The Formalization of Syntax-Based Mathematical Algorithms Using Quotation and Evaluation*

William M. Farmer**

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

5 August 2013

Abstract. Algorithms like those for differentiating functional expressions manipulate the syntactic structure of mathematical expressions in a mathematically meaningful way. A formalization of such an algorithm should include a specification of its computational behavior, a specification of its mathematical meaning, and a mechanism for applying the algorithm to actual expressions. Achieving these goals requires the ability to integrate reasoning about the syntax of the expressions with reasoning about what the expressions mean. A *syntax framework* is a mathematical structure that is an abstract model for a syntax reasoning system. It contains a mapping of expressions to *syntactic values* that represent the syntactic structures of the expressions; a language for reasoning about syntactic values; a *quotation* mechanism to refer to the syntactic value of an expression; and an *evaluation* mechanism to refer to the value of the expression represented by a syntactic value. We present and compare two approaches, based on instances of a syntax framework, to formalize a syntax-based mathematical algorithm in a formal theory T. In the first approach the syntactic values for the expressions manipulated by the algorithm are members of an inductive type in T, but quotation and evaluation are functions defined in the metatheory of T. In the second approach every expression in T is represented by a syntactic value, and quotation and evaluation are operators in T itself.

1 Introduction

A great many of the algorithms employed in mathematics work by manipulating the syntactic structure of mathematical expressions in a mathematically meaningful way. Here are some examples:

* Appears in *Intelligent Computer Mathematics* (proceedings of CICM 2013), *Lecture Notes in Computer Science*, Vol. 7961, pp. 35–50, Springer-Verlag, 2013. The final publication is available at `link.springer.com`. This research was supported by NSERC.

** `wmfarmer@mcmaster.ca`.

1. Arithmetic operations applied to numerals.
2. Operations such as factorization applied to polynomials.
3. Simplification of algebraic expressions.
4. Operations such as transposition performed on matrices.
5. Symbolic differentiation and antidifferentiation of expressions with variables.

The study and application of these kinds of algorithms is called *symbolic computation.* For centuries symbolic computation was performed almost entirely using pencil and paper (and similar devices). However, today symbolic computation can be performed by computer, and algorithms that manipulate mathematical expressions are the main fare of computer algebra systems.

In this paper we are interested in the problem of how to formalize syntax-based mathematical algorithms. These algorithms manipulate members of a formal language in a computer algebra system, but their behavior and meaning are usually not formally expressed in a computer algebra system. However, we want to use these algorithms in formal theories and formally understand what they do. We are interested in employing existing external implementations of these algorithms in formal theories as well as implementing these algorithms directly in formal theories.

As an illustration, consider an algorithm, say named $\mathsf{RatPlus}$, that adds rational number numerals, which are represented in memory in some suitable way. (An important issue, that we will not address, is how the numerals are represented to optimize the efficiency of $\mathsf{RatPlus}$.) For example, if the numerals $\frac{2}{5}$ and $\frac{3}{8}$ are given to $\mathsf{RatPlus}$ as input, the numeral $\frac{31}{40}$ is returned by $\mathsf{RatPlus}$ as output. What would we need to do to use $\mathsf{RatPlus}$ to add rational numbers in a formal theory T and be confident that the results are correct? First, we would have to introduce values in T to represent rational number numerals as syntactic structures, and then define a binary operator O over these values that has the same input-output relation as $\mathsf{RatPlus}$. Second, we would have to prove in T that, if $O(a, b) = c$, then the sum of the rational numbers represented by a and b is the rational number represented by c. And third, we would have to devise a mechanism for using the definition of O to add rational numbers in T.

The second task is the most challenging. The operator O, like $\mathsf{RatPlus}$, manipulates numerals as syntactic structures. To state and then prove that these manipulations are mathematically meaningful requires the ability to express the interplay of how the numerals are manipulated and what the manipulations mean with respect to rational numbers. This is a formidable task in a traditional logic in which there is no mechanism for directly referring to the syntax of the expressions in the logic. We need to reason about a rational number numeral $\frac{2}{5}$ both as a syntactic structure that can be deconstructed into the integer numerals 2 and 5 and as an expression that denotes the rational number $2/5$.

Let us try to make the problem of how to formalize syntax-based mathematical algorithms like $\mathsf{RatPlus}$ more precise. Let T be a theory in a traditional logic like first-order logic or simple type theory, and let A be an algorithm that manipulates certain expressions of T. To formalize A in T we need to do three things:

1. *Define an operator O_A in T that represents A*: Introduce values in T that represent the expressions manipulated by A. Introduce an operator O_A in T that maps the values that represent the input expressions taken by A to the values that represent the output expressions produced by A. Write a sentence named $\mathsf{CompBehavior}$ in T that specifies the *computational behavior* of O_A to be the same as that of A. That is, if A takes an input expression e and produces an output expression e', then $\mathsf{CompBehavior}$ should say that O_A maps the value that represents e to the value that represents e'.
2. *Prove in T that O_A is mathematically correct*: Write a sentence named $\mathsf{MathMeaning}$ in T that specifies the *mathematical meaning* of O_A to be the same as that of A. That is, if the value of an input expression e given to A is related to the value of the corresponding output expression e' produced by A in a particular way, then $\mathsf{MathMeaning}$ should say that the value of the expression representing e should be related to the value of the expression representing e' in the same way. Finally, prove $\mathsf{MathMeaning}$ from $\mathsf{CompBehavior}$ in T.
3. *Devise a mechanism for using O_A in T*: An application $O_A(a_1, \ldots, a_n)$ of O_A to the values $a_1, \ldots, a_n$ can be used in T by instantiating $\mathsf{MathMeaning}$ with $a_1, \ldots, a_n$ and then simplifying the resulting formula to obtain a statement about the value of $O_A(a_1, \ldots, a_n)$. For the sake of convenience or efficiency, we might want to use A itself to compute $O_A(a_1, \ldots, a_n)$. We will know that results produced by A are correct provided A and O_A have the same computational behavior.

If we believe that A works correctly and we are happy to do our computations with A outside of T, we can skip the writing of $\mathsf{CompBehavior}$ and use $\mathsf{MathMeaning}$ as an axiom that asserts A has the mathematical meaning specified by $\mathsf{MathMeaning}$ for O_A. The idea of treating specifications of external algorithms as axioms is a key component of the notion of a *biform theory* [2, 6].

So to use A in T we need to formalize A in T, and to do this, we need a system that integrates reasoning about the syntax of the expressions with reasoning about what the expressions mean. A *syntax framework* [9] is a mathematical structure that is an abstract model for a syntax reasoning system. It contains a mapping of expressions to *syntactic values* that represent the syntactic structures of the expressions; a language for reasoning about syntactic values; a quotation mechanism to refer to the syntactic value of an expression; and an evaluation mechanism to refer to the value of the expression represented by a syntactic value. A syntax framework provides the tools needed to reason about the interplay of syntax and semantics. It is just what we need to formalize syntax-based mathematical algorithms.

Reflection is a technique to embed reasoning about a reasoning system (i.e., metareasoning) in the reasoning system itself. Reflection has been employed in logic [13], theorem proving [12], and programming [5]. Since metareasoning very often involves the syntactic manipulation of expressions, a syntax framework is a natural subcomponent of a reflection mechanism.

This paper attacks the problem of formalizing a syntax-based mathematical algorithm A in a formal theory T using syntax frameworks. Two approaches are presented and compared. The first approach is local in nature. It employs a syntax framework in which there are syntactic values only for the expressions manipulated by A. The second approach is global in nature. It employs a syntax framework in which there are syntactic values for all the expressions of T. We will see that these two approaches have contrasting strengths and weaknesses. The local approach offers an incomplete solution at a low cost, while the global approach offers a complete solution at a high cost.

The two approaches will be illustrated using the familiar example of polynomial differentiation. In particular, we will discuss how the two approaches can be employed to formalize an algorithm that differentiates expressions with variables that denote real-valued polynomial functions. We will show that algorithms like differentiation that manipulate expressions with variables are more challenging to formalize than algorithms like symbolic arithmetic that manipulate numerals without variables.

The following is the outline of the paper. The next section, Section 2, presents the paper's principal example, polynomial differentiation. The notion of a syntax framework is defined in Section 3. Sections 4 and 5 present the local and global approaches to formalizing syntax-based mathematical algorithms. And the paper concludes with Section 6.

2 Example: Polynomial Differentiation

We examine in this section the problem of how to formalize a symbolic differentiation algorithm and then prove that the algorithm actually computes derivatives. We start by defining what a derivative is.

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function over the real numbers and $a \in \mathbb{R}$. The *derivative of f at a*, written $\mathsf{deriv}(f, a)$, is

$$\lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$

if this limit exists. The *derivative of f*, written $\mathsf{deriv}(f)$, is the function

$$\lambda\, x : \mathbb{R} \,.\, \mathsf{deriv}(f, x).$$

Notice that we are using the traditional definition of a derivative in which a derivative of a function is defined pointwise.

Differentiation is in general the process of finding derivatives which ultimately reduces to finding limits. *Symbolic differentiation* is the process of mechanically transforming an expression with variables that represents a function over the real numbers into an expression with variables that represents the derivative of the function. For example, the result of symbolically differentiating the expression $\mathsf{sin}(x^2)$ which represents the function $\lambda\, x : \mathbb{R} \,.\, \mathsf{sin}(x^2)$ is the expression $2 \cdot x \cdot \mathsf{cos}(x^2)$ which represents the function $\lambda\, x : \mathbb{R} \,.\, 2 \cdot x \cdot \mathsf{cos}(x^2)$.

Symbolic differentiation is performed by applying certain *differentiation rules* and *simplification rules* to a starting expression until no rule is applicable.

Let us look at how symbolic differentiation works on polynomials. A *polynomial* is an expression constructed from real-valued constants and variables by applying addition, subtraction, multiplication, and natural number exponentiation. For example, $x \cdot (x^2 + y)$ is a polynomial. The symbolic differentiation of polynomials is performed using the following well-known differentiation rules:

Constant Rule

$$\frac{d}{dx}(c) = 0 \quad \text{where } c \text{ is a constant or a variable different from } x.$$

Variable Rule

$$\frac{d}{dx}(x) = 1.$$

Sum and Difference Rule

$$\frac{d}{dx}(u \pm v) = \frac{d}{dx}(u) \pm \frac{d}{dx}(v).$$

Product Rule

$$\frac{d}{dx}(u \cdot v) = \frac{d}{dx}(u) \cdot v + u \cdot \frac{d}{dx}(v).$$

Power Rule

$$\frac{d}{dx}(u^n) = \begin{cases} 0 & \text{if } n = 0 \\ n \cdot u^{n-1} \cdot \frac{d}{dx}(u) & \text{if } n > 0. \end{cases}$$

Written using traditional Leibniz notation, the rules specify how symbolic differentiation is performed with respect to the variable x. The symbols u and v range over polynomials that may contain x as well as other variables, and the symbol n ranges over natural numbers. Notice that these rules are not meaning preserving in the usual way; for example, the rule $\frac{d}{dx}(c) = 0$ is not meaning preserving if we view c as a value and not as an expression.

Let PolyDiff be the algorithm that, given a polynomial u and variable x, applies the five differentiation rules above to the starting expression $\frac{d}{dx}(u)$ until there are no longer any expressions starting with $\frac{d}{dx}$ and then simplifies the resulting expression using the rules $0 + u = u + 0 = 0$ and $1 \cdot u = u \cdot 1 = u$ and collecting like terms. Applied to $x \cdot (x^2 + y)$, PolyDiff would perform the following steps:

$$\begin{aligned}
\frac{d}{dx}(x \cdot (x^2 + y)) &= \frac{d}{dx}(x) \cdot (x^2 + y) + x \cdot \frac{d}{dx}(x^2 + y) && (1) \\
&= 1 \cdot (x^2 + y) + x \cdot \left(\frac{d}{dx}(x^2) + \frac{d}{dx}(y)\right) && (2) \\
&= 1 \cdot (x^2 + y) + x \cdot \left(2 \cdot x^1 \cdot \frac{d}{dx}(x) + 0\right) && (3) \\
&= 1 \cdot (x^2 + y) + x \cdot (2 \cdot x^1 \cdot 1 + 0) && (4) \\
&= 3 \cdot x^2 + y && (5)
\end{aligned}$$

Line (1) is by the Product Rule; (2) is by the Variable and Sum and Difference Rules; (3) is by the Power and Constant Rules; (4) is by the Variable Rule; and (5) is by the simplification rules. Thus, given the function

$$f = \lambda\, x : \mathbb{R} \,.\, x \cdot (x^2 + y),$$

using PolyDiff we are able to obtain the derivative

$$\lambda\, x : \mathbb{R} \,.\, 3 \cdot x^2 + y$$

of f via mechanical manipulation of the expression $x \cdot (x^2 + y)$.

Algorithms similar to PolyDiff are commonly employed in informal mathematics. In fact, they are learned and applied by every calculus student. They should be as available and useful in formal mathematics as they are in informal mathematics. We thus need to formalize them as described in the Introduction.

The main objective of this paper is to show how syntax-based mathematical algorithms can be formalized using PolyDiff as an example. We will begin by making the task of formalizing PolyDiff precise.

Let a *theory* be a pair $T = (L, \Gamma)$ where L is a formal language and Γ is a set of sentences in L that serve as the axioms of the theory. Define $T_R = (L_R, \Gamma_R)$ to be a theory of the real numbers in (many-sorted) simple type theory. We assume that L_R is a set of expressions over a signature that includes a type $\mathbb{R}$ of the real numbers, constants for each natural number, and constants for addition, subtraction, multiplication, natural number exponentiation, and the unary and binary deriv operators defined above. We assume that Γ_R contains the axioms of a complete ordered field as well as the definitions of all the defined constants in L_R (see [8] for further details).

Let $L_{\mathrm{var}} \subseteq L_R$ be the set of variables of type $\mathbb{R}$ and $L_{\mathrm{poly}} \subseteq L_R$ be the set of expressions constructed from members of L_{var}, constants of type $\mathbb{R}$, addition, subtraction, multiplication, and natural number exponentiation. Finally, assume that PolyDiff : $L_{\mathrm{poly}} \times L_{\mathrm{var}} \to L_{\mathrm{poly}}$ is the algorithm described in the previous section adapted to operate on expressions of L_R.

Thus to formalize PolyDiff we need to:

1. Define an operator O_{pd} in T_R that represents PolyDiff.
2. Prove in T_R that O_{pd} is mathematically correct.
3. Devise a mechanism for using O_{pd} in T_R.

Formalizing PolyDiff should be much easier than formalizing differentiation algorithms for larger sets of expressions that include, for example, rational expressions and transcendental functions. Polynomial functions are total (i.e., they are defined at all points on the real line) and their derivatives are also total. As a result, issues of undefinedness do not arise when specifying the mathematical meaning of PolyDiff.

However, functions more general than polynomial functions as well as their derivatives may be undefined at some points. Thus using a differentiation algorithm to compute the derivative of one of these more general functions requires care in determining the precise domain of the derivative. For example, differentiating the rational expression x/x using the well-known Quotient Rule yields the expression 0, but the derivative of $\lambda\, x : \mathbb{R} \,.\, x/x$ is not $\lambda\, x : \mathbb{R} \,.\, 0$. The derivative is actually the partial function

$$\lambda\, x : \mathbb{R} \,.\, \mathsf{if}\ x \neq 0\ \mathsf{then}\ 0\ \mathsf{else}\ \bot.$$

We restrict our attention to differentiating polynomial functions so that we can focus on reasoning about syntax without being concerned about issues of undefinedness.

3 Syntax Frameworks

A syntax framework [9] is a mathematical structure that is intended to be an abstract model of a system for reasoning about the syntax of an interpreted language (i.e., a formal language with a semantics). It will take several definitions from [9] to present this structure.

Definition 1 (Interpreted Language). An *interpreted language* is a triple $I = (L, D_{\rm sem}, V_{\rm sem})$ where:

1. L is a formal language, i.e, a set of expressions.[1]
2. $D_{\rm sem}$ is a nonempty domain (set) of *semantic values*.
3. $V_{\rm sem} : L \rightarrow D_{\rm sem}$ is a total function, called a *semantic valuation function*, that assigns each expression $e \in L$ a semantic value $V_{\rm sem}(e) \in D_{\rm sem}$. □

A syntax representation of a formal language is an assignment of syntactic values to the expressions of the language:

Definition 2 (Syntax Representation). Let L be a formal language. A *syntax representation* of L is a pair $R = (D_{\rm syn}, V_{\rm syn})$ where:

1. $D_{\rm syn}$ is a nonempty domain (set) of *syntactic values*. Each member of $D_{\rm syn}$ represents a syntactic structure.
2. $V_{\rm syn} : L \rightarrow D_{\rm syn}$ is an injective, total function, called a *syntactic valuation function*, that assigns each expression $e \in L$ a syntactic value $V_{\rm syn}(e) \in D_{\rm syn}$ such that $V_{\rm syn}(e)$ represents the syntactic structure of e. □

A syntax language for a syntax representation is a language of expressions that denote syntactic values in the syntax representation:

Definition 3 (Syntax Language). Let $R = (D_{\rm syn}, V_{\rm syn})$ be a syntax representation of a formal language $L_{\rm obj}$. A *syntax language* for R is a pair $(L_{\rm syn}, I)$ where:

[1] No distinction is made between how expressions are constructed in this definition as well as in subsequent definitions. In particular, expressions constructed by binding variables are not treated in any special way.

1. $I = (L, D_{\text{sem}}, V_{\text{sem}})$ in an interpreted language.
2. $L_{\text{obj}} \subseteq L$, $L_{\text{syn}} \subseteq L$, and $D_{\text{syn}} \subseteq D_{\text{sem}}$.
3. V_{sem} restricted to L_{syn} is a total function $V'_{\text{sem}} : L_{\text{syn}} \to D_{\text{syn}}$. □

Finally, we are now ready to define a syntax framework:

Definition 4 (Syntax Framework in an Interpreted Language). Let $I = (L, D_{\text{sem}}, V_{\text{sem}})$ be an interpreted language and L_{obj} be a sublanguage of L. A *syntax framework* for (L_{obj}, I) is a tuple $F = (D_{\text{syn}}, V_{\text{syn}}, L_{\text{syn}}, Q, E)$ where:

1. $R = (D_{\text{syn}}, V_{\text{syn}})$ is a syntax representation of L_{obj}.
2. (L_{syn}, I) is a syntax language for R.
3. $Q : L_{\text{obj}} \to L_{\text{syn}}$ is an injective, total function, called a *quotation function*, such that:
   Quotation Axiom. For all $e \in L_{\text{obj}}$,
   $$V_{\text{sem}}(Q(e)) = V_{\text{syn}}(e).$$
4. $E : L_{\text{syn}} \to L_{\text{obj}}$ is a (possibly partial) function, called an *evaluation function*, such that:
   Evaluation Axiom. For all $e \in L_{\text{syn}}$,
   $$V_{\text{sem}}(E(e)) = V_{\text{sem}}(V_{\text{syn}}^{-1}(V_{\text{sem}}(e)))$$
   whenever $E(e)$ is defined. □

A syntax framework is depicted in Figure 1. For $e \in L_{\text{obj}}$, $Q(e)$ is called the *quotation* of e. $Q(e)$ denotes a value in D_{syn} that represents the syntactic structure of e. For $e \in L_{\text{syn}}$, $E(e)$ is called the *evaluation* of e. If it is defined, $E(e)$ denotes the same value in D_{sem} that the expression represented by the value of e denotes. Since there will usually be different $e_1, e_2 \in L_{\text{syn}}$ that denote the same syntactic value, E will usually not be injective. Q and E correspond to the quote and eval operators in Lisp and other languages.

Common examples of syntax frameworks are based on representing the syntax of expressions by Gödel numbers, strings, and members of an inductive type. Programming languages that support metaprogramming — such as Lisp, F# [10], MetaML [18], MetaOCaml [15], reFLect [11], and Template Haskell [16] — are instances of a syntax framework if mutable variables are disallowed. See [9] for these and other examples of syntax frameworks.

The notion of a syntax framework can be easily lifted from an interpreted language to an interpreted theory. This is the version of a syntax framework that we will use in this paper.

Definition 5 (Model). Let $T = (L, \Gamma)$ be a theory. A *model* of T is a pair $M = (D^M_{\text{sem}}, V^M_{\text{sem}})$ such that D^M_{sem} is a nonempty set of semantic values that includes the truth values T (true) and F (false) and $V^M_{\text{sem}} : L \to D^M_{\text{sem}}$ is a total function such that, for all sentences $A \in \Gamma$, $V^M_{\text{sem}}(A) = \text{T}$. □

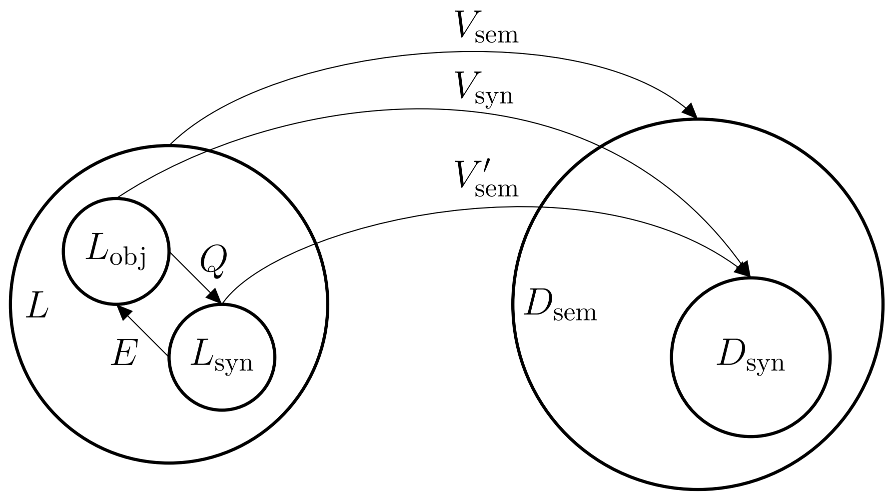

Fig. 1. A Syntax Framework

Definition 6 (Interpreted Theory). An *interpreted theory* is a pair $I = (T, \mathcal{M})$ where T is a theory and $\mathcal{M}$ is a set of models of T. (If $T = (L, \Gamma)$, $(L, D^M_{\rm sem}, V^M_{\rm sem})$ is obviously an interpreted language for each $M \in \mathcal{M}$.) $\Box$

Definition 7 (Syntax Framework in an Interpreted Theory). Let $I = (T, \mathcal{M})$ be an interpreted theory where $T = (L, \Gamma)$ and $L_{\rm obj} \subseteq L$. A *syntax framework* for $(L_{\rm obj}, I)$ is a triple $F = (L_{\rm syn}, Q, E)$ where:

1. $L_{\rm syn} \subseteq L$.
2. $Q : L_{\rm obj} \rightarrow L_{\rm syn}$ is an injective, total function.
3. $E : L_{\rm syn} \rightarrow L_{\rm obj}$ is a (possibly partial) function.
4. For all $M = (D^M_{\rm sem}, V^M_{\rm sem}) \in \mathcal{M}$, $F^M = (D^M_{\rm syn}, V^M_{\rm syn}, L_{\rm syn}, Q, E)$ is a syntax framework for $(L_{\rm obj}, (L, D^M_{\rm sem}, V^M_{\rm sem}))$ where $D^M_{\rm syn}$ is the range of $V^M_{\rm sem}$ restricted to $L_{\rm syn}$ and $V^M_{\rm syn} = V^M_{\rm sem} \circ Q$.

Let $I = (L, D, V)$ be an interpreted language, $L_{\rm obj} \subseteq L$, and $F = (D_{\rm syn}, V_{\rm syn}, L_{\rm syn}, Q, E)$ be a syntax framework for $(L_{\rm obj}, I)$. F has *built-in quotation* if there is an operator (which we will denote as quote) such that, for all $e \in L_{\rm obj}$, $Q(e)$ is the syntactic result of applying the operator to e (which we will denote as $\mathsf{quote}(e)$). F has *built-in evaluation* if there is an operator (which we will denote as eval) such that, for all $e \in L_{\rm syn}$, $E(e)$ is the syntactic result of applying the operator to e (which we will denote as $\mathsf{eval}(e)$) whenever $E(e)$ is defined. There are similar definitions of built-in quotation and evaluation for syntax frameworks in interpreted theories.

A syntax framework F for $(L_{\rm obj}, I)$, where I is either an interpreted language or an interpreted theory, is *replete* if $L_{\rm obj} = L$ and F has both built-in quotation and evaluation. If F is replete, it has the facility to reason about the syntax of all of L within L itself. Examples of a replete syntax framework are rare. The programming language Lisp with a simplified semantics is the best known example of a replete syntax framework [9]. T. Æ. Mogensen's self-interpretation of lambda calculus [14] and the logic Chiron [7], derived from classical NBG set theory, are two other examples of replete syntax frameworks [9].

4 Local Approach

In order to formalize $\mathsf{PolyDiff}$ in T_R we need the ability to reason about the polynomials in $L_{\rm poly}$ as syntactic structures (i.e., as syntax trees). This can be achieved by constructing a syntax framework for $(L_{\rm poly}, I'_R)$ where $I'_R = (T'_R, \mathcal{M}')$ is an interpreted theory such that T'_R is a conservative extension of T_R. Since we seek to reason about just the syntax of $L_{\rm poly}$ instead of a larger language, we call this the *local approach*.

The construction of the syntax framework requires the following steps:

1. Define in T_R an inductive type whose members are the syntax trees of the polynomials in $L_{\rm poly}$. The inductive type should include a new type symbol $\mathbb{S}$ and appropriate constants for constructing and deconstructing expressions of type $\mathbb{S}$. Let $L_{\rm syn}$ be the set of expressions of type $\mathbb{S}$. For example, if $x+3$ is a polynomial in $L_{\rm poly}$, then an expression like $\mathsf{plus}(\mathsf{var}(\mathsf{s}_x), \mathsf{con}(\mathsf{s}_3))$ could be the expression in $L_{\rm syn}$ that denotes the syntax tree of $x+3$. Next add an unspecified "binary" constant $O_{\rm pd}$ of type $\mathbb{S} \rightarrow (\mathbb{S} \rightarrow \mathbb{S})$ to L_R (that is intended to represent $\mathsf{PolyDiff}$). Let $T'_R = (L'_R, \Gamma'_R)$ be the resulting extension of T_R. T'_R is clearly a conservative extension of T_R.
2. In the metatheory of T'_R define an injective, total function $Q : L_{\rm poly} \rightarrow L_{\rm syn}$ such that, for each polynomial $u \in L_{\rm poly}$, $Q(u)$ is an expression e that denotes the syntax tree of u. For example, $Q(x+3)$ could be $\mathsf{plus}(\mathsf{var}(\mathsf{s}_x), \mathsf{con}(\mathsf{s}_3))$.
3. In the metatheory of T'_R define a total mapping $E : L_{\rm syn} \rightarrow L_{\rm poly}$ such that, for each expression $e \in L_{\rm syn}$, $E(e)$ is the polynomial whose syntax tree is denoted by e. For example, $E(\mathsf{plus}(\mathsf{var}(\mathsf{s}_x), \mathsf{con}(\mathsf{s}_3)))$ would be $x+3$.

Let $(L_{\rm poly}, I'_R)$ where $I'_R = (T'_R, \mathcal{M}')$ and $\mathcal{M}'$ is the set of standard models of T'_R in simple type theory (see [8]). It is easy to check that $F = (L_{\rm syn}, Q, E)$ is a syntax framework for $(L_{\rm poly}, I'_R)$. Notice that E is the left inverse of Q and hence the *law of disquotation* holds: For all $u \in L_{\rm poly}$, $E(Q(u)) = u$.

We are now ready to formalize $\mathsf{PolyDiff}$ in T'_R. First, we need to define an operator in T'_R to represent $\mathsf{PolyDiff}$. We will use $O_{\rm pd}$ for this purpose. We write a sentence $\mathsf{CompBehavior}$

$$\lambda\, a, b : \mathbb{S} \mathrel{.} \mathsf{is\text{-}var}(b) \Rightarrow B(a, b, O_{\rm pd}(a)(b))$$

in T'_R where, for all $u \in L_{\rm poly}$ and $x \in L_{\rm var}$,

$$B(Q(u), Q(x), O_{\rm pd}(Q(u))(Q(x)))$$

holds iff

$$\mathsf{PolyDiff}(u, x) = E(O_{\rm pd}(Q(u))(Q(x))).$$

That is, we specify the computational behavior of O_{pd} to be the same as that of $\mathsf{PolyDiff}$.

Second, we need to prove that O_{pd} is mathematically correct. We write the sentence $\mathsf{MathMeaning}$

$$\text{for all } u \in L_{\mathrm{poly}}, \mathsf{deriv}(\lambda\, x : \mathbb{R}\, .\, u) = \lambda\, x : \mathbb{R}\, .\, E(O_{\mathrm{pd}}(Q(u))(Q(x)))$$

in the metatheory of T'_R that says O_{pd} computes a syntactic value that represents an expression that denotes the derivative of $\lambda\, x : \mathbb{R}\, .\, u$ at x. And then we prove in T'_R that $\mathsf{MathMeaning}$ follows from $\mathsf{CompBehavior}$. The proof requires showing that $E(O_{\mathrm{pd}}(Q(u))(Q(x)))$ equals $\mathsf{deriv}((\lambda\, x : \mathbb{R}\, .\, u), x)$, which is

$$\lim_{h \to 0} \frac{(\lambda\, x : \mathbb{R}\, .\, u)(x + h) - (\lambda\, x : \mathbb{R}\, .\, u)(x)}{h}.$$

The details of the proof are found in any good calculus textbook such as [17].

Third, we need to show how $\mathsf{PolyDiff}$ can be used to compute the derivative of a function $\lambda\, x : \mathbb{R}\, .\, u$ at x in T'_R. There are two ways. The first way is to simplify $E(O_{\mathrm{pd}}(Q(u))(Q(x)))$ in $\mathsf{MathMeaning}$ (e.g., by beta-reduction). The second way is to replace $E(O_{\mathrm{pd}}(Q(u))(Q(x)))$ in $\mathsf{MathMeaning}$ with the result of applying $\mathsf{PolyDiff}$ to u and x. The first way requires that $\mathsf{PolyDiff}$ is implemented in T'_R as O_{pd}. The second way does not require that $\mathsf{PolyDiff}$ is implemented in T'_R, but only that its meaning is specified in T'_R.

The local approach is commonly used to reason about the syntax of expressions in a formal theory. It embodies a *deep embedding* [1] of the object language (e.g., L_{poly}) into the underlying formal language (e.g., L_R). The local approach to reason about syntax can be employed in almost any proof assistant in which it is possible to define an inductive type (e.g., see [1, 4, 20]).

The local approach has both strengths and weaknesses. These are the strengths of the local approach:

1. *Indirect Reasoning about the syntax of* L_{poly} *in the Theory.* In T'_R using L_{syn}, we can indirectly reason about the syntax of the polynomials in L_{poly}. This thus enables us to specify the computational behavior of $\mathsf{PolyDiff}$ via O_{pd}.
2. *Direct Reasoning about the syntax of* L_{poly} *in the Metatheory.* In the metatheory of T'_R using L_{syn}, Q, and E, we can directly reason about the syntax of the polynomials in L_{poly}. In particular, using $\mathsf{MathMeaning}$ and the formula

$$\text{for all } u \in L_{\mathrm{poly}}, x \in L_{\mathrm{var}}, \mathsf{PolyDiff}(u, x) = E(O_{\mathrm{pd}}(Q(u))(Q(x))),$$

we can specify the mathematical meaning of $\mathsf{PolyDiff}$.

And these are the weaknesses:

1. *Syntax Problem.* We cannot directly refer in T'_R to the syntax of polynomials. Also the variable x is free in $x+3$ but not in $Q(x+3) = \mathsf{plus}(\mathsf{var}(\mathsf{s}_x), \mathsf{con}(\mathsf{s}_3))$. As a result, Q and E cannot be defined in T'_R and thus $\mathsf{PolyDiff}$ cannot be fully formalized in T'_R. In short, we can reason about syntax in T'_R but not about the interplay of syntax and semantics in T'_R.

2. *Coverage Problem.* The syntax framework F can only be used for reasoning about the syntax of polynomials. It cannot be used for reasoning, for example, about rational expressions. To do that a new syntax framework must be constructed.
3. *Extension Problem.* L_{poly}, L_{syn}, Q, and E must be extended each time a new constant of type $\mathbb{R}$ is defined in T'_R .

In summary, the local approach only gives us indirect access to the syntax of polynomials and must be modified to cover new or enlarged contexts.

If L_{obj} (which is L_{poly} in our example) does not contain variables, then we can define E to be a total operator in the theory. (If the theory is over a traditional logic, we will still not be able to define Q in the theory.) This variant of the local approach is used, for example, in the Agda reflection mechanism [19].

5 Global Approach

The *global approach* described in this section utilizes a replete syntax framework. Assume that we have modified T_R and simple type theory so that there is a replete syntax framework $F = (L_{\mathrm{syn}}, Q, E)$ for (L_R, I_R) where $I_R = (T_R, \mathcal{M})$ and $\mathcal{M}$ is the set of standard models of T_R in the modified simple type theory. Let us also assume that L_{syn} is the set of expressions of type $\mathbb{S}$ and L_R includes a constant O_{pd} of type $\mathbb{S} \rightarrow (\mathbb{S} \rightarrow \mathbb{S})$. By virtue of F being replete, F embodies a deep embedding of L_R into itself.

As far as we know, no one has ever worked out the details of how to modify simple type theory so that it admits built-in quotation and evaluation for the full language of a theory. However, we have shown how NBG set theory can be modified to admit built-in quotation and evaluation for its entire language [7]. Simple type theory can be modified in a similar way. We plan to present a version of simple type theory with a replete syntax framework in a future paper.

We can formalize $\mathsf{PolyDiff}$ in T_R as follows. We will write $\mathsf{quote}(e)$ and $\mathsf{eval}(e)$ as $\ulcorner e \urcorner$ and $[\![e]\!]$, respectively. First, we define the operator O_{pd} in T_R to represent $\mathsf{PolyDiff}$. We write a sentence $\mathsf{CompBehavior}$

$$\lambda\, a, b : \mathbb{S} \,.\, \mathsf{is\text{-}poly}(a) \wedge \mathsf{is\text{-}var}(b) \Rightarrow B(a, b, O_{\mathrm{pd}}(a)(b))$$

in T_R where, for all $u \in L_{\mathrm{poly}}$ and $x \in L_{\mathrm{var}}$,

$$B(\ulcorner u \urcorner, \ulcorner x \urcorner, O_{\mathrm{pd}}(\ulcorner u \urcorner)(\ulcorner x \urcorner))$$

holds iff

$$\mathsf{PolyDiff}(u, x) = [\![O_{\mathrm{pd}}(\ulcorner u \urcorner)(\ulcorner x \urcorner)]\!].$$

That is, we specify the computational behavior of O_{pd} to be the same as that of $\mathsf{PolyDiff}$.

Second, we prove in T_R that O_{pd} is mathematically correct. We write the sentence $\mathsf{MathMeaning}$

$$\forall\, a : \mathbb{S} \,.\, \mathsf{is\text{-}poly}(a) \Rightarrow \mathsf{deriv}(\lambda\, x : \mathbb{R} \,.\, [\![a]\!]) = \lambda\, x : \mathbb{R} \,.\, [\![O_{\mathrm{pd}}(a)(\ulcorner x \urcorner)]\!]$$

in T_R that says O_{pd} computes a syntactic value that represents an expression that denotes the derivative of $\lambda\, x : \mathbb{R}\, .\, [\![a]\!]$ at x. And then we prove in T_R that $\mathsf{MathMeaning}$ follows from $\mathsf{CompBehavior}$.

Third, we use $\mathsf{PolyDiff}$ to compute the derivative of a function $\lambda\, x : \mathbb{R}\, .\, u$ at x in T_R in either of the two ways described for the local approach.

The strengths of the global approach are:

1. *Direct Reasoning about the syntax of polynomials in the Theory.* In T_R using L_{syn}, quote, and eval, we can directly reason about the syntax of the expressions in L_{poly}. As a result, we can formalize $\mathsf{PolyDiff}$ in T_R as described in the Introduction.
2. *Direct Reasoning about the syntax of all expressions in the Theory.* In T_R using L_{syn}, quote, and eval, we can directly reason about the syntax of the expressions in the entire language L_R. As a result, the syntax framework F can cover all current and future syntax reasoning needs. Moreover, we can express such things as syntactic side conditions, formula schemas, and substitution for a variable directly in T_R (see [7] for details).

In short, not only does the global approach enable us to formalize $\mathsf{PolyDiff}$ in T_R, it provides us with the facility to move syntax-based reasoning from the metatheory of T_R to T_R itself. This seems to be a wonderful result that solves the problem of formalizing syntax-based mathematical algorithms. Unfortunately, the global approach has the following serious weaknesses that temper the enthusiasm one might have for its strengths:

1. *Evaluation Problem.*

   Claim: eval cannot be defined on all expressions in L_R.

   Proof: Suppose eval is indeed total. T_R is sufficiently expressive, in the sense of Gödel's incomplete theorem, so apply the diagonalization lemma [3] to obtain a formula LIAR such that

   $$\mathsf{LIAR} = \ulcorner \neg [\![\mathsf{LIAR}]\!] \urcorner.$$

   Then

   $$[\![\mathsf{LIAR}]\!] = [\![\ulcorner \neg [\![\mathsf{LIAR}]\!] \urcorner]\!] = \neg [\![\mathsf{LIAR}]\!],$$

   which is a contradiction. □

   This means that the liar paradox limits the use of eval and, in particular, the law of disquotation does not hold universally, i.e., there are expressions e in L_R such that $[\![\ulcorner e \urcorner]\!] \neq e$.
2. *Variable Problem.* The variable x is not free in the expression $\ulcorner x + 3 \urcorner$ (or in any quotation). However, x is free in $[\![\ulcorner x + 3 \urcorner]\!]$ because $[\![\ulcorner x + 3 \urcorner]\!] = x + 3$. If the value of the variable e is $\ulcorner x + 3 \urcorner$, then both e and x are free in $[\![e]\!]$ because $[\![e]\!] = [\![\ulcorner x + 3 \urcorner]\!] = x + 3$.

   This example shows that the notions of a free variable, substitution for a variable, etc. are significantly more complex when expressions contain eval.

3. *Extension Problem.* We can define $L_{\mathrm{con}} \subseteq L_{\mathrm{syn}}$ in T_R as the language of expressions denoting the syntactic values of constants in L_R.

   Claim: Assume the set of constants in L is finite and $T'_R = (L'_R, \Gamma'_R)$ is an extension of T_R such that there is a constant in L'_R but not in L_R. Then T'_R is not a conservative extension of T_R.

   Proof: Let $\{c_1, \ldots, c_n\}$ be the set of constants in L. Then

   $$L_{\mathrm{con}} = \{\ulcorner c_1 \urcorner, \ldots, \ulcorner c_n \urcorner\}$$

   is valid in T_R but not in T'_R. □

   This shows that in the global approach the development of a theory via definitions requires that the notion of a conservative extension be weakened.
4. *Interpretation Problem.*

   Let $T = (L, \Gamma)$ and $T' = (L', \Gamma')$ in be two theories in a simple type theory that has been modified to admit built-in quotation and evaluation for the entire language of a theory.

   Claim: Let Φ be an interpretation of T in T' such that Φ is a homomorphism with respect to the logical operators of the underlying logic. Then Φ must be injective on the constants of L.

   Proof: Assume that Φ is not injective on constants. Then there are two different constants a, b such that $\Phi(a) = \Phi(b)$. $\ulcorner a \urcorner \neq \ulcorner b \urcorner$ is valid in T. Hence

   $$\Phi(\ulcorner a \urcorner \neq \ulcorner b \urcorner) = (\ulcorner \Phi(a) \urcorner \neq \ulcorner \Phi(b) \urcorner)$$

   since Φ is a homomorphism, and the latter inequality must be valid in T' since Φ is an interpretation (which maps valid formulas of T to valid formulas of T'). However, our hypothesis $\Phi(a) = \Phi(b)$ implies $\ulcorner \Phi(a) \urcorner = \ulcorner \Phi(b) \urcorner$, which is a contradiction. □

   This shows that the use of interpretations is more cumbersome in a logic that admits quotation than one that does not.

6 Conclusion

Syntax-based mathematical algorithms are employed throughout mathematics and are one of the main offerings of computer algebra systems. They are difficult, however, to formalize since they manipulate the syntactic structure of expressions in mathematically meaningful ways. We have presented two approaches to formalizing syntax-based mathematical algorithms in a formal theory, one called the *local approach* and the other the *global approach*. Both are based on the notion of a *syntax framework* which provides a foundation for integrating reasoning about the syntax of expressions with reasoning about what the expressions mean. Syntax frameworks include a syntax representation, a syntax language for reasoning about the representation, and quotation and evaluation mechanisms. Common syntax reasoning systems are instances of a syntax framework.

The local approach and close variants are commonly used for formalizing syntax-based mathematical algorithms. Its major strength is that it provides the means to formally reason about the syntactic structure of expressions, while its major weakness is that the mathematical meaning of a syntax-based mathematical algorithm cannot be expressed in the formal theory. Another weakness is that an application of the local approach cannot be easily extended to cover new or enlarged contexts.

The global approach enables one to reason in a formal theory T directly about the syntactic structure of the expressions in T as well as about the interplay of syntax and semantics in T. As a result, it is possible to fully formalize syntax-based algorithms like PolyDiff and move syntax-based reasoning, like the use of syntactic side conditions, from the metatheory of T to T itself. Unfortunately, these highly desirable results come with a high cost: Significant change must be made to the underlying logic as illustrated by the Evaluation, Variable, Extension, and Interpretation Problems given in the previous section.

One of the main goals of the MathScheme project [2], led by J. Carette and the author, is to see if the global approach can be used as a basis to integrate axiomatic and algorithmic mathematics. The logic Chiron [7] demonstrates that it is possible to modify a traditional logic to support the global approach. Although we have begun an implementation of Chiron, it remains an open question whether a logic modified in this way can be effectively implemented. As part of the MathScheme project, we are now pursuing this problem as well as developing the techniques needed to employ the global approach.

Acknowledgments.

The author would like to thank Jacques Carette and Pouya Larjani for many fruitful discussions on ideas related to this paper. The author is also grateful to the referees for their comments and careful review of the paper.

References

1. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*, pages 129–156. North-Holland, 1993.
2. J. Carette and W. M. Farmer. Mathscheme: Project description. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–288. Springer-Verlag, 2011.
3. R. Carnap. *Die Logische Syntax der Sprache*. Springer-Verlag, 1934.
4. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Frontiers of Combining Systems*, volume 4720 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2007.

5. F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: A short comparative study. In *IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.
6. W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer-Verlag, 2007.
7. W. M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University, 2007. Revised 2012. Available at `http://imps.mcmaster.ca/doc/chiron-tr.pdf`.
8. W. M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6:267–286, 2008.
9. W. M. Farmer and P. Larjani. Frameworks for reasoning about syntax that utilize quotation and evaluation. McSCert Report No. 9, McMaster University, 2013. Available at `http://imps.mcmaster.ca/doc/syntax.pdf`.
10. The F# Software Foundation.
11. J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16, 2006.
12. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995. Available at `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.ps.gz`.
13. P. Koellner. On reflection principles. *Annals of Pure and Applied Logic*, 157:206–219, 2009.
14. T. Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2:345–364, 1994.
15. Rice University Programming Languages Team. Metaocaml: A compiled, type-safe, multi-stage programming language. `http://www.metaocaml.org/`, 2011.
16. T. Sheard and S. P. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37:60–75, 2002.
17. M. Spivak. *Calculus*. Publish or Perish, fourth edition, 2008.
18. W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248:211–242, 2000.
19. P. van der Walt. Reflection in Agda. Master's thesis, Universiteit Utrecht, 2012.
20. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *TPHOLs*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer-Verlag, 2004.