A Formal Framework for Managing Mathematics

William M. Farmer
Martin v. Mohrenschildt

Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1, Canada

{wmfarmer,mohrens}@mcmaster.ca

17 October 2001

Abstract

Mathematics is a process of creating, exploring, and connecting mathematical models. This paper present 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 deductive and computational rules, and developing networks of biform theories linked by interpretations. The framework is not tied to a specific underlying logic; it can be used with many popular logics such as first order logic, simple type theory, and set theory. Many of the ideas and mechanisms used in the framework are inspired by the IMPS Interactive Mathematical Proof System.

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

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.

(2) *Model exploration.* The models are explored by stating and proving conjectures and by performing computations.

(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 relations $=$ and $<$ on N, and the usual binary operations $+$ 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, nonmachine-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.

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

1.1 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 pair (L, Γ) where L is a formal language 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 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 [41], Coq [2], EVES [14], HOL [31], IMPS [25], Isabelle [43], Mizar [46], Nqthm [5], Nuprl [13], Otter [38], and PVS [42]. 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 with exactly two nonlogical constants:

(1) An individual constant 0.

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

(The binary predicate constant $=$ 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 $= (L, \Gamma)$ 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 relation $<$, the operations $+$ and $*$, and the natural numbers $0, 1, \ldots$, 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 $\mathbf{PA}'$ be **PA** plus definitions for $<$, $+$, $*$, and each natural number n. $\mathbf{PA}'$ is not finitely axiomatizable (even in second-order logic), which means that $\mathbf{PA}'$ 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.

1.2 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 pair (L, Γ) where L is a formal language 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 [36], Macsyma [35], Maple [10], and Mathematica [49]. 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 terms of type boole (the type of truth values) and nat (the type of natural numbers) formed from the following primitive symbols:

(1) Constant symbols of type nat: $0, 1, \ldots$.

(2) Operator symbols:

$$\begin{array}{lcl} = & : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{boole}. \\ < & : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{boole}. \\ + & : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}. \\ * & : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}. \end{array}$$

A *numeral* is a member of $\{0, 1, \ldots\}$, and a *numeric term* is a term that does not contain $=$ or $<$.

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 term t of type boole, returns true [false] if t is a "true" ["false"] equation or inequality.

NNA $= (L, \{\mathsf{eval}, \mathsf{reduce}\})$ is an algorithmic theory of natural number arithmetic. **NNA** specifies the standard model of the natural numbers. □

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.

1.3 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 deductive and computational rules called *theoremoids*. Theoremoids are constructed from theorems and *axiomoids*, the primitive theoremoids of a biform theory, by applying *theoremoid constructors* 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 [22, 25, 26]. 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 Maple.

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 10.

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 notions of a *transformer* for representing expression manipulating algorithms and a *formuloid* for representing asserted formulas and transformers are presented in section 3. Section 4 defines the central notion of a biform theory, while section 5 defines an interpretation of one biform theory in another. Derivation, theoremoid construction, and theory development are the subjects of sections 6, 7, and 8, respectively. The paper then ends with a conclusion in section 9 and a survey of related work in section 10.

2 Logics

The background logic for FFMM is required to satisfy a small set of properties. These properties are expressed in the notion of an "admissible logic" defined in this section. The syntactic properties of an admissible logic's language are given by the definition of an "admissible language", while the semantic properties are embodied in definition of a "model" for an admissible language.

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

(1) $\mathcal{T}$ is a set of syntactic objects called the *types* of L.

(2) $\mathcal{E}$ is a set of syntactic objects called the *expressions* of L.

(3) $\tau : \mathcal{E} \to \mathcal{T}$ is a total function.

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

Let E be an expression of L. A *subexpression* of E is an expression E' of L that occurs at some position p in E, while a *subexpression occurrence* in E is a position p in E at which some expression E' of L occurs. Let E_1 be a subexpression of E that occurs at position p in E, and let E_2 be an expression of L such that $\tau(E_1) = \tau(E_2)$. 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 a language satisfies the following additional property:

(4) If E, E_1, E_2 are expressions of L such that E_1 is a subexpression of E that occurs at position p in E and $\tau(E_1) = \tau(E_2)$, then $E[p/E_2]$ is an expression E' of L such that $\tau(E') = \tau(E)$.

A language $L = (\mathcal{T}, \mathcal{E}, \tau)$ is *admissible* if the following conditions are satisfied:

(1) $* \in \mathcal{T}$. ($*$ denotes the type of truth values.)

(2) true and false are expressions of L of type $*$.

(3) If $E_1, E_2 \in \mathcal{E}$ with $\tau(E_1) = \tau(E_2) = \alpha$, then $(E_1 \simeq E_2)$ is an expression of L of type $*$.

(4) If $E, E' \in \mathcal{E}$ with $\tau(E) = \tau(E') = *$, then $\neg E$ and $(E \supset E')$ are expressions of L of type $*$.

(5) If $E_1, \ldots, E_n \in \mathcal{E}$ with $\tau(E_1) = \cdots = \tau(E_n) = *$ and $n \geq 0$, then $\wedge(E_1, \ldots, E_n)$ and $\vee(E_1, \ldots, E_n)$ are expressions of L of type $*$.

(6) If $E_1, E_2, E_3 \in \mathcal{E}$ with $\tau(E_1) = *$ and $\tau(E_2) = \tau(E_3) = \alpha$, then $\mathsf{if}(E_1, E_2, E_3)$ is an expression of type α.

true and false denote the truth values true and false. $(E_1 \simeq E_2)$ asserts the equivalence of E_1 and E_2.[1] $\neg E_1$, $(E_1 \supset E_2)$, $\wedge(E_1, \ldots, E_n)$, and $\vee(E_1, \ldots, E_n)$ assert the usual propositional combinations. $\mathsf{if}(E_1, E_2, E_3)$ denotes E_2 if E_1 is true and denotes E_3 otherwise. An admissible language may contain other kinds of types and expressions.

A *formula* of an admissible language L is an expression of L of type $*$. Parentheses in expressions may be suppressed when meaning is not lost.

Example 2.1 (Standard Language of Proposition Logic) Let $\mathcal{P}_{\rm pl}$ be an infinite set $\{P_1, P_2, \ldots\}$ of symbols called *propositional variables* or *letters*. The set $\mathcal{E}_{\rm pl}$ of expressions is defined inductively by:

(1) If $P \in \mathcal{P}_{\rm pl}$, then $P \in \mathcal{E}_{\rm pl}$.

(2) $\mathsf{true}, \mathsf{false} \in \mathcal{E}_{\rm pl}$.

(3) If $E, E' \in \mathcal{E}_{\rm pl}$, then $\neg E, (E \supset E') \in \mathcal{E}_{\rm pl}$.

The other kinds of expressions of an admissible language are introduced by the following definitions:

$\vee()$	stands for	false.
$\vee(E_1, \ldots, E_n)$	stands for	$(\neg E_1 \supset \vee(E_2, \ldots, E_n))$ where $n \geq 1$.
$\wedge(E_1, \ldots, E_n)$	stands for	$\neg\vee(\neg E_1, \ldots, \neg E_n)$ where $n \geq 0$.
$(E_1 \simeq E_2)$	stands for	$\wedge((E_1 \supset E_2), (E_2 \supset E_1))$.
$\mathsf{if}(E_1, E_2, E_3)$	stands for	$\wedge((E_1 \supset E_2), (\neg E_1 \supset E_3))$.

Let $\mathcal{T}_{\rm pl} = \{*\}$ and $\tau_{\rm pl} : \mathcal{E}_{\rm pl} \to \mathcal{T}_{\rm pl}$. Then $L_{\rm pl} = (\mathcal{T}_{\rm pl}, \mathcal{E}_{\rm pl}, \tau_{\rm pl})$ is an admissible language which we call the *standard language of propositional logic*. □

A *model* for an admissible language $L = (\mathcal{T}, \mathcal{E}, \tau)$ is a pair $M = (\mathcal{D}, V)$ such that:

(1) $\mathcal{D}$ is a set $\{\mathcal{D}_\alpha : \alpha \in \mathcal{T}\}$ of nonempty domains such that $\mathcal{D}_* = \{\mathrm{T}, \mathrm{F}\}$. ($\mathrm{T} \neq \mathrm{F}$.)

(2) V is a partial function[2] such that, if $E \in \mathcal{E}$ and $V(E)$ is defined, then $V(E) \in \mathcal{D}_{\tau(E)}$.

[1]In a standard logic, $(E_1 \simeq E_2)$ means that E_1 and E_2 denote the same value, while in a partial logic like LUTINS [16, 17, 18], 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.

[2]The *domain of definition* of a function f is the set D_f of values at which f is defined, and the *domain of application* of f is the set D^*_f of values to which f may be applied. A function f is *total* if $D_f = D^*_f$ and *partial* if $D_f \subseteq D^*_f$. Thus a total function is a special case of a partial function.

(3) $V(\mathsf{true}) = \mathrm{T}$ and $V(\mathsf{false}) = \mathrm{F}$.

(4) Let $E_1, E_2 \in \mathcal{E}$ with $\tau(E_1) = \tau(E_2) = \alpha$. If $V(E_1)$ and $V(E_2)$ are defined, then $V(E_1 \simeq E_2) = \mathrm{T}$ if $V(E_1) = V(E_2)$, and $V(E_1 \simeq E_2) = \mathrm{F}$ otherwise.

(5) Let $E, E' \in \mathcal{E}$ with $\tau(E) = \tau(E') = *$. If $V(E)$ and $V(E')$ are defined, then:

  (a) $V(\neg E) = \mathrm{T}$ if $V(E) = \mathrm{F}$, and $V(\neg E) = \mathrm{F}$ if $V(E) = \mathrm{T}$.

  (b) $V(E \supset E') = \mathrm{F}$ if $V(E) = \mathrm{T}$ and $V(E') = \mathrm{F}$, and $V(E \supset E') = \mathrm{T}$ otherwise.

(6) Let $E_1, \ldots, E_n \in \mathcal{E}$ with $\tau(E_1) = \cdots = \tau(E_n) = *$ and $n \geq 0$. If $V(E_1), \ldots, V(E_n)$ are defined, then:

  (a) $\wedge(E_1, \ldots, E_n) = \mathrm{T}$ if, for all i with $1 \leq i \leq n$, $V(E_i) = \mathrm{T}$, and $\wedge(E_1, \ldots, E_n) = \mathrm{F}$ otherwise.

  (b) $\vee(E_1, \ldots, E_n) = \mathrm{F}$ if, for all i with $1 \leq i \leq n$, $V(E_i) = \mathrm{F}$, and $\vee(E_1, \ldots, E_n) = \mathrm{T}$ otherwise.

(7) Let $E_1, E_2, E_3 \in \mathcal{E}$ with $\tau(E_1) = *$ and $\tau(E_2) = \tau(E_3) = \alpha$. If $V(E_1)$, $V(E_2)$, and $V(E_3)$ are defined, then $V(\mathsf{if}(E_1, E_2, E_3)) = V(E_2)$ if $V(E_1) = \mathrm{T}$ and $V(\mathsf{if}(E_1, E_2, E_3)) = V(E_3)$ if $V(E_1) = \mathrm{F}$.

(8) Let $E, E_1 \simeq E_2 \in \mathcal{E}$ such that E_1 is a subexpression of E that occurs at position p in E. If $V(E_1 \simeq E_2) = \mathrm{T}$, then $V(E \simeq E[p/E_2]) = \mathrm{T}$.

When $V(E)$ is defined, $V(E)$ is called the *value* of E in M. When $V(E)$ is undefined, E is said to be *undefined* or *nondenoting* in M. M is a *total* model if V is total.

Example 2.2 (Models of Propositional Logic) Let $\mathcal{D}_* = \{\mathrm{T}, \mathrm{F}\}$ and $V : \mathcal{E}_{\mathrm{pl}} \to \mathcal{D}_*$ such that:

(1) For all $P \in \mathcal{P}_{\mathrm{pl}}$, $V(P)$ is defined.

(2) $V(\mathsf{true}) = \mathrm{T}$ and $V(\mathsf{false}) = \mathrm{F}$.

(3) For all $E \in \mathcal{E}_{\mathrm{pl}}$, $V(\neg E) = \mathrm{T}$ if $V(E) = \mathrm{F}$, and $V(\neg E) = \mathrm{F}$ if $V(E) = \mathrm{T}$.

(4) For all $E, E' \in \mathcal{E}_{\mathrm{pl}}$, $V(E \supset E') = \mathrm{F}$ if $V(E) = \mathrm{T}$ and $V(E') = \mathrm{F}$, and $V(E \supset E') = \mathrm{T}$ otherwise.

$(\{D_*\}, V)$ is a total model for $L_{\rm pl}$ which we call a *standard model of propositional logic*. □

An *admissible logic* is a pair $\mathbf{K} = (L, \mathcal{M})$ such that L is an admissible language and $\mathcal{M}$ is a set of models for L. L is called the *language* of $\mathbf{K}$.[3] Many common logics can be formulated as admissible logics (including propositional logic, first-order logic, and simple type theory).

Let $\mathbf{K} = (L, \mathcal{M})$ be an admissible logic. Let $M = (\mathcal{D}, V) \in \mathcal{M}$, A be a formula of L, and Σ be a set of formulas of L. M *satisfies* A, written $M \models A$, if $V(A) = \mathrm{T}$. M *satisfies* Σ, written $M \models \Sigma$, if M satisfies each $B \in \Sigma$. A is *valid*, written $\models A$, if every model in $\mathcal{M}$ satisfies A. A is a *logical consequence* of Σ, written $\Sigma \models A$, if every model in $\mathcal{M}$ that satisfies Σ also satisfies A.

Proposition 2.3 *Let $\boldsymbol{K} = (L, \mathcal{M})$ be an admissible logic, A be a formula of L, Σ and Σ' be sets of formulas of L, and E and $E_1 \simeq E_2$ be expressions of L.*

(1) If $M \in \mathcal{M}$, then $M \models \mathsf{true}$.

(2) $\Sigma \models A$ iff $\Sigma \models A \simeq \mathsf{true}$.

(3) $\Sigma \models \mathsf{false}$ iff there is no model in $\mathcal{M}$ that satisfies Σ.

(4) If $\Sigma \subseteq \Sigma'$ and $\Sigma \models A$, then $\Sigma' \models A$.

(5) If E_1 is a subexpression of E that occurs at position p in E, then $\{E_1 \simeq E_2)\} \models E \simeq E[p/E_2]$.

Proof Follows immediately from the definition of a model for an admissible language. □

Let E and $E_1 \simeq E_2$ be expressions of L such that E_1 is a subexpression of E at position p, and let C be a set of formulas occurring in E. C is a *local context* in E at p if, for all sets Σ of formulas of L,

$$\Sigma \cup C \models E_1 \simeq E_2$$

implies

$$\Sigma \models E \simeq E[p/E_2].$$

[3] We could have defined an admissible logic to include a family of languages. Instead an admissible logic includes a single language (which will usually contain an infinite reservoir of each kind of symbol).

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 $A \supset B$. The method of local contexts [40] is a powerful idea that is applicable to both deduction and computation. See [25, 27] for examples of how local contexts are used in IMPS to facilitate deduction and computation.

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

(1) $(L, \mathcal{M})$ is an admissible logic.

(2) κ is function such that, for each expression E of L and subexpression occurrence p in E, $\kappa(E, p)$ is a local context of E at p.

Example 2.4 (Propositional Logic) Let $\mathbf{K}_{\rm pl} = (L_{\rm pl}, \mathcal{M}_{\rm pl}, \kappa_{\rm pl})$ where $\mathcal{M}_{\rm pl}$ is the set of standard models of propositional logic and $\kappa_{\rm pl}(E, p)$ is defined as follows:

(1) If p is the subexpression occurrence of E in E itself, then $\kappa_{\rm pl}(E, p) = \emptyset$.

(2) If $\neg E'$ occurs in E at position p' and E' occurs at position p in E, then $\kappa_{\rm pl}(E, p) = \kappa_{\rm pl}(E, p')$.

(3) If $(E' \supset E'')$ occurs in E at position p' and E' occurs at position p in E, then $\kappa_{\rm pl}(E, p) = \kappa_{\rm pl}(E, p')$.

(4) If $(E' \supset E'')$ occurs in E at position p' and E'' occurs at position p in E, then $\kappa_{\rm pl}(E, p) = \kappa_{\rm pl}(E, p') \cup \{E'\}$.

Notice that, if $E = \wedge(E_1, \ldots, E_n)$ and p is the subexpression occurrence of E_i in E for some i with $1 \leq i \leq n$, then $\kappa_{\rm pl}(E_i, p)$ is logically equivalent to $\{E_1, \ldots, E_{i-1}\}$.

$\mathbf{K}_{\rm pl}$ is a formulation of classical propositional logic as an admissible logic with local contexts . $\Box$

3 Transformers and Formuloids

Let $L_i = (\mathcal{T}_i, \mathcal{E}_i, \tau_i)$ 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 $L = (\mathcal{T}, \mathcal{E}, \tau)$ if Π is a transformer from L to L and, for all expressions $E \in$

$\mathsf{dom}(\Pi)$, $\tau(E) = \tau(\Pi(E))$. $\Pi_{E_1 \mapsto E_2}$ denotes a transformer that implements $\pi : \mathcal{E}_1 \rightharpoonup \mathcal{E}_2$ such that the domain of π is $\{E_1\}$ and $\pi(E_1) = E_2$.

A transformer is intended to be an expression transforming algorithm such as 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.

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 that occurs 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 whose type equals $\tau(E)$ if $\Pi(E_1)$ is defined.)

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

(1) $k \in \{0,1,2,3\}$ is the *kind* of θ.

(2) If $k = 0$, then X is a formula of L.

(3) If $k \in \{1,2,3\}$, then X is a transformer Π residing in L.

(4) If $k \in \{2,3\}$, then, for all expressions $E \in \mathsf{dom}(\Pi)$, $\tau(E) = *$.

A formuloid is *formulary* if its kind is 0 and is *transformational* if its kind is 1, 2, or 3. The purpose of a formulary formuloid is to assert that its formula is true, and the purpose of a transformational formuloid is to assert that each member of a certain set of formulas generated by its transformer is true.

Let $\theta = (k, X)$ be a formuloid where X is a formula A of L or a transformer Π residing in L. The *span* of θ, written $\mathsf{span}(\theta)$, is the set of formulas of L defined as follows:

(1) If $k = 0$, then $\mathsf{span}(\theta) = \{A\}$.

(2) If $k = 1$, then $\mathsf{span}(\theta) = \{E \simeq \Pi(E) : E \in \mathsf{dom}(\Pi)\}$.

(3) If $k = 2$, then $\mathsf{span}(\theta) = \{E \supset \Pi(E) : E \in \mathsf{dom}(\Pi)\}$.

(4) If $k = 3$, then $\mathsf{span}(\theta) = \{\Pi(E) \supset E : E \in \mathsf{dom}(\Pi)\}$.

The *operation* of θ, written $\mathsf{oper}(\theta)$, is the transformational formuloid $(1, \Pi_{A \mapsto \mathsf{true}})$ if $k = 0$ and is θ itself if $k \in \{1,2,3\}$.

A formuloid has two meanings. Its *axiomatic* meaning is its span of formulas, and its *algorithmic* meaning is its operation.

Remark 3.1 The great majority of commonly used rules of inference can be represented by transformational formuloids. The exceptions include rules like universal generalization ($\forall$-introduction) and existential instantiation ($\exists$-elimination). In [20], we show how rules of inference of this kind can be realized with conservative extensions made from "profiles" (see section 8). $\Box$

4 Biform Theories

In this section we introduce the central 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 deduction and computation.

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

(1) $\mathbf{K} = (L, \mathcal{M}, \kappa)$ is an admissible logic with local contexts.

(2) 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: $(L, \mathsf{span}(T))$ is its form as an axiomatic theory and $(L, \mathsf{oper}(T))$ is its form as an algorithmic theory.

A *model* of T is a model $M \in \mathcal{M}$ such that $M \models \mathsf{span}(T)$. T is *satisfiable* if there is a model of T. Let A be a formula of L. A is an *axiom* of T if $A \in \mathsf{span}(T)$. A is a *theorem* of T, written $T \models A$, if $\mathsf{span}(T) \models A$. 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.

Example 4.1 (Standard Theory of Propositional Logic) A popular axiomatization of classical propositional logic (formulated with the connectives $\neg$ and $\supset$) consists of three axiom schemata and the rule of inference modus ponens [39]. Following this axiomatization, we will formalize proposition logic as a biform theory. Let $T_{\rm pl} = (\mathbf{K}_{\rm pl}, \Gamma_{\rm pl})$ where

$$\Gamma_{\rm pl} = \{\theta_1, \theta_2, \theta_3, \textsf{modus-ponens}, \vee\textsf{-def}, \wedge\textsf{-def}, \simeq\textsf{-def}, \textsf{if-def}, \textsf{reduce}\}$$

is a set of (transformational) formuloids of $L_{\rm pl}$ such that:

(1) $\theta_1 = (1, \Pi))$ where, if E has the form

$$(A \supset (B \supset A)),$$

then $\Pi(E) = \textsf{true}$, and otherwise $\Pi(E)$ is undefined.

(2) $\theta_2 = (1, \Pi))$ where, if E has the form

$$((A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))),$$

then $\Pi(E) = \textsf{true}$, and otherwise $\Pi(E)$ is undefined.

(3) $\theta_3 = (1, \Pi))$ where, if E has the form

$$((\neg A \supset \neg B) \supset (B \supset A)),$$

then $\Pi(E) = \textsf{true}$, and otherwise $\Pi(E)$ is undefined.

(4) $\textsf{modus-ponens} = (1, \Pi))$ where, if E has the form

$$\wedge(A, (A \supset B)),$$

then $\Pi(E) = B$, and otherwise $\Pi(E)$ is undefined.

(5) $\vee\textsf{-def} = (1, \Pi))$ where, if $E = \vee()$, then $\Pi(E) = \textsf{false}$; if E has the form

$$\vee(A_1, \ldots, A_n)$$

where $n \geq 1$, then $\Pi(E) = (\neg A_1 \supset \vee(A_2, \ldots, A_n))$; and otherwise $\Pi(E)$ is undefined.

(6) $\wedge$-def $= (1, \Pi))$ where, if E has the form

$$\wedge(A_1, \ldots, A_n),$$

then $\Pi(E) = \neg\vee(\neg A_1, \ldots, \neg A_n)$, and otherwise $\Pi(E)$ is undefined.

(7) $\simeq$-def $= (1, \Pi))$ where, if E has the form

$$(E_1 \simeq E_2),$$

then $\Pi(E) = \wedge((E_1 \supset E_2), (E_2 \supset E_1))$, and otherwise $\Pi(E)$ is undefined.

(8) if-def $= (1, \Pi))$ where, if E has the form

$$\mathsf{if}(E_1, E_2, E_3),$$

then $\Pi(E) = \wedge((E_1 \supset E_2), (\neg E_1 \supset E_3))$, and otherwise $\Pi(E)$ is undefined.

(9) reduce $= (1, \Pi))$ where $\Pi(E)$ is some simplified conjunctive normal form of E. (For example, if E contain no members of $\mathcal{P}_{\text{pl}}$, then $\Pi(E)$ is either true or false.)

T_{pl} is a biform theory which we call the *standard biform theory of propositional logic.*

T_{pl} contains the machinery of both an axiomatic and an algorithmic formalization of classical propositional logic; its axiomoids provide the basis for both deduction and computation. □

Let $T_i = (\mathbf{K}, \Gamma_i)$ 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 $\Gamma_1 \subseteq \Gamma_2$. If $T = (\mathbf{K}, \Gamma)$ is a biform theory and Σ is a set of formulas of the language of $\mathbf{K}$, then $T[\Sigma]$ is the extension $T' = (\mathbf{K}, \Gamma \cup \Gamma')$ of T where $\Gamma' = \{(0, A) : A \in \Sigma\}$. For a single formula A, let $T[A]$ mean $T[\{A\}]$.

5 Interpretations

Let $\mathbf{K}_i = (L_i, \mathcal{M}_i, \kappa_i)$ be an admissible logic with local contexts and $T_i = (\mathbf{K}_i, \Gamma_i)$ 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 such that:

(1) $\Phi(\mathsf{true}) = \mathsf{true}$ and $\Phi(\mathsf{false}) = \mathsf{false}$.

(2) If $E_1 \simeq E_2$, $\neg A$, $A_1 \supset A_2$, $\wedge(A_1, \ldots, A_n)$, $\vee(A_1, \ldots, A_n)$, and $\mathsf{if}(A, E_1, E_2)$ are expressions of L_1 and $\Phi(E_1)$, $\Phi(E_2)$, $\Phi(A)$, $\Phi(A_1)$, $\ldots$, $\Phi(A_n)$ are defined, then

   (a) $\Phi(E_1 \simeq E_2) = \Phi(E_1) \simeq \Phi(E_2)$

   (b) $\Phi(\neg A) = \neg\Phi(A)$.

   (c) $\Phi(A_1 \supset A_2) = \Phi(A_1) \supset \Phi(A_2)$.

   (d) $\Phi(\wedge(A_1, \ldots, A_n)) = \wedge(\Phi(A_1), \ldots, \Phi(A_n))$.

   (e) $\Phi(\vee(A_1, \ldots, A_n)) = \vee(\Phi(A_1), \ldots, \Phi(A_n))$.

   (f) $\Phi(\mathsf{if}(A, E_1, E_2)) = \mathsf{if}(\Phi(A), \Phi(E_1), \Phi(E_2))$.

(Notice that, if E_1 and E_2 are expressions of L_1 of the same type and $\Phi(E_1)$ and $\Phi(E_2)$ are defined, then $\Phi(E_1)$ and $\Phi(E_2)$ are also of the same type.) 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 [15, 18, 47]).

Interpretations are a powerful mechanism for connecting biform theories with similar structure. They serve as conduits for passing information (in the form of theorems) from abstract theories to more concrete theories, or indeed to other equally abstract theories. They enable the *little theories method* [24], in which mathematical knowledge and reasoning is distributed across a network of theories, to be applied to biform theories.

In the rest of this section, let Φ be a translation from T_1 to T_2.

Proposition 5.1 (Relative Satisfiability) *If Φ is an interpretation of T_1 in T_2 and T_2 is satisfiable, then T_1 is also satisfiable.*

Proof Assume T_2 is satisfiable. Then there is some $M \in \mathcal{M}_2$ such that $M \models A$ for every theorem A of T_2. Now assume that T_1 is not satisfiable. Then $T_1 \models \mathsf{false}$ by part (3) of Proposition 2.3. Since Φ is an interpretation of T_1 in T_2, $T_2 \models \Phi(\mathsf{false})$ and hence $T_2 \models \mathsf{false}$, which contradicts the satisfiability of T_2 by part (3) of Proposition 2.3. $\Box$

Interpretations can be used to transport (both formulary and transformational) theoremoids, as well as theorems, from one biform theory to another.

Suppose Π is a transformer residing in L_1. Let $\Phi(\Pi)$ be an algorithm that implements the function that maps $\Phi(E)$ to $\Phi(\Pi(E))$ for each $E \in \mathsf{dom}(\Phi) \cap \mathsf{dom}(\Pi)$ with $\Pi(E) \in \mathsf{dom}(\Phi)$. If $\Phi(\Pi)$ is defined (i.e., the algorithm exists), $\Phi(\Pi)$ is clearly a transformer residing in L_2.

Suppose θ is a formuloid of L_1. Let $\Phi(\theta)$ be the formuloid of L_2 defined as follows:

(1) If $\theta = (0, A)$, then $\Phi(\theta) = (0, \Phi(A))$ if $\Phi(A)$ is defined and $\Phi(\theta)$ is undefined otherwise.

(2) If $\theta = (k, \Pi)$ where $k \in \{1, 2, 3\}$, then $\Phi(\theta) = (k, \Phi(\Pi))$ if $\Phi(\Pi)$ is defined and $\Phi(\theta)$ is undefined otherwise.

Proposition 5.2 *If Φ is an interpretation of T_1 in T_2, θ is a theoremoid of T_1, and $\Phi(\theta)$ is defined, then $\Phi(\theta)$ is a theoremoid of T_2.*

Proof Assume Φ is an interpretation of T_1 in T_2.

Let $\theta = (0, A)$ be a theoremoid of T_1. Then $T_1 \models A$. Assume $\Phi(A)$ is defined. To show that $\Phi(\theta) = (0, \Phi(A))$ is a theoremoid of T_2, we must show that $T_2 \models \Phi(A)$, but this follows immediately from Φ being an interpretation of T_1 in T_2.

Now let $\theta = (1, \Pi)$ be a theoremoid of T_1 (where Π is a transformer residing in L_1). Assume $\Phi(\Pi)$ is defined. To show that $\Phi(\theta) = (1, \Phi(\Pi))$ is a theoremoid of T_2, we must show $T_2 \models \Phi(E) \simeq \Phi(\Pi(E))$ for all $E \in \mathsf{dom}(\Phi) \cap \mathsf{dom}(\Pi)$ with $\Pi(E) \in \mathsf{dom}(\Phi)$. Let $E \in \mathsf{dom}(\Phi) \cap \mathsf{dom}(\Pi)$ with $\Pi(E) \in \mathsf{dom}(\Phi)$. Then $T_1 \models E \simeq \Pi(E)$ since θ is a theoremoid of T_1 of kind 1, and thus $T_2 \models \Phi(E \simeq \Pi(E))$ since Φ is an interpretation of T_1 in T_2. Therefore, $T_2 \models \Phi(E) \simeq \Phi(\Pi(E))$ since $\Phi(E \simeq \Pi(E)) = \Phi(E) \simeq \Phi(\Pi(E))$.

When $\theta = (k, \Pi)$ is a theoremoid of T_1 for $k = 2$ or 3, the argument is similar to the case when $k = 1$. $\Box$

6 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 inextricably 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 with local contexts. Our solution is the

Name	**Tuple**[4]	**Meaning**
node implication	$(1, N_1, N_2)$ where N_1, N_2 are formulary and $T_1 = T_2$.	$\{N_1\} \Rightarrow N_2$
node equivalence	$(2, N_1, N_2)$ where N_1, N_2 are formulary and $T_1 \leq T_2$.	$\{N_1\} \Leftrightarrow \{N_2\}$
one-to-many	$(3, N_0, \{N_1, \ldots, N_m\})$ where $N_0, N_1, \ldots, N_m$ are formulary and $T = T_1 = \cdots = T_m$.	$\{N_0\} \Leftrightarrow \{N_1, \ldots, N_m\}$
assumption	$(4, N_1, N_2)$ where $T_1 \leq T_2$.	$N_1 \gg N_2$ [5]
unconditional equivalence	$(5, N_1, N_2)$ where $T_1 = T_2$.	$N_1 \equiv N_2$
conditional equivalence	$(6, N_0, N_1, N_2)$ where N_0 is formulary and $T_1 = T_2$.	If $T_0 \models E_0$, then $N_1 \equiv N_2$.

Table 1: Derivation Graph Connectors

notion of a "derivation graph" defined in this section. It is a generalization of the notion of a *deduction graph* employed in IMPS.

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

Let Δ_1 and Δ_2 be finite sets of formulary nodes of $\mathbf{K}$. $\Delta_1 \Rightarrow \Delta_2$ means that, if $T_1 \models A_1$ for each $N_1 = (T_1, A_1) \in \Delta_1$, then $T_2 \models A_2$ for each $N_2 = (T_2, A_2) \in \Delta_2$. $\Delta_1 \Leftrightarrow \Delta_2$ means both $\Delta_1 \Rightarrow \Delta_2$ and $\Delta_2 \Rightarrow \Delta_1$ hold, and $\Delta_1 \Rightarrow N$ means $\Delta_1 \Rightarrow \{N\}$.

Let $N_1 = (T_1, E_1)$ and $N_2 = (T_2, E_2)$ be nodes of $\mathbf{K}$. $N_1 \gg N_2$ means $T_2 \models E_1 \simeq E_2$. $N_1 \equiv N_2$ means $N_1 \gg N_2$ and $T_1 = T_2$ hold.

There are six kinds of *(derivation graph) connectors*. They are given in Table 1. Each connector is a tuple consisting of a *kind* $k \in \{1, \ldots, 6\}$ 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

[4] $N_i = (T_i, E_i)$ for i with $0 \leq i \leq m$.

[5] If N_1, N_2 are formulary and $E_1 = E_2$, then the meaning of $(4, N_1, N_2)$ is $\{N_1\} \Rightarrow N_2$.

to an *inference node* in an IMPS deduction graph.

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 formulary nodes connected by the six kinds of connectors represent deductions, while sequences of nodes connected by assumption, unconditional equivalence, and conditional equivalence connectors represent computations.

The derivation graph $(\emptyset, \emptyset)$ is the *empty derivation graph*. Derivation graphs are built from the empty derivation graph by applying "operations" that add new nodes and connectors to a derivation graph.

There are nine primitive *(derivation graph) operations* for adding nodes and connectors to a derivation graph. They are defined in Table 2. Each operation takes a derivation graph $G = (\mathcal{N}, \mathcal{C})$ (of an admissible logic $\mathbf{K} = (L, \mathcal{M}, \kappa)$ with local contexts) 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. add-assumption creates a new node by adding a new axiom to the theory of a node in the derivation graph. split-implication, split-conjunction, and split-conditional are restructuring operations. And apply-k-thmoid, for $k \in \{0, 1, 2, 3\}$, are operations for applying theoremoids 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.

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

Remark 6.1 Composite derivation graph operations that apply primitive derivation graph operations in certain specified ways can be introduced in the style of *tactics* [30]. For example, a tactic to "weaken" a node $N = (T[A], E)$ to the node $N' = (T, E)$ in a derivation graph G could use add-node to add the node (T, E) to G and then use add-assumption to add the connector $(4, N', N)$ to G. $\Box$

Name	**Input Objects**	**Output Objects**
add-node	$T = (\mathbf{K}, \Gamma)$ is a biform theory E is an expression of L	$N = (T, E)$
add-assumption	$N = (T, E) \in \mathcal{N}$ A is a formula of L	$N' = (T[A], E)$ $C = (4, N, N')$
split-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-conditional	$N = (T, E) \in \mathcal{N}$ p is a position of $\mathsf{if}(A, E_1, E_2)$ in E	$N_0 = (T[\kappa(E, p)], A)$ $N'_0 = (T[\kappa(E, p)], \neg A)$ $N_1 = (T, E[p/E_1])$ $N_2 = (T, E[p/E_2])$ $C_1 = (6, N_0, N, N_1)$ $C_2 = (6, N'_0, N, N_2)$
apply-0-thmoid	$N = (T, E) \in \mathcal{N}$ p is a subexpression occurrence of A in E $\theta = (0, A) \in$ $\mathsf{thmoid}(T[\kappa(E, p)])$	$N' = (T, E[p/\mathsf{true}])$ $C = (5, N, N')$
apply-1-thmoid	$N = (T, E) \in \mathcal{N}$ p is a subexpression occurrence in E $\theta = (1, \Pi) \in$ $\mathsf{thmoid}(T[\kappa(E, p)])$ and $\Pi(E, p)$ is defined	$N' = (T, \Pi(E, p))$ $C = (5, N, N')$
apply-2-thmoid	$N = (T, E) \in \mathcal{N}$ $\theta = (2, \Pi) \in \mathsf{thmoid}(T)$ and $\Pi(E)$ is defined	$N' = (T, \Pi(E))$ $C = (1, N, N')$
apply-3-thmoid	$N = (T, E) \in \mathcal{N}$ $\theta = (3, \Pi) \in \mathsf{thmoid}(T)$ and $\Pi(E)$ is defined	$N' = (T, \Pi(E))$ $C = (1, N', N)$

Table 2: The Primitive Derivation Graph Operations

Let $G = (\mathcal{N}, \mathcal{C})$ be an admissible derivation graph, and let $\Delta \cup \{N\} \subseteq \mathcal{N}$ be a set of formulary nodes. (Δ, N) is an *atomic deduction* in G if one of the following conditions is satisfied:

(1) $\Delta = \{N\}$.

(2) $\Delta = \{N'\}$ and $\mathcal{C}$ contains one of the following connectors: $(1, N', N)$, $(2, N', N)$, $(2, N, N')$, $(4, N', N)$, $(5, N', N)$, or $(5, N, N')$.

(3) $\Delta = \{N'\}$ and $\mathcal{C}$ contains $(3, N', \{N_1, \ldots, N_m\})$ where $N \in \{N_1, \ldots, N_m\}$.

(4) $\Delta = \{N_1, \ldots, N_m\}$ and $\mathcal{C}$ contains $(3, N, \{N_1, \ldots, N_m\})$.

(5) $\Delta = \{N', N''\}$ and $\mathcal{C}$ contains $(6, N', N'', N)$ or $(6, N', N, N'')$.

A set $\mathcal{D}$ of atomic deductions in G is a *deduction* from Δ to N in G if one of the following conditions is satisfied:

(1) $\mathcal{D} = \{(\Delta, N)\}$.

(2) $\mathcal{D} = \mathcal{D}' \cup \mathcal{D}''$ such that $\mathcal{D}'$ is a deduction from Δ' to N, $\mathcal{D}''$ is a deduction from Δ'' to N', $N' \in \Delta'$, and $\Delta = (\Delta' \setminus N') \cup \Delta''$.

Lemma 6.2 (Soundness of Deductions) *Let G be an admissible derivation graph. If there is a deduction from Δ to N in G, then $\Delta \Rightarrow N$.*

Proof Let $G = (\mathcal{N}, \mathcal{C})$ be an admissible derivation graph and $\mathcal{D}$ be a deduction from Δ to $N = (T, A)$ in G. Our proof will be by induction on the cardinality of $\mathcal{D}$. The proof of the induction step is obvious, so we will only prove the basis. Let (Δ, N) be an atomic deduction in G. We must show that, if $T' \models A'$ for each $N' = (T', A') \in \Delta$, then $T \models A$.

Case 1: $\Delta = \{N\}$. Obvious.

Case 2a: $\Delta = \{N'\}$ and $C = (1, N', N) \in \mathcal{C}$. Let $N' = (T', A')$. Then C must have been added to $\mathcal{C}$ by an application of apply-2-thmoid to N' and a theoremoid of kind 2 or of apply-3-thmoid to N and a theoremoid of kind 3. This implies, in both cases, that $T = T'$ and $T \models A' \supset A$. Hence, if $T' \models A'$, then $T \models A$.

Case 2b: $\Delta = \{N'\}$ and $(C_1 = (2, N', N) \in \mathcal{C}$ or $C_2 = (2, N, N') \in \mathcal{C})$. Let $C_1 \in \mathcal{C}$ and $N' = (T', A')$. Then C_1 must have been added to $\mathcal{C}$ by an application of split-implication to N'. This implies that $A' = A'' \supset A$ and $T = T'[A'']$. Hence, if $T' \models A'$, then $T \models A$. The case when $C_2 \in \mathcal{C}$ is similar.

Case 2c: $\Delta = \{N'\}$ and $C = (4, N', N) \in \mathcal{C}$. Let $N' = (T', A')$. Then C must have been added to $\mathcal{C}$ by an application of $\mathsf{add\text{-}assumption}$ to N' and A''. This implies $T = T'[A'']$ and $A = A'$. Hence, if $T' \models A'$, then $T \models A$ by part (4) of Proposition 2.3.

Case 2d: $\Delta = \{N'\}$ and $(C_1 = (5, N', N) \in \mathcal{C}$ or $C_2 = (5, N, N') \in \mathcal{C})$. Let $C_1 \in \mathcal{C}$ and $N' = (T', A')$. Then C_1 must have been added to $\mathcal{C}$ by an application of $\mathsf{apply\text{-}0\text{-}thmoid}$ or $\mathsf{apply\text{-}1\text{-}thmoid}$ to N'. In the first case, $T = T'$, p is subexpression occurrence of B in A', $(0, B) \in \mathsf{thmoid}(T[\kappa(A', p)])$, and $A = A'[p/\mathsf{true}]$. Then $T \models A' \simeq A$ by part (2) of Proposition 2.3 and the fact that $\mathbf{K}$ is an admissible logic with local contexts. In the second case, $T = T'$, p is a subexpression occurrence in A', $(1, \Pi) \in \mathsf{thmoid}(T[\kappa(A', p)])$, and $A = \Pi(A', p)$. Then $T \models A' \simeq A$ by the fact that $\mathbf{K}$ is an admissible logic with local contexts. Hence, in both cases, if $T' \models A'$, then $T \models A$. The case when $C_2 \in \mathcal{C}$ is similar.

Case 3: $\Delta = \{N'\}$ and $C = (3, N', \{N_1, \ldots, N_m\}) \in \mathcal{C}$ where $N \in \{N_1, \ldots, N_m\}$. Let $N' = (T', A')$. Then C must have been added to $\mathcal{C}$ by an application of $\mathsf{split\text{-}conjunction}$ to N'. This implies that $T = T'$ and $A' = \wedge(A_1, \ldots, A_n)$ with $A \in \{A_1, \ldots, A_n\}$. Hence, if $T' \models A'$, then $T \models A$.

Case 4: $\Delta = \{N_1, \ldots, N_m\}$ and $C = (3, N, \{N_1, \ldots, N_m\}) \in \mathcal{C}$. Let $N_i = (T_i, A_i)$ for $i = 1, \ldots, m$. Then C must have been added to $\mathcal{C}$ by an application of $\mathsf{split\text{-}conjunction}$ to N. This implies that $T = T_1 = \cdots = T_m$ and $A = \wedge(N_1, \ldots, N_n)$. Hence, if $T_i \models A_i$ for all i with $1 \leq i \leq n$, then $T \models A$.

Case 5: $\Delta = \{N', N''\}$ and $(C_1 = (6, N'', N', N) \in \mathcal{C}$ or $C_2 = (6, N'', N, N') \in \mathcal{C})$. Let $C_1 \in \mathcal{C}$ and $N' = (T', A')$. Then C_1 must have been added to $\mathcal{C}$ by an application of $\mathsf{split\text{-}conditional}$ to N' and a position p of some expression $\mathsf{if}(A_0, A_1, A_2)$ in A'. In one case, $T = T'$, $N'' = (T[\kappa(A, p)], A_0)$ and $A = A'[p/A_1]$; while in another case, $T = T'$, $N'' = (T[\kappa(A, p)], \neg A_0)$ and $A = A'[p/A_2]$. Let $N'' = (T'', A'')$. Hence, in both cases, if $T'' \models A''$, then $T \models A' \simeq A$, and if also $T' \models A'$, then $T \models A$. The case when $C_2 \in \mathcal{C}$ is similar. $\Box$

A deduction from Δ to N in G is *grounded* if each member of Δ has the form (T, true). A *proof* of a node N in G is a grounded deduction (from some Δ) to N in G.

Let $N, N' \in \mathcal{N}$ where $N = (T, E)$ and $N' = (T', E')$. A *computation* from N' to N in G is a sequence $\langle N_1, \ldots, N_n \rangle$ of nodes of G with $N' = N_1$, $N = N_n$, and $n \geq 2$ such that, for all i with $1 \leq i \leq n - 1$, one of the following conditions holds:

(1) $(4, N_i, N_{i+1}) \in \mathcal{C}$.

(2) $(5, N_i, N_{i+1}) \in \mathcal{C}$ or $(5, N_{i+1}, N_i) \in \mathcal{C}$.

(3) There is a proof of a node N in G and $(6, N, N_i, N_{i+1}) \in \mathcal{C}$ or $(6, N, N_{i+1}, N_i) \in \mathcal{C}$.

Notice that, if $\langle N_1, \ldots, N_n \rangle$ is a computation in G and $N_i = (T_i, E_i)$ for all i with $1 \leq i \leq n$, then $T_1 \leq \cdots \leq T_n$.

Lemma 6.3 (Soundness of Computations) *Let G be an admissible derivation graph. If there is a computation from N' to N in G, then $N' \gg N$.*

Proof Let $G = (\mathcal{N}, \mathcal{C})$ be an admissible derivation graph and $\langle N_1, \ldots, N_n \rangle$ be a computation from $N' = (T', E')$ to $N = (T, E)$ in G. Our proof will by induction on n. The proof of the induction step is obvious, so we will only prove the basis. Let $n = 2$. We must show that $T \models E' \simeq E$.

Case 1: $C = (4, N', N) \in \mathcal{C}$. Then C must have been added to $\mathcal{C}$ by an application of $\mathsf{add\text{-}assumption}$ to N'. This implies $E = E'$. Hence $T \models E' \simeq E$.

Case 2: $C_1 = (5, N', N) \in \mathcal{C}$ or $C_2 = (5, N, N') \in \mathcal{C}$. Let $C_1 \in \mathcal{C}$. Then C_1 must have been added to $\mathcal{C}$ by an application of $\mathsf{apply\text{-}0\text{-}thmoid}$ or $\mathsf{apply\text{-}1\text{-}thmoid}$ to N'. In the first case, $T = T'$, p is subexpression occurrence of A in E', $(0, A) \in \mathsf{thmoid}(T[\kappa(E', p)])$, and $E = E'[p/\mathsf{true}]$. Then $T \models E' \simeq E$ by part (2) of Proposition 2.3 and the fact that $\mathbf{K}$ is an admissible logic with local contexts. In the second case, $T = T'$, p is a subexpression occurrence in E', $(1, \Pi) \in \mathsf{thmoid}(T[\kappa(E', p)])$, and $E = \Pi(E', p)$. Then $T \models E' \simeq E$ by the fact that $\mathbf{K}$ is an admissible logic with local contexts. Hence, in both cases, $T \models E' \simeq E$. The case when $C_2 \in \mathcal{C}$ is similar.

Case 3: There is a proof of N'' in G and $C_1 = (6, N'', N', N) \in \mathcal{C}$ or $C_2 = (6, N'', N, N') \in \mathcal{C}$. Let $C_1 \in \mathcal{C}$. Then C_1 must have been added to $\mathcal{C}$ by an application of $\mathsf{split\text{-}conditional}$ to N' and a position p of some expression $\mathsf{if}(A, E_1, E_2)$ in E'. In one case, $T = T'$, $N'' = (T[\kappa(E, p)], A)$ and $E = E'[p/E_1]$; while in another case, $T = T'$, $N'' = (T[\kappa(E, p)], \neg A)$ and $E = E'[p/E_2]$. Hence, in both cases, $T \models E' \simeq E$ since there is a proof of N'' in G. The case when $C_2 \in \mathcal{C}$ is similar. □

For each biform theory $T = (\mathbf{K}, \Gamma)$, let $\mathsf{thm}(T, G)$ be the set of formulas of L defined by the following statements:

(1) If there is a proof of $N = (T, A)$ in G, then $A \in \mathsf{thm}(T, G)$.

(2) If there is a computation from $N' = (T', E')$ to $N = (T, E)$ in G, then $E' \simeq E \in \mathsf{thm}(T, G)$.

Theorem 6.4 (Soundness of Derivation Graphs) *Let T be a biform theory and G be an admissible derivation graph. Then, for all formulas $A \in \mathsf{thm}(T,G)$, $T \models A$, i.e., each member of $\mathsf{thm}(T,G)$ is a theorem of T.*

Proof Let $A \in \mathsf{thm}(T,G)$.

Case 1: There is a proof of $N = (T,A)$ in G. By definition, there is a deduction from some Δ to N in G such that each member of Δ has the form (T', true). By Lemma 6.2, $\Delta \Rightarrow N$, and by part (1) of Proposition 2.3, $T' \models \mathsf{true}$ for all $N' = (T', \mathsf{true}) \in \Delta$. Therefore, $T \models A$.

Case 2: There is a computation from $N' = (T', E')$ to $N = (T,E)$ in G. By Lemma 6.3, $N' \gg N$. Therefore, $T \models E' \simeq E$. □

The Soundness of Derivation Graphs theorem shows that derivation graphs are a means to derive theorems.

7 Theoremoid Construction

In FFMM, the essence of derivation—and deduction and computation as special cases—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, a formulary theoremoid $(0, A)$ of T is obtained by constructing a derivation graph G such that $A \in \mathsf{thm}(T,G)$. 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 $k \in \{1,2,3\}$, (k, Π) is a theoremoid of T.

FFMM does not include a system for proving that a given transformational formuloid is a theoremoid of a particular theory. Instead, it includes a collection of *theoremoid constructors* inspired by the *macete constructors* of IMPS. From theorems and transformational theoremoids, theoremoid constructors construct transformational formuloids for which theoremoidhood is guaranteed by the construction itself. Several examples of theoremoid constructors are presented in this section. Together with axiomoids and theorems, they form a powerful programming language for building sound deductive and computational tools in the form of transformational theoremoids.

In this section let $\mathbf{K} = (L, \mathcal{M}, \kappa)$ be an admissible logic with local contexts.

7.1 Theorem-to-Theoremoid Constructors

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

Suppose A is a theorem of a biform theory $T = (\mathbf{K}, \Gamma)$.

For the first example, define $\mathsf{thm\text{-}to\text{-}thmoid\text{-}1}(A)$ to be $(1, \Pi^1_A)$ where $\Pi^1_A = \Pi_{A \mapsto \mathsf{true}}$. (Recall that $\Pi_{E_1 \mapsto E_2}$ is a transformer Π such that $\mathsf{dom}(\Pi) = \{E_1\}$ and $\Pi(E_1) = E_2$.) $\mathsf{thm\text{-}to\text{-}thmoid\text{-}1}(A)$ is a transformational formuloid of L that uses A as a theorem. (Notice that $\mathsf{thm\text{-}to\text{-}thmoid\text{-}1}(A)$ has exactly the same utility as the formulary theoremoid $(0, A)$.)

For the second, let A have the form $E_1 \simeq E_2$ and define $\mathsf{thm\text{-}to\text{-}thmoid\text{-}2}(A)$ to be $(1, \Pi^2_A)$ where $\Pi^2_A = \Pi_{E_1 \mapsto E_2}$. $\mathsf{thm\text{-}to\text{-}thmoid\text{-}2}(A)$ is a transformational formuloid of L that uses A as a rewrite rule.

For the third, let A have the form $A_1 \supset A_2$ and define $\mathsf{thm\text{-}to\text{-}thmoid\text{-}3}(A)$ to be $(2, \Pi^3_A)$ where $\Pi^3_A = \Pi_{A_1 \mapsto A_2}$. $\mathsf{thm\text{-}to\text{-}thmoid\text{-}3}(A)$ is a transformational formuloid of L that uses A as a forwardchaining rule of inference (a backchaining rule of inference $(3, \Pi^{3'}_A)$ could be defined in a similar way).

It is important to emphasize that each of these transformational formuloids can be generated automatically from a theorem and that the theoremoidhood of each formuloid is guaranteed by its construction (as stated in the next proposition).

Proposition 7.1 *Let* $A_1, A_2, A_3 \in \mathsf{thm}(T)$ *with* $A_2 = E_1 \simeq E_2$ *and* $A_3 = B_1 \supset B_2$*. Then* $\mathsf{thm\text{-}to\text{-}thmoid\text{-}}m(A_m) \in \mathsf{thmoid}(T)$ *for all* $m \in \{1, 2, 3\}$*.*

If the background logic $\mathbf{K}$ allows quantification in its language, much more powerful transformational theoremoids can be generated from theorems. For example, suppose the theorem A is a formula

$$\forall\, x^1_{\alpha_1}, \ldots, x^n_{\alpha_n} \,.\, A' \supset E_1 \simeq E_2$$

where each $x^i_{\alpha_i}$ is a quantified variable of type α_i that may occur in A', E_1, and E_2. Define $\mathsf{thm\text{-}to\text{-}thmoid\text{-}4}(A)$ to be $(1, \Pi^4_A)$ where Π^4_A is a transformer residing in L defined as follows: If E is alpha-equivalent to $E_1\sigma$, where σ is a substitution with domain $\{x^1_{\alpha_1}, \ldots, x^n_{\alpha_n}\}$, then $\Pi(E) = \mathsf{if}(A'\sigma, E_2\sigma, E)$; otherwise $\Pi(E)$ is undefined. $\mathsf{thm\text{-}to\text{-}thmoid\text{-}4}(A)$ is a transformational theoremoid of T that applies A as a conditional rewrite rule (the "reverse" conditional rewrite rule can also be generated from A). When $\mathsf{thm\text{-}to\text{-}thmoid\text{-}4}(A)$ is applied in a derivation graph, it introduces a conditional expression that can be resolved by simplification or split with the split-conditional operation. Other examples of theorem-to-theoremoid constructors using quantification are given in [28].

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.

7.2 The Composition Constructor

The composition constructor combines two given transformational theoremoids by composing their transformers.

Let $k \in \{1, 2, 3\}$, $T = (\mathbf{K}, \Gamma)$ be a biform theory, and $\theta_i = (k, \Pi_i) \in \mathsf{thmoid}(T)$ for $i = 1, 2$. Define $\mathsf{composition}(\theta_1, \theta_2)$ to be (k, Π) where $\Pi = \Pi_1 \circ \Pi_2$. It is easy to see that $\mathsf{composition}(\theta_1, \theta_2) \in \mathsf{thmoid}(T)$.

7.3 The Fixpoint Constructor

The fixpoint constructor 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 $k \in \{1, 2, 3\}$, $T = (\mathbf{K}, \Gamma)$ be a biform theory, and $\theta = (k, \Pi) \in \mathsf{thmoid}(T)$. 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 $n \geq 1$, $\Pi^1(E) \neq \Pi^2(E) \neq \cdots \neq \Pi^n(E)$ and $E' = \Pi^n(E) = \Pi^{n+1}(E)$. $\Pi'(E)$ is undefined otherwise. It is easy to see that $\mathsf{fixpoint}(\theta) \in \mathsf{thmoid}(T)$.

7.4 The Conditional Constructor

The conditional constructor combines two given transformational theoremoids with a given formula used as a conditional.

Let $k \in \{1, 2, 3\}$, $T = (\mathbf{K}, \Gamma)$ be a biform theory, $\theta_i = (k, \Pi_i) \in \mathsf{thmoid}(T)$ for $i = 1, 2$, and A be a formula of L. Define $\mathsf{conditional}(A, \theta_1, \theta_2)$ to be (k, Π) where Π is defined as follows. Let E be an expression of L. $\Pi'(E) = \mathsf{if}(A, \Pi_1(E), \Pi_2(E))$ if $\Pi_1(E)$ and $\Pi_2(E)$ are defined. $\Pi'(E)$ is undefined otherwise. It is easy to see that $\mathsf{conditional}(A, \theta_1, \theta_2) \in \mathsf{thmoid}(T)$.

7.5 The Broadcast Constructor

The $\mathsf{broadcast}$ constructor builds a transformational theoremoid whose transformer has the effect of applying the transformer of a given transformational theoremoid to all the subexpressions of an expression.

Let $T = (\mathbf{K}, \Gamma)$ be a biform theory and $\theta = (1, \Pi) \in \mathsf{thmoid}(T)$. Let E be an expression of L. A *target* of Π in E is an occurrence p of a subexpression E' of E such that $\Pi(E')$ is defined. p is *maximal* if there no other target p' of Π in E such that the expression occurring at p in E is a proper subexpression of the expression occurring at p' in E. Notice that the expressions occurring at two maximal targets of Π in E are disjoint from each other.

Define $\mathsf{broadcast}(\theta)$ to be $(1, \Pi')$ where Π' is defined as follows. Let E be an expression of L and $\{p_1, \ldots, p_m\}$ be the set of maximal targets of Π in E. For i with $1 \leq i \leq m$, define $\Pi^i(E)$ to be $\Pi(E, p_1)$ if $i = 1$ and to be $\Pi(\Pi^{i-1}(E), p_i)$ otherwise. If $m \neq 0$, $\Pi'(E) = \Pi^m(E)$, and if $m = 0$, $\Pi'(E)$ is undefined. It is easy to see that $\mathsf{broadcast}(\theta) \in \mathsf{thmoid}(T)$.

7.6 The Make-Sound Constructor

The $\mathsf{make\text{-}sound}$ constructor builds a transformational theoremoid whose transformer has the effect of applying a possibly unsound transformer to an expression only when the application is actually sound.

Let $T = (\mathbf{K}, \Gamma)$ be a biform theory, Π be a transformer residing in L, and $\theta_i = (1, \Pi_i) \in \mathsf{thmoid}(T)$ for $i = 1, 2$. Define $\mathsf{make\text{-}sound}(\Pi, \theta_1, \theta_2)$ to be $(1, \Pi')$ where Π' is defined as follows. Let E be an expression of L. $\Pi'(E) = \Pi(E)$ if $\Pi_1(E)$ and $\Pi_2(\Pi(E))$ are defined and $\Pi_1(E) = \Pi_2(\Pi(E))$, and $\Pi'(E)$ is undefined otherwise.

θ_1 and θ_2 are used to check the correctness of an application of Π. Since θ_1 and θ_2 are theoremoids of kind 1, $\Pi_1(E) = \Pi_2(\Pi(E))$ implies $T \models E \simeq \Pi(E)$. Thus $\mathsf{make\text{-}sound}(\Pi, \theta_1, \theta_2) \in \mathsf{thmoid}(T)$ even when $(1, \Pi)$ is not a theoremoid of T.

7.7 The Transport Constructor

The $\mathsf{transport}$ constructor "transports" a formulary or transformational theoremoid from one biform theory to another via an interpretation.

Let $\mathbf{K}_i = (L_i, \mathcal{M}_i, \kappa_i)$ be an admissible logic with local contexts and $T_i = (\mathbf{K}_i, \Gamma_i)$ be a biform theory for $i = 1, 2$; Φ be an interpretation of T_1 in T_2; $\theta = (k, \Pi)$ be a theoremoid of T_1 with $k \in \{0, 1, 2, 3\}$; and $\Phi(\theta)$ be defined. Then define $\mathsf{transport}(\theta, \Phi)$ to be the formuloid $\Phi(\theta)$. By Proposition 5.2, $\Phi(\theta) \in \mathsf{thmoid}(T_2)$.

8 Theory Development

In FFMM, derivation is performed on top of a network of biform theories connected by 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 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 [20] 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 $\mathbf{K} = (L, \mathcal{M}, \kappa)$ be an admissible logic with local contexts. A *constant* of a language L is an expression of L which contains no subexpressions. Let $T = (\mathbf{K}, \Gamma)$ be a biform theory. A constant c of L is *mentioned* in a biform theory $T = (\mathbf{K}, \Gamma)$ if c occurs in some member of Γ. An expression E of L is *proper* in T if every constant c occurring in E is mentioned in T. Let $T_i = (\mathbf{K}, \Gamma_i)$ 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 that are proper in T_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* $T_0 = (\mathbf{K}, \Gamma_0)$, a *current (biform) theory* $T = (\mathbf{K}, \Gamma)$ 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 interpretations, theorems, theoremoids, and definitions, and profiles (see below).

A *definition* D consists of a constant c and a defining expression E. D is installed in a biform theory $T = (\mathbf{K}, \Gamma)$ by adding $c \simeq E$ to Γ. (c is intended to be unmentioned in T.) The result is a new theory $T[D]$ with a constant that has the value expressed by E. The installation of D in T is only allowed if $T \trianglelefteq T[D]$. Hence, a definition is a means to introduce new machinery into T without compromising its original machinery. Moreover, the defined constant c can be "eliminated" from expressions of L using $c \simeq E$ as a rewrite rule.

A *profile* P consists of a finite set $\mathcal{C}$ of constants and a profiling formula A in which the members of $\mathcal{C}$ occur. P is installed in a biform theory $T = (\mathbf{K}, \Gamma)$ by adding A to Γ. (The members of $\mathcal{C}$ are intended to be unmentioned in T.) The result is a new theory $T[P]$ in which the members of $\mathcal{C}$ satisfy the property expressed by A. A profile is thus a generalization of a definition. The installation of P in T is only allowed if $T \trianglelefteq T[P]$. 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 constants in $\mathcal{C}$ 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, a profile with the profiling formula $(\sqrt{2})^2 = 2$ could be used to introduce a constant $\sqrt{2}$ in R whose sign 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 to $\mathbf{T}$ 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 interpretations. When the current theory T of a biform theory object $\mathbf{T}$ is replaced by an extension $T[X]$ of T, the stored interpretations of T would not normally be defined on the defined or profiled constants of X. Three basic solutions to this problem are discussed in [20]. The first two solutions extend the old interpretations of T automatically to new interpretations of $T[X]$, while the third solution is to provide the user with a primitive operation for extending interpretations. With a mechanism for extending interpretations, it is often advantageous to create an interpretation of the base theory of 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 example, consider the operation for installing theorems of a deduction graph defined as follows. Let G be a derivation graph and $\mathbf{T}$ be a biform theory object whose current theory is T. Given G and $\mathbf{T}$, the operation creates a theorem object O_A that stores A and then installs O_A in $\mathbf{T}$ for each $A \in \mathsf{thm}(T, G)$. Other operations could be defined for transporting theorems, 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.

9 Conclusion

In this paper we have proposed a formal framework for managing the mathematics process and the mathematics knowledge produced by the process called FFMM. 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 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 serve as raw material for constructing new theoremoids. The theoremoids are tools for reasoning and computation built using the theoremoid construction facility.

The mathematics process in FFMM is thus divided into a triad of symbiotic processes: Biform theories are created and incrementally enriched with new language and derivation tools. Theorems are derived by applying the derivation tools of biform theories. And new derivation tools are constructed from the theorems and the derivation tools 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, Maple, and other mechanized mathematics systems. Moreover, we view the success of the IMPS implementation as a proof of concept for our framework proposal.

An implementation of FFMM would be a kernel for an *interactive mathematics laboratory* (IML) [19, 21] 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.

10 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 Frank Pfenning's Web guide to logical frameworks [44]). 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 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, 12, 48]).

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 (BBDs), linear arithmetic [6], and generic algebraic simplification in IMPS.

(2) *Mechanisms for applying theorems and rules of inference*: LCF-style tactics [30], IMPS macetes, and IMPS proof scripts [23].

Second, deductive capabilities are added to a computer algebra system. Examples include:

(1) The incorporation of logic into the computer algebra system Axiom [45].

(2) Analytica [11], a computer theorem proving system for mathematical analysis implemented in Mathematica.

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 [32, 33].

(2) Frameworks and techniques for integrating computer theorem proving and computer algebra systems [1, 3, 4, 29, 34, 37].

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.

(2) The framework FFMM proposed in this paper.

References

[1] A. Armando and D. Zini. The TH∃OREM∀ project: A progress report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 33–48. A. K. Peters, 2001.

[2] B. Barros et al. The Coq proof assistant reference manual, version 6.1. Available at `ftp://ftp.inria.fr/INRIA/coq/V6.1/doc/Reference-Manual.dvi.gz`, 1997.

[3] C. Benzmüller, M. Jamnik, M. Kerber, and V. Sorge. An agent-oriented approach to reasoning. In S. Linton and R. Sebastiani, editors, *CALCULEMUS-2001*, pages 48–63, 2001.

[4] P. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. *Fundamenta Informaticae*, 39, 1999.

[5] R. Boyer and J Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[6] R. S. Boyer and J S. 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.

[7] B. Buchberger. Symbolic computation: Computer algebra and logic. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems*, Applied Logic Series, pages 193–220. Kluwer Academic Publishers, 1996.

[8] B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Văsaru, and W. Windsteiger. The TH∃OREM∀ project: A progress report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 98–113. A. K. Peters, 2001.

[9] Calculemus Project: Systems for Integrated Computation and Deduction. Web site at `http://www.mathweb.org/calculemus/`.

[10] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual*. Springer-Verlag, 1991.

[11] 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.

[12] 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`.

[13] 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.

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

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

[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. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[18] W. M. Farmer. Theory interpretation in simple type theory. In J. Heering et al., editor, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.

[19] W. M. Farmer. The interactive mathematics laboratory. In *Proceedings of the 31st Annual Small College Computing Symposium (SCCS '98)*, pages 84–94, April 1998.

[20] W. M. Farmer. An infrastructure for intertheory reasoning. In D. McAllester, editor, *Automated Deduction—CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 115–131. Springer-Verlag, 2000.

[21] W. M. Farmer. A proposal for the development of an interactive mathematics laboratory for mathematics education. In E. Melis, editor, *CADE-17 Workshop on Deduction Systems for Mathematics Education*, pages 20–25, 2000.

[22] W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega. IMPS: An updated system description. In M. McRobbie and J. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, 1996.

[23] W. M. Farmer, J. D. Guttman, M. E. Nadel, and F. J. Thayer. Proof script pragmatics in IMPS. In A. Bundy, editor, *Automated Deduction—CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 356–370. Springer-Verlag, 1994.

[24] 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.

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

[26] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 1993. Available at `http://imps.mcmaster.ca/doc/`.

[27] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Contexts in mathematical reasoning and computation. *Journal of Symbolic Computation*, 19:201–216, 1995.

[28] W. M. Farmer and M. v. Mohrenschildt. Transformers for symbolic computation and formal deduction. In S. Colton, U. Martin, and V. Sorge, editors, *CADE-17 Workshop on the Role of Automated Deduction in Mathematics*, pages 36–45, 2000.

[29] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning theories. *Journal of Automated Reasoning*, 26:291–331, 2001.

[30] 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.

[31] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[32] J. Harrison and L. Théry. A skeptic's approach to combining HOL and maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

[33] K. Homann and J. Calmet. Combining theorem proving and symbolic mathematical computing. In J. Calmet and J. A. Campbell, editors, *Integrating Symbolic Mathematical Computation and Artificial Intelligence*, volume 958 of *Lecture Notes in Computer Science*, 1995.

[34] K. Homann and J. Calmet. Structures for symbolic mathematical reasoning and computation. In J. Calmet, editor, *DISCO'96: Design and Implementation of Symbolic Computation Systems*, volume 1128 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 1996.

[35] Macsyma Inc. *Macsyma Mathematics and System Reference Manual.* Macsyma Inc., 1996.

[36] R. D. Jenks and R. S. Sutor. *Axiom : The Scientific Computation System.* Springer-Verlag, 1992.

[37] M. Kerber, M. Kohlhase, and V. Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, 21:327–355, 1998.

[38] W. McCune. OTTER 2.0. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 663–664. Springer-Verlag, 1990.

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

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

[41] R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics.* North Holland, 1994.

[42] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[43] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[44] F. Pfenning. Logical Frameworks. Web site at `http://www.cs.cmu.edu/afs/cs.cmu.edu/user/fp/www/lfs.html`.

[45] E. Poll and S. Thompson. Adding the axioms to Axiom: Towards a system of automated reasoning in aldor. Technical Report 6-9, Computing Laboratory, University of Kent, 1998.

[46] P. Rudnicki. An overview of the MIZAR project. Technical report, Department of Computing Science, University of Alberta, 1992.

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

[48] Theorema Project: Computer Supported Mathematical Theorem Proving, Research Institute for Symbolic Computation (RISC). Web site at `http://www.theorema.org/`.

[49] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1991.