Towards Specifying Symbolic Computation

Jacques Carette and William M. Farmer
Computing and Software, McMaster University, Canada
http://www.cas.mcmaster.ca/~carette
http://imps.mcmaster.ca/wmfarmer
4 April 2019

Abstract. Many interesting and useful symbolic computation algorithms manipulate mathematical expressions in mathematically meaningful ways. Although these algorithms are commonplace in computer algebra systems, they can be surprisingly difficult to specify in a formal logic since they involve an interplay of syntax and semantics. In this paper we discuss several examples of syntax-based mathematical algorithms, and we show how to specify them in a formal logic with undefinedness, quotation, and evaluation.

1 Introduction

Many mathematical tasks are performed by executing an algorithm that manipulates mathematical expressions in a mathematically meaningful way. For instance, children learn to do addition, subtraction, multiplication, and division by executing algorithms that manipulate strings of digits that represent numbers. A syntax-based mathematical algorithm (SBMA) is an algorithm of this kind that performs a mathematical task by manipulating the syntactic structure of certain mathematical expressions. SBMAs are commonplace in mathematics, and so it is no surprise that they are standard components of computer algebra systems.

SBMAs involve an interplay of syntax and semantics. The computational behavior of an SBMA is the relationship between its input and output expressions, while the mathematical meaning of an SBMA is the relationship between the mathematical meanings of its input and output expressions. Understanding what a SBMA does requires understanding how its computational behavior affects its mathematical meaning.

A complete specification of an SBMA is often much more complex than one might expect. This is because (1) manipulating syntax is complex in itself, (2) the interplay of syntax and semantics can be difficult to disentangle, and (3) seemingly benign syntactic manipulations can generate undefined expressions. An SBMA specification has both a syntactic component and a semantic component, but these components can be intertwined. Usually the more they are separated, the easier it is to understand the specification.

* This research is supported by NSERC.
As a result of the inherent complexity of SBMA specifications, SBMAs can be tricky to implement in mathematical software systems. Dealing with the semantic component is usually the bigger challenge for computer algebra systems, while the syntactic component is usually the bigger obstacle for proof assistants.

In this paper, we examine four representative examples of SBMAs, present their specifications, and show how their specifications can be written in \textit{ctt} \textit{qe} \cite{8}, a formal logic that is well suited for expressing the interplay of syntax and semantics. The paper is organized as follows. Section 2 presents background information about semantic notions and \textit{ctt} \textit{uqe}. Section 3 discusses the issues concerning SBMAs for factoring integers. Normalizing rational expressions and functions is examined in section 4. Symbolic differentiation algorithms are considered in section 5. Section 6 give as brief overview of related work. And the paper ends with a short conclusion in section 7.

2 Background

2.1 Definedness, Equality, and Quasi-Equality

Let \( e \) be a mathematical expression and \( D \) be a domain of mathematical values. We say \( e \) is defined in \( D \) if \( e \) denotes a member of \( D \). When \( e \) is defined in \( D \), the value of \( e \) in \( D \) is the element in \( D \) that \( e \) denotes. When \( e \) is undefined in \( D \) (i.e., \( e \) does not denote a member of \( D \)), the value of \( e \) in \( D \) is undefined. Two expressions \( e \) and \( e' \) are equal in \( D \), written \( e =_D e' \), if they are both defined in \( D \) and they have the same values in \( D \) and are quasi-equal in \( D \), written \( e \simeq_D e' \), if either \( e =_D e' \) or \( e \) and \( e' \) are both undefined in \( D \).

2.2 \textit{CTT} \textit{qe} and \textit{CTT} \textit{uqe}

\textit{CTT} \textit{qe} \cite{8} is a version of Church’s type theory with a built-in \textit{global reflection infrastructure} with global quotation and evaluation operators that is well-suited for reasoning about the interplay of syntax and semantics and, in particular, for specifying, defining, applying, and reasoning about SBMAs. The syntax and semantics of \textit{CTT} \textit{qe} is presented in \cite{8}. A proof system for \textit{CTT} \textit{qe} that is sound for all formulas and complete for eval-free formulas is also presented in \cite{8}. (An expression is \textit{eval-free} if it does not contain the evaluation operator.) By modifying HOL Light \cite{9}, we have produced a rudimentary implementation of \textit{CTT} \textit{qe} called HOL Light QE \cite{2}.

\textit{CTT} \textit{uqe} \cite{7} is a variant of \textit{CTT} \textit{qe} that has built-in support for partial functions and undefinedness based on the traditional approach to undefinedness \cite{5}. It is well-suited for specifying SBMAs that manipulate expressions that may be undefined. Its syntax and semantics are presented in \cite{7}. A proof system for \textit{CTT} \textit{uqe} is not given in \cite{7}, but a proof system can be straightforwardly derived by merging the proof systems for \textit{CTT} \textit{qe} \cite{8} and \textit{Q} \textit{u} \textit{0} \cite{6}.

The global reflection infrastructure of \textit{CTT} \textit{uqe} (and \textit{CTT} \textit{qe}) consists of three components. The first component is an inductive type \( \epsilon \) of \textit{syntactic values}. A
syntactic value typically represents the syntax tree of an eval-free expression of CTT_uqe. Each expression of type \( \epsilon \) denotes a syntactic value. Thus reasoning about the syntactic structure of expressions can be performed by reasoning about syntactic values via the expressions of type \( \epsilon \). The second component is a quotation operator \( \lceil \cdot \rceil \) such that, if \( A_\alpha \) is an eval-free expression (of some type \( \alpha \)), then \( \lceil A_\alpha \rceil \) is an expression of type \( \epsilon \) that denotes the syntactic value that represents the syntax tree of \( A_\alpha \). And the third component is an evaluation operator \( \llbracket \cdot \rrbracket_\alpha \) such that, if \( E_\epsilon \) is an expression of type \( \epsilon \), then \( \llbracket E_\epsilon \rrbracket_\alpha \) denotes the value of type \( \alpha \) denoted by the expression \( B \) represented by \( E_\epsilon \) (provided the type of \( B \) is \( \alpha \)). In particular the law of disquotation \( \llbracket \lceil A_\alpha \rceil \rrbracket_\alpha = A_\alpha \) holds in CTT_uqe (and CTT_qe).

The reflection infrastructure is global since it can be used to reason about the entire set of eval-free expressions of CTT_uqe. This is in contrast to local reflection infrastructures that are constructed by defining an inductive type of syntactic values only for the expressions of the logic that are relevant to a particular problem. See [8] for discussion about the difference between local and global reflection infrastructures and the design challenges that stand in the way of developing a global reflection infrastructure within a logic.

The type \( \epsilon \) includes syntax values for all eval-free expressions of all types as well as syntax values for ill-formed expressions like \((x_\alpha x_\alpha)\) in which the types are mismatched. Convenient subtypes of \( \epsilon \) can be represented via predicates of type \( \epsilon \to o \). (\( o \) is the type of boolean values.) In particular, CTT_uqe contains a predicate \( \text{is-exp}_\alpha^{\epsilon \to o} \) for every type \( \alpha \) that represents the subtype of syntax values for expressions of type \( \alpha \).

Unlike CTT_qe, CTT_uqe admits undefined expressions and partial functions.

3 Factoring Integers

3.1 Task

Here is a seemingly simple mathematical task: Factor (over \( \mathbb{N} \)) the number 12. One might expect the answer 12 = \( 2^2 \times 3 \) — but this is not actually the answer one gets in many systems! This is because, in any system with built-in beta-reduction (including all computer algebra systems as well as theorem provers based on dependent type theory), the answer is simplified to 12 = 12, which is certainly not very informative.

3.2 Problem

So why is \( 2^2 \times 3 \) not an answer? Because it involves a mixture of syntax and semantics. A better answer would be \( \lceil 2^2 \times 3 \rceil \) (the quotation of \( 2^2 \times 3 \)) that
would then make it clear that \( \ast \) represents multiplication rather than \( \text{being} \) multiplication. In other words, this is about intension and extension: we want to be able to both represent operations and perform operations. In Maple, one talks about \textit{inert forms}, while in Mathematica, there are various related concepts such as \texttt{Hold}, \texttt{Inactive} and \texttt{Unevaluated}. They both capture the same fundamental dichotomy about passive representations and active computations.

### 3.3 Solution

Coming back to integer factorization, interestingly both Maple and Mathematica choose a fairly similar option to represent the answer — a list of pairs, with the first component being a prime of the factorization and the second being the multiplicity of the prime (i.e., the exponent). Maple furthermore gives a leading unit (-1 or 1), so that one can also factor negative numbers. In other words, in Maple, the result of \( \texttt{ifactors}(12) \) is

\[
[1, [2, 2], [3, 1]]
\]

where lists are used (rather than proper pairs) as the host system is untyped. Mathematica does something similar.

### 3.4 Specification in Maple

Given the following Maple routine\(^1\)

```maple
remult := proc(l :: [{-1,1}, list([prime,posint])])
    local f := proc(x, y) (x[1] ^ x[2]) * y end proc;
    l[1] * foldr(f, 1, op(l[2]))
end proc;
```

then the specification for \( \texttt{ifactors} \) is that, for all \( n \in \mathbb{Z} \), (A) \( \texttt{ifactors}(n) \) represents a signed prime decomposition and

(B) \( \texttt{remult} (\texttt{ifactors}(n)) = n. \)

(A) is the syntactic component of the specification and (B) is the semantic component.

### 3.5 Specification in CTT\textsubscript{uqe}

We specify the factorization of integers in a theory \( T \) of CTT\textsubscript{uqe} using CTT\textsubscript{uqe}'s reflection infrastructure. We start by defining a theory \( T_0 = (L_0, \Gamma_0) \) of integer arithmetic. \( L_0 \) contains a base type \( i \) and the constants \( 0_i, 1_i, 2_i, \ldots, -i \), \( +_{i \rightarrow i}, -_{i \rightarrow i}, \) and \( ^{i \rightarrow i} \). \( \Gamma_0 \) contains the usual axioms of integer arithmetic.

Next we extend \( T_0 \) to a theory \( T_1 = (L_1, \Gamma_1) \) by defining the following two constants using the machinery of \( T_0 \):

\(\text{There are nonessential Maple-isms in this routine: because of how} \ \texttt{foldr} \text{ is defined,} \ \texttt{op} \text{ is needed to transform a list to an expression sequence; in other languages, this is unnecessary. Note however that it is possible to express the type extremely precisely.}\)
1. **Numeral**$_{\epsilon \rightarrow o}$ is a predicate representing the subtype of $\epsilon$ that denotes the subset $\{0_i, 1_i, 2_i, \ldots\}$ of expressions of type $i$. Thus, **Numeral**$_{\epsilon \rightarrow o}$ is the subtype of numerals and, for example, **Numeral**$_{\epsilon \rightarrow o}$ $\{2_i\}$ is valid in $T_1$.

2. **PrimeDecomp**$_{\epsilon \rightarrow o}$ is a predicate representing the subtype of $\epsilon$ that denotes the subset of expressions of type $i$ of the form $0_i$ or $\pm 1 \ast p_0^{e_0} \ast \cdots \ast p_k^{e_k}$

where parentheses and types have been dropped, the $p_i$ are numerals denoting unique prime numbers in increasing order, the $e_i$ are also numerals, and $k \geq 0$. Thus **PrimeDecomp**$_{\epsilon \rightarrow o}$ is a subtype of signed prime decompositions and, for example, **PrimeDecomp**$_{\epsilon \rightarrow o}$ $\{1 \ast 2^2 \ast 3^1\}$ (where again parentheses and types have been dropped) is valid in $T_2$.

Finally, we can extend $T_1$ to a theory $T = (L, \Gamma)$ in which $L$ contains the constant **factor**$_{\epsilon \rightarrow \epsilon}$ and $\Gamma$ contains the following axiom specFactor$_o$:

$$\forall u_e .$$

if (Numeral$_{\epsilon \rightarrow o}$ $u_e$)

(PrimeDecomp$_{\epsilon \rightarrow o}$ (factor$_{\epsilon \rightarrow \epsilon}$ $u_e$) $\land \|u_e\|_i = \|\text{factor}_{\epsilon \rightarrow \epsilon} u_e\|_i$)

(factor$_{\epsilon \rightarrow \epsilon}$ $u_e$)$\uparrow$

specFactor$_o$ says that **factor**$_{\epsilon \rightarrow \epsilon}$ is only defined on numerals and, when $u_e$ is a numeral, **factor**$_{\epsilon \rightarrow \epsilon}$ $u_e$ is a signed prime decomposition (the syntactic component) and denotes the same integer as $u_e$ (the semantic component). Notice that specFactor$_o$ does not look terribly complex on the surface, but there is a significant amount of complexity embodied in the definitions of **Numeral**$_{\epsilon \rightarrow o}$ and **PrimeDecomp**$_{\epsilon \rightarrow o}$.

### 3.6 Discussion

Why do neither of Maple or Mathematica use their own means of representing intensional information? History! In both cases, the integer factorization routines predates the intensional features by more than two decades. And backward compatibility definitely prevents them from making that change.

Furthermore, factoring as an operation produces output in a very predictable shape: $s \ast p_0^{e_0} \ast p_1^{e_1} \ast \cdots \ast p_k^{e_k}$. To parse such a term’s syntax to extract the information is tedious and error prone, at least in an untyped system. Such a shape could easily be coded up in a typed system using a very simple algebraic data type that would obviate the problem. But computer algebra systems are very good at manipulating lists and thus this output composes well with other system features.

---

$^2$ This is unsurprising given that the builders of both Maple and Mathematica were well acquainted with Macsyma which was implemented in Lisp.
Nevertheless, the main lesson is that a simple mathematical task, such as
factoring the number 12, which seems like a question about simple integer arith-
metic, is not. It is a question that can only be properly answered in a context
with a significantly richer term language that includes either lists or pairs, or
an inductive type of syntactic values, or access to the expressions of the term
language as syntactic objects.

All the issues we have seen with the factorization of integers appear again
with the factorization of polynomials.

4 Normalizing Rational Expressions and Functions

Let $\mathbb{Q}$ be the field of rational numbers, $\mathbb{Q}[x]$ be the ring of polynomials in $x$
 over $\mathbb{Q}$, and $\mathbb{Q}(x)$ be the field of fractions of $\mathbb{Q}[x]$. We may assume that $\mathbb{Q} \subseteq
\mathbb{Q}[x] \subseteq \mathbb{Q}(x)$.

The language $\mathcal{L}_{re}$ of $\mathbb{Q}(x)$ is the set of expressions built from the symbols
$x, 0, 1, +, *, -, -1$, elements of $\mathbb{Q}$, and parentheses (as necessary). For greater
readability, we will take the liberty of using fractional notation for $-1$ and the
exponential notation $x^n$ for $x * \cdots * x$ ($n$ times). A member of $\mathcal{L}_{re}$ can be something
simple like $x^4 - 1$ or something more complicated like

$$\frac{\frac{1-x}{3/2x^3+x+1}}{\frac{1}{9854+2x+3x^2+4x^3-1/5}} + 3 \times x - \frac{12}{x}.$$

The members of $\mathcal{L}_{re}$ are called rational expressions (in $x$ over $\mathbb{Q}$). They denote
elements in $\mathbb{Q}(x)$. Of course, a rational expression like $x/0$ is undefined in $\mathbb{Q}(x)$.

Let $\mathcal{L}_{rf}$ be the set of expressions of the form $(\lambda x : Q, r)$ where $r \in \mathcal{L}_{re}$. The
members of $\mathcal{L}_{rf}$ are called rational functions (in $x$ over $\mathbb{Q}$). That is, a rational
function is a lambda expression whose body is a rational expression. Rational
functions denote functions from $\mathbb{Q}$ to $\mathbb{Q}$. Though rational expressions and rational functions look similar, they have very different meanings due to the role
of $x$. The $x$ in a rational expression is an indeterminant that does not denote a
value, while the $x$ in a rational function is a variable ranging over values in $\mathbb{Q}$.

4.1 Task 1: Normalizing Rational Expressions

Normalizing a rational expression is a useful task. We are taught that, like for
members of $\mathbb{Q}$ (such as $5/15$), there is a normal form for rational expressions.
This is typically defined to be a rational expression $p/q$ for two polynomials
$p, q \in \mathbb{Q}[x]$ such that $p$ and $q$ are themselves in polynomial normal form and
$\text{gcd}(p, q) = 1$. The motivation for the latter property is that we usually want to
write the rational expression $\frac{x^4-1}{x^2-1}$ as $x^2 + 1$ just as we usually want to write $5/15$
as $1/3$. Thus, the normal forms of $\frac{x^4-1}{x^2-1}$ and $\frac{x}{x}$ are $x^2 + 1$ and 1, respectively.
This definition of normal form is based on the characteristic that the elements
of the field of fractions of a integral domain $D$ can be written as quotients $r/s$
of elements of $D$ where $r_0/s_0 = r_1/s_1$ if and only if $r_0 * s_1 = r_1 * s_0$ in $D$. 
We would like to normalize a rational expression by putting it into normal form. Let \( \text{normRatExpr} \) be the SBMA that takes \( r \in \mathcal{L}_{re} \) as input and returns the \( r' \in \mathcal{L}_{re} \) as output such that \( r' \) is the normal form of \( r \). How should \( \text{normRatExpr} \) be specified?

4.2 Problem 1

\( \text{normRatExpr} \) must normalize rational expressions as expressions that denote members of \( \mathbb{Q}(x) \), not members of \( \mathbb{Q} \). Hence \( \text{normRatExpr}(x/x) \) and \( \text{normRatExpr}(1/x - 1/x) \) should be 1 and 0, respectively, even though \( x/x \) and \( 1/x - 1/x \) are undefined when the value of \( x \) is 0.

4.3 Solution 1

The hard part of specifying \( \text{normRatExpr} \) is defining exactly what rational expressions are normal forms and then proving that two normal forms denote the same member of \( \mathbb{Q}(x) \) only if the two normal forms are identical. Assuming we have adequately defined the notion of a normal form, the specification of \( \text{normRatExpr} \) is that, for all \( r \in \mathcal{L}_{re} \), (A) \( \text{normRatExpr}(r) \) is a normal form and (B) \( r \simeq_{\mathbb{Q}(x)} \text{normRatExpr}(r) \). (A) is the syntactic component of the specification, and (B) is the semantic component. Notice that (B) implies that, if \( r \) is undefined in \( \mathbb{Q}(x) \), then \( \text{normRatExpr}(r) \) is also undefined in \( \mathbb{Q}(x) \). For example, since \( r = \frac{1}{x-2} \) is undefined in \( \mathbb{Q}(x) \), \( \text{normRatExpr}(r) \) should be the (unique) undefined normal form (which, for example, could be the rational expression \( 1/0 \)).

4.4 Task 2: Normalizing Rational Functions

Normalizing a rational function is another useful task. Let \( f = (\lambda x : \mathbb{Q} . r) \) be a rational function. We would like to normalize \( f \) by putting its body \( r \) in normal form of some appropriate kind. Let \( \text{normRatFun} \) be the SBMA that takes \( f \in \mathcal{L}_{rf} \) as input and returns a \( f' \in \mathcal{L}_{rf} \) as output such that \( f' \) is the normal form of \( f \). How should \( \text{normRatFun} \) be specified?

4.5 Problem 2

If \( f_i = (\lambda x : \mathbb{Q} . r_i) \) are rational functions for \( i = 1, 2 \), one might think that \( f_1 =_{\mathbb{Q} \rightarrow \mathbb{Q}} f_2 \) if \( r_1 =_{\mathbb{Q}(x)} r_2 \). But this is not the case. For example, the rational functions \( (\lambda x : \mathbb{Q} . x/x) \) and \( (\lambda x : \mathbb{Q} . 1) \) are not equal as functions over \( \mathbb{Q} \) since \( (\lambda x : \mathbb{Q} . x/x) \) is undefined at 0 while \( (\lambda x : \mathbb{Q} . 1) \) is defined everywhere. But \( x/x =_{\mathbb{Q}(x)} 1! \) Similarly, \( (\lambda x : \mathbb{Q} . (1/x - 1/x)) \neq_{\mathbb{Q} \rightarrow \mathbb{Q}} (\lambda x : \mathbb{Q} . 0) \) and \( (1/x - 1/x) =_{\mathbb{Q}(x)} 0 \). (Note that, in some contexts, we might want to say that \( (\lambda x : \mathbb{Q} . x/x) \) and \( (\lambda x : \mathbb{Q} . 1) \) do indeed denote the same function by invoking the concept of removable singularities.)
4.6 Solution 2

As we have just seen, we cannot normalize a rational function by normalizing its body, but we can normalize rational functions if we are careful not to remove points of undefinedness. Let a quasinormal form be a rational expression \( p/q \) for two polynomials \( p, q \in \mathbb{Q}[x] \) such that \( p \) and \( q \) are themselves in polynomial normal form and there is no irreducible polynomial \( s \in \mathbb{Q}[x] \) of degree \( \geq 2 \) that divides both \( p \) and \( q \). One should note that this definition of quasinormal form depends on the field \( \mathbb{Q} \) because, for example, the polynomial \( x^2 - 2 \) is irreducible in \( \mathbb{Q} \) but not in \( \mathbb{R} \) (since \( x^2 - 2 = \sqrt{2}(x - \sqrt{2})(x + \sqrt{2}) \)).

We can then normalize a rational function by quasinormalizing its body. So the specification of \( \text{normRatFun} \) is that, for all \((\lambda x : \mathbb{Q} . r) \in \mathcal{L}_t\), \( (A) \ \text{normRatFun}(\lambda x : \mathbb{Q} . r) = (\lambda x : \mathbb{Q} . r') \) where \( r' \) is a quasinormal form and \( (B) \ (\lambda x : \mathbb{Q} . r) \cong_{\mathbb{Q}} \text{normRatFun}(\lambda x : \mathbb{Q} . r) \). \( (A) \) is the syntactic component of its specification, and \( (B) \) is the semantic component.

4.7 Specification in \( \text{CTT}_{\text{uqe}} \)

We specify \( \text{normRatExpr} \) and \( \text{normRatFun} \) in a theory of \( \text{CTT}_{\text{uqe}} \) using again \( \text{CTT}_{\text{uqe}} \)'s reflection infrastructure. A complete development of \( T \) would be long and tedious, so we will only sketch the development of \( T \).

The first step is to define a theory \( T_0 = (L_0, \Gamma_0) \) that axiomatizes \( \mathbb{Q} \), the field of rational numbers. \( L_0 \) contains a base type \( q \) and constants \( 0_q, 1_q, +_{q\to q\to q}, \times_{q\to q\to q}, -_{q\to q}, \) and \( -1_{q\to q} \) representing the standard elements and operators of a field. \( \Gamma_0 \) contains axioms that say the type \( q \) is the field of rational numbers.

The second step is to extend \( T_0 \) to a theory \( T_1 = (L_1, \Gamma_1) \) that axiomatizes \( \mathbb{Q}(x) \), the field of fractions of the ring \( \mathbb{Q}[x] \). \( L_1 \) contains a base type \( f \); constants \( 0_f, 1_f, +_{f\to f\to f}, \times_{f\to f\to f}, -_{f\to f}, \) and \( -1_{f\to f} \) representing the standard elements and operators of a field; and a constant \( X_f \) representing the indeterminant of \( \mathbb{Q}(x) \). \( \Gamma_1 \) contains axioms that say the type \( f \) is the field of fractions of \( \mathbb{Q}[x] \). Notice that the types \( q \) and \( f \) are completely separate from each other since \( \text{CTT}_{\text{uqe}} \) does not admit subtypes as in \( \text{CTT}_0 \).

The third step is to extend \( T_1 \) to a theory \( T_2 = (L_2, \Gamma_2) \) that is equipped to express ideas about the expressions of type \( q \) and \( q \to q \) that have the form of rational expressions and rational functions, respectively. \( T_2 \) is obtained by defining the following constants using the machinery of \( T_1 \):

1. \( \text{RatExpr}_{\to o} \) is the predicate representing the subtype of \( e \) that denotes the set of expressions of type \( q \) that have the form of rational expressions in \( x_q \) (i.e., the expressions of type \( q \) built from the variable \( x_q \) and the constants representing the field elements and operators for \( q \)). So, for example, \( \text{RatExpr}_{\to o} \ x_q/x_q \) is valid in \( T_2 \).

2. \( \text{RatFun}_{\to o} \) is the predicate representing the subtype of \( e \) that denotes the set of expressions of type \( q \to q \) that are rational functions in \( x_q \) (i.e., the expressions of the form \( (\lambda x_q . R_q) \) where \( R_q \) is an expression having the form of a rational expression in \( x_q \)). So, for example, \( \text{RatFun}_{\to o} \ (\lambda x_q . x_q/x_q) \) is valid in \( T_2 \).
3. val-in-\( f_{\epsilon \rightarrow f} \) is a partial function that maps each member of the subtype \( \text{RatExpr}_{\epsilon \rightarrow o} \) to its denotation in \( f \). So, for example,

\[
\text{val-in-} f_{\epsilon \rightarrow f} \quad x_q + q \rightarrow q \rightarrow q \quad 1_q \gamma = X_f + f \rightarrow f \rightarrow f \quad 1_f
\]

and (\( \text{val-in-} f_{\epsilon \rightarrow f} \rightarrow 1_q/0_q \gamma \)) \( \uparrow \) are valid in \( T_2 \). Notice that the function is partial on the subtype \( \text{RatExpr}_{\epsilon \rightarrow o} \) since an expression like \( 1_q/0_q \) does not denote a member of \( f \).

4. \( \text{Norm}_{\epsilon \rightarrow o} \) is the predicate representing the subtype of \( \epsilon \) that denotes the subset of the subtype \( \text{RatExpr}_{\epsilon \rightarrow o} \) whose members are normal forms. So, for example, \( \neg (\text{Norm}_{\epsilon \rightarrow o} \rightarrow x_q/x_q \gamma) \) and \( \text{Norm}_{\epsilon \rightarrow o} \rightarrow 1_q \gamma \) are valid in \( T_2 \).

5. \( \text{Quasinorm}_{\epsilon \rightarrow o} \) is the predicate representing the subtype of \( \epsilon \) that denotes the subset of the subtype \( \text{RatExpr}_{\epsilon \rightarrow o} \) whose members are quasinormal forms. So, for example, \( \text{Quasinorm}_{\epsilon \rightarrow o} \rightarrow x_q/x_q \gamma \) and \( \neg (\text{Quasinorm}_{\epsilon \rightarrow o} \rightarrow A_q/A_q \gamma) \), where \( A_q = x_q^2 + q \rightarrow q \rightarrow q \). \( 1_q \gamma \) are valid in \( T_2 \).

6. \( \text{body}_{\epsilon \rightarrow \epsilon} \) is a partial function that maps each member of \( \epsilon \) denoting an expression of the form \( (\lambda x_\epsilon . B_\beta) \) to the member of \( \epsilon \) that denotes \( B_\beta \) and is undefined on the rest of \( \epsilon \).

The final step is to extend \( T_2 \) to a theory \( T = (L, \Gamma) \) in which \( L \) has two additional constants \( \text{normRatExpr}_{\epsilon \rightarrow \epsilon} \) and \( \text{normFun}_{\epsilon \rightarrow \epsilon} \) and \( \Gamma \) has two additional axioms \( \text{specNormRatExpr}_o \) and \( \text{specNormRatFun}_o \) that specify \( \text{normRatExpr}_{\epsilon \rightarrow \epsilon} \) and \( \text{normFun}_{\epsilon \rightarrow \epsilon} \). \( \text{specNormRatExpr}_o \) is the formula

\[
\forall u_\epsilon . \quad (\text{RatExpr}_{\epsilon \rightarrow o} u_\epsilon) \quad (\text{Norm}_{\epsilon \rightarrow o} (\text{normRatExpr}_{\epsilon \rightarrow o} u_\epsilon) \land (\text{val-in-} f_{\epsilon \rightarrow f} u_\epsilon \simeq \text{val-in-} f_{\epsilon \rightarrow f} (\text{normRatExpr}_{\epsilon \rightarrow o} u_\epsilon))) \quad (\text{specNormRatExpr}_{\epsilon \rightarrow o} u_\epsilon) \quad (\text{specNormRatFun}_{\epsilon \rightarrow o} u_\epsilon) \quad (\text{specNormRatFun}_{\epsilon \rightarrow o} u_\epsilon)
\]

(3) says that, if the input to \( \text{RatExpr}_{\epsilon \rightarrow o} \) represents a rational expression in \( x_\epsilon \), then the output represents a rational expression in \( x_\epsilon \) in normal form (the syntactic component). (4) says that, if the input represents a rational expression in \( x_\epsilon \), then either the input and output denote the same member of \( f \) or they both do not denote any member of \( f \) (the semantic component). And (5) says that, if the input does not represent a rational expression in \( x_\epsilon \), then the output is undefined.

\( \text{specNormRatFun}_o \) is the formula

\[
\forall u_\epsilon . \quad (\text{RatFun}_{\epsilon \rightarrow o} u_\epsilon) \quad (\text{RatFun}_{\epsilon \rightarrow o} (\text{normRatFun}_{\epsilon \rightarrow o} u_\epsilon) \land (\text{Quasinorm}_{\epsilon \rightarrow \epsilon} \rightarrow \text{body}_{\epsilon \rightarrow \epsilon} (\text{normRatExpr}_{\epsilon \rightarrow o} u_\epsilon)) \land (\text{specNormRatFun}_{\epsilon \rightarrow o} u_\epsilon) \quad (\text{specNormRatFun}_{\epsilon \rightarrow o} u_\epsilon) \quad (\text{specNormRatFun}_{\epsilon \rightarrow o} u_\epsilon)
\]

(5)
10

(3–4) say that, if the input to $\text{RatFun}_{\epsilon \to o}$ represents a rational function in $x_q$, then the output represents a rational function in $x_q$ whose body is in quasinormal form (the syntactic component). (5) says that, if the input represents a rational function in $x_q$, then input and output denote the same (possibly partial) function on the rational numbers (the semantic component). And (6) says that, if the input does not represent a rational function in $x_q$, then the output is undefined.

Not only is it possible to specify the algorithms $\text{normRatExpr}$ and $\text{normRatFun}$ in $\text{CTT}_{\text{uqe}}$, it is also possible to define the functions that these algorithms implement. Then applications of these functions can be evaluated in $\text{CTT}_{\text{uqe}}$ using a proof system for $\text{CTT}_{\text{uqe}}$.

4.8 Discussion

So why are we concerned about rational expressions and rational functions? Every computer algebra system implements functions that normalize rational expressions in several indeterminants over various fields guaranteeing that the normal form will be 0 if the rational expression equals 0 in the corresponding field of fractions. However, computer algebra systems make little distinction between a rational expression interpreted as a member of a field of fractions and a rational expression interpreted as a rational function.

For example, one can always evaluate an expression by assigning values to its free variables or even convert it to a function. In Maple\textsuperscript{3} these are done respectively via $\text{eval}(e, x = 0)$ and $\text{unapply}(e, x)$. This means that, if we normalize the rational expression $\frac{x^4 - 1}{x^2 - 1}$ to $x^2 + 1$ and then evaluate the result at $x = 1$, we get the value 2. But, if we evaluate $\frac{x^4 - 1}{x^2 - 1}$ at $x = 1$ without normalizing it, we get an error message due to division by 0. Hence, if a rational expression $r$ is interpreted as a function, then it is not valid to normalize it, but computer algebra system lets the user do exactly that since there is no distinction made between $r$ as a rational expression and $r$ as representing a rational function, as we have already mentioned.

The real problem here is that the normalization of a rational expression and the evaluation of an expression at a value are not compatible with each other. Indeed the function $g_q : \mathbb{Q}(x) \to \mathbb{Q}$ where $q \in \mathbb{Q}$ that maps a rational expression $r$ to the rational number obtained by replacing each occurrence of $x$ in $r$ with $q$ is not a homomorphism! In particular, $x/x$ is defined in $\mathbb{Q}(x)$, but $g_0(x/x)$ is undefined in $\mathbb{Q}$.

To avoid unsound applications of $\text{normRatExpr}$, $\text{normRatFun}$, and other SBMAs in mathematical systems, we need to carefully, if not formally, specify what these algorithms are intended to do. This is not a straightforward task to do in a traditional logic since SBMAs involve an interplay of syntax and semantics and algorithms like $\text{normRatExpr}$ and $\text{normRatFun}$ can be sensitive to definedness considerations. We can, however, specify these algorithm, as we have shown, in a logic like $\text{CTT}_{\text{uqe}}$.

\textsuperscript{3} Mathematica has similar commands.
5 Symbolically Differentiating Functions

5.1 Task

A basic task of calculus is to find the derivative of a function. Every student who studies calculus quickly learns that computing the derivative of \( f : \mathbb{R} \rightarrow \mathbb{R} \) is very difficult to do using only the definition of a derivative. It is a great deal easier to compute derivatives using an algorithm that repeatedly applies symbolic differentiation rules. For example,

\[
\frac{d}{dx} \sin(x^2 + x) = \cos(x^2 + x)(2x + 1)
\]

by applying the chain, sine, sum, power, and variable differentiation rules, and so the derivative of

\[
\lambda x : \mathbb{R} . \sin(x^2 + x)
\]

is

\[
\lambda x : \mathbb{R} . \cos(x^2 + x)(2x + 1).
\]

Notice that the symbolic differentiation algorithm is applied to expressions (e.g., \( \sin(x^2 + x) \)) that have a designated free variable (e.g., \( x \)) and not to the function \( \lambda x : \mathbb{R} . \sin(x^2 + x) \) the expression represents.

5.2 Problem

Let \( f = \lambda x : \mathbb{R} . \ln(x^2 - 1) \) and \( f' \) be the derivative of \( f \). Then

\[
\frac{d}{dx} \ln(x^2 - 1) = \frac{2x}{x^2 - 1}
\]

by standard symbolic differentiation rules. But

\[
g = \lambda x : \mathbb{R} . \frac{2x}{x^2 - 1}
\]

is not \( f' \)! The domain of \( f \) is \( D_f = \{ x \in \mathbb{R} \mid x < -1 \text{ or } x > 1 \} \) since the natural log function \( \ln \) is undefined on the nonpositive real numbers. Since \( f' \) is undefined wherever \( f \) is undefined, the domain \( D_{f'} \) of \( f' \) must be a subset of \( D_f \). But the domain of \( g \) is \( D_g = \{ x \in \mathbb{R} \mid x \neq -1 \text{ and } x \neq 1 \} \) which is clearly a superset of \( D_f \). Hence symbolic differentiation does not reliably produce derivatives.

5.3 Solution

Let \( \mathcal{L} \) be the language of expressions of type \( \mathbb{R} \) built from \( x \), the rational numbers, and operators for the following functions: \( +, \ast, -, ^{-1} \), the power function, the natural exponential and logarithm functions, and the trigonometric functions.
Let \( \text{diff} \) be the SBMA that takes \( e \in \mathcal{L} \) as input and returns the \( e' \in \mathcal{L} \) by repeatedly applying standard symbolic differentiation rules in some appropriate manner. The specification of \( \text{diff} \) is that, for all \( e \in \mathcal{L} \), (A) \( \text{diff}(e) \in \mathcal{L} \) and (B), for \( a \in \mathbb{R} \), if \( f = \lambda x : \mathbb{R}. e \) is differentiable at \( a \), then the derivative of \( f \) at \( a \) is \((\lambda x : \mathbb{R}. \text{diff}(e))(a)\). (A) is the syntactic component and (B) is the semantic component.

### 5.4 Specification in CTT\(_{uqe}\)

We specify \( \text{diff} \) in a theory \( T \) of CTT\(_{uqe}\) once again using CTT\(_{uqe}\)'s reflection infrastructure. Let \( T_0 = (L_0, \Gamma_0) \) be a theory of real numbers (formalized as the theory of a complete ordered field) that contains a base type \( r \) representing the real numbers and the usual individual and function constants.

We extend \( T_0 \) to a theory \( T_1 = (L_1, \Gamma_1) \) by defining the following two constants using the machinery of \( T_0 \):

1. \( \text{DiffExpr}_{\epsilon \rightarrow o} \) is a predicate representing the subtype of \( \epsilon \) that denotes the subset of expressions of type \( r \) built from \( x_r \), constants representing the rational numbers, and the constants representing \(+, -, ^{-1}, \) the power function, the natural exponential and logarithm functions, and the trigonometric functions. Thus, \( \text{DiffExpr}_{\epsilon \rightarrow o} \) is the subtype of expressions that can be symbolically differentiated and, for example, \( \text{DiffExpr}_{\epsilon \rightarrow o} \ interpret \frac{\ln(x^2) \cdot (x - 1)}{\ln(x^2 - 1)} \) (where parentheses and types have been dropped) is valid in \( T_1 \).

2. \( \text{deriv}_{r \rightarrow r \rightarrow r} \) is a function such that, if \( f \) and \( a \) are expressions of type \( r \rightarrow r \) and \( r_r \), respectively, then \( \text{deriv}_{r \rightarrow r \rightarrow r} f a \) is the derivative of \( f \) at \( a \) if \( f \) is differentiable at \( a \) and is undefined otherwise.

Finally, we can extend \( T_1 \) to a theory \( T = (L, \Gamma) \) in which \( L \) contains the constant \( \text{diff}_{\epsilon \rightarrow \epsilon} \) and \( \Gamma \) contains the following axiom \( \text{specDiff}_{\epsilon} \):

\[
\forall u_\epsilon. \quad \text{if } (\text{DiffExpr}_{\epsilon \rightarrow o} u_\epsilon) \quad \text{(1)}
\]
\[
(\text{DiffExpr}_{\epsilon \rightarrow o}(\text{diff}_{\epsilon \rightarrow \epsilon} u_\epsilon)) \land \quad \text{(2)}
\]
\[
\forall a_r. \quad (\text{deriv}_{r \rightarrow r \rightarrow r} (\lambda x_r. [u_\epsilon]_r) a_r) \downarrow \quad \text{(3)}
\]
\[
(\text{deriv}_{r \rightarrow r \rightarrow r} (\lambda x_r. [u_\epsilon]_r) a_r) = (\lambda x_r. [\text{diff}_{\epsilon \rightarrow \epsilon} u_\epsilon]_r) a_r \quad \text{(4)}
\]
\[
(\text{diff}_{\epsilon \rightarrow \epsilon} u_\epsilon) \uparrow \quad \text{(5)}
\]

(3) says that, if the input \( u_\epsilon \) to \( \text{specDiff}_{\epsilon} \) is a member of the subtype \( \text{DiffExpr}_{\epsilon \rightarrow o} \), then the output is also a member of \( \text{DiffExpr}_{\epsilon \rightarrow o} \) (the syntactic component). (4–6) say that, if the input is a member of \( \text{DiffExpr}_{\epsilon \rightarrow o} \) and, for all real numbers \( a \), if the function \( f \) represented by \( u_\epsilon \) is differentiable at \( a \), then \( f \) equals the function represented by \( \text{diff}_{\epsilon \rightarrow \epsilon} u_\epsilon \) at \( a \) (the semantic component). And (7) says that, if the input is not a member of \( \text{DiffExpr}_{\epsilon \rightarrow o} \), then the output is undefined.
5.5 Discussion

We have seen that symbolic differentiation alone does not always produce the derivative of a function. This is because symbolic differentiation does include an analysis of where a function is differentiable. A specification of a symbolic differentiation algorithm must include both a specification of what the algorithm does syntactically and where the output of the algorithm is semantically correct. Even though many symbolic differentiation engines found on the Web work purely syntactically, correct results can only be guaranteed with a differentiability analysis.

6 Related Work

The literature on the formal specification of symbolic computation algorithms is fairly modest; it includes the papers \[3,12,13,14\]. One of first systems to implement SBMAs in a formal setting is MATHPERT \[1\] (later called MathXpert), the mathematics education system developed by Michael Beeson. Another system in which SBMAs are formally implemented is the computer algebra system built on top of HOL Light \[9\] by Cezary Kaliszyk and Freek Wiedijk \[11\]. Both systems deal in a careful way with the interplay of syntax and semantics that characterize SBMAs. Kaliszyk addresses in \[10\] the problem of simplifying the kind of mathematical expressions that arise in computer algebra system resulting from the application of partial functions in a proof assistant in which all functions are total. Stephen Watt distinguishes in \[15\] between symbolic computation and computer algebra which is very similar to the distinction between syntax-based and semantics-based mathematical algorithms.

There is an extensive review in \[8\] of the literature on metaprogramming, metareasoning, reflection, quotation, theories of truth, reasoning in lambda calculus about syntax, and undefinedness related to CTT_{qe} and CTT_{uqe}.

7 Conclusion

Commonplace in mathematics, SBMAs are interesting and useful algorithms that manipulate the syntactic structure of mathematical expressions to achieve a mathematical task. Specifications of SBMAs are often complex because manipulating syntax is complex by its own nature, the algorithms involve an interplay of syntax and semantics, and undefined expressions are often generated from the syntactic manipulations. SBMAs can be tricky to implement in mathematical software systems that do not provide good support for the interplay of syntax and semantics that is inherent in these algorithms. For the same reason, they are challenging to specify in a traditional formal logic that provides little built-in support for reasoning about syntax.

In this paper, we have examined representative SBMAs that fulfill basic mathematical tasks. We have shown the problems that arise if they are not implemented carefully and we have delineated their specifications. We have
also sketched how their specifications can be written in CTTuqe [7], a version of Church’s type that is well suited for expressing the interplay of syntax and semantics by virtue of its global reflection infrastructure.

We would like to continue this work first by writing complete specifications of SBMAs in CTTuqe [7], CTTqe [8], and other logics. Second by formally defining SBMAs in CTTuqe and CTTqe. Third by formally proving in CTTuqe [7] and CTTqe [8] the mathematical meanings of SBMAs from their formal definitions. And fourth by further developing HOL Light QE [2] so that these SBMA definitions and the proofs of their mathematical meanings can be performed and machine checked in HOL Light QE. As a small startup example, we have defined a symbolic differentiation algorithm for polynomials and proved its mathematical meaning from its definition in [8 subsections 4.4 and 9.3].

References

1. Beeson, M.: Logic and computation in MATHPERT: An expert system for learning mathematics. In: Kaltofen, E., Watt, S. (eds.) Computers and Mathematics, pp. 202–214. Springer (1989)
2. Carette, J., Farmer, W.M., Laskowski, P.: HOL Light QE. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. Lecture Notes in Computer Science, Springer (2018), forthcoming
3. Dunstan, M., Kelsey, T., Linton, S., Martin, U.: Lightweight formal methods for computer algebra systems. In: Weispfenning, V., Trager, B.M. (eds.) Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation. pp. 80–87. ACM (1998)
4. Farmer, W.M.: A simple type theory with partial functions and subtypes. Annals of Pure and Applied Logic 64, 211–240 (1993)
5. Farmer, W.M.: Formalizing undefinedness arising in calculus. In: Basin, D., Rusinowitch, M. (eds.) Automated Reasoning—IJCAR 2004. Lecture Notes in Computer Science, vol. 3097, pp. 475–489. Springer (2004)
6. Farmer, W.M.: Andrews’ type system with undefinedness. In: Benzmüller, C., Brown, C., Siekmann, J., Statman, R. (eds.) Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on his 70th Birthday, pp. 223–242. Studies in Logic, College Publications (2008)
7. Farmer, W.M.: Theory morphisms in Church’s type theory with quotation and evaluation. In: Geuvers, H., Englund, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 10383, pp. 147–162. Springer (2017)
8. Farmer, W.M.: Incorporating quotation and evaluation into Church’s type theory. Information and Computation 260, 9–50 (2018)
9. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 5674, pp. 60–66. Springer (2009)
10. Kaliszyk, C.: Automating side conditions in formalized partial functions. In: Autexier, A., Campbell, J., Rubio, J., Suzuki, M., Wiedijk, F. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 5144, pp. 300–314. Springer (2008)
11. Kaliszyk, C., Wiedijk, F.: Certified computer algebra on top of an interactive theorem prover. In: Kauers, M., Kerber, M., Miner, R.R., Windsteiger, W. (eds.) Towards Mechanized Mathematical Assistants. Lecture Notes in Computer Science, vol. 4573, pp. 94–105. Springer (2007)

12. Khan, M.T.: Formal Specification and Verification of Computer Algebra Software. Ph.D. thesis, RISC, Johannes Kepler Universität Linz (2014)

13. Khan, M.T., Schreiner, W.: Towards the formal specification and verification of maple programs. In: Jeuring, J., Campbell, J., Carette, J., Reis, G.D., P.Sojka, Wenzel, M., Sorge, V. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 7362, pp. 231–247. Springer (2012)

14. Limongelli, C., M.Temperini: Abstract specification of structures and methods in symbolic mathematical computation. Theoretical Computer Science 104, 89–107 (1992)

15. Watt, S.M.: Making computer algebra more symbolic. In: Proceedings of Transgressive Computing 2006. pp. 43–49. Granada, Spain (2006)