Contexts in Mathematical Reasoning and Computation*

William M. Farmer, Joshua D. Guttman, F. Javier Thayer

7 February 1994

Abstract

Contexts are sets of formulas used to manage the assumptions that arise in the course of a mathematical deduction or calculation. This paper describes some techniques for symbolic computation that are dependent on using contexts, and are implemented in IMPS, an Interactive Mathematical Proof System.

Contents

*Supported by the MITRE-Sponsored Research program. Preliminary version entitled "Reasoning with Contexts" appears in A. Miola, ed., *Design and Implementation of Symbolic Computation Systems (Proceedings, International Symposium, DISCO '93, Gmunden, Austria, September 1993), Lecture Notes in Computer Science*, vol. 722, Springer-Verlag, Berlin, 1993, pp. 216–228. Authors' address: The MITRE Corporation, 202 Burlington Rd, Bedford MA 01730-1420 USA; Telephone: 617-271-2749; Fax: 617-271-3816; E-mail: {farmer,guttman,jt}@mitre.org.

1 Introduction

A *context* is a set of formulas $\Gamma = \{\varphi_1, \ldots, \varphi_n\}$. The formulas of a context ordinarily serve as background assumptions. They may contain both closed formulas, such as the axioms of a theory, and open formulas, such as:

- $x \neq 0$,
- $0 < y \wedge f$ is differentiable at y, and
- G is a finite group.

A formula ψ is true in the context Γ if the members of Γ logically imply ψ. Contexts are a natural device for managing the assumptions that arise in the course of a mathematical deduction or calculation. Contexts also provide a mechanism for keeping track of what can be legitimately assumed at various places in a formula. Thus, in the formula $0 < x^2 \supset x/x = 1$, one can assume that the variable x in the subexpression $x/x = 1$ is nonzero. A "context at a place in a formula" is called a *local context* [13].

The following examples illustrate the need for contexts in rigorous mathematical reasoning.

- If G is a set with a binary operation, then a theorem about finite groups can be applied to G only when "G is a finite group" is assumed.
- The term x/x can be simplified to 1 only if $x \neq 0$ is known to be true.
- The equation
$$\int_a^b x^\alpha dx = \frac{b^{\alpha+1} - a^{\alpha+1}}{\alpha + 1}$$
holds only when $\alpha \neq -1$.
- The power series expansion
$$\frac{1}{1-x} = \sum_{i=1}^{\infty} x^i$$
is legitimate only for $|x| < 1$.

Although symbolic computation can be as highly context-dependent as any other form of mathematical reasoning, existing symbolic manipulation systems usually are not sensitive to contexts. As a consequence, these systems may do such things as simplifying x/x to 1 even when there is no assumption that $x \neq 0$. Computations in these systems typically behave as though they were applying universally valid equalities without auxiliary conditions.

This simple approach is problematic because many indispensable equalities, like $x/x = 1$, *do* have auxiliary conditions. Keeping track of such auxiliary conditions within complex expressions can become extremely difficult, and without precautions, can very easily lead to falsehoods. One might hope to eliminate side conditions by using generic quantification, that is, validity "almost everywhere" of a formula. For instance, if $P(x, y)$ is the formula $y - x^2 \neq 0$, then P holds for all values of the parameter $(x, y) \in \mathbf{R}^2 \setminus K$, where the set K is the parabola given by the equation $y = x^2$. K is a small subset of $\mathbf{R}^2$ (in measure), so that it is customary to say "P is true for almost all values of (x, y)." However, this kind of quantification is fraught with difficulties in that the most basic logical operations, such as instantiation of universals, fail. Thus it is not true that for an arbitrary polynomial function $\varphi : \mathbf{R} \to \mathbf{R}^2$, $P(\varphi(t))$ holds outside some small set in $\mathbf{R}$. Indeed, if $\operatorname{ran} \varphi \subseteq K$, then $P(\varphi(t))$ is always false.

Logically, assumptions can be incorporated into an expression by turning the expression into a conditional. For example, $|x + |3 - x||$ has the same value as the conditional

$$\begin{cases} 2x - 3 & \text{if } x \geq 3 \\ 3 & \text{otherwise} \end{cases}$$

However, this reduction is usually undesirable, firstly because the conditional is much harder to understand than the original expression, and secondly because there are no simple algebraic rules for reducing conditionals. Alternatively, one could consider the original expression in two different contexts: one context Γ_1 contains the assumption $x \geq 3$ and the other Γ_2 contains the assumption $x < 3$. In the context Γ_1, the expression $|x + |3 - x||$ simplifies to $P_1 = 2x - 3$, while in Γ_2, it simplifies to $P_2 = 3$. The original expression is thus replaced with two polynomial expressions P_i in the variable x.

IMPS Reasoning with contexts is one of the main design principles of IMPS, an Interactive Mathematical Proof System [10, 8]. IMPS is intended to support the traditional techniques of mathematical reasoning. For instance, the logic of IMPS [7, 6, 11] allows functions to be partial and terms to be undefined. The system consists of a database of mathematics and a collection of tools for exploring, applying, extending, and communicating the mathematics in the database. One of the chief tools is a facility for developing formal proofs. In contrast to the formal proofs described in logic textbooks, IMPS proofs are a blend of computation and high-level inference. Consequently, they resemble intelligible informal proofs, but unlike informal proofs, all details of an IMPS proof are machine checked.

Reasoning in IMPS is always performed with respect to some context. In the process of constructing a proof, many different contexts are usually needed. IMPS represents each context with a separate data structure that holds the set of assumptions of the context as well as various kinds of derived information. The entries in an IMPS proof are compound formulas, called *sequents*, consisting of a single formula called the *assertion* together with a context. Low-level reasoning is performed by two kinds of context-dependent computation: automatic expression simplification and semi-automatic theorem application.

Each context in IMPS belongs to a *theory*, which is a formal language plus a set of closed formulas of the language called *axioms*. All the contexts created in course of a proof belong to the same theory. The assumptions of a context, by definition, form a superset of the axioms of its theory. Thus there are two classes of context assumptions: the theory axioms are closed formulas used as permanent assumptions, and the remaining assumptions are possibly open formulas used as temporary assumptions.

IMPS supports the "little theories" version of the axiomatic method [9]. In this approach, a number of theories are used in the course of developing a portion of mathematics. Different theorems are proved in different theories, depending on the amount and kind of mathematics that is required. Theories are linked together by interpretations [5] which serve as conduits to pass results from one theory to another. We argue in [9] that this way of organizing mathematics across a network of linked theories is advantageous for managing complex mathematics by means of abstraction and reuse. IMPS can also be used with the "big theory" version of the axiomatic method in which all reasoning is performed within a single powerful and highly expressive axiomatic theory, such as Zermelo-Fraenkel set theory. Both versions of the axiomatic method are well established in modern mathematical practice.

In the following pages, we will develop two main points. One concerns the broad applicability of contextual reasoning, while the second concerns how to integrate symbolic computation with the context-managing and theorem-proving facilities of a system like IMPS. This second main point is a more speculative one, and so our discussion of it is more tentative.

Reasoning with Contexts The first main point is that reasoning with contexts is useful and tractable to implement in a system for computerized mathematics. We will of course emphasize the way that the idea is realized within the IMPS theorem proving system; nevertheless, many issues and algorithms are also relevant to systems for symbolic computation which do not intend to support theorem proving.

We will discuss the IMPS mechanisms for expression simplification and theorem application. These mechanisms illustrate that managing symbolic computation with contexts is feasible as well as desirable. An example of an IMPS proof involving expression simplification and theorem application is given in Section 5, which also contains statistics about the set of contexts generated by that proof and others in the IMPS database.

A General Mathematics Laboratory The second main point is that a system combining the functionality of current systems for symbolic computation with the facilities of a theorem prover is now feasible. Such a system would allow rigorous computation and proof in a wide variety of axiomatic theories. Computation and inference would be well integrated, in that the results of a computation could be used in the course of a proof. Conversely, newly proved theorems could naturally be incorporated in computation as rewrite rules (possibly conditionalized, depending on the syntactic form of the theorem). Moreover, computation would normally be relative to a context; the course of a computation would frequently depend on low-level theorem proving, resolving conditional choices according to the consequences of the context.

We call such a system a general mathematics laboratory, because we believe it would provide computerized support for a wide range of characteristic mathematical activities, such as those identified in [2]. We also believe that such a system can play a crucial role in mathematics education, by allowing students to gain a concrete appreciation of the algorithmic and inferential consequences of different theories or techniques. A mathematics laboratory may then play a role in mathematics education somewhat analogous to the role of a language laboratory for foreign language students.

The ideal mathematics laboratory would be constructed from scratch to provide integrated computational and inferential support. However, that would be an expensive undertaking. A reasonable level of functionality can also be achieved by linking together IMPS and an existing symbolic computation system, even though such systems are not fully sound mathematically. In Section 6.1, we discuss how IMPS could be allowed to make calls to an external symbolic computation system, and check the results in a sound way. In Section 6.2 we discuss the converse linkage, in which a symbolic computation system would make calls on IMPS to manage contextual information and check whether conditions hold in the current context. The symbolic computation system can use the results to determine the remainder of its computation.

2 The Simplifier

The IMPS simplifier is responsible for substantial size steps in computation and inference. It is always invoked on an expression relative to a context Γ, and serves three primary purposes:

- To invoke a variety of theory-specific transformations on expressions, such as rewrite rules and simplification of polynomials (given that the theory has suitable algebraic structure, such as that of a ring);
- To make simplifications based on the logical structure of an expression, often at locations deeply nested within it;
- To discharge the great majority of the assertions about definedness needed to apply many inferences and lemmas.

This last goal, and many other aspects of the simplifier, are required by the relatively permissive IMPS logic. Since in the IMPS logic functions may be partial and terms may be undefined, term simplification must involve a considerable amount of definedness checking. For example, simplifying expressions naively may cancel undefined terms, reducing a possibly undefined expression such as $1/x - 1/x$ to 0, which is certainly defined. In this example, the previous replacement is valid if the context Γ can be seen to entail the definedness or "convergence" of $1/x$. In general, algebraic reductions of this kind produce intermediate definedness formulas to which the simplifier is applied recursively. These formulas are called *convergence requirements*.

The IMPS logic also allows subtypes. By a type we mean a basic domain of objects, such as the points of a space, the real numbers, or the partial real functions $\mathbf{R} \rightharpoonup \mathbf{R}$. A type may have several distinguished subtypes, which denote nonempty subsets of the whole domain of the type. A *sort* is a type or a subtype. A variable always ranges over some particular sort. The logic supplies a logical operator `defined-in`, written $\downarrow$, which can be used to assert that a term has a value in a particular sort, for instance $6.2/3.1 \downarrow \mathbf{Z}$. There is also a logical operator `is-defined`, written as $\downarrow$ without a trailing sort, which asserts only that the term takes some value. This is equivalent to asserting that the term is defined in the sort that its syntax predicts. For instance, $6.2/3.1\downarrow$ is equivalent to $6.2/3.1 \downarrow \mathbf{R}$. Thus by a convergence requirement we mean a formula of the form $t \downarrow \alpha$ or $t\downarrow$ which must be true to justify a manipulation.

The notion of quasi-equality serves as the correctness requirement for the simplifier. We say that s and t are quasi-equal, written $s \simeq t$, if

$$(s\downarrow \vee t\downarrow) \supset s = t.$$

Since in the IMPS logic $s = t$ can hold only if both terms are defined, $s \simeq t$ holds just in case the terms are either both undefined, or both defined with the same value. In IMPS, quasi-equality justifies substituting t in place of s at any occurrence at which t is free for s.

The simplifier is designed to transform an expression e to e' relative to the assumptions of a context Γ (in a theory $\mathcal{T}$) only if Γ entails $e \simeq e'$.

The algorithm traverses the expression recursively; as it traverses propositional connectives it does simplification with respect to a richer context. Thus, for instance, in simplifying an implication $A \supset B$, A may be assumed true in the "local context" relative to which B is simplified. Similarly, in simplifying the last conjunct C of a ternary conjunction $A \wedge B \wedge C$, A and B may be assumed in the "local context." On the other hand, when a variable-binding operator is traversed, and there are context assumptions in which the bound variable occurs free, then the simplifier must either rename the bound variable or discard the offending assumptions. The strategy of exploiting local contexts is justified in [13].

At any stage in this recursive descent, if a theory-specific procedure may successfully be used to transform the expression, it is applied. These procedures currently include:

- Algebraic simplification of polynomials, relative to a range of algebraic theories (see Section 3);
- A decision procedure for linear inequalities, based on the variable elimination method used in many other theorem provers, for instance by Boyer and Moore [1];
- Rewrite rules for the current theory $\mathcal{T}$, or for certain theories $\mathcal{T}_0$ for which IMPS can find interpretations of $\mathcal{T}_0$ in $\mathcal{T}$.

The procedures for algebraic simplification have been coded so as to record the convergence requirements for all the transformations they make. When the procedure returns, the simplified value is used only if the requirements can be discharged by a recursive invocation of the simplifier.

Rewrite rules also generate convergence requirements. Suppose that we have a theorem such as for instance

$$\forall x, y : \mathbf{Z} \, . \, x < y \Leftrightarrow x + 1 \leq y$$

which is being used as a rewrite rule from left to right. If a portion of an expression being simplified is of the form $s < t$, then we would like it to be rewritten to $s + 1 \leq t$, but only if s and t both take values in $\mathbf{Z}$. If one of the terms is undefined, or if it has a value in $\mathbf{R}$ but not in $\mathbf{Z}$, then the change is not justified as an instance of the theorem.

If no transform is applicable, then a simplification routine determined by the top-most constructor or quasi-constructor of the expression to be simplified is applied. These routines normally invoke the simplifier recursively on subexpressions, with different contexts. The routines for a few constructors, especially the definedness constructors (Section 2.1), use special routines exploiting information extracted from the axioms and theorems of the context's theory.

The emphasis on a powerful simplification procedure to allow large inference steps in the course of interactive proof development is shared with Eves [4] and its predecessor m-Eves [3], as well as the more recent PVS [14].

2.1 Reasoning about Definedness

Because simplification and theorem application involve large numbers of convergence requirements, it is important to automate, to the greatest extent possible, the process of checking that expressions are well-defined or defined with a value in a particular sort. This kind of reasoning must rely heavily on axioms and theorems of the axiomatic theory at issue. The algorithm for simplifying definedness assertions is separated into two layers, according to whether recursive calls to the simplifier are involved.

The Lower Level of Definedness Checking. In the lower level, there are no recursive calls to the simplifier; two kinds of information are used:

- *Totality* theorems of the form $\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, f(x_1, \ldots, x_n) \downarrow \alpha$.
- *Unconditional sort coercions* of the form $\forall x : \alpha \,.\, x \downarrow \beta$.

A set S of unconditional sort coercion theorems determine a pre-order $\ll_S$ on sorts. $\alpha \ll_S \beta$ if and only if in every model of S, the denotation of α is included in the denotation of β. The relation $\ll_S$ is a pre-order rather than a partial order because for two different syntactic sorts α and β, we may have $\alpha \ll_S \beta$ and $\beta \ll_S \alpha$; in this case α and β have the same denotation in every model of S. Fix some collection S of axioms and theorems of $\mathcal{T}$, with respect to which definedness checking is being carried out.

The relation $\ll_S$ together with the totality theorems are used in IMPS by an algorithm for checking definedness. We use totality information and unconditional sort coercions to extract "critical pairs of subterms and sorts," or simply critical pairs, from a term t and a sort α. By a set of critical pairs, we mean a set of pairs $\langle s_i, \beta_i \rangle$ such that:

- Each s_i is a subterm of t, and
- If $s_i \downarrow \beta_i$ holds for each i, then $t \downarrow \alpha$.

In particular, if the null set is a set of critical pairs for t and α, then $t \downarrow \alpha$ is true. Naturally, $\{\langle t, \alpha \rangle\}$ is always a set of critical pairs for t and α. More useful sets of critical pairs may be computed for many expressions using two main principles:

- Suppose that $C \cup \{\langle s_i, \beta_i \rangle\}$ is a set of critical pairs, where s_i is a variable, constant, or λ-expression, and γ is its syntactically declared sort. If $\gamma \ll_S \beta_i$, then $s_i \downarrow \beta_i$ is patently true, so C is also a set of critical pairs.
- Suppose that $\gamma \ll_S \alpha$, t is an application $f(a_1, \ldots, a_n)$, and S contains

  $$\forall x_1 : \beta_1, \ldots, x_n : \beta_n \,.\, f(x_1, \ldots, x_n) \downarrow \gamma.$$

  If C_i is a set of critical pairs for a_i and β_i, then $\bigcup_i C_i$ is a set of critical pairs for t and α.

These principles mechanize definedness checking for a fragment of the IMPS logic that corresponds to order sorted theories in higher order logic [12].

Frequently, a set of critical pairs will be relatively small, even if it is nonnull. Moreover, the terms it contains may be far smaller than t. For instance, consider the term

$$t = (i + j - k) \cdot (i - j + k) \cdot (i - k + j/2)$$

where k, j, i range over the integers $\mathbf{Z}$, and all of the function symbols denote the usual binary functions on the reals. The only critical pairs for t to be defined among the rationals $\mathbf{Q}$ is $\langle j/2, \mathbf{Q}\rangle$. In this case, we would like to combine the results of the lower level with the fact that

$$\forall p, q : \mathbf{Q} \,.\, q \neq 0 \supset p/q \downarrow \mathbf{Q}.$$

For this reason, the results of the lower level of definedness checking are passed to the upper layer, which uses this sort of conditional information.

The Upper Level of Definedness Checking In this layer, conditional information about definedness is consulted. The simplifier is invoked on the resulting assertions, in an attempt to reduce them to `truth`. This layer uses two primary kinds of information about the domain and range of functions, and the relations between sorts, in the theory.

- *Definedness conditions* of the form

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, \psi(x_1, \ldots, x_n) \supset f(x_1, \ldots, x_n) \downarrow \alpha.$$

- *Conditional sort coercions* of the form $\forall x : \beta \,.\, \varphi(x) \supset x \downarrow \alpha$.

To check the definedness of a term $f(t_1, \ldots, t_n)$ in sort α, we look for a definedness condition for f and α. If successful, we simplify the new goal $\psi(t_1, \ldots, t_n)$.

Alternatively, we look next for a sort coercion condition $\forall x : \beta \,.\, \varphi(x) \supset x \downarrow \alpha$, where β is the syntactic sort of $f(t_1, \ldots, t_n)$ (i.e., the declared range of f). If successful, we call the simplifier on the assertion $(\lambda x : \beta \,.\, \varphi)(f(t_1, \ldots, t_n))$. This is logically equivalent to the conjunction $f(t_1, \ldots, t_n) \downarrow \beta \wedge \varphi[f(t_1, \ldots, t_n)/x]$. The first conjunct may require a recursive call for definedness checking.

The assertions that, in IMPS, are expressed using partial functions and subtypes can also be expressed, more cumbrously, in ordinary simple type theory. Nevertheless, the machinery of subtypes and definedness assertions helps to guide IMPS's automated support. It provides syntactic cues that the reasoning embodied in these algorithms is likely to be useful. One might initially think that partial functions and subtyping would introduce restrictions that would make sound, effective simplification infeasible. Yet the IMPS simplifier is able to work effectively, partly because the necessary checking is guided by the syntactically explicit definedness operators of the logic.

3 Algebraic Processors

The IMPS simplifier makes repeated calls to procedures called transforms. For this discussion, a transform can be thought of as a function T on pairs (Γ, e), where Γ is a context and e is an expression, which returns two values: an expression $T(\Gamma, e)$ and a set of formulas $\{c_1(\Gamma, e), \ldots, c_n(\Gamma, e)\}$ called convergence requirements. The significance of this set of formulas is explained below.

The two kinds of transforms used by the IMPS simplifier are rewrite rule application and algebraic manipulation. Since any universally quantified equality can be used as a rewrite rule, the framework for applying rewrite rules is entirely general, and uses pattern matching and substitution in a familiar way. By contrast, the transforms that perform algebraic manipulation use specially coded procedures, and they are applied to expressions in a way that may not be easily expressed as patterns. Nevertheless, their validity, like the validity of rewrites, depends on theorems—many of which are universal, unconditional equalities (associative and commutative laws, for instance).

Two issues must be balanced in providing algebraic manipulation using specially coded procedures. On the one hand, ad hoc implementations have the benefit of allowing optimizations not possible with more generic procedures, and they may be coded in a rigorously correct way for particular kinds of structure. However, they are limited in scope: to provide similar computational facilities for a new kind of algebraic structure such as fields or vector spaces would require adding more special-purpose code to the reasoning machinery. With this approach, users who dealt with theories of an algebraic nature would either write new low-level code for algebraic simplification themselves (for example, using source code provided with the system as a model) or do without algebraic simplification altogether.

We have attempted to maintain the advantages of special-purpose algebraic manipulation while providing easy extensibility to a wide variety of algebraic structures. Instead of providing a fixed set transforms for manipulating expressions in a limited class of algebraic structures, we have implemented a facility for automatically generating and installing such transforms for general classes of algebraic structures. This is possible, since algorithmically the transforms are the same in many cases; only the names have to be changed, so to speak. The algebraic manipulation transform is one component of a data structure called an *algebraic processor.*

An algebraic processor has either two or three associated transforms. There is one transform for handling terms of the form $f(a,b)$ where f is one of the algebraic operations in the processor definition. A separate transform handles equalities of algebraic expressions. Frequently, when the structure has an ordering relation, there is a third transform for handling inequalities of algebraic expressions.

A transform T is valid if and only if for every context Γ and expression e the following holds:

- If all the convergence requirements $c_1(\Gamma,e),\cdots,c_n(\Gamma,e)$ are valid, then $T(\Gamma,e)\simeq e$.

- If any of the convergence requirements is false, then $T(\Gamma,e)$ is undefined.

When an algebraic processor is created, the system generates a set of formulas which must hold in order for its manipulations to be valid. The processor's transforms are installed in a theory $\mathcal{T}$ only if the system can ascertain that each of these formulas is a theorem of $\mathcal{T}$.

The user builds an algebraic processor using a specification such as

```
(def-algebraic-processor FIELD-ALGEBRAIC-PROCESSOR
  (language fields)
  (base ((operations
          (+ +_kk)
          (* *_kk)
          (- -_kk)
          (zero o_kk)
          (unit i_kk))
         commutes)))
```

This specification has the effect of building a processor with a transform for handling terms of the form $f(a, b)$, where f is one the arithmetic functions $+_{\mathbf{K}}, *_{\mathbf{K}}, -_{\mathbf{K}}$. For example, the entry `(+ +_kk)` tells the transform to treat the function symbol $+_{\mathbf{K}}$ as addition. The transforms are installed in the theory *Fields* by the specification

```
(def-theory-processors FIELDS
  (algebraic-simplifier
    (field-algebraic-processor *_kk +_kk -_kk))
  (algebraic-term-comparator field-algebraic-processor))
```

When this form is evaluated, IMPS checks that the conditions for doing algebraic manipulation do indeed hold in the theory *Fields*.

4 Computing with Theorems

In addition to its simplifier, IMPS also provides a separate mechanism for manipulating formal expressions through applying theorems. It is used when simplification does not go far enough or does the wrong thing. Unlike simplification, it is under the direct or indirect control of the user. The basic idea is that expression manipulation rules—for instance, conditional rewrite rules—are represented as theorems. The theory/context mechanism of IMPS provides a way of making sure that rules represented as theorems are applied correctly in as wide a scope as possible.

Application. How a theorem is applied is determined by its syntactic form. For example, consider the theorem $\forall x : \mathbf{R} \,.\, 0 \leq x \supset |x| = x$ which is proved in a theory $\mathcal{R}$ of the real numbers. This theorem has the form of a conditional rewrite rule; the rule replaces the absolute value of a nonnegative number with itself. Suppose we would like to apply this theorem to a subexpression e' of an expression e in a context in a theory $\mathcal{T}$. This can done in two ways.

The first and simplest way can be used when the home theory $\mathcal{R}$ of the theorem is a subtheory of $\mathcal{T}$; it is based on ordinary expression matching. First, e' is matched to the left hand side $|x|$ of the equation. If there is a substitution, say matching t with x, the simplifier is called to check whether $t \downarrow \mathbf{R}$. If it succeeds, then it is applied to the condition $0 \leq t$. If this also succeeds, e' is then replaced by t in this case, or generally by the instance of the right hand side of the equation under the substitution. If any one of the tests fails, e' is left unchanged.

Alternatively, if $\mathcal{R}$ is not a subtheory of $\mathcal{T}$, "translation matching" is used instead of ordinary matching. An expression e_1 in a theory $\mathcal{T}_1$ *translation matches* an expression e_2 in a theory $\mathcal{T}_2$ if there is a theory interpretation Φ of $\mathcal{T}_1$ in $\mathcal{T}_2$ and a substitution σ such that σ matches the translation of e_1 via Φ to e_2. Translation matching allows theorems to be applied outside of their home theory.

For theorems of different syntactic form, the behavior of theorem application differs. For instance, a form of the triangle inequality for metric spaces,

$$\forall x, y, z : \mathbf{P}, r : \mathbf{R} \,.\, \mathrm{dist}(x, z) + \mathrm{dist}(y, z) \leq r \supset \mathrm{dist}(x, y) \leq r,$$

causes a formula of the form $\mathrm{dist}(x, y) \leq r$ to be replaced by the corresponding existential assertion $\exists z : \mathbf{P} \,.\, \mathrm{dist}(x, z) + \mathrm{dist}(y, z) \leq r$. In a case such as this, in which an implication (rather than an equality or a biconditional) is being used, the theorem provides a form of backchaining. Thus, it is legitimate only when the target formula occurs positively in the goal being proved.[1]

Installation. In IMPS theorems are usually installed by the user (but in some cases theorems are installed automatically by the system). To install a theorem in a theory $\mathcal{T}$, the user has to supply a proof of the theorem in $\mathcal{T}$. Once the theorem is installed, it can be used in any theory containing $\mathcal{T}$ as well as any theory which contains the structure of $\mathcal{T}$ via a theory interpretation. Thus the user can add new expression manipulation rules to the system by simply proving and installing theorems.

Programming. Theorems are actually applied in IMPS by means of procedures called *macetes*. When a theorem is installed, the IMPS system automatically makes an *elementary macete* which will apply the theorem via ordinary matching. If requested by the user, the system will also make a *transportable macete* which will apply the theorem via translation matching.

[1] To define the notion of a positive occurrence, consider the ordering of the truth values in which $\mathbf{F}$ is less than $\mathbf{T}$. A location in a formula is positive if the formula is a monotonically nondecreasing function of the truth value of the subformula at that location. It is negative if it is monotonically nonincreasing. A location may also be neither positive nor negative. A syntactic approximation is easily defined by induction on the structure of formulas; for instance, if A occurs positively [negatively; neither] in B, then it occurs negatively [positively; neither] in $\neg B$ or $B \supset C$.

IMPS contains a rudimentary programming language for building more complex macetes. A *compound macete* is ultimately constructed from elementary and transportable macetes, together with a few special macetes such as beta-reduction and simplification, using a small number of *macete constructors*, which are just functions from macetes to macetes. Compound macetes provide a simple mechanism to apply a collection of theorems in an organized manner. Macetes are usually invoked directly by the user; the user decides what macete is applied on what subexpressions of the goal formula.

As an illustration, consider the compound macete specified by

```
(def-compound-macete REMOVE-ABSOLUTE-VALUES
  (repeat
    absolute-value-non-negative
    absolute-value-non-positive))
```

where absolute-value-non-negative and absolute-value-non-positive are, respectively, the names of the elementary macetes created from the theorem mentioned above and the theorem $\forall x : \mathbf{R} \,.\, x \leq 0 \supset |x| = -x$. This macete removes as many absolute value signs as possible from an expression.

Retrieval. Since theorem application via macetes is under the direct control of the user, it is important that the system have a facility for finding the macetes that are applicable to a particular goal. In IMPS this is done by applying each available macete to the goal using fast matching algorithms. In situations where there are over 500 macetes, the system rarely presents the user with more than 10 macetes as options. Any macete that is not presented is certainly not applicable. However, because the fast matching algorithms are not guaranteed to be sound, some of the presented macetes may not do anything when they are applied with the normal sound matching algorithms.

5 Example

To illustrate some of the preceding ideas, we present in this section an example of an IMPS proof.

Each theorem specification has three important components:

- The statement of the theorem, which is a formula in a formal language.
- The theory in which the theorem is valid. A theorem is *valid in* a theory if it is a semantic consequence of the set of assumptions encapsulated in the theory.
- The proof of the theorem. The proof of a theorem is a sequence of proof commands and control statements called a *script*. This script is executed prior to installing the theorem.

For example, the following entry installs the combinatorial identity as a theorem (called comb-ident).

```
(def-theorem COMB-IDENT
  "forall(k,m:zz,
     1<=k and k<=m implies comb(1+m,k)=comb(m,k-1)+comb(m,k))"
  (theory h-o-real-arithmetic)
  (proof
   ((unfold-single-defined-constant-globally comb)
    (apply-macete-with-minor-premises
      fractional-expression-manipulation)
    (label-node compound)
    direct-and-antecedent-inference-strategy
    (jump-to-node compound)
    (for-nodes
     (unsupported-descendents)
     (if (matches? "with(t:rr, #(t^[-1]))")
         (apply-macete-with-minor-premises
           definedness-manipulations)
         (block
          (apply-macete-with-minor-premises
            factorial-reduction)
          simplify))))))
```

The proof of the combinatorial identity has five main steps:

1. Expand the definition of `comb`, which in IMPS is given in terms of the factorial function. The resulting sequent consists of a context containing the formulas $k \leq m$ and $1 \leq k$ and the assertion

$$\frac{(1+m)!}{k! \cdot (1+m-k)!} = \frac{m!}{(k-1)!\,(m-(k-1))!} + \frac{m!}{k!\,(m-k)!}.$$

2. Apply the macete fractional-expression-manipulation to the sequent above. Among other things, this multiplies out denominators in equalities and inequalities. This expansion is not done by the simplifier since the necessary computations require extensive branching on cases (for instance, depending on the signs of denominators.) Since the simplifier is called at all levels of the IMPS system, making these reductions universally would be computationally too expensive.

3. Apply the command direct-and-antecedent-inference-strategy to the result of previous step. This reduces the proof to proving three sequents: two concerning definedness and one whose assertion is an equation.

4. For the sequents concerning definedness, use the macete definedness-manipulations. This combines a number of facts about the definedness of the arithmetic operations and the factorial function. One of these two sequents, for example, consists of a context containing $k \leq m$ and $1 \leq k$ and an assertion which says $(k! \cdot (1 + m + (-1) \cdot k)!)^{-1}$ is defined.

5. For the remaining sequent, use the macete factorial-reduction which combines the following two theorems:

   - $\forall m : \mathbf{Z} \, . \, 1 \leq m \supseteq m! = (m-1)! \, m$
   - $0! = 1$

The finished proof has the following characteristics:

- Number of sequents created by the proof: 14.
- Number of contexts created during the proof: 79.
- Average number of assumptions (other than axioms) per context: 5.61.

The current IMPS theory library contains approximately 1100 theorems with proofs. The context information for these proofs can be summarized as follows:

- Average number of sequents created by a proof: 24.83.
- Average number of contexts created during a proof: 22.63.
- Average number of assumptions (other than axioms) per context: 2.25.

6 Implementing a General Mathematics Laboratory

Though systems like MATHEMATICA and Maple are extremely powerful and useful, there is a great deal of mathematical activity they do not support [2]. The following is a list of facilities (most of which are from [2]) which a general mathematics laboratory that can claim to support mathematical activity should provide:

- Context-sensitive facilities for reducing terms. This means that the system can utilize contextual information whenever required.
- A mechanism for building and accessing a database of mathematical theorems.
- Mechanisms for making simple inferences; in the IMPS terminology these include:

– Unfolding definitions.
– Instantiating universal assumptions.
– Instantiating existentials.
– Computing with theorems.

In addition to providing these facilities, a general mathematics laboratory should be mathematically rigorous. This is an essential requirement, since otherwise symbolic computations which cannot be checked by hand cannot be used in a mathematical proof.

A general mathematics laboratory of the kind we are describing should ideally be designed and built from scratch in such a way that the computational and inference machinery are fully integrated. However, building such a system would be a major effort requiring sizable human and financial resources. Nevertheless, we believe that it is is possible to build a system satisfying most of the above requirements by appropriately linking a theorem proving system such as IMPS to an existing computer algebra system. In the following paragraphs we describe how such a linkage might be implemented in a straightforward way.

6.1 Rigorous Computations

As explained earlier (see also [10]), the macete mechanism is intended as a facility for computation. Macetes are built from atomic macetes using macete constructors, where atomic macetes correspond either to theorems or one of a number of special case macetes such as simplification or beta-reduction. The class of atomic macetes can include procedures which are not necessarily meaning preserving. In order to use one of these procedures in a mathematically rigorous way for computation, the macete must be used with the *sound* macete constructor as explained below.

One mechanism that can be used to augment the functionality of the macete mechanism is to allow atomic macetes to include special calls (via some interprocess communication mechanism) to a computer algebra system running concurrently. We will refer to macetes of this kind as *external-call macetes.* Moreover, using the sound macete constructor, we need not assume these external-call macetes are themselves meaning preserving. This macete constructor can be used to wrap any macete with a check to ascertain that the transformed expression can legitimately replace the original one. Typically the check involves using simplification or even beta-reduction. Many computations, such as factorization of polynomials or anti-differentiation, can easily be checked in IMPS by compound macetes.

As an example, suppose we had an external-call macete prime-factorization-external which takes an expression of sort **R** and by a call to a computer algebra system returns a factored expression. Then, we could build a sound macete as follows:

```
(def-compound-macete PRIME-FACTORIZATION
  (sound
    prime-factorization-external
    simplify
    simplify))
```

When the macete is applied an expression e in a given context, prime-factorization-external returns an expression e'. If e and e' simplify to the same expression, then we know that the denotation of e and e' is identical. In this case the macete prime-factorization returns e'; otherwise, it just returns e.

6.2 Use of Contextual Information

Many algorithms in symbolic mathematics require that certain expressions be fully evaluated in order to determine the branch of a computation. A simple example is integration of a monomial x^n, in which the truth value of the formula $n = -1$ must be known. As described earlier in the paper, the context mechanism in IMPS is tailored to organize and manipulate this information for use in mathematical proofs. Again by an appropriate linkage, IMPS can be utilized as a context manager for a computer algebra system. Thus a computer algebra system can request information about an expression in a specific context (such as the value of $n = -1$) from IMPS instead of requesting it directly from the user. Since the requested information may not be immediately available as a context entry, IMPS may need to perform additional computation in order to answer the request.

It is desirable that the link between a computer algebra system and IMPS (or any other appropriate theorem proving system) should allow for recursive calls between the two components. For instance, in answering a request from a computer algebra system, IMPS may need to use an external-call macete. Unfortunately, the possibility of making these recursive calls may lead to nonterminating computations. For example, a recursive call from IMPS to the computer algebra system may be identical to the original computation requested from the computer algebra system. The likelihood of this kind of nonterminating behavior can be reduced somewhat by keeping a table of previously requested values.

7 Conclusion

Systematic use of contexts is desirable for symbolic computation, and the IMPS system demonstrates that they are feasible to implement. Indeed, a context determines a class of identities that may be validly applied, and the constrained use of such theorems is in effect the heart of symbolic computation. IMPS uses these ideas in order to unify some of the strengths of theorem provers with symbolic computation.

References

[1] R. S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. Technical Report ICSCA-CMP-44, Institute for Computing Science, University of Texas at Austin, January 1985.

[2] B. Buchberger. *Mathematica:* a system for doing mathematics by computer? Preprint, 1993.

[3] D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, B. Pase, and M. Saaltink. m-EVES: A tool for verifying software. In *11th International Conference on Software Engineering (ICSE'11)*, Singapore, 1988.

[4] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. Eves system description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 771–775. Springer-Verlag, 1992.

[5] W. M. Farmer. Theory interpretation in simple type theory. In *Proceedings, International Workshop on Higher Order Algebra, Logic and Term Rewriting (HOA '93), Amsterdam, The Netherlands, September 1993*, Lecture Notes in Computer Science. Springer-Verlag. Forthcoming.

[6] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–91, 1990.

[7] W. M. Farmer. A simple type theory with partial functions and subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[8] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: System description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.

[9] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[10] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[11] J. D. Guttman. A proposed interface logic for verification environments. Technical Report M91-19, The MITRE Corporation, 1991.

[12] M. Kohlhase. Unification in order-sorted type theory. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 421–432. Springer-Verlag, July 1992.

[13] L. G. Monk. Inference rules using local contexts. *Journal of Automated Reasoning*, 4:445–462, 1988.

[14] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.