented as additional inference rules, which are, like the Gentzen rules, to be applied "backwards" with the aid of unification. We shall describe several of these rules. Like all the rules in sequent calculus, the premises and conclusion of these rules are sequents; but in all cases, the antecedent is unchanged from premise to conclusion, so when writing the rules below, we shall omit $\Gamma \Rightarrow$ in both premises and conclusion.

5.1 Finding Upper and Lower Bounds

Every mathematician knows that many a proof boils down to finding a suitable bound for some expression that does not depend on certain variables. We have implemented a pair of algorithms called *UpperBound* and *LowerBound*. *UpperBound* takes as input a term t to be bounded, and a list of variables on which the bound may not depend. Otherwise put, it tries to find a legal value for a metavariable M such that $|t| \leq M$ could be derived, with the specified list of variables forbidden to M. For example, *UpperBound* knows that $|\sin x| \leq 1$. A better example: if *UpperBound* is asked to bound x by a bound not depending on x, and the current assumptions include $a < x$ and $x < b$, then it will return the bound $|x| \leq \max(|a|, |b|)$. *UpperBound* is probably as good as a very good calculus student at what it does. *LowerBound* is similar, but it tries to find M such that $M \leq |t|$. The two algorithms are defined by mutual recursion.

UpperBound is added directly to *Weierstrass* as a rule of inference with no premises. That is, when we have a goal of the form $\alpha < M$, where M is a metavariable and α is some expression, we can directly terminate that proof branch, instantiating M to the expression produced by *UpperBound*, supplying as the second argument to *UpperBound* the list of variables forbidden to M.

5.2 Factor Bounding

The second new inference rule to be added is called *FactorBounding*. It says that if you want to prove $\beta\gamma$ is small, one way to do it is to prove that γ is small

and give a bound for β. The following rule is state for simplicity using only two factors, but the rule is implemented for a product of any number of factors:

$$\frac{\Gamma, |\alpha| < \delta \Rightarrow \gamma \leq M \qquad \Gamma, |\alpha| < \delta \Rightarrow |\beta| < \epsilon/(M+1)}{\Gamma, |\alpha| < \delta \Rightarrow |\beta\gamma| < \epsilon}$$

When this rule is implemented, we take M to be a fresh metavariable, and forbid to M all the variables that are forbidden to δ. In the present implementation of *Weierstrass*, the rule is used only when δ is a metavariable. The implementation also provides an algorithm for deciding which of several factors to bound: it first identifies the quantity in the antecedent that must be less then δ, and then looks for a factor which has a nonzero finite limit as that quantity tends to zero. Limit calculations are performed by symbolic code from *Mathpert*. These limit calculations do not enter the actual proof; they are only used to select the factor to try to bound.

At this point, you might want to turn to Example 1 in the next section, to see how *UpperBound* and *FactorBounding* are used to prove the continuity of $f(x) = x^3$.

5.3 Inequality Chaining

A notorious difficulty in inequality proving is the necessity of using transitivity chains, and the difficulty of finding the right chain of inequalities in an exponentially large search space. However, many useful chains are of length two, based on some standard "known" inequality. For example, if we want to prove $|\sin x| < \epsilon$, it will suffice to prove $|x| < \epsilon$ in view of the known inequality $|\sin x| < |x|$. Weierstrass implements this idea in an algorithm *UsefulBounds*. Described as in inference rule, this just looks like transitivity:

$$\frac{\alpha \leq \beta \qquad \beta < \epsilon}{\alpha < \epsilon}$$

When implemented, $\alpha \leq \beta$ is one of a list of specific known inequalities that have been supplied to *Weierstrass*. For example, a special case of the rule would be

$$\frac{|\sin x| \leq |x| \qquad |x| < \epsilon}{|\sin x| < \epsilon}$$

This rule of inference is needed by Weierstrass to prove the uniform continuity of $\sin x$. See the discussion of this example in the next section.

UpperBound is also capable of controlling some transitivity chaining through the inequalities present in the antecedent. For example, if it is trying to solve $x < M$, where x and y are forbidden to M, and the antecedent contains $x < y$ and $y < b$, the bound $x < b$ will be found, and M will get the value b.

5.4 Mean Value Theorem

Weierstrass can use the mean value theorem to prove an inequality. This is an interesting rule of inference, because it reduces a quantifier-free goal to a subgoal

involving quantifiers. The purely logical rules of *Weierstrass* use the cut-free rules of sequent calculus, which always reduce goals to logically simpler subgoals. Here is the rule of inference *MVT*:

$$\frac{\forall z(x \leq z \leq y \rightarrow f'(x) \leq M) \qquad |x-y| \leq \epsilon/M}{|f(x)-f(y)| \leq \epsilon}$$

There is another rule under the same name, in which the conclusion and the second premise have strict inequality. When implemented, M is a freshly-created metavariable, and x and y are added to the list of variables forbidden to M. Note that this would not be the case if the rule were stated with an existential quantifier over M in the premise (combining the two premises into a conjunction). It is by controlling the list of variable forbidden to M that Weierstrass is induced to look for a bound independent of x and y. Now, in general such a bound cannot exist unless the range of x and y is restricted by further inequalities, so some inequality chaining will generally be needed to find the bound M. As an example of such a proof, we consider in the next section, a proof of the uniform continuity of $\sqrt{x}$ on closed intervals $[a,b]$ with $a > 0$.

6 Examples of Proofs that Weierstrass Can Find

In this section we describe the key points of certain illustrative example proofs. The strict length limit does not permit the inclusion of the actual output of Weierstrass.

6.1 Uniform Continuity of $f(x) = x^3$ on Closed Intervals

. This example illustrates the use of *UpperBound* and *FactorBounding*. When *Weierstrass* is asked to prove the uniform continuity of $f(x) = x^3$ on closed intervals $[a,b]$, it soon arrives at the problem of finding a value for the metavariable δ such that, assuming $|x-y| < \delta$, we could derive $|x^3 - y^3| < \epsilon$. Factoring, this reduces to $|x-y||x^2+xy+y^2| < \epsilon$. At this point, the above rule will be used (in reverse, with $\alpha = |x-y|$), creating the two new goals $|x^2+xy+y^2| \leq M$ and $|x-y| < \epsilon/(M+1)$. The first one will be solved by using *UpperBound*, instantiating the metavariable M to $3\max(|a|,|b|)$ and the second will be solved by the axiom rule $\Gamma, A \Rightarrow A$, where A is the assumption $|x-y| < \delta$, instantiating the metavariable δ to $\epsilon/(M+1)$.[8]

6.2 Uniform Continuity of $\sin x$ and $\cos x$

These two theorems are proved by *Weierstrass* in a way similar to the above example. However, there are two new twists to the argument. First, *Weierstrass* needs to use the trig factoring operations, not just polynomial factoring, in order

[8] *Weierstrass* will be able to handle the case of $f(x) = x^n$ similarly, where n is an integer variable, as soon as *UpperBound* is extended to handle indexed sums, since an indexed sum arises when $x^n - y^n$ is factored.

to write $\sin x - \sin y$ as $2\sin(1/2(x-y))\cos((1/2)(x+y)$. Then in order to instantiate δ, it must use *UsefulBounds* to apply the known inequality $|\sin u| \leq |u|$, since *FactorBounding* will produce the subgoal $\sin(1/2(x-y)) < \epsilon/(M+1)$, which does not unify directly with $|x-y| < \delta$. Even after $|\sin u| \leq |u|$ is used, the 2 in the denominator still requires another step, which however *Weierstrass* takes without difficulty, since an inequality can be simplified by multiplying both sides by 2. This is an example of computation applied to a proposition rather than a mathematical term.

6.3 Continuity of $f(x) = \sqrt{x}$

More precisely, the example is the uniform continuity of $\sqrt{x}$ on closed intervals $[a,b]$ with $0 < a$. To handle the continuity of $\sqrt{x}$ by factoring, we would have to get *Weierstrass* to write

$$|\sqrt{x} - \sqrt{y}| = \frac{|x-y|}{\sqrt{x}+\sqrt{y}}$$

It would certainly be possible to do this, but it would be *ad hoc*, as the kind of computation rule that would do this would cause trouble elsewhere, so it would have to be added as a logical inference rule for this special sort of inequality. Rather than add an *ad hoc* rule, we chose to use this example as an illustration of the use of the Mean Value Theorem. *Weierstrass* will compute the derivative of $\sqrt{x}$ and bound it. Specifically, the inference rule MVT described above will introduce a new metavariable M and create the subgoals, $|x-y| < \epsilon/M$ and $\forall z(x \leq z \leq y \rightarrow |(1/2)z^{-2}| \leq M$. Note that the derivative is evaluated. The variables x,y, and z are forbidden to M. When *UpperBound* tries to bound z^{-2}, it calls *LowerBound* to bound z, and successfully finds the transitivity chain $a \leq x \leq z$, arriving at the bound $a \leq z$.

References

1. Beeson, M.: Logic and computation in *Mathpert*: an expert system for learning mathematics, in: Kaltofen, E., and Watt, S. M. (eds.), *Computers and Mathematics*, pp. 202–214, Springer-Verlag (1989).
2. Beeson, M.: Design Principles of Mathpert: Software to support education in algebra and calculus, in: Kajler, N. (ed.) *Human Interfaces to Symbolic Computation*, Springer-Verlag, Berlin/ Heidelberg/ New York (1996).
3. Beeson, M.: Using nonstandard analysis to ensure the correctness of symbolic computations, *International Journal of Foundations of Computer Science* **6**(3) (1995) 299-338.
4. Beeson, M.: Some applications of Gentzen's proof theory in automated deduction, in: Shroeder-Heister, P., *Extensions of Logic Programming*, Springer Lecture Notes in Computer Science **475**, pp. 101–156, Springer-Verlag (1991).
5. Beeson, M.: Unification in lambda-calculus, to appear in *Automated Deduction: CADE-15 - Proc. of the 15th International Conference on Automated Deduction*, Springer-Verlag, Berlin/Heidelberg (1998).

6. Bledsoe, W. W.: Some automatic proofs in analysis, pp. 89–118 in: W. Bledsoe and D. Loveland (eds.) *Automoated Theorem Proving: After 25 Years*, volume 29 in the *Contemporary Mathematics* series, AMS, Providence, R. I. (1984).
7. Boyer, R., and Moore, J.: *A Computational Logic*, Academic Press (1979).
8. Buchberger, B.: History and basic features of the critical-pair completeion procedure, *J. Symbolic Computation* **3**:3–88 (1987).
9. Clarke, E., and Zhao, X.: Analytica: A Theorem Prover in Mathematica, in: Kapur, D. (ed.), *Automated Deduction: CADE-11 - Proc. of the 11th International Conference on Automated Deduction*, pp. 761–765, Springer-Verlag, Berlin/Heidelberg (1992).
10. Coste, M., and Roy, M. F.: Thom's lemma, the coding of real algebraic numbers, and the computation of the topology of semi-algebraic sets, in: Arnon, D. S., and Buchberger, B., *Algorithms in Real Algebraic Geometry*, Academic Press, London (1988).
11. Harrison, J., and Thery, L.: Extending the HOL theorem prover with a computer algebra system to reason about the reals, in *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG '93*, pp. 174–184, Lecture Notes in Computer Science **780**, Springer-Verlag (1993).
12. Kleene, S. C., *Introduction to Metamathematics*, van Nostrand, Princeton, N. J. (1952).
13. McCune, W.: Otter 2.0, in: Stickel, M. E. (ed.), *10th International Conference on Automated Deduction* pp. 663-664, Springer-Verlag, Berlin/Heidelberg (1990).
14. Richardson, D., Some unsolvable problems involving elementary functions of a real variable, J. Symbolic Logic **33** 511–520 (1968).
15. Stoutemeyer, R.: Crimes and misdemeanors in the computer algebra trade, *Notices of the A.M.S* **38**(7) 779–785, September 1991.
16. Wu Wen-Tsum: Basic principles of mechanical theorem-proving in elementary geometries, *J. Automated Reasoning* **2** 221-252, 1986.

A Appendix: Output of *Weierstrass* on the Examples

Weierstrass produces an internal proof object, which can be viewed in either "trace view" or "proof tree view". These views both use two-dimensional display of formulas on the screen. When you choose `File | Save As`, you save a text representation of the proof, either as trace or as proof tree. Formulas are written in a parseable form, similar to TEX, but without backslashes, and enclosed in dollar signs. In the future, I intend to use these files with WebTEXto post proofs to the Web. For purposes of these appendices, I have simply included these files verbatim (inserting only some line breaks) to avoid any errors introduced by transcribing them into TEX, and to demonstrate exactly what the program produces. I have used trace view, since the files are more readable than with proof tree view. Even so, these files do not convey the process of proof construction well, since the metavariables are replaced by their final values; for example, we don't see how and when δ is found, but instead it appears to be "pulled out of a hat" near the beginning of the proof. It is interesting that this very phenomenon is often a problem in the presentation of proofs produced by human mathematicians!

A.1 Continuity of $f(x) = x^3$

```
Assuming $epsilon > 0$
Trying $exists(delta,all(x,y,a <= x,x <= b,a <= y,y <= b,
         abs(x-y) < delta->abs(x^3-y^3) < epsilon))$
 Trying $all(x,y,a <= x,x <= b,a <= y,y <= b,
         abs(x-y) < X->abs(x^3-y^3) < epsilon)$
   Assuming $a <= x,x <= b,a <= y,y <= b,abs(x-y) < X$
   Trying $abs(x^3-y^3) < epsilon$
   Factoring, it would suffice to prove:
       $abs(x-y) abs(x^2+x y+y^2) < epsilon$
   We have the following bound:
       $abs(x^2+x y+y^2) <= 3(max(abs(a),abs(b)))^2$
   So it would suffice to prove:
       $abs(x-y) < epsilon/(3(max(abs(a),abs(b)))^2+1)$
   Aha!  we have
       $abs(x-y) < epsilon/(3(max(abs(a),abs(b)))^2+1)$
   success
   Discharging
 success
success
Discharging
```

A.2 Continuity of $f(x) = \sin x$

```
Trying $epsilon > 0->exists(delta,all(x,y,abs(x-y) < delta->
               abs(sin(x)-sin(y)) < epsilon))$
 Assuming $epsilon > 0$
 Trying $exists(delta,all(x,y,abs(x-y) <
        delta->abs(sin(x)-sin(y)) < epsilon))$
  Trying $all(x,y,abs(x-y) < 1/2 epsilon->
        abs(sin(x)-sin(y)) < epsilon)$
    Assuming $abs(x-y) < 1/2 epsilon$
    Trying $abs(sin(x)-sin(y)) < epsilon$
    Using trigonometry, it would suffice to prove:
        $2abs(sin(x-y)/2) abs(cos(x+y)/2) < epsilon$
    Dividing both sides, it would suffice to prove:
        $abs(sin(x-y)/2) abs(cos(x+y)/2) < 1/2 epsilon$
    We have the following bound:
        $abs(cos(x+y)/2) <= 1$
    So it would suffice to prove:
        $abs(sin(x-y)/2) < epsilon/4$
    In view of the known inequality |sin x| < |x| we have:
        $abs(sin(x-y)/2) <= abs((x-y)/2)$
    it would therefore suffice to prove:
        $abs((x-y)/2) < epsilon/4$
```

```
    Simplifying, it would suffice to prove:
        $2abs(x-y) < epsilon$
    Dividing both sides, it would suffice to prove:
        $abs(x-y) < 1/2 epsilon$
    Aha!  we have $abs(x-y) < 1/2 epsilon$
    success
    Discharging
  success
 success
 Discharging
success
```

A.3 Continuity of $f(x) = \sqrt{x}$

```
 Assuming $a > 0,epsilon > 0$
 Trying $exists(delta,all(x,y,a <= x,x <= b,a <= y,y <= b,
        abs(x-y) < delta->abs(sqrt(x)-sqrt(y)) < epsilon))$
  Trying $all(x,y,a <= x,x <= b,a <= y,y <= b,abs(x-y) <
        epsilon/(1/2 a^(-1/2))->abs(sqrt(x)-sqrt(y)) < epsilon)$
    Assuming $a <= x,x <= b,a <= y,y <= b,
        abs(x-y) < epsilon/(1/2 a^(-1/2))$
    Trying $abs(sqrt(x)-sqrt(y)) < epsilon$
    Simplifying, it would suffice to prove:
        $abs(x^(1/2)-y^(1/2)) < epsilon$
    By the mean value theorem applied to $fz = z^(1/2)$
    it would suffice to prove:
        $all(z,x <= z,z <= y->abs(1/2 z^(-1/2))
        <= 1/2 a^(-1/2)),abs(x-y) < epsilon/(1/2 a^(-1/2))$
     Trying $all(z,x <= z,z <= y->abs(1/2 z^(-1/2)) <= 1/2 a^(-1/2))$
      Trying $x <= z,z <= y->abs(1/2 z^(-1/2)) <= 1/2 a^(-1/2)$
       Assuming $x <= z,z <= y$
       Trying $abs(1/2 z^(-1/2)) <= 1/2 a^(-1/2)$
       We have the bound: $abs(1/2 z^(-1/2)) <= 1/2 a^(-1/2)$
       success
       Discharging
      success
     success
     Trying $abs(x-y) < epsilon/(1/2 a^(-1/2))$
     Aha!  we have $abs(x-y) < epsilon/(1/2 a^(-1/2))$
     success
    success
    Discharging
  success
 success
 Discharging
```