Biform Theories in Chiron*

William M. Farmer

McMaster University
Hamilton, Ontario, Canada
wmfarmer@mcmaster.ca

23 May 2009

Abstract. An *axiomatic theory* represents mathematical knowledge declaratively as a set of *axioms*. An *algorithmic theory* represents mathematical knowledge procedurally as a set of *algorithms*. A *biform theory* is simultaneously an axiomatic theory and an algorithmic theory. It represents mathematical knowledge both declaratively and procedurally. Since the algorithms of algorithmic theories manipulate the syntax of expressions, biform theories—as well as algorithmic theories—are difficult to formalize in a traditional logic without the means to reason about syntax. *Chiron* is a derivative of von-Neumann-Bernays-Gödel (NBG) set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. It includes elements of type theory, a scheme for handling undefinedness, and a facility for reasoning about the syntax of expressions. It is an exceptionally well-suited logic for formalizing biform theories. This paper defines the notion of a biform theory, gives an overview of Chiron, and illustrates how biform theories can be formalized in Chiron.

1 Introduction

The mission of *mechanized mathematics* is to develop software systems that support the process people use to create, explore, and apply mathematics. There are historically two major approaches to mechanized mathematics, *computer theorem proving* and *computer algebra*. Computer theorem proving emphasizes the conjecture proving aspect of the mathematics process and usually represents mathematical knowledge as "axiomatic theories". On the other hand, computer algebra focuses on the computational aspect of the mathematics process and usually represents mathematical knowledge as "algorithmic theories".

An *axiomatic theory* is a set of formulas in a language L called *axioms* that serve as the background assumptions of the theory. The axioms encode a set of mathematical truths, namely, the formulas of L that are the logical consequences of the axioms. There is thus a clear demarcation between what is assumed (the axioms) and what is derived (the logical consequences of the axioms). The deduction and computation rules for reasoning within the theory are usually expressed

 Published in M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, eds., *Towards Mechanized Mathematical Assistants, LNCS*, 4573:66–79, 2007.

in the metalanguage of L, not in L itself. This is because deduction and computation rules cannot directly manipulate values such as numbers, functions, and sets; they can only manipulate the expressions that denote these values. Traditional logics do not usually provide a facility for formalizing the syntax of expressions. As a result, neither the specifications of deduction and computation rules nor the algorithms that implement them can be directly expressed in an axiomatic theory.

An *algorithmic theory* is a set of algorithms that manipulate expressions in a language L. The background assumptions of the theory and the specifications of the algorithms are usually not part of an algorithmic theory; they are instead part of the informal metatheory of the theory. An algorithmic theory can be used to manipulate expressions, but it cannot be used to understand what the results of the manipulations mean. Also, unlike an axiomatic theory, there is no clear demarcation between the algorithms that are primitive in the theory and those that are derived from the primitive algorithms.

A *biform theory* T is a set Ω of formulas and rules in a language L. A *rule* in L consists of an algorithm called a *transformer* that transforms a tuple of input expressions of L into an output expression of L and a *meaning formula* that specifies how the values of the input expressions are related to the value of the output expression. For each tuple I of input expressions, the meaning formula M reduces to a formula M_I that specifies the relationship between the values of the members of I and the value of the resulting output expression. M_I is said to be an *instance* of the rule.

The notion of a biform theory merges the notions of an axiomatic theory and an algorithmic theory. In fact, a biform theory is simultaneously both an axiomatic theory and an algorithmic theory. The axiomatic theory of T, written T_{axm}, is the set of formulas in Ω together with the set of the instances of all the rules in Ω, while the algorithmic theory of T, written T_{alg}, is the set of the transformers of all the rules in Ω.

The formulas and rules in Ω are called the *axioms* of T. They are implicit background assumptions of T, and the axioms of T_{axm} are the explicit background assumptions of T. A rule is a *logical consequence* of T if its instances are logical consequences of T_{axm}. Thus in a biform theory there is a clear demarcation between primitive formulas and rules whose correctness is assumed and derived formulas and rules whose correctness is a logical consequence of the primitive formulas and rules.

In summary, a biform theory includes both formulas and rules as primitive assumptions. A rule consists of an algorithm that manipulates expressions and a formula that specifies what the manipulations of the expressions mean semantically. A biform theory is simultaneously both an axiomatic theory and an algorithmic theory. The meaning of an algorithm of the algorithmic theory is understood in the context of the axiomatic theory. And there is a clear definition of what a derived formula or rule is in a biform theory.

The notion of a biform theory was first introduced as part of FFMM, a Formal Framework for Managing Mathematics [11] developed as part of the MathScheme

project [15] at McMaster University. One of the principal goals of FFMM is to integrate and generalize computer theorem proving and computer algebra. Biform theories play a central role in FFMM by providing a formal context in which deduction and computation can be merged. In general, biform theories are useful for formalizing mathematics in which deduction and computation are intimately related. For applications of biform theories outside of FFMM, see [4–6].

A mechanized mathematics system that utilizes biform theories to represent mathematics needs a logic in which biform theories can be expressed. At the very least, it must be possible to express in the logic the meaning formulas of rules. Otherwise, there is no formal basis for understanding what a transformer of a rule means. This is problematic because a meaning formula expresses statements both about the syntax of expressions and what the expressions mean. Traditional logics are usually not equipped with the means to express statements about syntax and to reason about syntax.

The transformer of a rule does not need to be expressed in the logic. As long as its corresponding meaning formula is expressed in the logic, it can treated as a black-box algorithm that is assumed to behave according to its meaning formula. In other words, the transformer's rule would be considered as an axiom of the biform theory. Hence an algorithm in the form of a program in a high-level programming language can be made into a perfectly legitimate rule if a meaning formula for it can be expressed in the logic.

Chiron [8, 9] is a derivative of von-Neumann-Bernays-Gödel (NBG) set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. It includes elements of type theory, a scheme for handling undefinedness, and a facility for reasoning about the syntax of expressions. Chiron has a high level of both theoretical and practical expressivity [8]. It is an exceptionally well-suited logic for formalizing biform theories. In particular, the meaning formulas of rules can be directly expressed in Chiron.

This paper defines the notion of a biform theory, gives an overview of Chiron, and illustrates how biform theories can be formalized in Chiron. Section 2 defines the notions of a transformer, a rule, and a biform theory. Section 3 gives a quick introduction to Chiron and shows how rules are expressed in Chiron. Section 4 sketches the development in Chiron of a nontrivial example of a biform theory. The paper ends with a conclusion in Section 5 that discusses related and future work.

2 Biform Theories

We present here a formulation of a biform theory that is simpler than the formulation given in [11].

2.1 General Logics

A *general language* is a pair $L = (\mathcal{E}, \mathcal{F})$ where $\mathcal{E}$ is a set of syntactic entities called the *expressions* of L and $\mathcal{F} \subseteq \mathcal{E}$ is a set of expressions called the *formulas*

of L. For example, if F is a first-order language, then $L_F = (\mathcal{T} \cup \mathcal{F}, \mathcal{F})$ is a general language where $\mathcal{T}$ and $\mathcal{F}$ are the sets of terms and formulas of F, respectively. In the rest of this paper, let $L = (\mathcal{E}, \mathcal{F})$ be a general language.

A *general logic* is a set of general languages with a notion of logical consequence. In the rest of this paper, let $\mathbf{K}$ be a general logic. L is a *language* of $\mathbf{K}$ if it is one of the general languages of $\mathbf{K}$. If L is a language of $\mathbf{K}$ and $\Sigma \cup \{A\}$ is a set of formulas of L, then $\Sigma \models_{\mathbf{K}} A$ means A is a logical consequence of Σ in $\mathbf{K}$. For example, let **FOL** be a general logic representation of first-order logic such that L is a language of **FOL** iff $L = L_F$ for some first order language F and $\Sigma \models_{\mathbf{FOL}} A$ means A is a logical consequence of Σ in first-order logic.

An *axiomatic theory* in $\mathbf{K}$ is a pair $T = (L, \Gamma)$ where $L = (\mathcal{E}, \mathcal{F})$ is a language of $\mathbf{K}$ and $\Gamma \subseteq \mathcal{F}$. L is the *language* of T, and Γ is the set of *axioms* of T. A formula A of L is a *logical consequence* of T if $\Gamma \models_{\mathbf{K}} A$.

2.2 Transformers

For $n \geq 0$, an n-ary *transformer in* L is a pair $\Pi = (\pi, \hat{\pi})$ where π is a symbol and $\hat{\pi}$ is an algorithm that implements a (possibly partial) function $f_{\hat{\pi}} : \mathcal{E}^n \to \mathcal{E}$. The symbol π serves as a name for the algorithm $\hat{\pi}$. There is no restriction on how the algorithm is presented. For example, it could be a lambda-expression of L or a program written in a high-level programming language like C or Java.

Let $\mathsf{dom}(\Pi)$ denote the domain of $\hat{\pi}$, i.e., the subset of $\mathcal{E}^n$ on which $f_{\hat{\pi}}$ is defined. Suppose $E_1, \ldots, E_n$ are expressions in $\mathcal{E}$. If $(E_1, \ldots, E_n) \in \mathsf{dom}(\Pi)$, the expression $\pi(E_1, \ldots, E_n)$ denotes the output of $\hat{\pi}$ when given $E_1, \ldots, E_n$ as input, i.e., it denotes $f_{\hat{\pi}}(E_1, \ldots, E_n) \in \mathcal{E}$ (and is thus *defined*). If $(E_1, \ldots, E_n) \notin \mathsf{dom}(\Pi)$, $\pi(E_1, \ldots, E_n)$ does not denote anything (and is thus *undefined*). The expression $\pi(E_1, \ldots, E_n)$ is not required to be in $\mathcal{E}$; it will usually be an expression of the metalanguage of L but not of L itself.

Example 1. Suppose $L_F = \{\mathcal{E}_F, \mathcal{F}_F\}$ is the general language corresponding to a first-order language F. Let $\Pi = (\pi, \hat{\pi})$ be a unary transformer in L_F such that:

1. $\pi(E)$ is defined iff $E \in \mathcal{F}_F$.
2. If $\pi(A)$ is defined, it denotes a formula $B \in \mathcal{F}_F$ that is in prenex normal form and is logically equivalent to A.

That is, the algorithm $\hat{\pi}$ transforms any formula of L_F into a logically equivalent formula in prenex normal form. The expression $\pi(E)$ cannot be an expression in $\mathcal{E}_F$ (without some mechanism, such as Gödel numbering, for formalizing the syntax of L_F in L_F itself). □

Example 2. Suppose $L_F = \{\mathcal{E}_F, \mathcal{F}_F\}$ is again the general language corresponding to a first-order language F. Let $\Pi = (\pi, \hat{\pi})$ be a ternary transformer in L_F such that:

1. $\pi(E_1, E_2, E_3)$ is defined iff E_1 is a term of F, E_2 is a variable of F, and E_3 is a formula of F.

2. If $\pi(t, x, A)$ is defined, it denotes the result of simultaneously substituting t for each free occurrence of x in A.

That is, given t, x, A, the algorithm $\hat{\pi}$ transforms the formula A into the formula $A[x \mapsto t]$. Again the expression $\pi(E_1, E_2, E_3)$ cannot be an expression in $\mathcal{E}_F$. □

Example 3. Let **STT** be a general logic representation of simple type theory [7]. Suppose $T = (L, \Gamma)$ is an axiomatic theory of a complete ordered field in **STT** and that we have defined in T a type real of real numbers and the basic concepts of calculus such as limits, continuity, derivatives, etc. Let $\Pi = (\pi, \hat{\pi})$ be a unary transformer in L such that:

1. $\pi(E)$ is defined iff E is an expression of L of type $\mathsf{real} \to \mathsf{real}$.
2. If $\pi(E)$ is defined, it is an expression of L of type $\mathsf{real} \to \mathsf{real}$ that denotes the derivative of the function denoted by E.

That is, $\hat{\pi}$ is an algorithm that differentiates expressions that denote functions on the real numbers. □

An *algorithmic theory* is a pair $T = (L, \Delta)$ where L is a general language and Δ is a set of transformers in L. L is called the *language* of T, and Δ is the set of *algorithms* of T. For more on transformers, see [10, 11].

2.3 Rules

A *rule* in L is a pair $R = (\Pi, M)$ where:

1. $\Pi = (\pi, \hat{\pi})$ is an n-ary transformer in L.
2. M is a formula that uses π to relate the values of the inputs to $\hat{\pi}$ to the value of the output of $\hat{\pi}$.

The *transformer* of R, written $\mathsf{trans}(R)$, is Π, and the *meaning formula* of R, written $\mathsf{mean}(R)$, is M. The meaning formula M, which specifies the semantic relationship between the tuple of inputs and the output of the algorithm $\hat{\pi}$, will usually be an expression of the metalanguage of L but not of L itself. For each n-tuple $I = (E_1, \ldots, E_n)$ of inputs to $\hat{\pi}$, we assume that M reduces to a formula M_I of L which is called the *instance* of M with respect to I. An instance of M specifies the relationship between the values of a given tuple of input expressions and the value of the resulting output expression. Let $\mathsf{inst}(R)$ be the set of instances of M. M can often be conveniently expressed as a formula schema.

Example 4. Let $R = (\Pi, M)$ where:

1. $\Pi = (\pi, \hat{\pi})$ is the transformer in L_F given in Example 1.
2. M is the formula schema

   $$A \equiv \pi(A)$$

   where A is a formula of L_F.

If A is the formula $p(c) \supset \forall x \,.\, q(x)$ (where c is a constant) and the result of applying $\hat{\pi}$ to (A) is $\forall x \,.\, p(c) \supset q(x)$, then

$$(p(c) \supset \forall x \,.\, q(x)) \equiv (\forall x \,.\, p(c) \supset q(x))$$

is the instance of M with respect to (A). □

Example 5. Let $R = (\Pi, M)$ where:

1. $\Pi = (\pi, \hat{\pi})$ is the transformer in L_F given in Example 2.
2. M is the formula schema

   $$(x = t \wedge A) \supset \pi(t, x, A)$$

   where t is a term, x is a variable, and A is a formula of L_F and t is free for x in A.

If t is a term, x is a variable, and A is $f(x, y) = g(x)$, then

$$(x = t \wedge f(x, y) = g(x)) \supset f(t, y) = g(t)$$

is the instance of M with respect to (t, x, A). □

Example 6. Let $R = (\Pi, M)$ where:

1. $\Pi = (\pi, \hat{\pi})$ is the transformer in the language L of the theory T given in Example 3.
2. M is the formula schema

   $$\mathsf{derivative}(E) = \pi(E)$$

   where E is of type $\mathsf{real} \to \mathsf{real}$. $\mathsf{derivative}$ is an expression of L of type

   $$(\mathsf{real} \to \mathsf{real}) \to (\mathsf{real} \to \mathsf{real})$$

   that maps a function to its derivative. M thus asserts that the derivative of the function denoted by E is the function denoted by $\pi(E)$.

If E is $\lambda\, x : \mathsf{real} \,.\, x^2$, then

$$\mathsf{derivative}(\lambda\, x : \mathsf{real} \,.\, x^2) = (\lambda\, x : \mathsf{real} \,.\, 2 \cdot x)$$

is the instance of M with respect to (E). □

For the sake of convenience, we will view a formula A of L as a (transformerless) rule in L and assume that $\mathsf{trans}(A)$ is undefined, $\mathsf{mean}(A) = A$, and $\mathsf{inst}(A) = \{A\}$.

2.4 Biform Theories

A *biform theory* in $\mathbf{K}$ is a pair $T = (L, \Omega)$ where L is a language of $\mathbf{K}$ and Ω is a set of rules in L. (Ω may include formulas of L viewed as transformer-less rules.) L is the *language* of T, and Ω is the set of *axioms* of T.

T can be viewed as simultaneously both an *axiomatic theory* and an *algorithmic theory*. The *axiomatic theory of* T is the axiomatic theory $T_{\mathrm{axm}} = (L, \Gamma)$ in $\mathbf{K}$ where

$$\Gamma = \bigcup_{R \in \Omega} \mathsf{inst}(R),$$

while the *algorithmic theory of* T is the algorithmic theory $T_{\mathrm{alg}} = (L, \Delta)$ where

$$\Delta = \{\mathsf{trans}(R) \mid R \in \Omega \text{ and } \mathsf{trans}(R) \text{ is defined}\}.$$

The axioms of T—which are formulas and rules—are the background assumptions of T in an implicit form. The axioms of T_{axm}—which are formulas alone—are the background assumptions of T in an explicit form. A rule R in L is a *logical consequence* of T if, for all formulas $A \in \mathsf{inst}(R)$, A is a logical consequence of T_{axm}. Thus, the axioms of T are trivially logical consequences of T. Notice also that, since we are assuming that the formulas of L are rules in L, every logical consequence of T_{axm} is also a logical consequence of T.

3 Chiron

A formal, complete presentation of the syntax and semantics of Chiron is given in [9], and a shorter, more informal presentation is given in [8].

3.1 Values

The semantics of Chiron is based on the notion of a *standard model* which is an elaboration of a model of NBG set theory. The basic values or elements in a model of NBG are classes (which include sets and proper classes).[1] A standard model M includes other values besides classes, but classes are the most important. M is derived from a structure, consisting of a nonempty domain D_{c} of classes and a membership relation $\in$ on D_{c}, that satisfies the axioms of NBG set theory as given, for example, in [13] or [16]. The *values* of M include sets, classes, superclasses, truth values, the undefined value, and operations.

A *class* of M is a member of D_{c}. A *set* of M is a member x of D_{c} such that $x \in y$ for some member y of D_{c}. That is, a set is a class that is itself a member of a class. A class is thus a collection of sets. A class is *proper* if it is not a set. A *superclass* of M is a collection of classes in D_{c}. We consider a class, as a collection of sets, to be a superclass itself. Let D_{v} be the domain of sets of

[1] Recall that values of a model of Zermelo-Fraenkel (ZF) set theory includes only sets, not proper classes.

op	type	formula	op-app	var
type-app	dep-fun-type	fun-app	fun-abs	if
exists	def-des	indef-des	quote	eval
true	false	set	class	expr
expr-op	expr-type	expr-term	expr-formula	in
type-equal	term-equal	formula-equal	not	or

Table 1. The Key Words of Chiron.

M and D_{s} be the domain of superclasses of M. The following inclusions hold: $D_{\mathrm{v}} \subset D_{\mathrm{c}} \subset D_{\mathrm{s}}$. D_{v} is the universal class (the class of all sets), and D_{c} is the universal superclass (the superclass of all classes).

T, F, and $\bot$ are distinct values of M not in D_{s}. T and F represent the truth values *true* and *false*, respectively. $\bot$ is the *undefined value* which serves as the value of undefined terms. For $n \geq 0$, an *n-ary operation* of M is a total mapping from $D_1 \times \cdots \times D_n$ to D_{n+1} where D_i is D_{s}, $D_{\mathrm{c}} \cup \{\bot\}$, or $\{\mathrm{T}, \mathrm{F}\}$ for each i with $1 \leq i \leq n+1$. Let D_{o} be the domain of operations of M. $D_{\mathrm{s}} \cup \{\mathrm{T}, \mathrm{F}, \bot\}$ and D_{o} are assumed to be disjoint.

3.2 Expressions

Let $\mathcal{S}$ be a fixed infinite set of symbols that includes the 30 *key words* in Table 1. The key words are used to classify expressions, identify different categories of expressions, and name the built-in operators (see below).

An *expression* of Chiron is defined inductively by:

1. Each symbol $s \in \mathcal{S}$ is an expression.
2. If $e_1, \ldots, e_n$ are expressions where $n \geq 0$, then $(e_1, \ldots, e_n)$ is an expression.

Hence, an expression is an S-expression (with commas in place of spaces) that exhibits the structure of a tree whose leaves are symbols in $\mathcal{S}$. Let $\mathcal{E}$ be the set of expressions of Chiron.

There are four special sorts of expressions: *operators*, *types*, *terms*, and *formulas*. An expression is *proper* if it is one of these special sorts of expressions, and an expression is *improper* if it is not proper. Proper expressions denote values of M, while improper expressions are nondenoting (i.e., they do not denote anything). Operators are used to construct expressions. They denote operations. Types are used to restrict the values of operators and variables and to classify terms by their values. They denote superclasses. Terms are used to describe classes. They denote classes or the undefined value $\bot$. Formulas are used to make assertions. They denote truth values. A *kind* is the key word type, a type, or the key word formula.

A term is *defined* if it denotes a class and is *undefined* if it denotes $\bot$. Every term is assigned a type. Suppose a term a is assigned a type α. Then a is said to be a *term of type* α. Suppose further α denotes a superclass Σ_α. If a is defined,

i.e., a denotes a class x, then x is in Σ_α. The value of an intuitively nondenoting term is the undefined value $\bot$, but the value of a intuitively nondenoting type or formula is D_c (the universal superclass) or F (false), respectively. That is, the values for intuitively nondenoting types, terms, and formulas are the default values D_c, $\bot$, and F, respectively. Hence every proper expression, even one that is intuitively nondenoting, denotes some value.

There are 13 proper expression categories. They are shown in Table 2 in both a compact notation in the middle and the official S-expression-like notation on the right. $O, P, Q, \ldots$ denote operators, $\alpha, \beta, \gamma, \ldots$ denote types, $a, b, c, \ldots$ denote terms, $A, B, C, \ldots$ denote formulas, $s, t, u, \ldots$ denote symbols, $e, e', \ldots$ denote expressions, and $k, k', \ldots$ denote kinds.

Expression Category	**Compact Notation**	**Official Notation**
Operator	$(s :: k_1, \ldots, k_{n+1})$	$(\mathsf{op}, s, k_1, \ldots, k_{n+1})$
Operator application	$(s :: k_1, \ldots, k_{n+1})$ $(e_1, \ldots, e_n)$	$(\mathsf{op\text{-}app}, (\mathsf{op}, s, k_1, \ldots, k_{n+1}),$ $e_1, \ldots, e_n)$
Variable	$(x : \alpha)$	$(\mathsf{var}, x, \alpha)$
Type application	$\alpha(a)$	$(\mathsf{type\text{-}app}, \alpha, a)$
Dependent Function Type	$(\Lambda\, x : \alpha\ .\ \beta)$	$(\mathsf{dep\text{-}fun\text{-}type}, (\mathsf{var}, x, \alpha), \beta)$
Function application	$f(a)$	$(\mathsf{fun\text{-}app}, f, a)$
Function abstraction	$(\lambda\, x : \alpha\ .\ b)$	$(\mathsf{fun\text{-}abs}, (\mathsf{var}, x, \alpha), b)$
Conditional term	$\mathsf{if}(A, b, c)$	(if, A, b, c)
Existential quantification	$(\exists\, x : \alpha\ .\ B)$	$(\mathsf{exists}, (\mathsf{var}, x, \alpha), B)$
Definite description	$(\iota\, x : \alpha\ .\ B)$	$(\mathsf{def\text{-}des}, (\mathsf{var}, x, \alpha), B)$
Indefinite description	$(\epsilon\, x : \alpha\ .\ B)$	$(\mathsf{indef\text{-}des}, (\mathsf{var}, x, \alpha), B)$
Quotation	$\lceil e \rceil$	(quote, e)
Evaluation	$[\![a]\!]_\mathrm{ty}$	$(\mathsf{eval}, a, \mathsf{type})$
	$[\![a]\!]_\alpha$	$(\mathsf{eval}, a, \alpha)$
	$[\![a]\!]_\mathrm{fo}$	$(\mathsf{eval}, a, \mathsf{formula})$

Table 2. Compact Notation

Table 3 defines additional compact notation for the built-in operators and the universal quantifier. The compact notation also includes some customary abbreviation rules (see [9]).

3.3 Quotation and Evaluation

If e is any expression, proper or improper, then

$$(\mathsf{quote}, e)$$

is a term of type E called a *quotation*. The value of the quotation is a set, called the *construction* of e, that represents the syntactic structure of the expression e. Thus a proper expression e has two different meanings:

Compact Notation	Defining Expression
T	$(\mathsf{true} :: \mathsf{formula})()$
F	$(\mathsf{false} :: \mathsf{formula})()$
V	$(\mathsf{set} :: \mathsf{type})()$
C	$(\mathsf{class} :: \mathsf{type})()$
E	$(\mathsf{expr} :: \mathsf{type})()$
E_{op}	$(\mathsf{expr\text{-}op} :: \mathsf{type})()$
E_{ty}	$(\mathsf{expr\text{-}type} :: \mathsf{type})()$
E_{te}	$(\mathsf{expr\text{-}term} :: \mathsf{type})()$
E_{fo}	$(\mathsf{expr\text{-}formula} :: \mathsf{type})()$
$(a \in b)$	$(\mathsf{in} :: \mathsf{V}, \mathsf{C}, \mathsf{formula})(a, b)$
$(\alpha =_{\mathrm{ty}} \beta)$	$(\mathsf{type\text{-}equal} :: \mathsf{type}, \mathsf{type}, \mathsf{formula})(\alpha, \beta)$
$(a =_{\alpha} b)$	$(\mathsf{term\text{-}equal} :: \mathsf{C}, \mathsf{C}, \mathsf{type}, \mathsf{formula})(a, b, \alpha)$
$(a = b)$	$(a =_{\mathsf{C}} b)$
$(A \equiv B)$	$(\mathsf{formula\text{-}equal} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})(A, B)$
$(\neg A)$	$(\mathsf{not} :: \mathsf{formula}, \mathsf{formula})(A)$
$(a \notin b)$	$(\neg(a \in b))$
$(a \neq b)$	$(\neg(a = b))$
$(A \vee B)$	$(\mathsf{or} :: \mathsf{formula}, \mathsf{formula}, \mathsf{formula})(A, B)$
$(\forall\, x : \alpha \,.\, A)$	$(\neg(\exists\, x : \alpha \,.\, (\neg A)))$

Table 3. Additional Compact Notation

1. The *semantic meaning* of e is the value denoted by e itself.
2. The *syntactic meaning* of e is the construction denoted by (quote, e).

If a is a term and k is a kind, then

$$(\mathsf{eval}, a, k)$$

is an expression called an *evaluation* that is a type if $k = \mathsf{type}$, a term of type k if k is a type, and a formula if $k = \mathsf{formula}$. Roughly speaking, if a denotes a construction that represents an expression e, then the evaluation denotes the value of e. If a denotes a construction that represents an expression in which the symbol eval occurs, then the evaluation is undefined. This provision is needed to block the liar paradox and similar semantically ungrounded expressions (see [9]).

3.4 Biform Theories in Chiron

Let L be a language of Chiron. An *n-ary transformer in L* is an n-ary transformer $\mathit{\Pi} = (\pi, \hat{\pi})$ where π is an n-ary operator $(s :: \mathsf{E}, \ldots, \mathsf{E})$ in L (with E occurring $n+1$ times). A *rule in L* is a rule $R = (\mathit{\Pi}, M)$ where $\mathit{\Pi} = (\pi, \hat{\pi})$ is an n-ary transformer in L and M is formula of Chiron having the form

$$\forall\, e_1 \,.\, \mathsf{E}_1, \ldots, e_n : \mathsf{E}_n \,.\, M'$$

where E_i is E, E_{op}, E_{ty}, E_{te}, or E_{fo} for all i with $1 \leq i \leq n$. If $a_1, \ldots, a_n$ are quotations (of type E), then the *instance* of M with respect to $(a_1, \ldots, a_n)$ is the result of replacing each occurrence of $\pi(a_1, \ldots, a_n)$ in

$$M[e_1 \mapsto a_1, \ldots, e_n \mapsto a_1]$$

with $f_{\hat{\pi}}(a_1, \ldots, a_n)$ if this is defined and with $\bot_{\mathsf{C}}$ (which denotes the undefined value) if this is undefined. A *biform theory in* Chiron is a pair $T = (L, \Omega)$ where L is a language of Chiron and Ω is a set of rules in L.

Example 7. Let R be the rule given in Example 4 expressed as a rule in a language of Chiron. Then M would be the formula

$$\forall\, e : \mathsf{E}_{\mathrm{fo}} \,.\, [\![e]\!]_{\mathrm{fo}} \equiv [\![\pi(e)]\!]_{\mathrm{fo}},$$

and the instance of M with respect to $(\ulcorner p(0) \supset \forall\, x \,.\, q(x) \urcorner)$ would be

$$[\![\ulcorner p(0) \supset \forall\, x \,.\, q(x) \urcorner]\!]_{\mathrm{fo}} \equiv [\![\ulcorner \forall\, x \,.\, p(0) \supset q(x) \urcorner]\!]_{\mathrm{fo}},$$

which reduces to

$$(p(0) \supset \forall\, x \,.\, q(x)) \equiv (\forall\, x \,.\, p(0) \supset q(x))$$

as desired. □

Example 8. Let R be the rule given in Example 5 expressed as a rule in a language of Chiron. Then M would be the formula

$$\forall\, e_1 : \mathsf{E}_{\mathrm{te}}, e_2 : \mathsf{E}_{\mathrm{te}}, e_3 : \mathsf{E}_{\mathrm{fo}} \,.$$
$$(\mathsf{is\text{-}var}(e_2) \wedge \mathsf{free\text{-}for}(e_1, e_2, e_3) \wedge [\![e_2]\!]_{\mathrm{te}} = [\![e_1]\!]_{\mathrm{te}} \wedge [\![e_3]\!]_{\mathrm{fo}}) \supset [\![\pi(e_1, e_2, e_3)]\!]_{\mathrm{fo}}$$

which says that, for all expressions E_1, E_2, E_3, if E_1 is a term t, E_2 is a variable x, and E_3 is a formula A such that t is free for x in A, then $x = t \wedge A$ implies the result of applying the algorithm $\hat{\pi}$ to (t, x, A). Notice that the syntactic side condition of the formula schema in Example 5 (that says t is free for x in A) has been directly incorporated into M. □

Example 9. Let R be the rule given in Example 6 expressed as a rule in a language of Chiron. Assume that $\mathsf{real} \to \mathsf{real}$ is the type

$$(\Lambda\, x : \mathsf{real} \,.\, \mathsf{real})$$

and deriv is the operator

$$(\mathsf{derivative} :: \mathsf{real} \to \mathsf{real}, \mathsf{real} \to \mathsf{real}).$$

Also let $(a \downarrow \alpha)$ mean that the term a is defined with a value in the denotation of the type α. Then M would be the formula

$$\forall\, e : \mathsf{E}_{\mathrm{te}} \,.\, ([\![e]\!]_{\mathrm{te}} \downarrow \mathsf{real} \to \mathsf{real}) \supset \mathsf{deriv}([\![e]\!]_{\mathrm{te}}) = [\![\pi(e)]\!]_{\mathrm{te}}$$

which says that, for all expressions E, if E is a term t that denotes a function f that maps real numbers to real numbers, then the result of applying the algorithm $\hat{\pi}$ to (t) is a term that denotes the derivative of f. □

See [9] for further details, discussion, examples, and references concerning Chiron.

4 An Example

In this section we will sketch the development of a nontrivial biform theory. We will start with a theory $T = (L, \Omega)$ of (higher-order) Peano arithmetic where:

- L contains operators $\mathsf{nat}, 0, S$ that represent the type of natural numbers, zero, and the successor function, respectively.
- Ω contains three formulas that express that 0 does not succeed another natural number, that the successor function is injective, and the full induction principle over all sets of natural numbers.

The next step is to extend T to $T' = (L', \Omega')$ by introducing defined operators $1, +, *$ for one, the addition function, and the multiplication function, respectively. 1 is defined as the successor of 0. $+$ and $*$ are defined recursively.

The last step is to extend T' to $T'' = (L'', \Omega'')$ by introducing the machinery to add and multiply binary numerals. Define a *(binary) numeral* to be an expression $(a_1, \ldots, a_n)$ where $n \geq 1$ and a_i is 0 or 1 for each i with $1 \leq i \leq n$. As defined, a numeral is an improper expression, and thus it denotes nothing. However, if n is a numeral, then $\lceil n \rceil$ is a proper expression that denotes the construction of n. We can introduce defined operators $\mathsf{num}, \mathsf{num\text{-}val}$ that represent the type of numerals and a function that maps the type of numerals onto the type of natural numbers.

We can then define a rule $R = (\Pi, M)$ for numeral addition where $\Pi = (\mathsf{add}, \mathsf{add\text{-}alg})$ is a binary transformer and M is the formula

$$\forall\, m, n : \mathsf{num}\ .$$
$$\mathsf{num\text{-}val}(m) + \mathsf{num\text{-}val}(n) = \mathsf{num\text{-}val}((\mathsf{add} :: \mathsf{num}, \mathsf{num}, \mathsf{num})(m, n)).$$

This formula says that the sum of the values of two numerals equals the value of the output of the algorithm $\mathsf{add\text{-}alg}$ when given the two numerals as input. The formula also says implicitly that

$$(\mathsf{add} :: \mathsf{num}, \mathsf{num}, \mathsf{num})(m, n)$$

is defined iff m and n both denote numerals. The algorithm $\mathsf{add\text{-}alg}$ could be implemented, for example, as a lambda-expression of L'' or as a program in some convenient programming language. We can introduce a rule for numeral multiplication in a similar way.

5 Conclusion

The notion of a biform theory enables axiomatic mathematics and algorithmic mathematics to be expressed together in one theory. A biform theory consists of a set of axioms that includes both formulas and rules. A rule is an expression-manipulating algorithm called a transformer coupled with a meaning formula that defines its semantics. A biform theory can be viewed both as an axiomatic theory and as an algorithmic theory. The algorithmic theory provides the deduction and computation rules for reasoning within the theory, while the axiomatic

theory provides the context in which to understand the reasoning that is done via these rules. The axioms of a biform theory are the implicit background assumptions of the theory that define what formulas and rules are logical consequences of the theory.

Since transformers are algorithms that manipulate expressions, the meaning formulas of biform theory rules can only be directly formalized in a logic with support for reasoning about the syntax of expressions. Traditional logics do not offer this kind of support. Chiron is a general-purpose logic with high theoretical and practical expressivity and a facility for reasoning about the syntax of expressions. As a result, it is exceptionally well-suited for formalizing biform theories. Meaning formulas—that would usually be expressed as formula schemas in tradition logics—can be directly expressed in Chiron.

Biform theories can also be formalized in other logics that provide a means to reason about syntax. Many approaches for formalizing the syntax of expressions have been proposed starting with K. Gödel's famous *arithmetization of syntax* via Gödel numbering [12]. Two good surveys of this research area are [14] and the extended version of [17].

A great deal of research has been directed to the problem of how to integrate computer theorem proving and computer algebra. Much of this research has been done in connection with the Calculemus Project [3] or has been presented at the Calculemus symposia that began in 1996. Two research initiatives that are closely related to biform theories and the MathScheme project are the Theorema project [2] at the RISC Research Institute for Symbolic Computation [18] and the work by H. Barendregt and F. Wiedijk on the foundations of computerized mathematics [1].

The development and application of Chiron is a long-range research project composed of the following four tasks:

1. The design of Chiron.
2. The design of a proof system for Chiron.
3. The development of an implementation of Chiron and its proof system.
4. The development of a series of applications to demonstrate Chiron's reach and level of effectiveness.

The first task is largely completed [8, 9]. The last three tasks have hardly been started. This paper begins the fourth task.

Acknowledgments

The author is grateful to Marc Bender and Jacques Carette for many valuable discussions on the design and use of Chiron. Over the course of these discussions, Dr. Carette convinced the author that Chiron needs to include a powerful facility for reasoning about the syntax of expressions. The author would also like to thank the referees for their suggestions on how to improve the paper.

References

1. H. Barendregt and F. Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:2351–2375, 2005.
2. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4:470–504, 2006.
3. Calculemus Project: Systems for Integrated Computation and Deduction. Web site at `http://www.calculemus.net/`.
4. J. Carette. Understanding expression simplification. In J. Gutierrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation (ISSAC 2004)*, pages 72–79. ACM Press, 2004.
5. J. Carette, W. M. Farmer, and V. Sorge. A rational reconstruction of a system for experimental mathematics. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, Lecture Notes in Computer Science, pages 13–26. Springer-Verlag, 2007.
6. J. Carette, W. M. Farmer, and J. Wajs. Trustable communication between mathematics systems. In T. Hardin and R. Rioboo, editors, *Calculemus 2003*, pages 58–68, Rome, Italy, 2003. Aracne.
7. W. M. Farmer. The seven virtues of simple type theory. SQRL Report No. 18, McMaster University, 2003. Revised 2006.
8. W. M. Farmer. Chiron: A multi-paradigm logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1–19. University of Białystok, 2007.
9. W. M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University, 2007. Revised 2008.
10. W. M. Farmer and M. von 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.
11. W. M. Farmer and M. von Mohrenschildt. An overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.
12. K. Gödel. Über formal unentscheidbare Sätze der Principia Mthematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
13. K. Gödel. *The Consistency of the Axiom of Choice and the Generalized Continuum Hypothesis with the Axioms of Set Theory*, volume 3 of *Annals of Mathematical Studies*. Princeton University Press, 1940.
14. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available at `http://www.cl.cam.ac.uk/ jrh13/papers/reflect.ps.gz`.
15. MathScheme: An Integrated Framework for Computer Algebra and Computer Theorem Proving. Web site at `http://imps.mcmaster.ca/mathscheme/`.
16. E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, fourth edition, 1997.
17. Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In A. Momigliano and R. Pollack, editors, *MERLIN'05: Proceedings of the Third*

ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding, pages 2–12. ACM Press, 2005. An extended version is available as a California Institute of Technology technical report, CaltechCSTR:2005.003.

18. RISC Research Institute for Symbolic Computation. Web site at `http://www.risc.uni-linz.ac.at//`.