An Overview of A Formal Framework For Managing Mathematics [*]

William M. Farmer (wmfarmer@mcmaster.ca)
Martin v. Mohrenschildt (mohrens@mcmaster.ca)
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1, Canada

04 February 2003

Abstract.
Mathematics is a process of creating, exploring, and connecting mathematical models. This paper presents an overview of a formal framework for managing the mathematics process as well as the mathematical knowledge produced by the process. The central idea of the framework is the notion of a *biform theory* which is simultaneously an *axiomatic theory* and an *algorithmic theory.* Representing a collection of mathematical models, a biform theory provides a formal context for both deduction and computation. The framework includes facilities for deriving theorems via a mixture of deduction and computation, constructing sound deduction and computation rules, and developing networks of biform theories linked by interpretations. The framework is not tied to a specific underlying logic; indeed, it is intended to be used with several background logics simultaneously. Many of the ideas and mechanisms used in the framework are inspired by the IMPS Interactive Mathematical Proof System and the Axiom computer algebra system.

Keywords: Mechanized mathematics, computer theorem proving, computer algebra, axiomatic method, little theories method

AMS subject classification: 00, 03, 68

1. Introduction

What is mathematics? Mathematics is a *process* of creation, exploration, and connection. It consists of three intertwined activities:

1. *Model creation.* Mathematical models representing mathematical aspects of the world are created.

[*] To appear in: B. Buchberger, G. Gonnet, and M. Hazewinkel, eds., *Mathematical Knowledge Management*, special issue of *Annals of Mathematics and Artificial Intelligence*, 2003.

2. *Model exploration.* The models are explored by stating and proving conjectures and by performing computations.[1]

3. *Model connection.* The models are connected to each other so that results obtained in one model can be used in other related models.

Although mathematical models come in many forms, most mathematical models can be considered as collections of objects related in certain ways. For example, the standard model of the natural numbers consists of an infinite set $N = \{0, 1, 2, \ldots\}$, the usual binary operations $+$ and $*$ on N, and the usual binary relations $=$ and $<$ on N.

By producing models and knowledge about models, the mathematics process enlarges the body of mathematical knowledge. Mathematical knowledge, in turn, fuels the mathematics process. Old ideas are joined and refined into new ideas. Old structures are extended and refashioned into new structures. Patterns are discovered and illuminated.

The mathematics process has produced a body of mathematical knowledge that is truly overwhelming in both size and complexity and that is being enlarged at an ever increasing rate. Compared to other disciplines, the system for managing the mathematics process and the knowledge it produces is very primitive and has changed relatively little in the last half century. Although computers are used extensively for performing computations, they are rarely used for the other parts of the mathematics process. The great majority of mathematical knowledge is expressed in the abbreviated, informal, non-machine-readable style mathematicians have employed for centuries. In this new century, how should mathematics be managed to best facilitate its further production via the mathematics process and its application in science and technology?

This question is one of the most important questions facing mathematics today. We believe that the answer to it should be a *formal framework* that meets the following goals:

1. *Model Representation.* The framework provides a way of representing models and knowledge about models.

2. *Process Facilitation.* The framework facilitates the full process of creating, exploring, and connecting models.

3. *Mechanization.* The framework can be effectively mechanized by a software system.

[1] *Visualization* is another important model exploration technique which is not considered in this paper.

There are two principal candidates for such a framework: *computer theorem proving* and *computer algebra.*

Computer Theorem Proving

Computer theorem proving emphasizes the conjecture proving aspect of the mathematics process. An *axiomatic theory* T is used to represent a collection of one or more mathematical models with similar structure. Formally, T is a triple $(\mathbf{K}, L, \Gamma)$ where $\mathbf{K}$ is a logic (consisting of a family of formal languages and a set of models for each language), L is a language of $\mathbf{K}$, and Γ is a set of formulas of L. The members of Γ are called the *axioms* of T. A *model* of T is a model of $\mathbf{K}$ for L in which each axiom of T holds. The language L provides a common vocabulary for making statements about the models of T. Each logical consequence of the axioms of T holds in each model of T. Example 1.1 below presents a formulation of Peano arithmetic, a famous axiomatic theory that represents the standard model of the natural numbers.

The computer theorem proving framework is mechanized by a wide range of different kinds of *computer theorem provers.* Examples include Automath [45], Coq [2], EVES [15], HOL [36], IMPS [28], Isabelle [47], Mizar [50], Nqthm [5], Nuprl [14], Otter [43], and PVS [46]. Most theorem provers are primarily used to prove conjectures in the context of an axiomatic theory. Other aspects of the mathematics process are usually not well supported. However, some can be used to manage the creation, extension, and connection of axiomatic theories, and some can perform computations in the process of proving conjectures.

EXAMPLE 1.1 (Peano Arithmetic). Let L be a language of second-order logic (SOL) with exactly two nonlogical constants:

1. An individual constant 0.

2. A unary function constant S (the successor function).

(L also contains the binary predicate constant $=$ which is considered a logical constant.)

Let Γ be the set of the following three formulas of L:

1. $\forall\, x \,.\, S(x) \neq 0$ (0 is not a successor).

2. $\forall\, x, y \,.\, S(x) = S(y) \supset x = y$ (S is injective).

3. $\forall\, P \,.\, [P(0) \wedge (\forall\, x \,.\, P(x) \supset P(S(x)))] \supset \forall\, x \,.\, P(x)$ (induction axiom).

PA $= (\text{SOL}, L, \Gamma)$ is the (second-order) theory of Peano arithmetic. **PA** specifies a single model (up to isomorphism), namely, the standard model of the natural numbers. (The operations $+$ and $*$, the relation $<$, and the natural numbers $1, 2, \ldots$, are definable in **PA**.) $\Box$

PA is a powerful theory which is well suited for proving general theorems about the standard model of the natural numbers. However, it has some significant shortcomings. First, let **PA**$'$ be **PA** plus definitions for $+$, $*$, $<$, and each nonzero natural number n. **PA**$'$ is not finitely axiomatizable (even in second-order logic), which means that **PA**$'$ cannot be represented in a computer system without some kind of procedural mechanism for encoding the infinite set of definitions $\{1 = S(0), 2 = S(S(0)), \ldots\}$. Second, to prove an equation such as $4671 * 8334 = 38928114$ directly from the axioms of **PA**$'$ requires a prodigious number of steps, while it can be proved with one calculation using a simple calculator.

Computer Algebra

Computer algebra emphasizes the computational aspect of the mathematics process. An *algorithmic theory* T usually represents a single mathematical model. Formally, T is a triple $(\mathbf{K}, L, \Gamma)$ where $\mathbf{K}$ is a logic, L is a formal language of $\mathbf{K}$, and Γ is a set of algorithms that take expressions of L as input and return expressions of L as output. The language L provides a vocabulary for making statements about the model T represents. The algorithms exhibit the behavior that the model possesses. Example 1.2 below presents a simple algorithmic theory that represents the standard model of the natural numbers.

The computer algebra framework is mechanized by *computer algebra systems.* Examples include Axiom [40], Macsyma [42], Maple [10], and Mathematica [54]. Most computer algebra systems are designed primarily for performing computations. Computations are performed at great speed, but the results are not always reliable. The algorithmic theories in which computation is performed are usually not represented as explicit, manageable units. Conjecture proving is generally not possible since mathematical knowledge is represented algorithmically.

EXAMPLE 1.2 (Natural Number Arithmetic). Let L be a language of first-order logic (FOL) with the following nonlogical constants:

1. Infinitely many individual constants $0, 1, 2, \ldots$.
2. Two binary function constants $+$ and $*$.
3. One binary predicate constant $<$.

(L also includes the binary predicate constant $=$ which is considered a logical constant.) A *numeral* of L is a member of $\{0, 1, 2, \ldots\}$, a *numeric term* of L is a variable-free term of L, and a *numeric formula* of L is an equation $t_1 = t_2$ or inequality $t_1 < t_2$ where t_1 and t_2 are numeric terms of L.

Let eval be an algorithm that, given a numeric term t of L, returns the numeral that "equals" t. Let reduce be an algorithm that, given a numeric formula A, returns the logical constant true or false if A is "true" or "false", respectively. **NNA** $= (\mathrm{FOL}, L, \{\mathsf{eval}, \mathsf{reduce}\})$ is an algorithmic theory of natural number arithmetic. **NNA** specifies the standard model of the natural numbers. $\square$

NNA is a powerful theory for evaluating (variable-free) numeric terms and deciding equations and inequalities between (variable-free) numeric terms. However, it is not at all suitable for proving abstract properties about the natural numbers. For example, it does not provide the means to prove the fundamental theorem of arithmetic that says every natural number > 1 can be factored into a product of primes that is unique up to the order of the factors.

Neither computer theorem proving nor computer algebra fulfills our requirements for a formal framework for managing mathematics. First, some knowledge about mathematical models is best encoded declaratively using axioms, while other knowledge is best encoded procedurally using algorithms that manipulate expressions. A formal framework should allow models and knowledge about models to be represented in both ways. Second, the full process of creating, exploring, and connecting mathematical models should be supported. Emphasizing just conjecture proving or just computation is not enough. The power of the mathematics process comes from the rich interplay of creating models, exploring them using both deduction and computation, and connecting them when they share structure.

Our Proposal for a Formal Framework

In this paper we propose a Formal Framework for Managing Mathematics (FFMM). It is for managing both the mathematics process and the knowledge it produces. FFMM is based on the notion of a *biform theory* which is simultaneously an axiomatic theory and an algorithmic theory. A biform theory represents a collection of mathematical models by encoding knowledge about the models both declaratively and procedurally. FFMM is intended to support the full mathematics process; it provides the means to manage the creation, exploration, and connection of biform theories. FFMM includes facilities for "derivation", "theoremoid construction", and "theory development". *Derivation* is a merger of deduction and computation which is driven by the application of deduction and computation rules called *theoremoids.* Theoremoids are constructed from theorems and *axiomoids*, the primitive theoremoids of a biform theory, using techniques that guarantee soundness. And networks of biform theories are developed by creating biform theories, linking them with interpretations, and installing theorems, theoremoids, and definitions in them.

Many of the ideas and mechanisms used in FFMM are inspired by the IMPS Interactive Mathematical Proof System [31, 28, 29] and the Axiom [40] computer algebra system. The mechanization of FFMM is not discussed in this paper. We believe that FFMM can be mechanized using ideas embodied in computer theorem proving systems like IMPS and computer algebra systems like Axiom and Maple.

This paper presents an overview of FFMM; proofs and many other details are not presented. For a detailed presentation of FFMM, see the technical report [33].

There is a large body of work related to our proposal concerning (1) *logical frameworks* for managing logical systems and investigating metalogical issues and (2) the problem of integrating computer theorem proving and computer algebra. This related work is discussed at the end of the paper in section 11.

The rest of the paper is organized as follows. The properties that a background logic for FFMM must satisfy are discussed in section 2. The central notion of a biform theory is defined in section 3. Two techniques for constructing theoremoids are discussed in section 4. Derivation, the process for exploring biform theories, is introduced in section 5. Connecting biform theories using translations and interpretations is the subject of section 6. A brief overview of the FFMM infrastructure for developing networks of biform theories is found in section 7. Sections 8 and 9 present two special computational devices, algebraic processors and computational models. The paper then ends with a conclusion in section 10 and a survey of related work in section 11.

2. Logics

FFMM is not based on one particular background logic. Instead, mathematics can be formalized within a variety of traditional logics such as first-order logic and simple type theory. Moreover, several background logics can be used in FFMM simultaneously, and mathematical results can be shared between related logics (see section 6).

Each background logic must satisfy a relatively small set of properties. These properties are expressed in the notion of an “admissible logic" defined precisely in [33]. An admissible logic is a classical nonconstructive, two-valued logic defined as a family of "admissible languages" (that usually share a common structure).

Languages

A *language* is a triple $L = (\mathcal{S}, \mathcal{E}, \sigma)$ where:

1. $\mathcal{S}$ is a set of syntactic objects called the *sorts* of L.
2. $\mathcal{E}$ is a set of syntactic objects called the *expressions* of L.
3. $\sigma : \mathcal{E} \rightarrow \mathcal{S}$ is a total function.

The phrase "E is an expression of L of sort α" means that $E \in \mathcal{E}$ and $\sigma(E) = \alpha$.

Let E be an expression of L, E_1 be a subexpression of E at position p in E, and E_2 be an expression of L such that $\sigma(E_1) = \sigma(E_2)$. The result of replacing E_1 at position p in E with E_2 is the syntactic object denoted by $E[p/E_2]$. We assume that $E[p/E_2]$ is an expression of L of the same sort as E. That is, we assume that all languages are closed under the replacement of a subexpression with another expression of the same sort. E is *atomic* if it contains no subexpressions other than itself.

Let $L_i = (\mathcal{S}_i, \mathcal{E}_i, \sigma_i)$ be a language for $i = 1, 2$. L_1 is a *sublanguage* of L_2 and L_2 is an *extension* of L_1, written $L_1 \leq L_2$, if $\mathcal{S}_1 \subseteq \mathcal{S}_2$, $\mathcal{E}_1 \subseteq \mathcal{E}_2$, and σ_1 is a subfunction of σ_2.

A language $L = (\mathcal{S}, \mathcal{E}, \sigma)$ is *admissible* if the following conditions hold:

1. $\mathcal{S}$ contains $*$, which denotes the sort of truth values.
2. true and false are expressions of L of sort $*$.
3. If $E, E', E_1, \ldots, E_n$ are expressions of L of sort $*$ $(0 \leq n)$, then $\neg E$ (negation), $(E \supset E')$ (implication), $\wedge(E_1, \ldots, E_n)$ (multiary conjunction), and $\vee(E_1, \ldots, E_n)$ (multiary disjunction) are expressions of L of sort $*$.
4. If E_1 and E_2 are expressions of L of the same sort, then $(E_1 \simeq E_2)$ is an expression of L of sort $*$ (called an *equation*) that asserts the equality of E_1 and E_2.[2]

[2] In a standard logic, $(E_1 \simeq E_2)$ means that E_1 and E_2 denote the same value, while in a logic that admits undefinedness like LUTINS [18, 19, 20], the logic of IMPS, $(E_1 \simeq E_2)$ means that either E_1 and E_2 denote the same value or E_1 and E_2 are both undefined.

5. If E_1, E_2, E_3 are expressions of L with $\sigma(E_1) = *$ and $\sigma(E_2) = \sigma(E_3) = \alpha$, then $\mathsf{if}(E_1, E_2, E_3)$ is an expression of sort α (called a *conditional*) that denotes E_2 if E_1 is true and denotes E_3 if E_1 is false.

In summary, a language is admissible if it contains certain basic machinery for forming propositional statements, equations, and conditionals. Let a *formula* be an expression of an admissible language of sort $*$.

Models

A semantics for an admissible language L is given by a set of *models* for L (see [33]). Let $\mathcal{M}$ be a set of models for L, $M = (\mathcal{D}, V) \in \mathcal{M}$, A be a formula of L, and Σ be a set of formulas of L. M *satisfies* A, written $M \models A$, if $V(A) = \mathrm{T}$ (i.e., A is true in M). M *satisfies* Σ, written $M \models \Sigma$, if M satisfies each $B \in \Sigma$. A is *valid* with respect to $\mathcal{M}$, written $\models_{\mathcal{M}} A$, if every model in $\mathcal{M}$ satisfies A. A is a *logical consequence* of Σ with respect to $\mathcal{M}$, written $\Sigma \models_{\mathcal{M}} A$, if every model in $\mathcal{M}$ that satisfies Σ also satisfies A.

Local Contexts

Let E and $E_1 \simeq E_2$ be expressions of L such that E_1 is a subexpression of E at position p, $\mathcal{M}$ be a set of models for L, and C be a set of formulas occurring in E. C is a *local context* in E at p with respect to $\mathcal{M}$ if, for all sets Σ of formulas of L,

$$\Sigma \cup C \models_{\mathcal{M}} E_1 \simeq E_2$$

implies

$$\Sigma \models_{\mathcal{M}} E \simeq E[p/E_2].$$

In other words, a local context in an expression E at a position p is a set of formulas in E that govern the subexpression of E occurring at p. For example, $\{A\}$ is a local context at the position where B occurs in $A \supset B$ and $\{A_1, \ldots, A_{i-1}\}$ is a local context at the position where A_i occurs in $\wedge(A_1, \ldots, A_n)$.

The method of local contexts [44] is a powerful idea that is applicable to both deduction and computation. See [28, 30] for examples of how local contexts are used in IMPS to facilitate deduction and computation. In FFMM, local contexts are used to control the local application of deduction and computation rules (see section 5).

Admissible Logics

An *admissible logic* is a triple $\mathbf{K} = (\mathcal{L}, \mu, \kappa)$ such that:

1. $\mathcal{L}$ is a set of admissible languages.
2. For each $L \in \mathcal{L}$, $\mu(L)$ is a set of models for L.
3. For each expression E of $L \in \mathcal{L}$ and position p in E, $\kappa(E, p)$ is a local context in E at p with respect to $\mu(L)$.

Many traditional logics (including propositional logic, first-order logic, and simple type theory) can be formulated as admissible logics. Logics that admit undefinedness, such as LUTINS [18, 19, 20], the logic of IMPS, and other related logics (see [24, 25]), can also be formulated as admissible logics.[3]

For examples later in the paper, let $\mathbf{K}_{\rm stt} = (\mathcal{L}, \mu, \kappa)$ be an admissible logic formulation of Church's simple type theory [11].

3. Theories

In this section we introduce the central notion of a "biform theory", which is a generalization of both an axiomatic theory and an algorithmic theory. The "axioms" of a biform theory have both an axiomatic meaning and an algorithmic meaning. As a result, a biform theory is simultaneously an axiomatic theory and an algorithmic theory. Representing a collection of mathematical models, a biform theory provides a formal context for deduction and computation.

Transformers

Deduction and computation rules are represented in FFMM as algorithms called "transformers" that map expressions of one admissible language to expressions of another.

Let $L_i = (\mathcal{S}_i, \mathcal{E}_i, \sigma_i)$ be an admissible language for $i = 1, 2$. A *transformer* Π from L_1 to L_2 is an algorithm that implements a partial function $\pi : \mathcal{E}_1 \rightharpoonup \mathcal{E}_2$. For $E \in \mathcal{E}_1$, let $\Pi(E)$ mean $\pi(E)$, and let $\mathsf{dom}(\Pi)$ denote the domain of π, i.e., the subset of $\mathcal{E}_1$ on which π is defined. Π *resides in* a language $L = (\mathcal{S}, \mathcal{E}, \sigma)$ if Π is a transformer from L to L and, for all expressions $E \in \mathsf{dom}(\Pi)$, $\sigma(E) = \sigma(\Pi(E))$.

A transformer is intended to be an expression transforming algorithm that preserves meaning or modifies meaning in a prescribed way.

[3] For the logics LUTINS and STMM [24], this requires adding machinery to the framework to handle "subsorts".

For instance, a transformer can represent an evaluator, a simplifier, a rewrite rule, a rule of inference, a decision procedure, or a translation from one language to another. Various examples of transformers will be given later in the paper.

REMARK 3.1. Context-sensitive deduction and computation rules, that give different values under different assumptions, can be represented by transformers that return conditionals.

Suppose Π is a transformer residing in a language L. Let E be an expression of L and E_1 be a subexpression of E at position p in E. The application of Π to E at p, written $\Pi(E, p)$, is the expression $E[p/\Pi(E_1)]$. $\Pi(E, p)$ is undefined if $\Pi(E_1)$ is undefined. (Since L is a language, $E[p/\Pi(E_1)]$ is an expression of L of sort $\sigma(E)$ if $\Pi(E_1)$ is defined.)

Formuloids

A "formuloid" generalizes a formula and an algorithm.

Let L be an admissible language. A *formuloid* of L is a pair $\theta = (k, X)$ where:

1. $k \in \{0, 1, 2, \ldots\}$ is the *kind* of θ.
2. If $k = 0$, then X is a formula A of L and θ is called an *assertional formuloid.*
3. If $k > 0$, then X is a transformer Π residing in L and θ is called a *transformational formuloid.*

A formuloid of kind 1 is called an *equational formuloid.* In this paper, we will concentrate our attention on formuloids of kind 0 and 1, i.e., on assertional and equational formuloids. Many other useful kinds of transformational formuloids can be defined (see [33] for examples).

The *span* of θ, written $\mathsf{span}(\theta)$, is a set of formulas of L. If $k = 0$, $\mathsf{span}(\theta) = \{A\}$. If $k = 1$, $\mathsf{span}(\theta) = \{E \simeq \Pi(E) : E \in \mathsf{dom}(\Pi)\}$. For a transformational formuloid of kind $k > 1$, $\mathsf{span}(\theta)$ will normally be a set of formulas relating the input of Π to its output. For example, $\mathsf{span}(\theta)$ could be a set of implications $E \supset \Pi(E)$, inequalities $E < \Pi(E)$, transitive relations $R(E, \Pi(E))$, etc.

The *operation* of θ, written $\mathsf{oper}(\theta)$, is a transformational formuloid. If $k = 0$, $\mathsf{oper}(\theta) = (1, \Pi_{A \mapsto \mathsf{true}})$ where $\Pi_{A \mapsto \mathsf{true}}$ is a transformer that maps A to true but is undefined on all other expressions. If $k \geq 1$, $\mathsf{oper}(\theta) = \theta$. A formuloid has two meanings. Its *axiomatic* meaning is its span of formulas, and its *algorithmic* meaning is its operation.

Biform Theories

A *biform theory* is a triple $T = (\mathbf{K}, L, \Gamma)$ where:

1. $\mathbf{K} = (\mathcal{L}, \mu, \kappa)$ is an admissible logic called the *logic* of T.

2. L is a member of $\mathcal{L}$ called the *language* of T.

3. Γ is a set of formuloids of L called the *axiomoids* of T.

The *span* of T, written $\mathsf{span}(T)$, is the union of the spans of the axiomoids of T, i.e.,

$$\bigcup_{\theta \in \Gamma} \mathsf{span}(\theta).$$

The *operations* of T, written $\mathsf{oper}(T)$, is the set of operations of the axiomoids of T, i.e.,

$$\{\mathsf{oper}(\theta) : \theta \in \Gamma\}.$$

T can be viewed as having two forms simultaneously: $(\mathbf{K}, L, \mathsf{span}(T))$ is its form as an axiomatic theory and $(\mathbf{K}, L, \mathsf{oper}(T))$ is its form as an algorithmic theory.

A *model* of T is a model $M \in \mu(L)$ such that $M \models \mathsf{span}(T)$. A is an *axiom* of T if $A \in \mathsf{span}(T)$. A is a *theorem* of T, written $T \models A$, if $\mathsf{span}(T) \models_{\mu(L)} A$. Let $\mathsf{thm}(T)$ denote the set of theorems of T. A *theoremoid* of T is a formuloid θ of L such that, for each $A \in \mathsf{span}(\theta)$, $T \models A$. Obviously, each axiomoid of T is also a theoremoid of T. Let $\mathsf{thmoid}(T)$ denote the set of theoremoids of T.

REMARK 3.2. The great majority of commonly used rules of inference can be represented by transformational formuloids of one kind or another. The exceptions include rules like universal generalization ($\forall$-introduction) and existential instantiation ($\exists$-elimination). In [22], we show how rules of inference of this kind can be realized with conservative extensions made from "profiles" (see section 7). □

Let $\mathbf{K} = (\mathcal{L}, \mu, \kappa)$ be an admissible logic and $T_i = (\mathbf{K}, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. T_1 is a *subtheory* of T_2 and T_2 is an *extension* of T_1, written $T_1 \leq T_2$, if $L_1 \leq L_2$ and $\mathsf{thm}(T_1) \subseteq \mathsf{thm}(T_2)$.[4] If $T = (\mathbf{K}, L, \Gamma)$ is a biform theory and Δ is a set of formuloids of

[4] Notice that this definition does not require that $\Gamma_1 \subseteq \Gamma_2$, i.e., that T_2 contain all the axiomoids of T_1. Thus two theories that are equivalent in the sense of being subtheories of each other could be based on entirely different axiomatic and algorithmic assumptions.

languages in $\mathcal{L}$, then $T[\Delta]$ is the extension $T' = (\mathbf{K}, L', \Gamma \cup \Delta)$ of T where L' is the smallest language in $\mathcal{L}$ such that $L \leq L'$ and each $\theta \in \Delta$ is a formuloid of L'. ($T[\Delta]$ may not always be defined.) Obviously, if $T[\Delta]$ is defined, $T \leq T[\Delta]$. For a set of formulas Σ, let $T[\Sigma]$ mean $T[\{(0, A) : A \in \Sigma\}]$, and for a single formula A, let $T[A]$ mean $T[\{A\}]$.

For examples later in the paper, let $T_{\mathrm{stt}} = (\mathbf{K}_{\mathrm{stt}}, L_{\mathrm{stt}}, \Gamma_{\mathrm{stt}})$ be a biform theory formulation of the logical theory of $\mathbf{K}_{\mathrm{stt}}$.

EXAMPLE 3.3 (Biform Theory of Peano Arithmetic). Let $T_{\mathrm{pa}} = (\mathbf{K}_{\mathrm{stt}}, L_{\mathrm{pa}}, \Gamma_{\mathrm{pa}})$ be an extension of T_{stt} such that:

1. $L_{\mathrm{stt}} \leq L_{\mathrm{pa}}$.
2. L_{pa} includes atomic expressions $0, 1, 2, \ldots$ of sort ι; an atomic expression S of sort $(\iota \rightarrow \iota)$; atomic expressions $+$ and $*$ of sort $(\iota \rightarrow (\iota \rightarrow \iota))$; and an atomic expression $<$ of sort $(\iota \rightarrow (\iota \rightarrow *))$. ($\iota$, the sort of individuals, is intended to denote the set of natural numbers.)
3. $\Gamma_{\mathrm{pa}} = \Gamma_{\mathrm{stt}} \cup \{\theta_1, \ldots, \theta_8\}$ where:
   a) θ_1, θ_2, and θ_3 are assertional axiomoids representing the three axioms of the axiomatic theory **PA** presented in Example 1.1.
   b) θ_4, θ_5, and θ_6 are assertional axiomoids that define the operations $+$ and $*$ and the relation $<$, respectively.
   c) θ_7 and θ_8 are equational axiomoids representing the algorithms eval and reduce of the algorithmic theory **NNA** presented in Example 1.2.

T_{pa} is biform theory formulation of Peano arithmetic that combines the axiomatic and algorithmic machinery of **PA** and **NNA**. □

It is important to emphasize that the axiomoids of a biform theory are assumptions. The members of their spans are assumed to be true and the results of their operations are assumed to be sound. Thus, the transformational axiomoids θ_7 and θ_8 of Example 3.3 are taken as assumptions and whether or not they are sound in T_{pa} is not a meaningful question.

4. Theoremoid Construction

In the next section we will present the notion of "derivation", a merger of deduction and computation. We will see that the essence of derivation is the application of theoremoids. Effective derivation requires a well-stocked toolbox of theoremoids for each employed biform theory. Of course, the toolbox of theoremoids for a theory contains the axiomoids of the theory, but these may not embody all the reasoning and computational techniques that are desired. How are other theoremoids obtained?

There are two parts to the answer for a biform theory T. First, if A is a theorem of T, then $(0, A)$ will always be an assertional theoremoid of T. Second, a transformational theoremoid of T is obtained by constructing a transformer Π residing in the language of T and then proving that, for some $k \in \{1, 2, \ldots\}$, (k, Π) is a theoremoid of T.

FFMM does not include a general system for proving that a given transformational formuloid is a theoremoid of a particular theory. Instead, it includes a collection of techniques for constructing transformational formuloids for which theoremoidhood is guaranteed by the construction itself. Two of the techniques are described in this section. A third technique, based on theory interpretation, is described in section 6. Together these techniques constitute a toolkit for building sound deduction and computation tools in the form of transformational theoremoids.

Computing with Theorems

One can define a family of *theorem-to-theoremoid constructors* for $\mathbf{K}_{\mathrm{stt}}$ that automatically generate transformational theoremoids from theorems, in the style of *theorem macetes* as employed in IMPS (see [28, 30]). The form of the generated transformational theoremoid depends on the syntactic form of the theorem.

For example, suppose the theorem A is a formula of the form

$$\forall\, x_1 : \alpha_1, \ldots, x_n : \alpha_n \,.\, B \supset E_1 \simeq E_2$$

where each x_i is a quantified variable of sort α_i that may occur in B, E_1, and E_2. Define θ_A to be $(1, \Pi_A)$ where Π_A is a transformer residing in L defined as follows: If E is alpha-equivalent to $E_1\tau$, where τ is a substitution with domain $\{x_1, \ldots, x_n\}$, then $\Pi(E) = \mathsf{if}(B\tau, E_2\tau, E)$; otherwise $\Pi(E)$ is undefined. θ_A is an equational theoremoid of T that applies A as a conditional rewrite rule (the "reverse" conditional rewrite rule can also be generated from A). When θ_A is applied in a derivation graph, it introduces a conditional that can be resolved by simplification or split with the split-conditional operator (see section 5). Other examples of theorem-to-theoremoid constructors using quantification are given in [32, 33].

This method of generating transformational theoremoids from theorems is a very powerful technique. Transformational theoremoids are designed by writing formulas of the right form, are verified by proving that the formulas are theorems, and, finally, are constructed automatically. Thus the construction of a large collection of useful transformational theoremoids is reduced to essentially just theorem formulation and proving.

Combining Theoremoids

A *theoremoid combinator* forms a new transformational theoremoid from a set of existing transformational theoremoids. Theoremoid combinators are inspired by the *macete constructors* of IMPS (see [28, 30]).

For example, the fixpoint combinator builds a transformational theoremoid whose transformer has the effect of repeating applying the transformer of a given transformational theoremoid to an expression until the expression remains unchanged. Let $T = (\mathbf{K}, L, \Gamma)$ be a biform theory and $\theta = (k, \Pi) \in \mathsf{thmoid}(T)$ where $0 < k$. For $n \geq 1$, define Π^n to be Π if $n = 1$ and $\Pi \circ \Pi^{n-1}$ otherwise. Now define $\mathsf{fixpoint}(\theta)$ to be (k, Π') where Π' is defined as follows. Let E be an expression of L. $\Pi'(E) = E'$ if, for some $n \geq 1$, each of $\Pi^1(E), \ldots, \Pi^{n+1}(E)$ is defined, $\Pi^i(E) \neq \Pi^{i+1}(E)$ for all i with $1 \leq i < n$, and $E' = \Pi^n(E) = \Pi^{n+1}(E)$. $\Pi'(E)$ is undefined otherwise. It is easy to see that whenever θ is an equational theoremoid of T (i.e., $k = 1$), $\mathsf{fixpoint}(\theta)$ is also a equational theoremoid of T.

Several other theoremoid combinators are given in [32, 33].

5. Derivation

In FFMM, a biform theory provides a context for performing both deductions and computations, and more importantly, operations in which deduction and computation are intertwined. We want to replace the unfortunate separation between deduction and computation with a new notion that combines the two, which we will call *derivation.*

We need a formal workspace for building derivations, that is, intertwined deductions and computations. The workspace should work with biform theories of any fixed admissible logic. Our solution is the notion of a "derivation graph" defined in this section. It is a generalization of the notion of a *deduction graph* employed in IMPS (see [28]).

Table I. Derivation Graph Connectors

Name	**Tuple**	**Meaning**
implication	$(1, N_1, N_2)$ where N_1, N_2 are assertional	$N_1 \Rightarrow N_2$
biimplication	$(2, N_1, N_2)$ where N_1, N_2 are assertional	$N_1 \Leftrightarrow N_2$
one-to-many	$(3, N_0, \{N_1, \ldots, N_n\})$ where $N_0, N_1, \ldots, N_n$ are assertional $(1 \leq n)$	$N_0 \Leftrightarrow N_1 \mathbin{\&} \cdots \mathbin{\&} N_n$
computation	$(4, N_1, N_2, N)$ where N is assertional	N

Derivation graphs have several levels. We will concentrate on the base level of a derivation graph that defines derivation at the lowest level. Above the base level are several other levels of structure that are intended to raise derivation to a level at which mathematics practitioners will be comfortable.

Derivation Graphs: Base Level

Let $\mathbf{K}$ be an admissible logic. A *(derivation graph) node* N of $\mathbf{K}$ is a pair (T, E) such that $T = (\mathbf{K}, L, \Gamma)$ is a biform theory and E is an expression of L. The node (T, E) is intended to represent the expression E in the context of the biform theory T. N is *assertional* if E is a formula. An assertional node $N = (T, A)$ represents the assertion $T \models A$. It is analogous to a *sequent node* in an IMPS deduction graph; T plays the role of the *context* and A plays the role of the *assertion*.

Let $N_1 = (T_1, A_1)$ and $N_2 = (T_2, A_2)$ be assertional nodes of $\mathbf{K}$. $N_1 \Rightarrow N_2$ means that, if $T_1 \models A_1$, then $T_2 \models A_2$. $N_1 \mathbin{\&} N_2$ means both $T_1 \models A_1$ and $T_2 \models A_2$ hold. And, $N_1 \Leftrightarrow N_2$ means both $N_1 \Rightarrow N_2$ and $N_2 \Rightarrow N_1$ hold.

There are four kinds of *(derivation graph) connectors* named *implication*, *biimplication*, *one-to-many*, and *computation*. They are given in Table I. Each connector is a tuple consisting of a *kind* $k \in \{1, 2, 3, 4\}$, and a certain collection of nodes. The intended meaning of each kind of connector is given in the table. A derivation graph connector is analogous to an *inference node* in an IMPS deduction graph; in fact, a inference node can be represented as a combination of an implication connector and a one-to-many connector.

The computation connector is used to connect two nodes $N_1 = (T_1, E_1)$ and $N_2 = (T_2, E_2)$ related to each other by a computation. The relationship between E_1 and E_2 is recorded by an assertional node $N = (T, A)$. The form of A is not restricted. For example, A could be an equation $E_1 \simeq E_2$, an inequality $E_1 < E_2$, a transitive relation $R(E_1, E_2)$, etc.

A *derivation graph* of $\mathbf{K}$ is a pair $G = (\mathcal{N}, \mathcal{C})$ such that $\mathcal{N}$ is a finite set of nodes of $\mathbf{K}$ and $\mathcal{C}$ is a finite set of connectors that contain only nodes in $\mathcal{N}$. A derivation graph is intended to record a web of deductions and computations. Trees of assertional nodes connected by the first three kinds of connectors represent deductions, while sequences of nodes connected by computation connectors represent computations.

Let a *truth node* be an assertional node of the form (T, true). For each G, let N_G^{true} be a canonical truth node. An *initial derivation graph* is a derivation graph G of the form $(\{N_G^{\mathsf{true}}\}, \emptyset)$. Derivation graphs are built from an initial derivation graph G by applying "operators" that add new nodes and connectors to G.

There are nine primitive *(derivation graph) operators* for adding nodes and connectors to a derivation graph. They are defined in Table II. Each operator takes a derivation graph $G = (\mathcal{N}, \mathcal{C})$ (of an admissible logic $\mathbf{K}$) and other objects (the inputs), and returns a derivation graph

$$G' = (\mathcal{N} \cup \mathcal{N}', \mathcal{C} \cup \mathcal{C}')$$

obtained by adding a finite set $\mathcal{N}'$ of nodes (the output nodes) and a finite set $\mathcal{C}'$ of connectors (the output connectors) to G. (The output nodes are required to be nodes of $\mathbf{K}$.)

add-node simply adds a node to the derivation graph. strengthen-1 and strengthen-2 create new nodes by replacing the theory T of a node in the derivation graph with an extension of T. eliminate-implication, split-conjunction, split-biimplication, and split-conditional are restructuring operators. apply-0-thmoid and apply-1-thmoid are operators for applying theoremoids of kind 0 and 1, respectively, to nodes in the derivation graph. They provide the means to employ the formulas and transformers asserted by the axiomoids and other theoremoids of biform theories. Notice that these operators exploit the local context at the position where a theoremoid is to be applied. For each kind of theoremoid of kind $k > 1$, there will be a corresponding operator theoremoid apply-k-thmoid.

Each operator is well defined in the sense that, if an operator is applied to a derivation graph, the result is still a derivation graph. A derivation graph is *admissible* if it is an initial derivation graph or it is the result of applying an operator to an admissible derivation graph.

Table II. The Primitive Derivation Graph Operators

Name	**Input Objects**	**Output Objects**
add-node	$T = (\mathbf{K}, L, \Gamma)$ is a biform theory E is an expression of L	$N = (T, E)$
strengthen-1	$N = (T, A) \in \mathcal{N}$ is assertional T' is a biform theory with $T \leq T'$	$N' = (T', A)$ $C = (1, N, N')$
strengthen-2	$N = (T, E) \in \mathcal{N}$ T' is a biform theory with $T \leq T'$	$N' = (T', E)$ $N_0 = (T', E \simeq E)$ $C_1 = (4, N, N', N_0)$ $C_2 = (2, N_0, N_G^{\mathsf{true}})$
eliminate-implication	$N = (T, A_1 \supset A_2) \in \mathcal{N}$	$N' = (T[A_1], A_2)$ $C = (2, N, N')$
split-conjunction	$N = (T, \wedge(A_1, \ldots, A_n)) \in \mathcal{N}$	$N_i = (T, A_i)$ for $i = 1, \ldots, n$ $C = (3, N, \{N_1, \ldots, N_n\})$
split-biimplication	$N = (T, A_1 \simeq A_2) \in \mathcal{N}$ where A_1 and A_2 are formulas	$N_1 = (T, A_1)$ $N_2 = (T, A_2)$ $C = (2, N_1, N_2)$
split-conditional	$N = (T, E) \in \mathcal{N}$ p is a position of $\mathsf{if}(A, E_1, E_2)$ in E	$N_0 = (T[\kappa(E, p)], A)$ $N' = (T, E[p/E_1])$ $N_1 = (T, E \simeq E[p/E_1])$ $C_1 = (4, N, N', N_1)$ $C_2 = (1, N_0, N_1)$
apply-0-thmoid	$N = (T, E) \in \mathcal{N}$ p is a position of A in E $\theta = (0, A) \in \mathsf{thmoid}(T[\kappa(E, p)])$	$N' = (T, E[p/\mathsf{true}])$ $N_0 = (T, E \simeq E[p/\mathsf{true}])$ $C_1 = (4, N, N', N_0)$ $C_2 = (2, N_0, N_G^{\mathsf{true}})$
apply-1-thmoid	$N = (T, E) \in \mathcal{N}$ p is a position in E $\theta = (1, \Pi) \in \mathsf{thmoid}(T[\kappa(E, p)])$ and $\Pi(E, p)$ is defined	$N' = (T, \Pi(E, p))$ $N_0 = (T, E \simeq \Pi(E, p))$ $C_1 = (4, N, N', N_0)$ $C_2 = (2, N_0, N_G^{\mathsf{true}})$

5.1. Deductions and Computations

Fix an admissible derivation graph $G = (\mathcal{N}_G, \mathcal{C}_G)$. A *deduction* of N from $\mathcal{N}$ in G is a pair $\Delta = (\mathcal{N}_\Delta, \mathcal{C}_\Delta)$ such that $\{N\} \cup \mathcal{N} \subseteq \mathcal{N}_\Delta \subseteq \mathcal{N}_G$ and $\mathcal{C}_\Delta \subseteq \mathcal{C}_G$ and one of the following is true:

1. $\mathcal{N}_\Delta = \mathcal{N} = \{N\}$ and $\mathcal{C}_\Delta = \emptyset$.

2. $\Delta' = (\mathcal{N}_{\Delta'}, \mathcal{C}_{\Delta'})$ is a deduction of N' from $\mathcal{N}$ in G, $\mathcal{N}_\Delta = \mathcal{N}_{\Delta'} \cup \{N\}$, and $\mathcal{C}_\Delta = \mathcal{C}_{\Delta'} \cup \{C\}$ where C is $(1, N', N)$, $(2, N', N)$, or $(2, N, N')$.

3. $\Delta_i = (\mathcal{N}_{\Delta_i}, \mathcal{C}_{\Delta_i})$ is a deduction of N_i from $\mathcal{N}_i$ in G for all i with $1 \leq i \leq n$ $(1 \leq n)$, $\mathcal{N}_\Delta = \mathcal{N}_{\Delta_1} \cup \cdots \cup \mathcal{N}_{\Delta_n} \cup \{N\}$, $\mathcal{N} = \mathcal{N}_1 \cup \cdots \cup \mathcal{N}_n$, and $\mathcal{C}_\Delta = \mathcal{C}_{\Delta_1} \cup \cdots \cup \mathcal{C}_{\Delta_n} \cup \{(3, N, \{N_1, \ldots, N_n\})\}$.

A *proof* of N in G is a deduction of N from a set of truth nodes in G.

THEOREM 5.1 (Soundness of Deductions). *Let G be an admissible derivation graph. If there is a deduction of N from $\mathcal{N} = \{N_1, \ldots, N_n\}$ in G, then $N_1 \;\&\; \cdots \;\&\; N_n \Rightarrow N$. Moreover, if there is a proof of $N = (T, A)$ in G, then $T \models A$.*

Let N and N' be nodes of G and T be a biform theory. An *equational computation* from N to N' in T and G is a sequence

$$\langle N_0, (4, N_0, N_1, N^1), N_1, \ldots, N_{n-1}, (4, N_{n-1}, N_n, N^n), N_n \rangle$$

$(0 < n)$ of alternating nodes and computation connectors in G such that:

1. $N = N_0$ and $N' = N_n$.

2. $N_i = (T_i, E_i)$ for all i with $0 \leq i \leq n$.

3. $N^i = (T_i, E_{i-1} \simeq E_i)$ for all i with $1 \leq i \leq n$.

4. $T_i \leq T$ for all i with $1 \leq i \leq n$.

THEOREM 5.2 (Soundness of Equational Computations). *Let T be a biform theory and G be an admissible derivation graph. If there is an equational computation from $N_1 = (T_1, E_1)$ to $N_2 = (T_2, E_2)$ in T and G, then $T \models E_1 \simeq E_2$.*

Derivation Graphs: Higher Levels

At the base level, a derivation graph consists of set of nodes and connectors and is construction by the application of operators. A derivation graph has at least four additional levels of structure above the base level:

1. *Annotations.* At this level, a derivation graph is annotated with the operator applications used to construct it.

2. *Tactics.* Composite derivation graph operators that apply primitive derivation graph operators in certain specified ways can be introduced in the style of *tactics* [35]. For example, a tactic to *split a conditional negatively* applied to a conditional $C =$

$\mathsf{if}(A, E_1, E_2)$ in a node N would (1) apply apply-1-thmoid to N with an appropriate theoremoid to create a new node N' in which C is replaced by $C' = \mathsf{if}(\neg A, E_2, E_1)$ and then (2) apply split-conditional to C' in N'. At this level, a derivation graph is constructed by the application of tactics, and primitive derivation graph operators can only be applied indirectly via tactics.

3. *Scripts.* A *derivation graph script* is a list of tactic applications for building a derivation graph. It serves as a compact prescriptive representation of the derivation graph. Scripts are convenient for storing derivation graphs, building new tactics, and reusing parts of a derivation graph construction. (Proof scripts and their use are discussed in [26].) At this level, a derivation graph includes a derivation graph script that records the history of how it was constructed via tactics.

4. *Tracking.* A derivation graph can easily become a mess of incomplete deductions and computations. At this level, the partial deductions and computations within a derivation graph and the theorems they produce along the way are tracked as the derivation graph is constructed.

6. Translations and Interpretations

"Translations" are transformers that connect one biform theory to another. "Interpretations" are meaning preserving translations.

Let $\mathbf{K}_i$ be an admissible logic and $T_i = (\mathbf{K}_i, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. A *translation* from T_1 to T_2 is a transformer Φ from L_1 to L_2 that respects sorts, i.e., if E_1 and E_2 are expressions of L_1 of the same sort and $\Phi(E_1)$ and $\Phi(E_2)$ are defined, then $\Phi(E_1)$ and $\Phi(E_2)$ are also of the same sort. In the case when the two logics are the same, i.e., $\mathbf{K}_1 = \mathbf{K}_2$, a translation will normally be a homomorphism with respect to the syntactic structure of the logic (see [20]). Hence, for example, $\Phi(A_1 \supset A_2)$ would equal $\Phi(A_1) \supset \Phi(A_2)$ whenever $\Phi(A_1)$ and $\Phi(A_2)$ are defined.

An *interpretation* of T_1 in T_2 is a translation Φ from T_1 to T_2 such that, for all formulas A of L_1, if $T_1 \models A$ and $\Phi(A)$ is defined, then $T_2 \models \Phi(A)$. In other words, an interpretation is a translation that maps theorems to theorems (see [17, 20, 51]).

Translations and interpretations are a powerful mechanism for connecting biform theories with similar structure. They serve as conduits for passing information (in the form of formulas) from one theory to another. Translations transport problems (i.e., conjectures), while interpretations transport solutions (i.e., theorems). A translation may not preserve meaning, but an interpretation connects an abstract theory to a more concrete theory, or an equally abstract theory, in a meaning preserving way. Interpretations enable the *little theories method* [27], in which mathematical knowledge and reasoning is distributed across a network of theories, to be applied to biform theories.

EXAMPLE 6.1. Let M be a biform theory of monoids, and let $\Phi_{0,+}$ and $\Phi_{1,*}$ be interpretations of M in $T_{\rm pa}$ (see Example 3.3) that interpret the unit and binary operator of M by 0 and $+$ of $T_{\rm pa}$ and 1 and $*$ of $T_{\rm pa}$, respectively. These interpretations enable theorems about monoids to be "transported" to $T_{\rm pa}$. $\Box$

REMARK 6.2. In some cases, an interface for a mechanized mathematics system S can be formalized as a biform theory T_S. Suppose S_1 and S_2 are mechanized mathematics systems with biform theory interfaces T_{S_1} and T_{S_2}. Then problems and solutions could be passed between S_1 and S_2 using translations and interpretations, respectively, between T_{S_1} and T_{S_2}.

Theoremoid Instantiation

Theoremoid instantiation is another technique for constructing transformational formuloids for which theoremoidhood is guaranteed by the construction itself.

Let Φ be a translation from T_1 to T_2. Suppose Π is a transformer residing in L_1. Π' is an *instance* of Π via Φ if Π' implements the function that maps $\Phi(E)$ to $\Phi(\Pi(E))$ for each $E \in \mathsf{dom}(\Phi) \cap \mathsf{dom}(\Pi)$ with $\Pi(E) \in \mathsf{dom}(\Phi)$. Π' is clearly a transformer residing in L_2.

Let θ be a formuloid of L_1. For $\theta = (0, A)$ such that $\Phi(A)$ is defined, *the instance* of θ via Φ is the formuloid $(0, \Phi(A))$. For $\theta = (k, \Pi)$ where $k \in \{1, 2, \ldots\}$, *an instance* of θ via Φ is a formuloid (k, Π') where Π' is an instance of Π via Φ.

PROPOSITION 6.3. *If Φ be an interpretation of T_1 in T_2, then the instance of an assertional theoremoid of T_1 via Φ is an assertional theoremoid of T_2.*

PROPOSITION 6.4. *Let Φ be an interpretation of T_1 in T_2 such that, for all expressions E_1 and E_2 in the language of T_1, if $\Phi(E_1)$ and $\Phi(E_2)$ are defined, then $\Phi(E_1 \simeq E_2) = \Phi(E_1) \simeq \Phi(E_2)$. Then each instance of an equational theoremoid of T_1 via Φ is an equational theoremoid of T_2.*

We can now describe the technique of theoremoid instantiation: Create a formuloid θ residing in the language of a theory T that is instantiable via interpretations of T. Prove, in one way or another, that θ is an equational theoremoid of T. Then, by Proposition 6.4, each instance of θ via an appropriate interpretation of T in T' will be a theoremoid of T'. Theoremoid instantiation also works for constructing other kinds of transformational theoremoids.

7. Theory Development

In FFMM, derivation is performed on top of a network of biform theories connected by translations and interpretations. The network is not static. Like the collection of models in informal mathematics itself, it needs to be continuously expanded and enriched. FFMM therefore includes a facility for developing theories that provides services to:

1. Create new biform theories.
2. Create new links between biform theories using translations and interpretations.
3. Store derived theorems.
4. Store constructed theoremoids.
5. Add new objects and concepts to biform theories using conservative extensions.

Our theory development facility for biform theories is based on the infrastructure for developing axiomatic theories presented in [22] and partially implemented in IMPS. It consists of several kinds of storage objects and a collection of primitive operations for creating and modifying the storage objects. In this section, we will give just a brief overview of the theory development infrastructure for FFMM.

Let $\mathbf{K} = (\mathcal{L}, \mu, \kappa)$ be an admissible logic. Recall that an *atomic expression* of a language L is an expression of L that contains no subexpressions other than itself. Let $T_i = (\mathbf{K}, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. T_2 is a *conservative extension* of T_1, written $T_1 \trianglelefteq T_2$, if $T_1 \leq T_2$ and, for all formulas A of L_1, if $T_2 \models A$, then $T_1 \models A$.

A *biform theory object* $\mathbf{T}$ stores a "development" of a biform theory. More specifically, $\mathbf{T}$ includes a *base (biform) theory* $T_0 = (\mathbf{K}, L, \Gamma_0)$, a *current (biform) theory* $T = (\mathbf{K}, L, \Gamma)$ such that $T_0 \trianglelefteq T$, a set of *derived theorems* of T, and a set of *constructed theoremoids* of T. There are also objects for storing translations, interpretations, theorems, theoremoids, definitions, and profiles (see below).

A *definition* D is a pair $(\mathcal{A}, \theta)$ where $\mathcal{A}$ is a set of atomic expressions and $\theta = (1, \Pi)$ is an equational formuloid with $\mathsf{dom}(\Pi) = \mathcal{A}$ called the *defining axiomoid* of D. D is installed in a biform theory T by changing T to $T[\{\theta\}]$. The result is a new theory $T[D]$ in which a has the value $\Pi(a)$ for each $a \in \mathcal{A}$. The installation of D in T is only allowed if $T[D]$ is defined and $T \trianglelefteq T[D]$. Hence, a definition is a means to introduce new machinery into T without compromising its original machinery. Moreover, each defined atomic expression $a \in \mathcal{A}$ can be "eliminated" from expressions of $T[D]$ using $a \simeq \Pi(a)$ as a rewrite rule. D is *simple* if $\mathcal{A}$ is a singleton. Simple definitions are much more commonly used in practice than nonsimple definitions.

EXAMPLE 7.1. Let $T'_{\rm pa} = (\mathbf{K}_{\rm stt}, L'_{\rm pa}, \Gamma'_{\rm pa})$ be the biform obtained from $T_{\rm pa}$ in Example 3.3 by removing the atomic expressions $1, 2, 3, \ldots$ from $L_{\rm pa}$ and the axiomoids θ_7 and θ_8 from $\Gamma_{\rm pa}$. $T'_{\rm pa}$ is a biform theory formulation of the axiomatic theory **PA** presented in Example 1.1 (with definitions for $+$, $*$, and $<$). Let $D = (\mathcal{A}, (1, \Pi))$ be the nonsimple definition such that $\mathcal{A} = \{1, 2, 3, \ldots\}$ and, for each $n \in \mathcal{A}$, $\Pi(n) = S(\cdots(S(0))\cdots)$ where S is applied n times. $T'_{\rm pa} \trianglelefteq T'_{\rm pa}[D]$ and θ_7 and θ_8 are theoremoids of $T'_{\rm pa}[D]$. Moreover, $T'_{\rm pa}[D]$ is equivalent to $T_{\rm pa}$, i.e., $T'_{\rm pa}[D] \leq T_{\rm pa}$ and $T_{\rm pa} \leq T'_{\rm pa}[D]$. $\Box$

A *profile* P is a pair $(\mathcal{A}, \theta)$ where $\mathcal{A}$ is a set of atomic expressions and θ is a formuloid called the *profiling axiomoid* of P. P is installed in a biform theory T by changing T to $T[\{\theta\}]$. The result is a new theory $T[P]$ in which the members of $\mathcal{A}$ satisfy the properties expressed by θ. A profile is thus a generalization of a definition. The installation of P in T is only allowed if $T[P]$ is defined and $T \trianglelefteq T[P]$. Hence, like a definition, a profile is a means to introduce new machinery into T without compromising its original machinery. But, unlike a definition, the profiled atomic expressions in $\mathcal{A}$ may not be eliminable. Profiles can introduce abstract machinery that is impossible to introduce with direct definitions.

For example, in a biform theory R of real number arithmetic consider a profile with the assertional profiling axiomoid $(0, A)$ where $A = (\sqrt{2})^2 - 2 \simeq 0$ is obtained by applying the predicate $\lambda\, x \,.\, x^2 - 2 \simeq 0$ to the atomic expression $\sqrt{2}$. The profile introduces $\sqrt{2}$ in R as an expression that denotes one of the two square roots of 2 (but which one is unspecified). For another example, a profile can be used to introduce in a biform theory an abstract algebra or data type consisting of a collection of objects plus a set of operations on the objects.

There are primitive operations for creating each kind of storage object and for "installing" theorem, theoremoid, definition, and profile objects in a biform theory object $\mathbf{T}$. The result of installing a theorem or theoremoid object in $\mathbf{T}$ is that the theorem or theoremoid is added to the derived theorems or constructed theoremoids of $\mathbf{T}$, respectively. The result of installing a definition or profile object X in $\mathbf{T}$ is that the current theory T of $\mathbf{T}$ is replaced by $T[X]$. Replacement is appropriate because $T[X]$ is a conservative extension of T.

There is also a primitive operation for extending translations (and interpretations). When the current theory T of a biform theory object $\mathbf{T}$ is replaced by an extension $T[X]$ of T, the stored translations of T would not normally be defined on the defined or profiled atomic expressions of X. Three basic solutions to this problem are discussed in [22]. The first two solutions extend the old translations of T automatically to new translations of $T[X]$, while the third solution is to provide the user with a primitive operation for extending translations. With a mechanism for extending translations, it is often advantageous to create a translation of the base theory of a biform theory object and then extend it later as needed. This is the reason why the base theory of $\mathbf{T}$—which is the initial current theory of $\mathbf{T}$—is permanently stored in $\mathbf{T}$.

Many useful theory development operations could be defined using these primitive operations. For examples, operations could be defined for transporting theorems, theoremoids, definitions, and profiles from one biform theory object to another and for creating new biform theory objects by instantiating an existing biform theory object, in both cases using interpretations (see [22]).

8. Algebraic Processors

By changing atomic expressions, a transformer for simplifying expressions in a general biform theory T (say of fields) can often be turned into a transformer for simplifying expressions in a more specialized biform theory T' (say of real arithmetic). Moreover, if the transformer is sound for T and T' has the right structure, the modified transformer will be sound for T'. This is the idea behind the notion of an "algebraic processor", an equational theoremoid that can be specialized via an interpretation.

An *algebraic processor* is a pair $P = (T, \theta)$ where:

1. $T = (\mathbf{K}, L, \Gamma)$ is a biform theory that is intended to represent a class of mathematical structures (such as monoids, vector spaces, partial orders, etc.).

2. $\theta = (1, \Pi)$ is an equational theoremoid of T that is intended to simplify expressions of L.

T encodes the properties that are required for θ to be a theoremoid. An *instance* of P via an interpretation Φ of T in T' is an algebraic processor $P' = (T', \theta')$ where θ' is an instance of θ via Φ.

EXAMPLE 8.1. Suppose $P = (T, (1, \Pi))$ is an algebraic processor where T is a biform theory of commutative semirings with additive and multiplicative identities and natural number exponents. It would be reasonable for Π to simplify an expression

$$x + x * x * x + x$$

to the expression

$$x^{(1+1+1)} + (1 + 1) * x.$$

Notice that the first and second sum of 1s are actually very different expressions: the first denotes a natural number and the second denotes a semiring element. Moreover, if T contains appropriate computational models (see section 9), Π could actually simplify $x + x * x * x + x$ to $x^3 + 2 * x$.

Suppose further that Φ is an interpretation of T in T_{pa} such that there is an instance P' of P via Φ. Then P' will be algebraic processor for T_{pa} which is a specialization of P. $\Box$

An *algebraic processor constructor* is a procedure for creating algebraic processors from other algebraic processors. A small collection of algebraic processor constructors together with algebraic processor instantiation can be used to construct a large variety of algebraic processors. For example, IMPS provides two instantiable algebraic processor constructors, one for modules (i.e., vector spaces over a ring instead of a field) and one for partial orders (see [29]). They have been combined to make simplifiers for theories of real arithmetic, octet arithmetic, abstract fields, and vector spaces over the real numbers.

9. Computational Models

A *domain of computation* (*domain* for short) is a set of data structures that represents a set of mathematical elements (such as the integers) and a set of operations on the data structures that implement mathematical functions (such as addition and multiplication). Domains play a fundamental role in the Axiom computer algebra system [40] (and in the Aldor programming language [53]). In fact, Axiom is equipped with a sophisticated programming language for constructing simple domains (such as a domain of the integers) and complex domains from other simpler domains (such as a domain of the rational numbers constructed from a domain of the integers). However, the "background theory" of an Axiom domain is only implicit. The notion of a "computation model" cements an Axiom-style domain to a background theory in the form of a biform theory.

A *representation* is a triple $R = (L, \mathcal{S}, h)$ such that:

1. $L = (\{\alpha\}, \mathcal{E}, \sigma)$ is a (usually nonadmissible) language where each $E \in \mathcal{E}$ is an expression of sort α.
2. $\mathcal{S}$ is a (possibly infinite) set of data structures.
3. h is a bijection from $\mathcal{E}$ to $\mathcal{S}$.

An *instance* of R via a translation Φ is a representation

$$R' = ((\{\alpha'\}, \mathcal{E}', \sigma'), \mathcal{S}, h')$$

such that Φ is a bijective from $\mathcal{E}$ to $\mathcal{E}'$ and $h' = h \circ g$ where g is the restriction of the inverse of Φ to $\mathcal{E}'$.

EXAMPLE 9.1. Let $L_{\rm nn}$ be the sublanguage $(\{\iota\}, \mathcal{E}, \sigma)$ of $L_{\rm pa}$ (see Example 3.3) where $\mathcal{E} = \{0, 1, 2, \ldots\}$. Let $\mathcal{S}_{\rm nn}$ be a set of character strings of base-10 numerals that represent the natural numbers. And let $h_{\rm nn}$ be a function from $\mathcal{E}$ to $\mathcal{S}_{\rm nn}$ that maps each $n \in \mathcal{E}$ to the string $s \in \mathcal{S}_{\rm nn}$ that represents n. Then $R_{\rm nn} = (L_{\rm nn}, \mathcal{S}_{\rm nn}, h_{\rm nn})$ is a representation (of the natural numbers). $\Box$

A *computational model* is a tuple $M = (T, R, D, \Theta)$ such that:

1. $T = (\mathbf{K}, L, \Gamma)$ is a biform theory.
2. $R = (L', \mathcal{S}, h)$ is a representation such that $L' \leq L$.
3. D is a domain consisting of the set $\mathcal{S}$ of data structures and a set of operations.
4. Θ is a finite set of transformational theoremoids of T such that, for each $\theta = (k, \Pi) \in \Theta$, Π uses the data structures and operations of D to compute the values of expressions constructed from the expressions of L'.

An *instance* of M via an interpretation Φ of T in T' is a computational model $M' = (T', R', D, \Theta')$ such that R' is an instance of R via Φ and each $\theta' \in \Theta'$ is an instance of some $\theta \in \Theta$ via Φ. (Notice that the domain is the same in both M and M'.)

EXAMPLE 9.2. Recall the theory

$$T_{\rm pa} = (\mathbf{K}_{\rm stt}, L_{\rm pa}, \{\theta_1, \ldots, \theta_8\})$$

from Example 3.3 and the representation $R_{\rm nn} = (L_{\rm nn}, \mathcal{S}_{\rm nn}, h_{\rm nn})$ from Example 9.1. Let $D_{\rm nn}$ be a domain consisting of the set $\mathcal{S}_{\rm nn}$ of data structures and operations for adding and multiplying the members of $\mathcal{S}_{\rm nn}$. If θ_7 and θ_8 are defined using the data structures and operations of $D_{\rm nn}$, then $(T_{\rm pa}, R_{\rm nn}, D, \{\theta_7, \theta_8\})$ is a computational model. $\Box$

Two entities similar to computational models are provided by IMPS (see [29]). The first is for arithmetic over the integers, and the second, which extends the first, is for arithmetic over the rational numbers. Both are in the IMPS theory of real arithmetic.

A *computational model constructor* is a procedure for creating computational models. It is an expansion of the Axiom concept of a domain constructor. Simple computational models are created by constructors that take no arguments. Complex computational models are created by constructors that take one or more computational models as arguments. For example, a computational model constructor could build a model of polynomials with rational coefficients from a model of the rational numbers. A computational model constructor could also build a model of a field of quotients from a model that is an integral domain.

10. Conclusion

In this paper we have proposed a formal framework called FFMM for managing the mathematics process and the mathematics knowledge produced by the process. We claim that FFMM meets the three goals given in the Introduction.

Model Representation. A biform theory, which is simultaneously an axiomatic theory and an algorithmic theory, is used to represent a collection of mathematical models. The properties of the models are specified both declaratively and procedurally.

Process Facilitation. Mathematical models are created, explored, and connected via biform theories. The theory development facility provides operations for creating biform theories and storing them in biform theory objects. It also has operations for connecting biform theories with translations and interpretations and for developing biform theories by installing theorems, theoremoids, definitions, and profiles in biform theory objects. Biform theories are explored using the derivation facility. Driven by the application of theoremoids, derivation is a combination of deduction and computation that produces theorems. The theorems represent knowledge about models, and the theoremoids are tools for reasoning and computing.

The mathematics process in FFMM is thus divided into a triad of symbiotic processes: (1) Biform theories are created and incrementally enriched with new language and theoremoids. (2) Theorems are derived by applying the theoremoids of biform theories. And (3) new theoremoids are constructed from the theorems and the existing theoremoids of biform theories.

Mechanization. We have not discussed in this paper how FFMM can be mechanized as a computer system. It is a subject for an entirely separate paper. Although we have not produced a computer mechanization of FFMM, we believe that it is mechanizable and we intend to mechanize it in the future. We expect to borrow heavily from the implementation ideas employed in IMPS, Axiom, Maple, and other mechanized mathematics systems. Moreover, we view the success of the IMPS and Axiom implementations as a proof of concept for our framework proposal.

An implementation of FFMM would be a kernel for an *interactive mathematics laboratory* (IML) [21, 23] with which students, engineers, scientists, and even mathematicians could create, explore, and connect mathematics in countless ways that are not possible today—at least for the common mathematics practitioner. An IML that supports the full mathematics process, is equipped with a well-endowed mathematics library, and is accessible to a wide range of mathematics practitioners has the potential to revolutionize how mathematics is learned and practiced.

11. Related Work

A *logical framework* is a system for managing logical systems and investigating metalogical issues. There is a large literature on the design and use of logical frameworks (see F. Pfenning's Web guide to logical frameworks [48]). Many logical frameworks have been proposed which provide one or more of the following services:

S1 Representation of logical systems.

S2 Implementation of logical systems.

S3 Interoperation of logical systems.

S4 Analysis of metalogical issues.

FFMM is a logical framework that provides services **S1** and **S3**. FFMM manages logical systems represented as biform theories. As we have shown, in FFMM biform theories can be connected with translations and interpretations and incrementally enriched, and a derivation can involve many different biform theories (of the same logic). Most logical frameworks deal with logical systems for deduction, but biform theories are for mixed deduction and computational. The notion of a biform theory is both simple and abstract: the details about the syntax and semantics of a biform theory T are given by the underlying logic of T and the details about derivation in T are given by the axiomoids of T.

The problem of integrating computer theorem proving and computer algebra systems is one of the primary challenges in mechanized mathematics today. A mechanized mathematics system that combines the capabilities of a computer theorem proving system and a computer algebra system would be of great value to a wide range of mathematics practitioners. Unfortunately, there has historically been very little communication between the computer theorem proving and computer algebra communities. Recently, researchers in Europe and North America have begun to pursue ways of integrating computer theorem proving and computer algebra (for example, see [7, 9, 13, 52]).

There are four general approaches for creating an integrated system.

First, computational capabilities are added to a computer theorem proving system. Computation in various forms has been added to many computer theorem proving systems. Examples include (1) decision and simplification procedures (rewrite rule systems, propositional simplification using binary decision diagrams (BDDs), linear arithmetic [6], and generic algebraic simplification in IMPS [28, 30]) and (2) mechanisms for applying theorems and rules of inference (LCF-style tactics [35], IMPS macetes [28, 30], and IMPS proof scripts [26]).

Second, deductive capabilities are added to a computer algebra system. Examples include (1) Analytica [12], a computer theorem proving system for mathematical analysis implemented in Mathematica, (2) the incorporation of logic into the computer algebra system Axiom [49], and (3) the addition of formal proof capabilities to Maple using PVS libraries [16].

Third, a computer theorem proving system and a computer algebra system are combined. Examples include (1) systems combining a computer theorem proving system with a computer algebra system [37, 38] and (2) frameworks and techniques for integrating computer theorem proving and computer algebra systems [1, 3, 4, 34, 39, 41].

Fourth, a system is created in which deduction and computation are integrated at the bottom level. Examples include (1) the Theorema system [8] which is intended to support the full process of mathematical problem solving including conjecture proving and computation and (2) the framework FFMM proposed in this paper.

Acknowledgements

The authors thank Bell Canada and MITACS for supporting this research.

References

1. Armando, A. and D. Zini: 2001, 'Interfacing Computer Algebra and Deduction Systems'. In: M. Kerber and M. Kohlhase (eds.): *Symbolic Computation and Automated Reasoning.* A. K. Peters, pp. 49–64.
2. Barros et al., B.: 1997, 'The Coq Proof Assistant Reference Manual, Version 6.1'. Available at `ftp://ftp.inria.fr/INRIA/coq/V6.1/doc/Reference-Manual.dvi.gz`.
3. Benzmüller, C., M. Jamnik, M. Kerber, and V. Sorge: 2001, 'An Agent-Oriented Approach to Reasoning'. In: S. Linton and R. Sebastiani (eds.): *CALCULEMUS-2001.* pp. 48–63.
4. Bertoli, P., J. Calmet, F. Giunchiglia, and K. Homann: 1999, 'Specification and Integration of Theorem Provers and Computer Algebra Systems'. *Fundamenta Informaticae* **39**.
5. Boyer, R. and J. Moore: 1988, *A Computational Logic Handbook.* Academic Press.
6. Boyer, R. S. and J. S. Moore: 1985, '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.
7. Buchberger, B.: 1996, 'Symbolic Computation: Computer Algebra and Logic'. In: F. Baader and K. U. Schulz (eds.): *Frontiers of Combining Systems*, Applied Logic Series. Kluwer Academic Publishers, pp. 193–220.
8. Buchberger, B., C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Văsaru, and W. Windsteiger: 2001, 'The TH∃OREM∀ Project: A Progress Report'. In: M. Kerber and M. Kohlhase (eds.): *Symbolic Computation and Automated Reasoning.* A. K. Peters, pp. 98–113.
9. Calculemus Project, 'Calculemus Project: Systems for Integrated Computation and Deduction'. Web site at `http://www.mathweb.org/calculemus/`.
10. Char, B. W., K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt: 1991, *Maple V Language Reference Manual.* Springer-Verlag.
11. Church, A.: 1940, 'A Formulation of the Simple Theory of Types'. *Journal of Symbolic Logic* **5**, 56–68.
12. Clarke, E. and X. Zhao: 1992, 'Analytica—A Theorem Prover in Mathematica'. In: D. Kapur (ed.): *Automated Deduction—CADE-11*, Vol. 607 of *Lecture Notes in Computer Science.* pp. 761–765.
13. Computer Algebra and Automated Reasoning, 'Computer Algebra and Automated Reasoning (CAAP) Project, University of St. Andrews'. Web site at `http://www-theory.dcs.st-and.ac.uk/info/caar.html`.

14. Constable, R. L., 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: 1986, *Implementing Mathematics with the Nuprl Proof Development System*. Englewood Cliffs, New Jersey: Prentice-Hall.
15. Craigen, D., S. Kromodimoeljo, I. Meisels, B. Pase, and M. Saaltink: 1991, ‘EVES: An Overview’. Technical Report CP-91-5402-43, ORA Corporation.
16. Dunstan, M., H. Gottliebsen, T. Kelsey, and U. Martin: 2001, ‘Computer Algebra meets Automated Theorem Proving: A Maple-PVS Interface’. In: S. Linton and R. Sebastiani (eds.): *CALCULEMUS-2001*. pp. 107–119.
17. Enderton, H. B.: 1972, *A Mathematical Introduction to Logic*. Academic Press.
18. Farmer, W. M.: 1990, ‘A Partial Functions Version of Church’s Simple Theory of Types’. *Journal of Symbolic Logic* **55**, 1269–91.
19. Farmer, W. M.: 1993, ‘A Simple Type Theory with Partial Functions and Subtypes’. *Annals of Pure and Applied Logic* **64**, 211–240.
20. Farmer, W. M.: 1994, ‘Theory Interpretation in Simple Type Theory’. In: J. H. et al. (ed.): *Higher-Order Algebra, Logic, and Term Rewriting*, Vol. 816 of *Lecture Notes in Computer Science*. pp. 96–123.
21. Farmer, W. M.: 1998, ‘The Interactive Mathematics Laboratory’. In: *Proceedings of the 31st Annual Small College Computing Symposium (SCCS ’98)*. pp. 84–94.
22. Farmer, W. M.: 2000a, ‘An Infrastructure for Intertheory Reasoning’. In: D. McAllester (ed.): *Automated Deduction—CADE-17*, Vol. 1831 of *Lecture Notes in Computer Science*. pp. 115–131.
23. Farmer, W. M.: 2000b, ‘A Proposal for the Development of an Interactive Mathematics Laboratory for Mathematics Education’. In: E. Melis (ed.): *CADE-17 Workshop on Deduction Systems for Mathematics Education*. pp. 20–25.
24. Farmer, W. M.: 2001, ‘STMM: A Set Theory for Mechanized Mathematics’. *Journal of Automated Reasoning* **26**, 269–289.
25. Farmer, W. M. and J. D. Guttman: 2000, ‘A Set Theory with Support for Partial Functions’. *Studia Logica* **66**, 59–78.
26. Farmer, W. M., J. D. Guttman, M. E. Nadel, and F. J. Thayer: 1994, ‘Proof Script Pragmatics in IMPS’. In: A. Bundy (ed.): *Automated Deduction—CADE-12*, Vol. 814 of *Lecture Notes in Computer Science*. pp. 356–370.
27. Farmer, W. M., J. D. Guttman, and F. J. Thayer: 1992, ‘Little Theories’. In: D. Kapur (ed.): *Automated Deduction—CADE-11*, Vol. 607 of *Lecture Notes in Computer Science*. pp. 567–581.
28. Farmer, W. M., J. D. Guttman, and F. J. Thayer: 1993a, ‘IMPS: An Interactive Mathematical Proof System’. *Journal of Automated Reasoning* **11**, 213–248.
29. Farmer, W. M., J. D. Guttman, and F. J. Thayer: 1993b, ‘The IMPS User’s Manual’. Technical Report M-93B138, The MITRE Corporation. Available at `http://imps.mcmaster.ca/`.
30. Farmer, W. M., J. D. Guttman, and F. J. Thayer: 1995, ‘Contexts in Mathematical Reasoning and Computation’. *Journal of Symbolic Computation* **19**, 201–216.
31. Farmer, W. M., J. D. Guttman, and F. J. Thayer Fábrega: 1996, ‘IMPS: An Updated System Description’. In: M. McRobbie and J. Slaney (eds.): *Automated Deduction—CADE-13*, Vol. 1104 of *Lecture Notes in Computer Science*. pp. 298–302.
32. Farmer, W. M. and M. v. Mohrenschildt: 2000, ‘Transformers for Symbolic Computation and Formal Deduction’. In: S. Colton, U. Martin, and V.

Sorge (eds.): *CADE-17 Workshop on the Role of Automated Deduction in Mathematics*. pp. 36–45.

33. Farmer, W. M. and M. v. Mohrenschildt: 2002, 'A Detailed Description of A Formal Framework for Managing Mathematics'. Technical report, McMaster University. Available at `http://imps.mcmaster.ca/mathscheme/`.
34. Giunchiglia, F., P. Pecchiari, and C. Talcott: 2001, 'Reasoning Theories'. *Journal of Automated Reasoning* **26**, 291–331.
35. Gordon, M., R. Milner, and C. P. Wadsworth: 1979, *Edinburgh* LCF*: A Mechanised Logic of Computation*, Vol. 78 of *Lecture Notes in Computer Science*. Springer-Verlag.
36. Gordon, M. J. C. and T. F. Melham: 1993, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.
37. Harrison, J. and L. Théry: 1998, 'A Skeptic's Approach to Combining HOL and Maple'. *Journal of Automated Reasoning* **21**, 279–294.
38. Homann, K. and J. Calmet: 1995, 'Combining Theorem Proving and Symbolic Mathematical Computing'. In: J. Calmet and J. A. Campbell (eds.): *Integrating Symbolic Mathematical Computation and Artificial Intelligence*, Vol. 958 of *Lecture Notes in Computer Science*.
39. Homann, K. and J. Calmet: 1996, 'Structures for Symbolic Mathematical Reasoning and Computation'. In: J. Calmet (ed.): *DISCO'96: Design and Implementation of Symbolic Computation Systems*, Vol. 1128 of *Lecture Notes in Computer Science*. Springer, pp. 216–227.
40. Jenks, R. D. and R. S. Sutor: 1992, *Axiom : The Scientific Computation System*. Springer-Verlag.
41. Kerber, M., M. Kohlhase, and V. Sorge: 1998, 'Integrating Computer Algebra Into Proof Planning'. *Journal of Automated Reasoning* **21**, 327–355.
42. Macsyma: 1996, *Macsyma Mathematics and System Reference Manual*. Macsyma Inc.
43. McCune, W.: 1990, 'OTTER 2.0'. In: M. E. Stickel (ed.): *10th International Conference on Automated Deduction*, Vol. 449 of *Lecture Notes in Computer Science*. pp. 663–664.
44. Monk, L. G.: 1988, 'Inference Rules Using Local Contexts'. *Journal of Automated Reasoning* **4**, 445–462.
45. Nederpelt, R. P., J. H. Geuvers, and R. C. D. Vrijer (eds.): 1994, *Selected Papers on Automath*, Vol. 133 of *Studies in Logic and The Foundations of Mathematics*. North Holland.
46. Owre, S., S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas: 1996, 'PVS: Combining Specification, Proof Checking, and Model Checking'. In: R. Alur and T. A. Henzinger (eds.): *Computer Aided Verification: 8th International Conference, CAV '96*, Vol. 1102 of *Lecture Notes in Computer Science*. pp. 411–414.
47. Paulson, L. C.: 1994, *Isabelle: A Generic Theorem Prover*, Vol. 828 of *Lecture Notes in Computer Science*. Springer-Verlag.
48. Pfenning, F., 'Logical Frameworks'. Web site at `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/fp/www/lfs.html`.
49. Poll, E. and S. Thompson: 1998, 'Adding the Axioms to Axiom: Towards a System of Automated Reasoning in Aldor'. Technical Report 6-9, Computing Laboratory, University of Kent.
50. Rudnicki, P.: 1992, 'An Overview of the MIZAR Project'. Technical report, Department of Computing Science, University of Alberta.
51. Shoenfield, J. R.: 1967, *Mathematical Logic*. Addison-Wesley.

52. Theorema Project, ‘Theorema Project: Computer Supported Mathematical Theorem Proving, Research Institute for Symbolic Computation (RISC)’. Web site at `http://www.theorema.org/`.
53. Watt, S. M., P. A. Broadbery, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor: 2001, ‘Aldor Compiler User Guide’. Available at `http://www.aldor.org/docs/HTML/`.
54. Wolfram, S.: 1991, *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley.