IMPS: An Interactive Mathematical Proof System*

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

The MITRE Corporation

9 February 1993

Abstract

IMPS is an Interactive Mathematical Proof System intended as a general purpose tool for formulating and applying mathematics in a familiar fashion. The logic of IMPS is based on a version of simple type theory with partial functions and subtypes. Mathematical specification and inference are performed relative to axiomatic theories, which can be related to one another via inclusion and theory interpretation. IMPS provides relatively large primitive inference steps to facilitate human control of the deductive process and human comprehension of the resulting proofs. An initial theory library containing almost a thousand repeatable proofs covers significant portions of logic, algebra and analysis, and provides some support for modeling applications in computer science.

Key Words: interactive theorem proving, automated analysis, computing with theorems, theory interpretation, higher-order logic, partial functions

*Supported by the MITRE-Sponsored Research program and to appear in the *Journal of Automated Reasoning*. Authors' address: The MITRE Corporation, 202 Burlington Rd, Bedford MA 01730-1420 USA; Telephone: 617-271-2749; Email: farmer, guttman, jt@mitre.org.

1 Introduction

The primary goal of IMPS, an Interactive Mathematical Proof System, is to provide mechanized support for the traditional techniques of mathematical reasoning. The system consists of a data base of mathematics (represented as a collection of interconnected axiomatic theories) and a collection of tools for exploring, applying, and extending the mathematics in the data base. IMPS is distinguished by its logic, its methodology for formalizing mathematics, and its style of proof.

- *Logic.* The IMPS logic is intended to allow the user to formulate mathematical concepts and arguments in a natural and direct manner. It is a simple type theory with strong support for specifying and reasoning about functions. Unlike classical logic, functions may be partial and terms may be nondenoting, but formulas always have a standard truth value. Section 2 describes the IMPS logic.

- *Methodology.* Mathematics is formalized in IMPS as a network of axiomatic theories. The theories in the network are linked together by theory interpretations which serve as conduits to pass results from one theory to another. This way of formalizing mathematics—the "little theories" version of axiomatic method—has advantages for mechanized mathematics [19]. In particular, it fosters the reuse of theories and their constituents. Section 3 discusses the little theories approach in IMPS.

- *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 the details of an IMPS proof are machine checked. IMPS emphasizes interactive proof development. There is essentially no structural difference between completed proofs and partial proof attempts. The development of proofs in IMPS is the subject of Section 4.

The remaining sections of the paper discuss the IMPS user interface (Section 5), the use of IMPS for mathematical analysis (Section 6), and the IMPS mathematics data base (Section 7). A brief conclusion is given in Section 8.

2 Logic

The logic[1] of IMPS is called LUTINS[2], a Logic of Undefined Terms for Inference in a Natural Style. LUTINS is a conceptually simple implementation of higher-order predicate logic that closely conforms to mathematical practice. Partial functions are dealt with directly, and consequently, terms may be nondenoting. The logic, however, is bivalent; formulas are either true or false.

LUTINS is derived from the formal system $\mathbf{PF}^*$ [17], which in turn is derived from the formal system **PF** [16]. **PF** is a version of Church's simple theory of types [7, 1] in which functions may be partial, and $\mathbf{PF}^*$ is a many-sorted, multivariate simple type theory with partial functions, subtypes, and definite description operators. It is shown in [16] and [17] that **PF** and $\mathbf{PF}^*$, respectively, are complete with respect to a Henkin-style general models semantics [29]. LUTINS is essentially $\mathbf{PF}^*$ plus a number of convenient expression constructors, which are discussed below. The formal semantics of LUTINS is straightforwardly derived from the (standard models) semantics of $\mathbf{PF}^*$ in [17]. (See [28] for a detailed description of the syntax and semantics of LUTINS.)

2.1 Higher-Order Functions and Types

Higher-order logic (or type theory) was developed in the early part of this century to serve as a foundation for mathematics, but lost its popularity as a foundation for mathematics in the 1930's with the rise of set theory and first-order logic. Higher-order logic emphasizes the role of functions, in contrast to set theory, which emphasizes the role of sets. In type theory, functions

[1] By a *logic*, we mean in effect a function. Given a particular vocabulary, or set of (nonlogical) constants, the logic yields a triple consisting of a formal language $\mathcal{L}$, a class of models $\mathcal{A}$ for the language, and a satisfaction relation $\models$ between models and formulas. The function is normally determined by the syntax and semantics of a set of logical constants for the logic.

The satisfaction relation determines a *consequence* relation between sets of formulas and individual formulas. A formula P is a consequence of a set of formulas S if $\mathcal{A} \models P$ holds whenever $\mathcal{A} \models Q$ holds for every $Q \in S$.

When we speak of a *theory*, we mean in essence a language together with a set of axioms. A formula is a *theorem* of the theory if it is a consequence of the axioms.

[2] Pronounced as the word in French.

may be quantified and may take other functions as arguments. In order to avoid circularity, functions are organized according to a type hierarchy.

Type theory has a uniform syntax; it is based on familiar notions; and it is highly expressive. The use of λ-notation allows functions to be specified succinctly. Since type theory contains second-order logic, there are many things that can be expressed in it which cannot be directly expressed in first-order logic. For example, the induction principle for the natural numbers can be expressed completely and naturally by a single second-order formula. (See [4, 45] for discussion on the expressive power of second-order logic relative to first-order logic.)

The type hierarchy of LUTINS consists of base types and function types. Let $\mathcal{L}$ be a language in LUTINS. The *base types* of $\mathcal{L}$ are the type of propositions `prop` and $m \geq 1$ types of individuals.[3] The *function types* of $\mathcal{L}$ are inductively defined from its base types: if $\alpha_1, \ldots, \alpha_n, \alpha_{n+1}$ are (base or function) types where $n \geq 1$, then $\alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1}$ is a function type. Since m and n may be strictly greater than 1, the type structure is "many-sorted" and "multivariate," respectively.

A higher-order logic with this sort of type hierarchy is called a *simple type theory*. The automatic theorem proving system TPS developed at CMU [2], the proof development system HOL developed at the University of Cambridge [25], and the EHDM and PVS verification systems developed at SRI International [42, 39] are also based on simple type theories. However, in these systems function types contain only total functions, while in LUTINS, some types may contain partial functions. These are the *types of kind ind*. We say that a type α is of *kind ind* (or ι) if α is a base type of individuals or $\alpha = \alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1}$ and α_{n+1} is of kind ind. Otherwise, we say that α is of *kind prop* (or $*$).

When a function type α is known to be of kind ind, we prefer to write it in the form $\alpha_1, \ldots, \alpha_n \rightharpoonup \alpha_{n+1}$ instead of $\alpha_1, \ldots, \alpha_n \rightarrow \alpha_{n+1}$. This emphasizes that α contains partial functions as well as total functions.

Every formal expression in LUTINS has a unique type. The type of an expression serves both a semantic and syntactic role: An expression denotes an object in the denotation of its type (if the expression is defined), and the syntactic well-formedness of an expression is determined on the basis of the

[3] We will refer to these types by the names `ind`, $\texttt{ind}_1$, etc., although of course they may be given any convenient names.

types of its components. An expression is said to be of *kind ind* [respectively, *prop*] if its type is of kind ind [respectively, prop]. Expressions of kind ind are used to refer to mathematical objects; they may be undefined. Expressions of kind prop are primarily used in making assertions about mathematical objects; they are always defined.

2.2 Partial Functions

One of the primary distinguishing characteristics of LUTINS is its direct approach to specifying and reasoning about partial functions (i.e., functions which are not necessarily defined on all arguments). Partial functions are ubiquitous in both mathematics and computer science. If a term is constructed from simpler expressions by the application of an expression denoting a partial function f to an expression denoting an argument a which is outside the domain of f, then the term itself has no natural denotation. Such a term would violate the *existence assumption* of classical logic, which says that terms always have a denotation. Thus a direct handling of partial functions can only lie outside of classical logic.[4]

The semantics of LUTINS is based on five principles:

(1) Variables, constants, and λ-expressions always have a denotation.

(2) Expressions of type `prop` always denote a standard truth value.

(3) Expressions of kind ind may denote partial functions.

(4) An application of kind ind is undefined if its function or any of its arguments is undefined.

(5) An application of type `prop` is false if any of its arguments is undefined.

As a consequence of these principles, expressions of kind prop must be denoting. We have chosen this approach for dealing with partial functions because it causes minimal disruption to the patterns in reasoning familiar

[4]However, since the graph of a function (partial or total) can always be represented as a relation, the problem of nondenoting terms can in theory be easily avoided—at the cost of using unwieldy, verbose expressions packed with existential quantifiers. Hence, *if pragmatic concerns are not important*, classical logic is perfectly adequate for dealing with partial functions.

from classical logic and standard mathematical practice. (For a detailed discussion of various ways of handling partial functions in predicate logic, see [16].)

2.3 Constructors

The expressions of a language of LUTINS are constructed from variables and constants by applying *constructors*. Constructors serve as "logical constants" that are available in every language. LUTINS has approximately 20 constructors. (**PF** and **PF*** have only two constructors, application and λ-abstraction.) Logically, the most basic constructors are `apply-operator`, `lambda`, `iota`, and `equals`; in principle every expression of LUTINS could be built from these four.[5] The other constructors serve to provide economy of expression.

There is a full set of constructors for predicate logic: constants for true and false, propositional connectives, equality[6], and universal and existential quantifiers. LUTINS also has an definite description operator ι, an if-then-else operator `if`, and definedness constructors `is-defined` (denoted by the postfix symbol $\downarrow$) and `defined-in` ($\downarrow$ in between an expression and a sort). Although a few constructors (such as `implies` (infix $\supset$) and `not` ($\neg$)) correspond to genuine functions, most constructors do not. For example, the constructor `if` is nonstrict in its second and third arguments (e.g., the expression $\text{if}(0 = 0, 0, 1/0)$ is defined in a theory of arithmetic even though $1/0$ is undefined). Four constructors bind variables: `forall` ($\forall$), `forsome` ($\exists$), ι, and λ, the basic variable-binding constructor.

The ι constructor, the definite description operator of LUTINS, is a constructor that cannot be easily imitated in other logics. Using this constructor, one can create a term of the form $\iota x.P(x)$, where P is a predicate, which denotes the unique element described by P. More precisely, $\iota x.P(x)$ denotes the unique x that satisfies P if there is such an x and is undefined otherwise. In addition to being quite natural, this kind of definite description operator is

[5]Throughout this paper, constructors will be denoted using traditional symbology. For example, `lambda` and `iota` are denoted, respectively, by the variable-binding symbols λ and ι; `equals` is denoted by the usual infix symbol $=$; and `apply-operator` is denoted implicitly by the standard notation of function application.

[6]The meaning of the formula $s = t$ is that s and t denote the same value, and so $0/0 = 0/0$ is a false assertion.

very useful for specifying (partial) functions. For example, ordinary division (which is undefined whenever its second argument is 0) can be defined from the times function $*$ by a λ-expression of the form

$$\lambda x, y \,.\, \iota z \,.\, x * z = y.$$

In logics in which terms always have a denotation, there is no completely satisfactory way to formalize a definite description operator (see B. Russell's attempt [43]). This is because a definite description term $\iota x.P(x)$ must always have a denotation, even when there is no unique element satisfying P.

The IMPS implementation allows one to create macro/abbreviations called *quasi-constructors* which are defined in terms of the ordinary constructors. For example, the quasi-constructor `quasi-equals` (infix $\simeq$) is defined by the following biconditional:

$$e_1 \simeq e_2 \;\equiv\; (e_1\!\downarrow \vee e_2\!\downarrow) \supset e_1 = e_2.$$

A quasi-constructor is used in two different modes: as a device for constructing expressions with a common form and as if it were an ordinary constructor. The first mode is needed for proving basic theorems about quasi-constructors, while the second mode effectively gives the user a logic with a richer set of constructors. Quasi-constructors can be especially useful for formulating generic theories (e.g., a theory of finite sequences) and special-purpose logics within IMPS.

Constructors and quasi-constructors are polymorphic in the sense that they can be applied to expressions of several different types. For instance, the constructor `if` can take any three expressions as arguments as long as the type of the first expression is `prop` and the second and third expressions are of the same type.

2.4 Sorts

Superimposed on the type hierarchy of LUTINS is a system of subtypes. We call types and subtypes jointly *sorts*. The sort hierarchy consists of atomic sorts and compound sorts. Let $\mathcal{L}$ be a language in LUTINS. $\mathcal{L}$ contains a set of *atomic sorts* which includes the base types of $\mathcal{L}$. The *compound sorts* of $\mathcal{L}$ are inductively defined from the atomic sorts of $\mathcal{L}$ in the same way that function types of $\mathcal{L}$ are defined from the base types of $\mathcal{L}$. Every atomic sort α is assigned an *enclosing sort* $\xi(\alpha)$. $\preceq$ is the least reflexive, transitive binary relation on the sorts of $\mathcal{L}$ such that:

- If α is an atomic sort, then $\alpha \preceq \xi(\alpha)$.
- If $\alpha_1 \preceq \beta_1, \ldots, \alpha_{n+1} \preceq \beta_{n+1}$, then $\alpha_1, \ldots, \alpha_n \to \alpha_{n+1} \preceq \beta_1, \ldots, \beta_n \to \beta_{n+1}$.

It follows from the definition of a LUTINS language that the enclosing sort function ξ satisfies three properties:

- The enclosing sort of a base type is itself.
- The enclosing sort of an atomic sort is of kind prop iff the atomic sort is itself `prop`.
- $\preceq$ is noetherian, i.e., every ascending sequence of sorts,
$$\alpha_1 \preceq \alpha_2 \preceq \alpha_3 \preceq \cdots,$$
is eventually stationary.

These properties imply that:

- $\preceq$ is a partial order.
- For all sorts α, there is a unique type β, called the *type* of α, such that $\alpha \preceq \beta$.
- If two sorts have the same type, there is a least upper bound for them in $\preceq$.

A sort denotes a nonempty subset of the denotation of its type. Hence sorts may overlap, which is very convenient for formalizing mathematics. (The overlapping of sorts has been dubbed *inclusion polymorphism* [6].)

Since a partial function from a set A to a set B is also a partial function from any superset of A to any superset of B, compound sorts of kind ind have a very elegant semantics: The denotation of $\alpha = \alpha_1, \ldots, \alpha_n \rightharpoonup \alpha_{n+1}$ of type β of kind ind is the set of partial (and total) functions f of type β such that $f(a_1, \ldots, a_n)$ is undefined whenever at least one of its arguments a_i lies outside the denotation of α_i.

Sorts serve two main purposes. First, they help to specify the value of an expression. Every expression is assigned a sort (called the *syntactic sort* of the expression) on the basis of its syntax. If an expression is *defined*, it denotes an object in the denotation of its syntactic sort. Second, sorts are used to restrict the application of binding constructors. For example, if α is a sort of type β, then a formula of the form

$$\forall x : \alpha \, . \, P(x)$$

(which says, for all x of sort α, $P(x)$ holds) is equivalent to the formula

$$\forall y : \beta \, . \, (y \downarrow \alpha) \supset P(y).$$

Sorts are not directly used for determining the well-formedness of expressions. Thus, if f and a are expressions of sorts $\alpha \to \beta$ and α', respectively, then the application $f(a)$ is well-formed provided only that α and α' have the same type.

As a simple illustration of the effectiveness of this subtyping mechanism, consider the language of our basis theory of real numbers, *Complete Ordered Field*, in which we stipulate $\mathbf{N}$ is enclosed by $\mathbf{Z}$, which is enclosed by $\mathbf{Q}$, which is enclosed by $\mathbf{R}$, which is enclosed by the base type `ind`. So $\mathbf{N} \rightharpoonup \mathbf{R}$ denotes the set of all partial functions from the natural numbers to the real numbers. This set of functions is a subset of the denotation of ind $\rightharpoonup$ ind. A function constant specified to be of sort $\mathbf{R} \rightharpoonup \mathbf{R}$ would automatically be applicable to expressions of sort $\mathbf{N}$. Similarly, if f is a function constant declared to be of sort $\mathbf{N} \rightharpoonup \mathbf{N}$ and a is an expression of sort $\mathbf{R}$, then $f(a)$ is automatically *well-formed*, but $f(a)$ is *well-defined* only when a denotes a natural number. A subtyping mechanism of this kind would be quite awkward in a logic having only total functions.

Since LUTINS has a partially ordered set of sorts, it is an "ordered-sorted" logic. Ordered-sorted type theory [32] (and most weaker order-sorted logics) can be directly embedded in LUTINS.

2.5 Summary

LUTINS is a many-sorted, multivariate higher-order predicate logic with partial functions and subtypes. It has strong support for specifying and reasoning about functions: λ-notation, partial functions, a true definite description operator, and full quantification over functions. Its type hierarchy and sort mechanism are convenient and natural for developing many different kinds of mathematics. Although LUTINS contains no polymorphism in the sense of variables over types, polymorphism is achieved through the use of constructors and quasi-constructors, sorts, and theory interpretations (see Section 3.3).

Perhaps most importantly, the intuition behind LUTINS closely corresponds to the intuition used in everyday mathematics. The logical principles employed by LUTINS are derived from classical predicate logic and standard mathematical practice. This puts it in contrast to some other higher-order logics, such as Martin-Löf's constructive type theory [34], the Coquand-Huet Calculus of Constructions [10], and the logic of the Nuprl proof development system [9]. These logics—which are constructive as well as higher order—employ rich type constructors and incorporate the "propositions as types" isomorphism (see [31]). Motivated in part by a desire to model computational reasoning, they are a significant departure from traditional, classical mathematical practice. Moreover, they allow dependent types or quantification over type variables, which create more complicated type systems. However, the Martin-Löf-style systems provide simpler, specifically predicative [21], methods for defining mathematical objects, so that their domains are in this respect less complicated than those for classical simple type theories. The restriction to predicative definitions may or may not be an advantage; from the point of view of developing classical analysis, for instance, it is certainly an impediment [44, 50, 49, 21].

3 Little Theories Approach

IMPS supports the "little theories" version of the axiomatic method [19] as well as the "big theory" version in which all reasoning is performed within a single powerful and highly expressive axiomatic theory, such as Zermelo-Fraenkel set theory. In the little theories version, 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 logically linked together by translations called theory interpretations which serve as conduits to pass results from one theory to another. We argue in [19] that this way of organizing mathematics across a network of linked theories is advantageous for managing complex mathematics by means of abstraction and reuse.

3.1 Theories

Mathematically, a theory in IMPS consists of a language and a set of axioms. At the implementation level, however, theories contain additional structure which encodes this axiomatic information in procedural or tabular form. It facilitates various kinds of low-level reasoning within theories that are encapsulated in the IMPS expression simplifier (see Section 4.4.1).

A theory is constructed from a (possibly empty) set of subtheories, a language, and a set of axioms. Theories are related to each other in two ways: one theory can be the *subtheory* of another, and one theory can be *interpreted* in another by a theory interpretation. A theory may be enriched via the definition of new atomic sorts and constants and via the installation of theorems. Several examples of theories are discussed in Section 7.

3.2 Definitions

IMPS supports four kinds of definitions: atomic sort definitions, constant definitions, recursive function definitions, and recursive predicate definitions. In the following let $\mathcal{T}$ be an arbitrary theory.

Atomic sort definitions are used to define new atomic sorts from nonempty unary predicates. An *atomic sort definition* for $\mathcal{T}$ is a pair $\delta = (n, U)$ where n is a symbol intended to be the name of a new atomic sort of $\mathcal{T}$ and U is a nonempty unary predicate in $\mathcal{T}$ intended to specify the extension of the new sort. δ can be installed in $\mathcal{T}$ only if the formula $\exists x . U(x)$ is known to be a theorem of $\mathcal{T}$. As an example, the pair

$$(\mathbf{N},\ \lambda x : \mathbf{Z}\ .\ 0 \leq x)$$

defines $\mathbf{N}$ to be an atomic sort which denotes the natural numbers. Since the sort of an expression gives immediate information about the value of the expression, it is often very advantageous to define new atomic sorts rather than work directly with unary predicates.

Constant definitions are used to define new constants from defined expressions. A *constant definition* for $\mathcal{T}$ is a pair $\delta = (n, e)$ where n is a symbol intended to be the name of a new constant of $\mathcal{T}$ and e is a expression in $\mathcal{T}$ intended to specify the value of the new constant. δ can be installed in $\mathcal{T}$ only if the formula $e\downarrow$ is verified to be a theorem of $\mathcal{T}$. As an example, the pair

$$(\text{floor}, \ \lambda x : \mathbf{R} \, . \, \iota z : \mathbf{Z} \, . \, z \leq x \wedge x < 1 + z)$$

defines the floor function on reals using the ι constructor.

Recursive function definitions are used to define one or more functions by (mutual) recursion. They are essentially an implementation of the approach to recursive definitions presented by Y. Moschovakis in [38]. A *recursive definition* for $\mathcal{T}$ is a pair $\delta = ([n_1, \ldots, n_k], [F_1, \ldots, F_k])$ where $k \geq 1$, $[n_1, \ldots, n_k]$ is a list of distinct symbols intended to be the names of k new constants, and $[F_1, \ldots, F_k]$ is a list of functionals (i.e., functions which map functions to functions) of kind ind in $\mathcal{T}$ intended to specify, as a system, the values of the new constants. δ can be installed in $\mathcal{T}$ only if the functionals $F_1, \ldots, F_k$ are verified to be monotone in $\mathcal{T}$ with respect to the subfunction order $\sqsubseteq$.[7] The names $[n_1, \ldots, n_k]$ then denote the simultaneous least fixed point of the functionals $F_1, \ldots, F_k$. As an example, the pair

$$(\text{factorial}, \ \lambda f : \mathbf{Z} \rightharpoonup \mathbf{Z} \, . \, \lambda n : \mathbf{Z} \, . \, \text{if}(n = 0, 1, n * f(n-1)))$$

is a recursive definition of the factorial function in our standard theory of the real numbers.

This approach to recursive definitions is very natural in IMPS because expressions of kind ind are allowed to denote partial functions. Notice that there is no requirement that the functions defined by a recursive definition be total. In a logic in which functions must be total, a list of functionals can be a legitimate recursive definition only if it has a solution composed entirely of *total* functions. This is a difficult condition for a machine to check, especially when $k > 1$. Of course, in IMPS there is no need for a recursive definition to satisfy this condition since a recursive definition is legitimate as long as the defining functionals are monotone. IMPS has an automatic syntactic check sufficient for monotonicity that succeeds for many common recursive function definitions.

Recursive predicate definitions are used to define one or more predicates by (mutual) recursion. They are implemented in essentially the same way as

[7] $f \sqsubseteq g$ iff $f(a_1, \ldots, a_m) = g(a_1, \ldots, a_m)$ for all m-tuples $\langle a_1, \ldots, a_m \rangle$ in the domain of f.

recursive function definitions using the order $\subseteq$[8] on predicates. This approach is based on the classic theory of positive inductive definitions (see [37]). For an example, consider the pair

$$([\text{even}, \text{odd}],\ [F_1, F_2]),$$

where:

- $F_1 = \lambda e, o : \mathbf{N} \rightarrow \text{prop} \ . \ \lambda n : \mathbf{N} \ . \ \text{if}(n = 0, \text{truth}, o(n-1))$.
- $F_2 = \lambda e, o : \mathbf{N} \rightarrow \text{prop} \ . \ \lambda n : \mathbf{N} \ . \ \text{if}(n = 0, \text{falsehood}, e(n-1))$.

It defines the predicates even and odd on the natural numbers by mutual recursion. As with recursive function definitions, there is an automatic syntactic check sufficient for monotonicity that succeeds for many recursive predicate definitions.

3.3 Theory Interpretations

One of the chief virtues of the axiomatic method is that the theorems of a theory can be "transported" to any specialization of the theory. A theory interpretation is a syntactic device for translating the language of a source theory to the language of a target theory. By definition, it has the property that the image of a theorem of the source theory is always a theorem of the target theory. It then follows that any formula proved in the source theory translates to a theorem of the target theory. We use this method in a variety of ways (which are described below) to reuse mathematical results from abstract mathematical theories.

Theory interpretations are constructed in IMPS by giving an interpretation of the sorts and constants of the language of the source theory; this is the standard approach which is usually seen in logic textbooks (e.g., see [14, 35, 46]). We give below a summary of theory interpretations in IMPS; a detailed description of theory interpretations for $\mathbf{PF}^*$ is given in [17].

Let $\mathcal{T}$ and $\mathcal{T}'$ be theories over languages $\mathcal{L}$ and $\mathcal{L}'$, respectively. A *translation from $\mathcal{T}$ to $\mathcal{T}'$* is a pair $\Phi = (\mu, \nu)$, where μ is a mapping from the sorts of $\mathcal{L}$ to the sorts of $\mathcal{L}'$ and ν is a mapping from the constants of $\mathcal{L}$ to the expressions of $\mathcal{L}'$, such that:

[8] $p \subseteq q$ iff $p(a_1, \ldots, a_m) \supset q(a_1, \ldots, a_m)$ for all m-tuples $\langle a_1, \ldots, a_m \rangle$ in the common domain of p and q.

(1) $\mu(\text{prop}) = \text{prop}$.

(2) For each sort α of $\mathcal{L}$, α and $\mu(\alpha)$ are of the same kind.

(3) If α is a sort of $\mathcal{L}$ with type β, then $\mu(\alpha)$ and $\mu(\beta)$ have the same type.

(4) If c is a constant of $\mathcal{L}$ of sort α, then the type of $\nu(c)$ is the type of $\mu(\alpha)$.

Given an expression e of $\mathcal{L}$, $\Phi(e)$ denotes the expression of $\mathcal{L}'$, defined in the obvious way from μ and ν, that is the translation of e via Φ.[9]

Let Φ be a translation from $\mathcal{T}$ to $\mathcal{T}'$. An *obligation* of Φ is a formula $\Phi(\varphi)$ where φ is either:

(1) an axiom of $\mathcal{T}$;

(2) a formula asserting that a particular constant of $\mathcal{L}$ is defined in its sort; or

(3) a formula asserting that a particular atomic sort of $\mathcal{L}$ is a subset of its enclosing sort.

By a theorem called the *interpretation theorem* (see [17]), Φ is a *theory interpretation* from $\mathcal{T}$ to $\mathcal{T}'$ if each of its obligations is a theorem of $\mathcal{T}'$.

The IMPS system provides support for using theory interpretations in many different ways. The following are brief descriptions of some of the most important ways theory interpretations are used in IMPS. See [19] for further discussion on applications of theory interpretations in mechanical theorem proving.

Theorem reuse Mathematicians want to be able to formulate a result in the most general axiomatic framework that good taste and ease of comprehension allow. One major advantage of this approach is that a result proved in an abstract theory holds in all contexts that have the same structure as the abstract theory. In IMPS, theory interpretations are used foremost as a

[9]The translations available in IMPS are actually more general than what we describe here: μ is allowed to map sorts to *unary predicates*. When this occurs, expressions beginning with variable binders such as $\forall$ or λ must be "relativized." For example, if μ maps α to a unary predicate U on sort β, then $\Phi(\forall x : \alpha \,.\, \psi) = \forall x : \beta \,.\, U(x) \supset \Phi(\psi)$.

for every $a, b : \mathbf{K}, n : \mathbf{Z}$ implication

- conjunction
  - $1 \leq n$
  - $\neg(a = o_{\mathbf{K}})$
  - $\neg(b = o_{\mathbf{K}})$
- $(a + b)^n = \sum_{j=0}^{n} \binom{n}{j} \cdot b^j \cdot a^{n-j}$.

Figure 1: The Binomial Theorem in Fields

mechanism for realizing this advantage: theorems proved in abstract theories can be transported via a theory interpretation to all appropriate concrete structures. For instance, the binomial theorem may be proved in a theory of fields (see Figure 1).[10] Because the real numbers form a field, we can define a theory interpretation from the theory of fields to a theory of the reals. As a consequence, we can then "install" the usual binomial theorem for the real numbers.

Automatic application of theorems Theorems can be automatically applied in IMPS in two ways: (1) as rewrite rules (see Section 4.4.3) and (2) as macetes (see Section 4.4.4). Theorems can be applied both inside and outside of their home theories. A theorem is applied within a theory $\mathcal{T}$ which is outside of its home theory $\mathcal{H}$ by, in effect, transporting the theorem from $\mathcal{H}$ to $\mathcal{T}$ and then applying the new theorem directly within $\mathcal{T}$. The mechanism is based on a kind of polymorphic matching called *translation matching* [19]; the theory interpretation used to transport the theorem is either selected or constructed automatically by IMPS. See Sections 4.4.4 and 4.4.5 for more details.

Polymorphic operators As we noted in Section 2.3, constructors and quasi-constructors are polymorphic in the sense that they can be applied to expressions of several different types. This sort of polymorphism is not very useful unless we have results about constructors and quasi-constructors that

[10]In this formulation, $\mathbf{K}$ is the underlying sort of field elements. Figure 1 is printed exactly as formatted by the the TEX presentation facility of IMPS. Various switches are available, for instance to cause connectives to be printed in-line with the usual logical symbols instead of being written as words with subexpressions presented in itemized format.

could be used in proofs regardless of the actual types that are involved. For constructors, most of these "generic" results are coded in the form of rules, as described in Section 4.2. Since quasi-constructors, unlike constructors, can be introduced by IMPS users, it is imperative that there is some way to prove generic results about quasi-constructors. This can be done by proving theorems about quasi-constructors in a theory of generic types, and then transporting these results as needed to theories where the quasi-constructor is used. For example, consider the quasi-constructor `composition` (infix $\circ$) defined as follows, for expressions f and g of type $\beta \to \gamma$ and $\alpha \to \beta$, respectively:

$$f \circ g \;\equiv\; \lambda x : \alpha \,.\, f(g(x)).$$

The basic properties about $\circ$, such as associativity, can be proved in a generic theory having four base types but no constants, axioms, or other atomic sorts. See Section 7.2 for further discussion on using quasi-constructors as polymorphic operators.

Symmetry and duality proofs Theory interpretations can be used to formalize certain kinds of arguments involving symmetry and duality. For example, suppose we have proved a theorem in some theory and have noticed that some other conjecture follows from this theorem "by symmetry." This notion of symmetry can frequently be made precise by creating a theory interpretation from the theory to itself which translates the theorem to the conjecture. As an illustration, let $\mathcal{T}$ be a theory of groups where $*$ is a binary constant denoting group multiplication. Then the translation from $\mathcal{T}$ to $\mathcal{T}$ which takes $*$ to $\lambda x, y \,.\, y * x$ and holds everything else fixed maps the left cancellation law $x * y = x * z \supset y = z$ to the right cancellation law $y * x = z * x \supset y = z$. Since this translation is in fact a theory interpretation, we need only prove the left cancellation law to show that both cancellation laws are theorems of $\mathcal{T}$.

Parametric theories As argued by R. Burstall and J. Goguen (e.g., in [23, 24]), a flexible notion of *parametric theory* can be obtained with the use of ordinary theories and theory interpretations. The key idea is that the primitives of a subtheory of a theory are a collection of parameters which can be instantiated as a group via a theory interpretation. For example, consider a generic theory $\mathcal{T}$ of graphs which contains a subtheory $\mathcal{T}'$ of abstract nodes and edges, and another theory $\mathcal{U}$ containing graphs with a concrete representation. The general results about graphs in $\mathcal{T}$ can be transported to $\mathcal{U}$ by creating a theory interpretation Φ from $\mathcal{T}'$ to $\mathcal{U}$ and then lifting Φ, in a completely mechanical way, to a theory interpretation of $\mathcal{T}$ to an extension of $\mathcal{U}$. This use of theory interpretations has been implemented in OBJ3 as well as IMPS (but in OBJ3, which has no facility for theorem proving, translation obligations must be checked by hand). For a detailed description of this technique, see [15, 18].

Relative consistency If there is a theory interpretation from a theory $\mathcal{T}$ to a theory $\mathcal{T}'$, then $\mathcal{T}$ is consistent if $\mathcal{T}'$ is consistent. Thus, theory interpretations provide a mechanism for showing that one theory is consistent relative to another. One consequence of this is that IMPS can be used as a *foundational system*. In this approach, whenever one introduces a theory, one shows it to be consistent relative to a chosen foundational theory (such as perhaps our theory of real numbers *Real Arithmetic*, described in Section 7).

3.4 Theory Ensembles

Ordinarily, mathematicians use the term *theory* in a much broader sense than we use in this paper, or than is used by logicians generally. In this sense "metric space theory" refers not to the formal theory of a single metric space (which is from a mathematical point of view not very interesting) but at the very least a theory of metric spaces and mappings between them. For example, the notion of continuity for mappings involves two separate metric spaces, and is naturally defined in a theory which is the union of two copies of a theory of an abstract metric space. A family of theories organized in this way is implemented in IMPS as a *theory ensemble* which consists of a base theory, copies of the base theory called *theory replicas*, and unions of copies of the base theory called *theory multiples*. The various theories of a theory ensemble are connected by theory interpretations which rename constants. Theory interpretations are automatically created from the base theory to each theory replica, and theory interpretations between the theory multiples are created when needed by the user. The theory interpretations allow the user to make a definition or prove a theorem in just one place, and then transport the definition or theorem to other members of the theory ensemble as needed.

for every $f : \mathbf{P}_0 \rightharpoonup \mathbf{P}_1, g : \mathbf{P}_1 \rightharpoonup \mathbf{P}_2$ implication
- conjunction
  - continuous(f)
  - $\text{continuous}_{12}(g)$
- $\text{continuous}_{02}(g \circ f)$.

for every $f : \mathbf{P}_0 \rightharpoonup \mathbf{P}_1, g : \mathbf{P}_1 \rightharpoonup \mathbf{P}_2$ implication
- conjunction
  - continuous(f)
  - continuous(g)
- continuous($g \circ f$).

Figure 2: Composition Preserves Continuity (Printed without and with overloading)

As an illustration, consider the theory ensemble for the IMPS theory $\mathcal{M}$ of an abstract metric space. The points and the distance function of the metric space are denoted in $\mathcal{M}$ by an atomic sort $\mathbf{P}$ and a constant `dist` (of sort $\mathbf{P}, \mathbf{P} \rightharpoonup \mathbf{R}$). For $n \geq 0$, let $\mathcal{M}_n$ be a copy of $\mathcal{M}$ in which the set of points and distance function are denoted by $\mathbf{P}_n$ and $\mathtt{dist}_n$, and let $\mathcal{U}_n$ be the union of $\mathcal{M}_0, \ldots, \mathcal{M}_{n-1}$. (Usually, $n \leq 3$.) The theorem in Figure 2, which says that the composition of two continuous functions is itself a continuous function, is proved in $\mathcal{U}_3$. The constant `continuous` is defined in $\mathcal{U}_2$ by the user. IMPS introduces the constants $\mathtt{continuous}_{12}$ and $\mathtt{continuous}_{02}$ by transporting the definition of `continuous` to $\mathcal{U}_3$ via the obvious theory interpretations. Normally, the user would use the mechanism in IMPS for overloading constants so that each of the three continuous constants would be written for the user as `continuous` and the theorem would be printed in TeX in the second form given in Figure 2.

After this composition theorem is proved, it can be transported to other theory replicas and multiples of the theory ensemble. For example, to obtain the composition theorem for continuous functions on a single metric space, the theorem would be transported from $\mathcal{U}_3$ to $\mathcal{M}$ via the theory interpretation which maps each of $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2$ to $\mathbf{P}$ and each of $\mathtt{dist}_0, \mathtt{dist}_1, \mathtt{dist}_2$ to `dist`. The theory ensemble mechanism also supports the transportation of definitions and theorems from a theory multiple to one of its "instances." For example, the user can transport the definition of `continuous` to *Real*

Arithmetic by doing little more than specifying that both $\mathbf{P}_0, \mathbf{P}_1$ map to $\mathbf{R}$ and both $\mathtt{dist}_0, \mathtt{dist}_1$ map to $\lambda x, y : \mathbf{R} \,.\, |x - y|$.

4 Theorem Proving

In accordance with our emphasis on mathematically natural and intuitively understandable proofs, we distinguish two levels of reasoning in proving theorems in IMPS. Reasoning at the (lower) formula level is largely done automatically via an expression simplification routine. Reasoning at the proof structure level is done by user and the machine interactively. IMPS is designed to provide some automated support, but without giving free reign to the machine; the course of machine deduction is orchestrated and controlled by the user.

IMPS produces formal proofs; they serve as the basis for conveying why the theorem is true. Because they are intended for this purpose, they are very different from the formal proofs that are described in logic text books. Usually a text book formal proof is a sequence or tree constructed using a small number of low-level rules of inference. Formal proofs of this kind tend to be composed of a mass of small logical steps. Humans usually find these proofs to be unintelligible. In contrast, the steps in an IMPS proof can be large, and most low-level inference in the proof is performed by the expression simplification routine. Moreover, a number of these larger steps may be grouped together as the result of a single human-level command. Since inference is described at a high level, proofs constructed in IMPS resemble informal proofs in understandability, but unlike an informal proof, all the details of an IMPS proof have been checked by machine.

4.1 Deduction Graphs

Every proof is carried out within some formal theory. In the process of constructing a proof, IMPS builds a data structure representing the deduction, so that during the proof process the user has great freedom to decide the order in which he wants to work on different subgoals, and to try alternative strategies on a particular subgoal. At the end of a proof, this object, called a *deduction graph*, can be surveyed by the user, typeset automatically, or analyzed by software.

The items appearing in a deduction graph are not formulas, but *sequents*, in a sense derived from Gentzen [22]; see [36] for a discussion of the advantage of organizing deduction in this way. A sequent consists of a single formula called the *assertion* together with a *context*. The context is logically a finite set of assumptions, although the implementation caches various kinds of derived information with a context. In addition, the implementation associates each context with a particular theory. We will write a sequent in the form $\Gamma \Rightarrow A$, where Γ is a context and A is an assertion.

A deduction graph is a directed graph with nodes of two kinds, representing sequents and inferences respectively. If an arrow points from a sequent node to an inference node, then the sequent node represents a hypothesis to the inference. An inference node has exactly one arrow pointing at a sequent node, and that sequent node represents the conclusion of the inference. A sequent node is said to be *grounded* (i.e., known to be "valid" or "true") if at least one arrow comes into it from a grounded inference node; an inference node is grounded if, for every arrow coming into it, the source of the arrow is a grounded sequent node. In particular, an inference node with no arrows coming into it represents an inference with no hypotheses, and is thus "immediately grounded." A deduction graph has one distinguished sequent node as its *goal*; it represents the theorem to be proved. A deduction graph is a proof of each sequent represented by a grounded sequent node in the graph.

This representation of deductions has several advantages. First, because any number of inference nodes may share a common sequent node as their conclusion, the user (or a program) may try any number of alternative strategies for proving a given sequent. Second, loops in deduction graphs arise naturally; they indicate that either of two sequents may be derived from the other, possibly in combination with different sets of additional premises. Finally, at the end of a proof, the resulting deduction graph serves as a transcript for analyzing the reasoning used in the proof, and recollecting the ideas. On the other hand, the cost to store the objects is not significant: in the current IMPS data base of almost a thousand proofs, only 18 contain as many as a hundred sequent nodes; the average number is 23.

4.2 Building Deduction Graphs

A deduction graph is begun by "posting" the goal node, a sequent node representing a sequent to be proved. The deduction graph is then enlarged by posting additional sequent nodes and creating inferences. The building of a deduction graph usually stops when the goal node is marked as grounded. Inference nodes are created by procedures called *primitive inferences*. Primitive inferences provide the only means to add inference nodes to a deduction graph; there is no way to modify or delete existing inference nodes. Each primitive inference works in roughly the same way: Certain information is fed to the primitive inference; zero or more new sequent nodes are posted; and finally, an inference node is constructed that links the newly posted nodes with one or more previously posted nodes.

There are about 30 primitive inferences. Two of the primitive inferences are special: simplification makes an inference on the basis of simplifying expressions (see Section 4.4.1); macete-application makes an inference by applying a macete (see Section 4.4.4). Each of the remaining primitive inferences embody one of the basic laws of LUTINS (or is a variant of simplification or macete-application). For example, the primitive inference direct-inference applies an analogue of an introduction rule of Gentzen's sequent calculus (in reverse). It is selected according to the leading constructor of the assertion of the input sequent node, which will become the conclusion of the inference. The system also has primitive inferences for beta-reduction, universal generalization, existential generalization, equality substitution, contraposition, cut, backchaining, eliminating iota expressions, extensionality, unfolding defined constants, definedness assertions, raising if-then-else expressions, assuming theorems, introducing choice functions, and for modifying the context of a sequent in various ways. Although the primitive inferences are available in every theory, some of them, such as simplification and defined-constant-unfolding depend on the axioms and theorems in the theory.

Primitive inferences are not called directly by the user. Instead, the user invokes interactive proof *commands* which are procedures that call primitive inferences in useful patterns. They are akin to what are called tactics in some other systems, such as HOL [25], LCF [26], and Nuprl [9].

Commands are more useful than mere primitive inferences for three reasons. First, unlike primitive inferences, commands have an interface procedure for collecting information from the user. The interface procedure protects the user from the "primitive" nature of the arguments of a primitive inference. For instance, the command unfold-single-defined-constant collects a set of natural numbers, where the number n represents the nth occurrence of the defined constant to be unfolded. By contrast, the primitive inference defined-constant-unfolding requires a set of paths[11] to the defined constant that is to be unfolded. The interface procedure calculates a path for each natural number and then calls the primitive inference defined-constant-unfolding with this new information. More precisely, the interface procedure orchestrates a conversation in which information can be exchanged a number of times between the user and the system. For example, when the user applies the command unfold-single-defined-constant to some sequent node, the system will list the constants that occur in the assertion of sequent, the user will select one, and then the system will unfold the constant if there is only one occurrence of it in the sequent. If there is more than one occurrence, the system will ask the user for which occurrences to unfold.

Second, commands may combine primitive inferences into larger, more humanly understandable units. They may thus lift the user to a higher level of inference than that of primitive inferences. As an illustration, consider again the command unfold-single-defined-constant. After this command calls the primitive inference to unfold some specified occurrences of a defined constant, the beta-reduction primitive inference is called repeatedly until no beta-reductions are possible. This has the desirable effect of building in beta-reduction into constant unfolding.

And, third, commands provide the user with new inferences that realize a certain pattern of primitive inferences. These kind of commands, which we sometimes informally call *strategies*, usually add several new inference nodes to a deduction graph at one time. Some of the simplest and most useful strategies break down the logical structure of an assertion (e.g, by applying the direct-inference primitive inference repeatedly), or else instantiate universal assumptions, existential assertions, and theorems.

An extremely important strategy is used for proving theorems by induction. The strategy takes, among other arguments, an *inductor* which specifies what induction principle to use, how to apply the induction principle, and

[11] By a path we mean a sequence of natural numbers that "navigates" from the topmost node of an expression, regarded as a tree, to some subexpression. It is thus one way of formalizing and implementing the notion of an occurrence.

for every $n : \mathbf{Z}$ implication

- $0 \leq n$
- $\sum_{j=0}^{n} j^6 = n^7/7 + n^6/2 + n^5/2 - n^3/6 + n/42.$

Figure 3: The Sum of Sixth Powers

for every $f, g : \mathbf{Z} \rightharpoonup \mathbf{R}, a, b : \mathbf{Z}$ implication

- for every $z : \mathbf{Z}$ implication
  - $a \leq z \wedge z \leq b$
  - $f(z) \leq g(z)$
- $\sum_a^b f \leq \sum_a^b g.$

Figure 4: The Monotonicity of Summation

what heuristics to employ in trying to prove the basis and induction step. IMPS allows the user to build his own inductors; the induction principles are axioms or theorems of an appropriate form. For example, the induction principle for the integers in *Real Arithmetic* is just the full second-order induction axiom. The induction strategy is very effective on many theorems from elementary mathematics; in some simple cases, the strategy can produce a complete proof (two such formulas are printed in Figures 3–4), while in other cases it does part of the work and then returns control to the user.

IMPS also has "ending" strategies, the most basic of which is called prove-by-logic-and-simplification. These strategies correspond to statements like "and the theorem follows from the above lemmas" that are commonly given in informal proofs. They make complicated, but shallow inferences using lots of logical deduction and simplification. These strategies have the flavor of the proof search strategies of classic automated theorem provers; hence, they give IMPS an automated, as well as interactive, theorem proving capability.

4.3 Soundness

We intend, of course, that the user can only make sound inferences in IMPS. Our scheme for guaranteeing this is rather simple: IMPS allows the user to modify a deduction graph only by posting sequent nodes or by calling primitive inferences (either directly or indirectly). Since posting a sequent node does not affect the inferences encoded in a deduction graph, IMPS will be sound as long as each primitive inference is sound. The primitive inferences have been carefully implemented so that there is a high degree of assurance that they do indeed only make sound inferences. With this scheme, there is no problem about the soundness of commands since they ultimately only affect a deduction graph through the application of primitive inferences. Hence, our machinery of deduction graphs and primitive inferences makes a type discipline like ML's unnecessary for assuring that complex reasoning does not go awry.

4.4 Theory-Supported Reasoning

The logical content of a theory is determined by its language and set of axioms. As an IMPS object, a theory also has a variety of other characteristics, such as the sequence of defined constants that have been introduced, and the sequence of theorems that have been derived so far. This section will discuss mechanisms that support theory-specific reasoning, by which we mean reasoning that is sound only relative to the axiomatic content of particular theories.

4.4.1 Simplification

Humanly understandable proofs must take relatively large steps, so that the reader is not overwhelmed with a forest of detail. The expression simplifier is crucial to achieving human-sized proof steps. 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 definedness and sort-definedness assertions needed to apply many forms of inference.

The notion of quasi-equality, mentioned in Section 2.3, serves as the correctness requirement for the simplifier: If the simplifier transforms an expression

e to e' relative to the assumptions of a context Γ (in a theory $\mathcal{T}$), then $\mathcal{T}$ and Γ must together entail $e \simeq e'$. That is to say, either e and e' are both defined and share the same denotation, or else they are both undefined. In LUTINS, quasi-equality justifies substituting e' in place of e at any occurrence at which e' is free for 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 [36], and has since been incorporated in other work (such as [27]).

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:

1. Algebraic simplification of polynomials, relative to a range of algebraic theories (see Section 4.4.3);

2. A decision procedure for linear inequalities, based on the variable elimination method used in many other theorem provers, for instance by Boyer and Moore [5];

3. Rewrite rules for the current theory $\mathcal{T}$, or for certain theories $\mathcal{T}_0$ for which IMPS can find interpretations from $\mathcal{T}_0$ into $\mathcal{T}$ (see Section 4.4.5).

Since in LUTINS functions may be partial and terms may be undefined, term simplification in LUTINS 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.*

Rewrite rules also generate convergence requirements. Suppose that we have a theorem of the form

$$\forall x : \alpha \, . \, s[x] = s'[x]$$

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 $s'[t]$, but only if $t \downarrow \alpha$. If t is undefined, or if it has a value in the type of α but not in α, then the change is not justified as an instance of the theorem.

Despite these apparently stringent restrictions, the IMPS simplifier is able to work effectively. Although allowing partial functions in theories does require checking definedness of expressions, one of the significant lessons that we have learned from IMPS is that this difficulty can be overcome.

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 4.4.2), use special routines exploiting information extracted from the axioms and theorems of the context's theory.

The simplification procedures are used systematically in the course of building deduction graphs. For instance, if A simplifies to `truth` relative to Γ, then the sequent $\Gamma \Rightarrow A$ is recognized as valid without any further inference. In addition, the power of the simplifier ensures that the same proof idea may be successfully applied to different formulas when the differences between them are syntactic and superficial.

The emphasis on a powerful simplification procedure to allow large inference steps in the course of interactive proof development is shared with Eves and its predecessor m-Eves [11, 12, 40], as well as the more recent PVS [39].

4.4.2 Reasoning about Definedness

Because simplification involves 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 $\forall x : \alpha \,.\, x \downarrow \beta$.

The unconditional sort coercion theorems, together with the syntactic ordering on sorts $\preceq$, defined in Section 2, determines a pre-order $\ll$ on sorts. In particular, if S is a set of unconditional sort coercion formulas in a language $\mathcal{L}$, then $\ll_S$ is the weakest pre-order extending $\preceq$ for $\mathcal{L}$, such that:

- $\alpha \ll_S \beta$ if a formula of the form $\forall x : \alpha \,.\, x \downarrow \beta$ is in S;
- $\alpha_1, \ldots, \alpha_n \to \alpha_{n+1} \ll_S \beta_1, \ldots, \beta_n \to \beta_{n+1}$ whenever $\alpha_i \ll_S \beta_i$ for all i with $1 \leq i \leq n+1$.

$\alpha \ll_S \beta$ 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 t and 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. If t is a conditional term "if ϕ then s_1 else s_2," then critical pairs for s_1 and s_2 may be combined to provide a set for t.

These principles mechanize definedness checking for a fragment of LUTINS that corresponds to order sorted theories in higher order logic [32].

Frequently, a set of critical pairs will be relatively small, even if it is non-null. 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`.

The conditional theorems used in this level are stored in a *domain-range handler* for the theory. It contains three primary kinds of information about the domain and range of functions, and the relations between sorts, in the theory.

- *Definedness conditions* of the form

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

- *Value information* of the form

$$\forall x_1 : \alpha_1, \ldots, x_n : \alpha_n \, . \, \phi(x_1, \ldots, x_n, g(x_1, \ldots, x_n)).$$

  These theorems characterize the range of g, and can be used in checking the definedness of expressions of the form $f(\ldots g(t_1, \ldots, t_n) \ldots)$.

- *Conditional sort coercions* of the form

$$\forall x : \beta \, . \, \phi(x) \supset x \downarrow \alpha.$$

To check the definedness of a term $f(t_1, \ldots, t_n)$ in sort α, we look for a definedness condition

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

or alternatively a sort coercion condition

$$\forall x : \beta \, . \, \phi(x) \supset x \downarrow \alpha,$$

where β is the syntactic sort of $f(t_1, \ldots, t_n)$ (i.e., the declared range of f).

If a definedness condition for α is found, then we form the new goal $\psi(t_1, \ldots, t_n)$. Moreover, for each subterm t_i that is of the form $g(s_1, \ldots, s_m)$ and has a value condition ϕ_i, we add $\phi_i(s_1, \ldots, s_m, g(s_1, \ldots, s_m))$ to Γ, thus forming an expanded context Γ'. Finally, we call the simplifier on Γ' and $\psi(t_1, \ldots, t_n)$.

If instead only a sort coercion is found, we call the simplifier on the assertion $(\lambda x : \beta \, . \, \phi)(f(t_1, \ldots, t_n))$. As part of establishing this, IMPS must ensure that $f(t_1, \ldots, t_n) \downarrow \beta$. In the course of doing so, a definedness condition for β may be used. Recursive calls of yet greater depth are, however, almost certain to be in vain, and are prevented by the implementation.

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.

4.4.3 Transforms

Each theory contains a table with information used by the simplifier. This table is organized as a hash table of procedures (called transforms) each of which will transform an expression in a sound manner. Look-up in this table is done by using constructor and the leftmost function constant as keys. Rewrite rules are implemented in this way, as are algebraic simplification procedures that would be impractical to represent as rewrite rules.

In IMPS some of the transforms can be generated in a uniform way, independently of the specific constants which play the role of the algebraic operations. This means that the simplifier can be crafted to provide particular forms of simplification, when the constants have certain algebraic properties. For instance, algebraic simplification for an arbitrary field, for real arithmetic, and for modular arithmetic are derived from the same entity, called an algebraic processor. An algebraic processor is applied by establishing a correspondence between the operators of the processor (e.g., the addition and multiplication operators) and specific constants of the theory. In the IMPS theory of fields, where the field elements form the type $\mathbf{K}$, the algebraic processor is configured by stipulating that the multiplication operator is the function constant $\times_{\mathbf{K}}$, the addition operator is the function constant $+_{\mathbf{K}}$, the zero is the individual constant $o_{\mathbf{K}}$, and so on. Certain operators need not be used; for instance, modular arithmetic does not have a division operator in general. Depending on the correspondence between operators and constants, the algebraic processor generates a set of formulas that must be theorems in the theory in order for its manipulations to be correct.

4.4.4 Macetes

In IMPS we have used the name *macete* (in Portuguese, a macete is a clever trick) to denote user-definable extensions of the simplifier which are under direct control of the user. Formally, a macete is a function which takes as arguments a context and an expression and returns an expression. Macetes are used to apply a theorem or a collection of theorems to a sequent in a deduction graph. Individual theorems are applied by *theorem macetes* built automatically when a theorem is installed in a theory. Compound macetes are constructed ultimately from theorem macetes, together with a few special macetes such as beta-reduction and simplification, using a few simple *macete* *constructors*, which are just functions from macetes to macetes. They include constructs to apply a number of macetes in succession, or repeatedly until no further changes can be made. Compound macetes provide a simple mechanism for applying lists of theorems in a manner which is under direct user control.

One kind of theorem macete based on straightforward matching of expressions is called an *elementary macete*. To explain their behavior, we need two auxiliary notions. An expression e *matches* a pattern expression p if and only if there is a substitution σ such that σ applied to p is α-equivalent to e. If Γ is a context and σ is a substitution, we say that Γ *immediately entails* σ *is defined* if, for each component $x \mapsto t$ of σ, with x of sort α, simplification reduces $t \downarrow \alpha$ to `truth`.

Though any kind of theorem can be used to generate an elementary macete, for the purposes of this exposition, let us assume the theorem is the universal closure of a conditional equality of the form $\phi \supset p_1 = p_2$. When applied to a context-expression pair (Γ, e), the macete works as follows. The left-hand side p_1 is matched to e. If the matching succeeds, then the resulting substitution σ is applied to the formula ϕ. If the $\phi[\sigma]$ is entailed by Γ, and if Γ immediately entails that σ is defined, then the macete returns $p_2[\sigma]$, the result of applying the substitution σ to the right-hand side p_2. If any stage of this process fails, then the macete simply returns e. (This mechanism is described in more detail in [48].) Elementary macetes are used to apply a theorem within its home theory.

Another kind of theorem macete is called a *transportable macete*. It is based on a much more interesting kind of matching we call *translation matching*, which allows for inter-theory matching of expressions. A translation match is essentially a two-fold operation consisting of a theory interpretation and ordinary matching. An expression e is a translation match to a pattern expression p if and only if there is a theory interpretation Φ and a substitution σ such that σ applied to the translation of p under Φ is α-equivalent to e. After using translation matching, transportable macetes work in much the same way as elementary macetes. Transportable macetes are used to apply a theorem outside of its home theory.

We end this section with three simple examples of elementary and transportable macetes chosen from the hundreds of examples contained in the

IMPS initial theory library. The theorem

$$\forall x : \mathbf{R} \,.\, 0 \leq x \supset |x| = x$$

of *Real Arithmetic* generates an elementary macete which rewrites an expression of the form $|s|$ to s provided the simplifier can verify that s is a nonnegative real number (in the local context of the expression). The theorem

$$\forall x : \mathbf{R}, y : \mathbf{Z} \,.\, 0 < x \supset 0 < x^y$$

generates an elementary macete which reduces a goal of the form $0 < r^n$ to a new goal $0 < r$ provided the simplifier can verify that r and n are defined in the reals and integers, respectively. Finally, the theorem

$$\forall x, y : \mathbf{U} \,.\, x \in \{y\} \equiv x = y$$

of *Indicators* generates a transportable macete which rewrites an expression of the form $a \in \{b\}$ to $a = b$, regardless of what are the sorts of a and b.

4.4.5 Transportable Rewrite Rules

Transportable *rewrite rules* make use of translation matching also, but within the simplifier. If an unconditional equality $p_1 = p_2$ (or a formula of some other suitable syntactic form) is a theorem of $\mathcal{T}_0$, then it may be installed into the simplifier for use within a theory $\mathcal{T}_1$. When simplifying an expression e, we use translation matching to find Φ and σ as before. If we are successful, and if the context ensures that σ is defined, then we may replace e by $(\Phi(p_2))[\sigma]$. Transportable rewrite rules are used to provide efficient simplification based on rewrites from generic theories about such things as mappings and finite sequences. For instance, the theorem

$$\forall t, u : \mathbf{N} \rightharpoonup \mathrm{ind}_1, e : \mathrm{ind}_1 \,.\quad \mathrm{append}\{(e :: t), u\} = (e :: \mathrm{append}\{t, u\})$$

rewrites expressions in which a `cons` is nested within an `append`. The role of translation matching in this case is simply to select a sort σ as the instance for $\mathtt{ind}_1$.

5 User Interface

The IMPS user interface, which is a removable component of IMPS, is primarily written in GNU Emacs [47]; the IMPS core is written in T [33, 41], a sophisticated version of Scheme. The user interface is implemented using the subordinate process mechanism of GNU Emacs, which allows a program executing in T to issue commands to Emacs, and *vice versa*. Thus IMPS can request that formulas and derivations be presented to the user, specially formatted by Emacs, while conversely the user can frame his requests to IMPS using the interactive machinery of Emacs. The interface provides additional facilities in case Emacs is running under the X Window System, for instance a menu-driven mode.

The user interface provides for three major activities: interactive theorem proving, theory development, and parsing and printing. Most of the actual interface code is devoted to the first activity. Each of these activities is discussed below.

5.1 Interactive Theorem Proving

The IMPS user interface provides facilities for directing, monitoring, recording, and replaying proofs. The facilities to monitor the state of the proof include graphical displays of the deduction graph as a tree, TEX typesetting of the proof history, and TEX typesetting of individual subgoals in the deduction graph. The graphical display of the deduction graph permits the user to visually determine the set of unproven subgoals. The TEX typesetting facilities allow the user to examine each sequent in the proof or an entire proof in a mathematically more appealing notation than is possible by raw textual presentation alone.

There are various facilities to direct proofs. For example, for any particular subgoal, the interface presents the user with a well-pruned list of macetes which can be applied to that subgoal. This list is obtained by using syntactic and semantic information which is made available to the interface by the IMPS supporting machinery. In situations where over 400 theorems are available to the user, there are rarely more than 10 macetes presented to the user as options.

The interface assists the user with command syntax for commands which require additional arguments. For example, in order to apply the command instantiate-universal-assumption the user must specify the universal assumption to be instantiated and the instantiations of the variables. In such cases, the interface will prompt the user for the necessary arguments, if it cannot first determine them from other available information. Thus in the previous example, when there is only one universal assumption, the interface will not ask the user which formula to instantiate.

Finally, there is a mechanism for producing a transcript of an interactive proof. The resulting transcript is a segment of text which can be edited and replayed fully or partially, in much the same way that a text editing macro replays a sequence of commands entered at the keyboard. This is especially useful for building new proofs which differ in small ways from previously constructed ones.

5.2 Theory Development

The IMPS user creates and modifies a theory, theory interpretation, theory constituent (such as a definition or theorem), or other IMPS object by evaluating an expression called a *definition form* (or *def-form* for short). The approximately 30 def-forms provide a mechanism for extending the mathematics of the IMPS system. The user interface provides templates to the user for writing def-forms. Def-forms are stored in files which can be loaded as needed into a running IMPS process.

For instance, the def-form introducing the natural numbers as a defined sort within *Real Arithmetic* is:

```
(def-atomic-sort NN
  "lambda(x:zz, 0<=x)"
  (theory h-o-real-arithmetic)
  (witness "0"))
```

This stipulates that the natural numbers will be those integers satisfying $\lambda x : \mathbf{Z} \,.\, 0 \leq x$; it also advises IMPS to consider 0 when checking that this predicate is non-empty.

The def-form that introduces the symmetry translation reversing group multiplication, mentioned in Section 3.3, reads:

```
(def-translation MUL-REVERSE
   (source groups)
   (target groups)
   (fixed-theories h-o-real-arithmetic)
   (constant-pairs
     (mul "lambda(x,y:gg, y mul x)"))
   (theory-interpretation-check using-simplification))
```

It stipulates that the theory of groups forms both the source and the target of the translation, and that vocabulary defined in *Real Arithmetic* should be left unchanged. Only the binary function symbol `mul` is to be translated, to the lambda-expression shown. IMPS is requested to use the simplifier to ascertain that the theory interpretation obligations of this translation are in fact met, so that the translation is an interpretation.

5.3 Parsing and Printing

Interaction with IMPS requires an extensive amount of reading of expressions from the keyboard, or from files, and of displaying of expressions on the screen, or writing them to files. Abstractly, an expression is a tree-like structure determined by the IMPS logic. In the implementation of IMPS an expression is a data structure which corresponds very closely to its tree structure, but has in addition a large amount of cached information available to the deductive machinery. For the IMPS user, an expression is typically something which can be represented as text, for instance $\int_a^b \ln x dx$. The correspondence of an expression as a data structure to an external representation for input or output is determined by the user syntax which is employed. IMPS allows multiple user syntaxes, so for example, the syntax that is used for reading in expressions (usually text) may be different from the syntax used to display expressions (which could be text or text interspersed with TEX commands.) This flexible arrangement means users can freely change from one syntax to another, even during the course of an interactive proof. In other words, the core machinery is completely syntax independent.

6 IMPS and Mathematical Analysis

To a large extent, the development of IMPS has been guided by our attempts to prove theorems in mathematical analysis—both theorems about the real numbers and theorems about more abstract objects such as Banach spaces or spaces of continuous functions from one metric space to another. The IMPS logic and little theories methodology have made it possible to develop parts of graduate-level analysis without sacrificing either clarity or naturalness.

With partial functions, higher-order operators, and subtypes, LUTINS is well-suited to be a logic for analysis. The value of having a natural way of dealing with partial functions in the development of analysis cannot be overestimated. Many of the important operators of analysis, such as the integral of a function or the limit of a sequence, are most conveniently treated as *partial* higher-order functions. For example, the limit of a sequence $\xi = (x_n)_{n \in \mathbf{N}}$ is defined whenever there is a unique real number a satisfying a familiar predicate $\varphi(\xi, a)$, and when $\varphi(\xi, a)$ holds, the limit of ξ is the number a. Having a logic with partial functions and possibly nondenoting terms, we can define the limit operator by

$$\lim(\xi) = \iota\, a : \mathbf{R} \,.\, \varphi(\xi, a)$$

without having to specify separately its domain.

Analysis is rife with various types of spaces and classes of functions. The little theories approach is an especially good framework for organizing this kind of mathematics. An example of little theories used to advantage in IMPS is the proof of an open mapping theorem (for certain Banach space functions close to the identity) given in [19]. Other examples are briefly described in subsection 7.3.

Mathematical analysis has traditionally served as a ground for testing the adequacy of formalizations of mathematics, because analysis requires great expressive power for constructing proofs. Nonetheless, most work in automated theorem proving has been in areas other than analysis. One notable exception to this is the work of W. Bledsoe and his students which has dealt with problems in analysis and general topology beginning in the early 70's (see Bledsoe's discussion in [3]). In particular, this group has built a series of powerful provers combining resolution and other techniques (such as variable elimination) to reason about formulas involving real inequalities. One recent prover in this direction developed by L. Hines is described in [30].

An entirely different approach to automated theorem proving in analysis is taken by E. Clarke and X. Zhao [8]. They have successfully implemented a system which can reason about a large class of expressions encountered in real analysis, including trigonometric functions, real inequalities, limits, infinite summations, derivatives, and integrals. Clarke and Zhao's system, called Analytica is implemented on top of the commercially available system Mathematica. Mathematica provides it with a wide range of facilities not possessed by other provers. These include sophisticated algebraic manipulation, reduction rules that apply analytic identities and the ability to determine closed forms for transforms of functions in many cases, and solutions of differential equations. Though Analytica combines theorem proving capabilities with very sophisticated symbolic manipulation capabilities in an interesting way, it has several drawbacks. Firstly, since Mathematica is unsound (for example in doing beta-reduction), the soundness of Analytica itself becomes an issue. Moreover, it is not clear how Analytica can relate the formal facilities for manipulating objects offered by Mathematica with the underlying semantics of these objects. For example, is there a definition for the integral or a set of axioms characterizing the integral, or are the manipulations performed by Mathematica code the ultimate arbiter of what integration means? Finally, it is not clear how Analytica can handle abstract objects such as Banach spaces, which are very useful in all kinds of analysis, even "classical" analysis.

7 Initial Theory Library

A *theory library* is a collection of theories, theory interpretations, and theory constituents (e.g., definitions and theorems) which serves as a data base of mathematics. A theory library is composed of *sections*; each section is a particular body of knowledge that is stored in a set of files consisting of def-forms. A section can be loaded as needed into a running IMPS process. In the course of using IMPS, an IMPS user builds his or her own theory library on top of the *initial theory library* that is supplied with IMPS.

The IMPS initial theory library contains a large variety of basic mathematics. It offers the user a well-developed starting point for her or his own theory library. It also is a rich source of examples that illustrates some of the diverse ways mathematics can be formulated in IMPS. The initial theory library includes formalizations of the real number system and objects like sets and sequences; theories of abstract mathematical structures such as groups and metric spaces; and theories to support specific applications of IMPS in computer science.

This section describes the major theories that are contained in the IMPS initial theory library. Along the way, we point out some of the more important theorems we have proven in these theories.

7.1 Real Numbers

Two theories of the real numbers are contained in the initial theory library. These theories are equivalent in the sense that each one can be interpreted in the other; moreover, the two interpretations compose to the identity. These interpretations are constructed in the theory library using the IMPS translation machinery.

The first is *Complete Ordered Field*, a theory in which the real numbers are specified as a complete ordered field and the rational numbers and integers are specified axiomatically as substructures of the real numbers. Exponentiation to an integer power is a defined constant denoting a partial function.

The second axiomatization is *Real Arithmetic*, which we consider to be our working theory of the real numbers. The axioms of *Real Arithmetic* include the axioms of *Complete Ordered Field*, formulas characterizing exponentiation as a primitive constant and formulas which are theorems proven in *Complete Ordered Field*. These theorems are needed for installing an algebraic processor and for utilizing the definedness machinery of the simplifier. The proofs of these theorems in the theory *Complete Ordered Field* require a large number of intermediate results with little independent interest. The use of two equivalent axiomatizations frees our working theory of the reals from the burden of recording these uninteresting results.

The theory *Real Arithmetic* is equipped with routines for simplifying arithmetic expressions and rational linear inequalities (see Section 4.4.1). These routines allow the system to perform a great deal of low-level reasoning without user intervention. The theory contains several defined entities; e.g., the natural numbers are a defined sort and the higher-order operators Σ and Π are defined recursively.

Real Arithmetic is a useful building block for more specific theories. If a theory has *Real Arithmetic* as a subtheory, the theory can be developed with the help of a large portion of basic, everyday mathematics. For example, in a theory of graphs which includes *Real Arithmetic*, one could introduce the concept of a weighted graph in which nodes or edges are assigned real numbers. We imagine that *Real Arithmetic* will be a subtheory of most theories formulated in IMPS.

7.2 Generic Objects

One of the advantages of working in a logic like LUTINS, with a rich structure of functions, is that generic objects like sets and sequences can be represented directly in the logic as certain kinds of functions. For instance, sets are represented in IMPS as *indicators*, which are similar to characteristic functions, except that x is a "member" of an indicator f iff $f(x)$ is *defined*. Operators on indicators and other functions representing generic objects are formalized in IMPS as quasi-constructors, and theorems about these operators are proved in "generic theories" that contain neither constants nor axioms (except for possibly the axioms of *Real Arithmetic*). Consequently, reasoning is performed in generic theories using only the purely logical apparatus of LUTINS (and possibly *Real Arithmetic*). Moreover, theorems about generic objects are easy to apply in other theories since the operators, as quasi-constructors, are polymorphic and since the theory interpretations involved usually have no obligations to check.

The initial theory library contains theorems about operators (i.e., quasi-constructors) on the following kinds of generic objects:

- *Sets.* There are operators for basic set operations such as union, intersection, complement, membership, subset, etc. Since sets are represented as indicators, most of the basic theorems are proved by just the command simplify-insistently.

- *Unary functions.* The operators formalize basic function notions such as composition, domain, range, inverse function, injectiveness, etc. These operators supplement the built-in function machinery of LUTINS.

- *Sequences.* Sequences over a sort α are represented as partial functions from the natural numbers to α. Lists are identified with sequences whose domain is a finite initial segment of the natural numbers. The operators include basic list operations: `nil`, `car`, `cdr`, `cons`, and `append`.

- *Pairs.* A pair of elements a, b of sort α, β is represented as a function whose domain equals the singleton set $\{\langle a, b\rangle\}$. The operators include a pair constructor and the two pair selectors.

This part of the initial theory library also contains theorems about the ι constructor and about the cardinality of sets. Several of the cardinality theorems (e.g., the Schröder-Bernstein theorem and the theorem that says a subset of a finite set is itself finite) are surprisingly difficult to prove in complete detail.

7.3 Analysis

The structure of the IMPS analysis theory library aggressively exploits the little theories approach outlined earlier in Section 3, providing users with an extensive network of theories and interpretations between them. This approach is desirable because it permits users to state and prove once and for all the basic facts in a high degree of generality and reuse the results in more specific contexts.

The analysis library consists of about 25 theories and over 100 theory interpretations. Most of the theory interpretations (close to 90 percent) are created automatically by the system when the theories are created or by the translation-match machinery. What follows is only a small sample of theories that are actually available:

- *Partial Order.* This is the theory of an abstract set $\mathbf{S}$ with a transitive, reflexive, and anti-symmetric relation $\prec$. This theory provides a framework for stating and proving general theorems about ordering relations, including definitions and characterizations of the supremum and infimum of a set. One of the more interesting theory interpretations that are explicitly constructed in this section of the library is the "order reversing" interpretation which takes the constant symbol $\prec$ into the defined constant symbol $\succ$ (defined by $x \succ y$ if and only if $y \prec x$.) This allows results about suprema to be immediately usable as results about infima. It is also clear that results in *Partial Order* can be interpreted in the theory *Real Arithmetic.*

- *Metric Space.* This is the theory of an abstract set $\mathbf{P}$ with a two-place real-valued function d on $\mathbf{P}$ which is non-negative, symmetric, satisfies the triangle inequality, and for which $d(x, y) = 0$ only if $x = y$. This theory is appropriate for defining metric and topological properties such as limits of sequences, open sets, closed sets, completeness, and sequential compactness. The theory library contains statements and proofs of numerous facts about these concepts.

- *Metric Spaces 2-Tuples.* This is the theory of a pair of abstract metric spaces. It is the natural setting for notions about mappings between spaces, such as continuity, uniform continuity, and the Lipschitz property. In this theory we can easily prove abstract versions of the intermediate value and Bolzano-Weierstrass theorems which assert, respectively, that the image of a connected or sequentially compact set is connected or sequentially compact (see [20] for details). By transporting machinery developed in this theory to the theory of metric spaces, we obtain several versions of the contractive-mapping fixed-point principle of Banach.

- *Mappings into a Pointed Metric Space.* A theory for spaces of bounded, everywhere defined functions from a set into a metric space with a distinguished point a. A function is bounded whenever its range is contained in a closed ball centered at a. Basic properties of these spaces (such as completeness conditions) are formulated in this theory.

- *Normed Spaces.* This is the theory of a real vector space with a norm function. A sample theorem in this theory is an open mapping theorem for certain functions which are near the identity in a suitable sense (for a discussion of the proof of this theorem in IMPS, see [19]).

- *Mappings from an Interval to a Normed Space.* This is the theory of an abstract normed space together with an arbitrary interval of real numbers. This theory is used for defining the fundamental notions of calculus of functions of one variable such as differentiation and integration. Important theorems proved in the library are the mean value theorems for differentiation and integration (see Figure 5).[12]

[12] In this formulation, $\mathbf{U}$ denotes the underlying sort of vectors and $\mathbf{I}$ denotes an arbitrary (possibly unbounded) interval of real numbers.

for every $a, b : \mathbf{I}, f : \mathbf{I} \rightharpoonup \mathbf{U}, m : \mathbf{R}$ implication

- conjunction
  - $\exists y : \mathbf{I} \quad a < y$
  - $\int_a^b f \downarrow$
  - $\forall x : \mathbf{I} \quad \|f(x)\| \leq m$
- $\| \int_a^b f\| \leq m \cdot |b - a|$.

Figure 5: The Mean Value Theorem for Integrals

7.4 Algebra

The initial library contains theories of the following algebraic structures:

- *Monoids.* A monoid is a set with an associative binary operation and an identity element. In the theory *monoids*, a constant `monoid%prod` is defined recursively as the iterated product of the primitive monoid operation. Basic properties of this constant are proven in *monoids* and then transported to theories with their own iterated product operators, such as *Real Arithmetic* with the operators Σ and Π.

- *Groups* and *Group Actions.* The rudiments of group theory are developed in the theory network consisting of these two theories and various interpretations of *Group Actions* in *Groups* and of *Groups* in itself. The theorem that the quotient of a group by a normal subgroup is itself a group is proved as well as the Fundamental Counting Theorem for group theory, of which Lagrange's theorem is an easy corollary.

- *Fields.* Basic operations for fields (exponentiation to an integer power and multiplication by an integer) are defined. The theory is developed sufficiently for installing an algebraic processor for simplification. Some useful identities, such as the binomial theorem, are proved.

7.5 Computer Science

The theory library for computer science is currently less developed than that for mathematics. However, three significant facilities exist.

for every p : state $\to$ prop $\iff$

- for every s : state implication
  - accessible(s)
  - $p(s)$
- conjunction
  - $\forall s$: state initial(s) $\supset p(s)$
  - $\forall s_1$: state, a : action (accessible(s_1) $\wedge$ $p(s_1)$ $\wedge$ next(s_1, a) $\downarrow$) $\supset$ p(next(s_1, a)).

Figure 6: Induction on Accessibility for Deterministic State Machines

State Machine Theories Reusable parametric theories (in the sense described in Section 3.3) characterize a number of different kinds of state machine. A state machine theory axiomatizes a state space, which need not be finite, together with a transition function or relation, depending on whether the machine being specified is known to be deterministic or not. Typical theorems include the induction theorems for accessible states, one of which is shown in Figure 6. When such a theorem has been proved, it may be applied with the full power of the induction command (Section 4.2).

In order to specify a particular state machine, say, a deterministic one, the user develops an axiomatic theory $\mathcal{T}$ characterizing the objects that will serve as states and as inputs. He then *instantiates* the parametric theory $\mathcal{P}$ of deterministic state machines. To do so, he supplies an interpretation Φ from $\mathcal{P}$ to $\mathcal{T}$. As a consequence, all defined expressions from $\mathcal{P}$ are made available in $\mathcal{T}$, possibly under a suitable renaming, and all the theorems of $\mathcal{P}$ are available through the interpretation Φ.

Safety theorems are then expressed in terms of the resulting accessibility predicate, and liveness assertions may be expressed directly. Refinement relations between two state machines may be formulated in the direct way, using a joint theory describing both individual machines.

Domain Theory for Denotational Semantics A simple parametric theory of continuous functions and related notions developed for denotational semantics. We have developed an IMPS theory representing the official Scheme denotational semantics using this as a basis.

Facility for Defining Free Recursive Datatypes Many applications in computing use datatypes that are constructed recursively by a number of operations from previously given objects and some atoms. These datatypes are often specified by a BNF-like notation. For instance, if `elt` denotes a class of previously given objects, then the finite lists composed of these elements may be specified by the clauses:

```
L   ::= NIL | CONS elt L
```

In other cases, for instance in the abstract syntax for a programming language, different sorts of objects, such as statements, expressions, and variables, may be defined by mutual recursion, starting from given objects such as identifiers:

```
s  ::= WHILE e s | SEMICOLON s1 s2 | ASSIGN v e | ...
e  ::= v | PLUS v1 v2 | BLOCK s e | ...
v  ::= VAR ident
```

A specification such as these is normally interpreted as denoting the free algebra generated by regarding each atom as a constant and each constructor as a function symbol. Hence, it justifies an induction principle and a principle of function definition by primitive recursion.

IMPS provides a procedure which given a legitimate specification will generate a new theory $\mathcal{T}$. The specification may stipulate an already known theory $\mathcal{T}_0$ to characterize the given objects (list elements and identifiers in the examples above). $\mathcal{T}$ will be a *model conservative extension* of $\mathcal{T}_0$, by which we mean that any model of $\mathcal{T}_0$ may be enlarged to form a model of $\mathcal{T}$ by adding suitable new objects. In particular, $\mathcal{T}$ is satisfiable if $\mathcal{T}_0$ is.

$\mathcal{L}(\mathcal{T})$ extends $\mathcal{L}(\mathcal{T}_0)$ by adding one new type τ. If the specification declares several new categories (such as statements, expressions, and variables in the example above), then each of these will be represented by a subsort of τ. A clause may be represented by a sort inclusion; for instance, the sort `v` is included within `e` above. Otherwise, it is represented by a function serving as a datatype constructor. Axioms of $\mathcal{T}$ ensure that the ranges of different datatype constructors are disjoint, and do not include the values of atoms; the domain of a datatype constructor is characterized exactly by the sorts of its arguments. The second order induction principle for τ is also an axiom. The primitive recursion principle is supplied as a theorem, although its proof is not generated at the time that $\mathcal{T}$ is constructed.

8 Conclusion

IMPS is an interactive proof development system intended to support standard mathematical notation, concepts, and techniques. In particular, it provides a flexible logical framework in which to specify axiomatic theories, prove theorems, and relate one theory to another via inclusion and theory interpretation. Theory interpretations are used extensively in IMPS for reusing theories and theorems. The IMPS logic is a conceptually simple, but highly expressive version of higher-order logic which allows partially defined (higher-order) functions and undefined terms. The simple types hierarchy of the logic is equipped with a subtyping mechanism. Proofs are developed in IMPS with the aid of several different deduction mechanisms, including expression simplification, automatic theorem application, and a mechanism for orchestrating applications of inference rules and theorems. The naturalness of the logic and the high level of inference in proofs make it possible to develop machine checked proofs in IMPS that are intuitive and readable.

The IMPS initial theory library provides evidence for the claim that IMPS supports the traditional methods of mathematics. The theory library now contains repeatable proofs of almost a thousand theorems, including significant portions of algebra and analysis. Sources include traditional presentations, such as parts of a graduate course in algebra and parts of Dieudonné's *Foundations of Modern Analysis* [13]. Because standard mathematical development is possible in IMPS, the system should be an accessible, effective tool to a wide range of mathematically educated users.

Acknowledgments

We are grateful to the MITRE-Sponsored Research program, which funded the development of IMPS.

Several of the key ideas behind IMPS were originally developed by Dr. Leonard Monk on the Heuristics Research Project, also funded by MITRE-Sponsored Research, during 1984–1987. Some of these ideas are described in [36].

The authors are also grateful for the suggestions received from the referee.

References

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Academic Press, 1986.

[2] P. B. Andrews, S. Issar, D. Nesmith, and F. Pfennig. The TPS theorem proving system (system abstract). In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 641–642. Springer-Verlag, 1990.

[3] W. W. Bledsoe. Some automatic proofs in analysis. In *Automated Theorem Proving: After 25 Years.* American Mathematical Society, 1984.

[4] G. S. Boolos. On second-order logic. *Journal of Philosophy*, 72:509–527, 1975.

[5] 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.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymophism. *Computing Surveys*, 17:471–522, 1985.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] E. Clarke and X. Zhao. Analytica—a theorem prover in mathematica. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 761–765. Springer-Verlag, 1992.

[9] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[10] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[11] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink. EVES: An overview. Technical Report CP-91-5402-43, ORA Corporation, 1991.

[12] 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.

[13] J. Dieudonné. *Foundations of Modern Analysis*. Academic Press, 1960.

[14] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[15] W. M. Farmer. Abstract data types in many-sorted second-order logic. Technical Report M87-64, The MITRE Corporation, 1987.

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

[17] W. M. Farmer. A simple type theory with partial functions and subtypes. Technical report, The MITRE Corporation, 1991. Submitted for publication.

[18] W. M. Farmer. A technique for safely extending axiomatic theories. Technical report, The MITRE Corporation, 1993.

[19] 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.

[20] W. M. Farmer and F. J. Thayer. Two computer-supported proofs in metric space topology. *Notices of the American Mathematical Society*, 38:1133–1138, 1991.

[21] S. Feferman. Systems of predicative analysis. *Journal of Symbolic Logic*, 29:1–30, 1964.

[22] G. Gentzen. Investigations into logical deduction (1935). In *The Collected Works of Gerhard Gentzen*. North Holland, 1969.

[23] J. A. Goguen. Principles of parameterized programming. Technical report, SRI International, 1987.

[24] J. A. Goguen and R. M. Burstall. Introducing institutions. In *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1984.

[25] M. Gordon. HOL: A proof-generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

[26] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh* LCF*: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[27] J. Grundy. Window inference in the HOL system. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 177–89. IEEE Computer Society Press, 1991.

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

[29] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[30] L. M. Hines. The central variable strategy of str†ve. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 35–49. Springer-Verlag, 1992.

[31] W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[32] 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.

[33] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for scheme. In *Proceedings of the* SIGPLAN *'86 Symposium on Compiler Construction*, volume 21, pages 219–233, 1986. Proceedings of the '86 Symposium on Compiler Construction.

[34] P. Martin-Löf. Constructive mathematics and computer programming. In L. J. Cohen, J. Los, H. Pfeiffer, and K. P. Podewski, editors, *Logic, Methodology, and Philosophy of Science VI*, pages 153–175, Amsterdam, 1982. North-Holland.

[35] J. D. Monk. *Mathematical Logic*. Springer-Verlag, 1976.

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

[37] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, 1974.

[38] Y. N. Moschovakis. Abstract recursion as a foundation for the theory of algorithms. In *Computation and Proof Theory, Lecture Notes in Mathematics 1104*, pages 289–364. Springer-Verlag, 1984.

[39] 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.

[40] B. Pase and S. Kromodimoeljo. m-Never system summary. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 738–39. Springer-Verlag, 1988.

[41] J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual*. Computer Science Department, Yale University, fifth edition edition, 1988.

[42] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-02, SRI International, 1991.

[43] B. Russell. On denoting. *Mind (New Series)*, 14:479–493, 1905.

[44] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

[45] S. Shapiro. Second-order languages and mathematical practice. *Journal of Symbolic Logic*, 50:660–696, 1985.

[46] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[47] R. M. Stallman. GNU *Emacs Manual (Version 18).* Free Software Foundation, sixth edition edition, 1987.

[48] F. J. Thayer. Obligated term replacements. Technical Report MTR-10301, The MITRE Corporation, 1987.

[49] H. Weyl. *Das Kontinuum.* Veit, Leipzig, 1918.

[50] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, 1910. Paperback version to section *56 published 1964.