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

6 May 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 expressions (syntax) in a "meaningful" way. For instance, children learn to perform arithmetic by executing algorithms that manipulate strings of digits that represent numbers. A *syntax-based mathematical algorithm (SBMA)* is such an algorithm, that performs a mathematical task by manipulating the syntactic structure of certain 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 *meaning*[1] of its input and output expressions. Understanding what a SBMA does requires understanding how its computational behavior is related to 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.

[1] I.e., denotation.

This inherent complexity of SBMA specifications makes SBMAs tricky to implement correctly. Dealing with the semantic component is usually the bigger challenge for computer algebra systems as they excel in the realm of computation but have weak reasoning facilities, while the syntactic component is usually the bigger obstacle for proof assistants, often due to partiality issues.

In this paper, we examine four representative examples of SBMAs, present their specifications, and show how their specifications can be written in CTT_{uqe} [12], a formal logic designed to make expressing the interplay of syntax and semantics easier than in traditional logics. The paper is organized as follows. Section 2 presents background information about semantic notions and CTT_{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 gives a brief overview of related work. And the paper ends with a short conclusion in section 7.

The principal contribution of this paper, in the author's opinion, is not the specifications themselves, but rather bringing to the fore the subtle details of SBMAs themselves, along with the fact that traditional logics are ill-suited to the specification of SBMAs. While here we use CTT_{uqe} for this purpose, the most important aspect is the ability to deal with two levels at once, syntax and semantics. The examples are chosen because they represent what are traditionally understood as fairly simple, even straightforward, symbolic algorithms, and yet they are nevertheless rather difficult to formalize properly.

2 Background

To be able to formally display the issues involved, it is convenient to first be specific about definedness, equality, quasi-equality, and logics that can deal with syntax and semantics directly.

2.1 Definedness, Equality, and Quasi-Equality

Let e be an expression and D be a domain of 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. When D is a domain of interest to mathematicians, we will call e a *mathematical expression*.

2.2 CTT_{qe} and CTT_{uqe}

CTT_{qe} [13] is a version of Church's type theory with a built-in *global reflection infrastructure* with global quotation and evaluation operators geared towards

reasoning about the interplay of syntax and semantics and, in particular, for specifying, defining, applying, and reasoning about SBMAs. The syntax and semantics of CTT_{qe} is presented in [13]. A proof system for CTT_{qe} that is sound for all formulas and complete for eval-free formulas is also presented in [13]. (An expression is *eval-free* if it does not contain the evaluation operator.) By modifying HOL Light [14], we have produced a rudimentary implementation of CTT_{qe} called HOL Light QE [5].

CTT_{uqe} [12] is a variant of CTT_{qe} that has built-in support for partial functions and undefinedness based on the traditional approach to undefinedness [10]. It is well-suited for specifying SBMAs that manipulate expressions that may be undefined. Its syntax and semantics are presented in [12]. A proof system for CTT_{uqe} is not given there, but can be straightforwardly derived by merging those for CTT_{qe} [13] and $\mathcal{Q}_0^{\text{u}}$ [11].

The global reflection infrastructure of CTT_{uqe} (and CTT_{qe}) consists of three components. The first is an inductive type ϵ of *syntactic values*: these typically represent the syntax tree of an eval-free expression of CTT_{uqe}. Each expression of type ϵ 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 ϵ. The second component is a *quotation operator* $\ulcorner \cdot \urcorner$ such that, if $\mathbf{A}_\alpha$ is an eval-free expression (of some type α), then $\ulcorner \mathbf{A}_\alpha \urcorner$ is an expression of type ϵ that denotes the syntactic value that represents the syntax tree of $\mathbf{A}_\alpha$. Finally, the third component is an *evaluation operator* $[\![\cdot]\!]_\alpha$ such that, if $\mathbf{E}_\epsilon$ is an expression of type ϵ, then $[\![\mathbf{E}_\epsilon]\!]_\alpha$ denotes the value of type α denoted by the expression $\mathbf{B}$ represented by $\mathbf{E}_\epsilon$ (provided the type of $\mathbf{B}$ is α). In particular the *law of disquotation* $[\![\ulcorner \mathbf{A}_\alpha \urcorner]\!]_\alpha = \mathbf{A}_\alpha$ 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* which constructs an inductive type of syntactic values only for the expressions of the logic that are relevant to a particular problem. See [13] 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 ϵ includes syntax values for all eval-free expressions of all types as well as syntax values for ill-formed expressions like $(\mathbf{x}_\alpha\, \mathbf{x}_\alpha)$ in which the types are mismatched. Convenient subtypes of ϵ can be represented via predicates of type $\epsilon \to o$. (o is the type of boolean values.) In particular, CTT_{uqe} contains a predicate $\mathsf{is\text{-}expr}^{\alpha}_{\epsilon \to o}$ for every type α that represents the subtype of syntax values for expressions of type α.

Unlike CTT_{qe}, CTT_{uqe} admits undefined expressions and partial functions. The formulas $\mathbf{A}_\alpha \downarrow$ and $\mathbf{A}_\alpha \uparrow$ assert that the expression $\mathbf{A}_\alpha$ is defined and undefined, respectively. Formulas (i.e., expressions of type o) are always defined. Evaluations may be undefined. For example, $[\![\ulcorner \mathbf{A}_\alpha \urcorner]\!]_\beta$ is undefined when $\alpha \neq \beta$. See [11,12] for further details.

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 $12 = 2^2 * 3$ — but this is not actually the answer one gets in many systems! The reason is, that 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 immediately evaluated to $12 = 12$, which is certainly not very informative.

3.2 Problem

So why is $2^2 * 3$ not an answer? Because it involves a mixture of *syntax* and *semantics*. A better answer would be $\ulcorner 2^2 * 3 \urcorner$ (the quotation of $2^2 * 3$) that would make it clear that $*$ *represents* multiplication rather than *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 inert forms, while in Mathematica, there are various related concepts such as `Hold`, `Inactive` and `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 `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[2]

```
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;
```

[2] There are nonessential Maple-isms in this routine: because of how foldr is defined, op 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.

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

(B) $\mathsf{remult}\,(\mathsf{ifactors}\,(n)) = n$.

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

3.5 Specification in $\mathbf{CTT_{uqe}}$

We specify the factorization of integers in a theory T of $\mathrm{CTT_{uqe}}$ using $\mathrm{CTT_{uqe}}$'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 $0_i, 1_i, 2_i, \ldots, -_{i\to i}$, $+_{i\to i\to i}$, $*_{i\to i\to i}$, and $\hat{}_{i\to i\to i}$. Γ_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:

1. $\mathsf{Numeral}_{\epsilon\to o}$ is a predicate representing the subtype of ϵ that denotes the subset $\{0_i, 1_i, 2_i, \ldots\}$ of expressions of type i. Thus, $\mathsf{Numeral}_{\epsilon\to o}$ is the subtype of numerals and, for example, $\mathsf{Numeral}_{\epsilon\to o}\ulcorner 2_i \urcorner$ is valid in T_1.
2. $\mathsf{PrimeDecomp}_{\epsilon\to o}$ is a predicate representing the subtype of ϵ that denotes the subset of expressions of type i of the form 0_i or

   $$\pm 1 * p_0^{e_0} * \cdots * p_k^{e_k}$$

   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 $\mathsf{PrimeDecomp}_{\epsilon\to o}$ is a subtype of signed prime decompositions and, for example, $\mathsf{PrimeDecomp}_{\epsilon\to o}\ulcorner 1 * 2^2 * 3^1 \urcorner$ (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 $\mathsf{factor}_{\epsilon\to\epsilon}$ and Γ contains the following axiom $\mathsf{specFactor}_o$:

$$\begin{array}{l}\forall\, u_\epsilon\ . \\ \quad \mathsf{if}\ (\mathsf{Numeral}_{\epsilon\to o}\, u_\epsilon) \\ \qquad (\mathsf{PrimeDecomp}_{\epsilon\to\epsilon}(\mathsf{factor}_{\epsilon\to\epsilon}\, u_\epsilon) \wedge [\![u_\epsilon]\!]_i = [\![\mathsf{factor}_{\epsilon\to\epsilon}\, u_\epsilon]\!]_i) \\ \qquad (\mathsf{factor}_{\epsilon\to\epsilon}\, u_\epsilon)\!\uparrow\end{array}$$

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

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*: $s * p_0^{e_0} * p_1^{e_1} * \cdots * p_k^{e_k}$. 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[3], and thus this output *composes* well with other system features.

It is worth noting that none of the reasons for the *correctness* of this representation is clearly visible: once the integers are partitioned into negative, zero and positive, and only positive natural numbers are subject to "prime factorization", their structure as a *free commutative monoid* on infinitely many generators (the primes) comes out. And so it is natural that *multisets* (also called *bags*) are the natural representation. The list-with-multiplicities makes that clear, while in some sense the more human-friendly syntactic representation $s*p_0^{e_0}*p_1^{e_1}*\cdots*p_k^{e_k}$ obscures that.

Nevertheless, the main lesson is that a simple mathematical task, such as factoring the number 12, which seems like a question about simple integer arithmetic, 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 $\mathcal{L}_{\text{re}}$ 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 $\mathcal{L}_{\text{re}}$ can be something simple like $\frac{x^4-1}{x^2-1}$ or something more complicated like

$$\frac{\frac{1-x}{3/2x^{18}+x+17}}{\frac{1}{9834*x^{19393874}-1/5}} + 3 * x - \frac{12}{x}.$$

[3] This is unsurprising given that the builders of both Maple and Mathematica were well acquainted with Macsyma which was implemented in Lisp.

The members of $\mathcal{L}_{\text{re}}$ 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 $\mathcal{L}_{\text{rf}}$ be the set of expressions of the form $(\lambda\, x : \mathbb{Q}\ .\ r)$ where $r \in \mathcal{L}_{\text{re}}$. The members of $\mathcal{L}_{\text{rf}}$ 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 $\mathbb{Q}$. Even 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 $\mathsf{gcd}(p, q) = 1$. 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 $\mathsf{normRatExpr}$ be the SBMA that takes $r \in \mathcal{L}_{\text{re}}$ as input and returns the $r' \in \mathcal{L}_{\text{re}}$ as output such that r' is the normal form of r. How should $\mathsf{normRatExpr}$ be specified?

4.2 Problem 1

$\mathsf{normRatExpr}$ must normalize rational expressions as expressions that denote members of $\mathbb{Q}(x)$, not members of $\mathbb{Q}$. Hence $\mathsf{normRatExpr}(x/x)$ and $\mathsf{normRatExpr}(1/x - 1/x)$ 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 $\mathsf{normRatExpr}$ 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 $\mathsf{normRatExpr}$ is that, for all $r \in \mathcal{L}_{\text{re}}$, (A) $\mathsf{normRatExpr}(r)$ is a normal form and (B) $r \simeq_{\mathbb{Q}(x)} \mathsf{normRatExpr}(r)$. (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 $\mathsf{normRatExpr}(r)$ is also undefined in $\mathbb{Q}(x)$. For example, since $r = \frac{1}{x-x}$ is undefined in $\mathbb{Q}(x)$, $\mathsf{normRatExpr}(r)$ 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 $\mathsf{normRatFun}$ be the SBMA that takes $f \in \mathcal{L}_{\mathrm{rf}}$ as input and returns a $f' \in \mathcal{L}_{\mathrm{rf}}$ as output such that f' is the normal form of f. How should $\mathsf{normRatFun}$ 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 $f_1 =_{\mathbb{Q}\to\mathbb{Q}} f_2$ 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, $(\lambda\, x : \mathbb{Q}\ .\ (1/x - 1/x)) \neq_{\mathbb{Q}\to\mathbb{Q}} (\lambda\, x : \mathbb{Q}\ .\ 0)$ 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 ≥ 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 $\overline{\mathbb{Q}}$ (the algebraic closure of $\mathbb{Q}$) or $\mathbb{R}$ (since $x^2 - 2 =_{\mathbb{R}[x]} (x - \sqrt{2})(x + \sqrt{2})$.)

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

4.7 Specification in $\mathrm{CTT}_{\mathbf{uqe}}$

We specify $\mathsf{normRatExpr}$ and $\mathsf{normRatFun}$ in a theory of $\mathrm{CTT}_{\mathrm{uqe}}$ again using $\mathrm{CTT}_{\mathrm{uqe}}$'s reflection infrastructure. A complete development of T would be long and tedious, thus we only sketch it.

The first step is to define a theory $T_0 = (L_0, \Gamma_0)$ that axiomatizes the field $\mathbb{Q}$; L_0 contains a base type q and constants 0_q, 1_q, $+_{q\to q\to q}$, $*_{q\to q\to q}$, $-_{q\to q}$, and $^{-1}{}_{q\to q}$ representing the standard elements and operators of a field. Γ_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, $+_{f\to f\to f}$, $*_{f\to f\to f}$, $-_{f\to f}$, and ${}^{-1_{f\to f}}$ representing the standard elements and operators of a field; and a constant X_f representing the indeterminant of $\mathbb{Q}(x)$. Γ_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 CTT_{uqe} does not admit subtypes as in [9].

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 obtain by defining the following constants using the machinery of T_1:

1. $\mathsf{RatExpr}_{\epsilon\to o}$ is the predicate representing the subtype of ϵ 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, $\mathsf{RatExpr}_{\epsilon\to o}{}^{\ulcorner} x_q/x_q {}^{\urcorner}$ is valid in T_2.
2. $\mathsf{RatFun}_{\epsilon\to o}$ is the predicate representing the subtype of ϵ 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 $(\lambda\, x_q\ .\ \mathbf{R}_q)$ where $\mathbf{R}_q$ has the form of a rational expression in x_q). For example $\mathsf{RatFun}_{\epsilon\to o}{}^{\ulcorner}\lambda\, x_q\ .\ x_q/x_q{}^{\urcorner}$ is valid in T_2.
3. $\mathsf{val\text{-}in}\text{-}f_{\epsilon\to f}$ is a partial function that maps each member of the subtype $\mathsf{RatExpr}_{\epsilon\to o}$ to its denotation in f. So, for example,

$$\mathsf{val\text{-}in}\text{-}f_{\epsilon\to f}{}^{\ulcorner} x_q +_{q\to q\to q} 1_q {}^{\urcorner} = X_f +_{f\to f\to f} 1_f$$

   and $(\mathsf{val\text{-}in}\text{-}f_{\epsilon\to f}{}^{\ulcorner} 1_q/0_q {}^{\urcorner})\!\uparrow$ are valid in T_2. $\mathsf{val\text{-}in}\text{-}f_{\epsilon\to f}$ is partial on is domain since an expression like $1_q/0_q$ does not denote a member of f.
4. $\mathsf{Norm}_{\epsilon\to o}$ is the predicate representing the subtype of ϵ that denotes the subset of the subtype $\mathsf{RatExpr}_{\epsilon\to o}$ whose members are normal forms. So, for example, $\neg(\mathsf{Norm}_{\epsilon\to o}{}^{\ulcorner} x_q/x_q {}^{\urcorner})$ and $\mathsf{Norm}_{\epsilon\to o}{}^{\ulcorner} 1_q {}^{\urcorner}$ are valid in T_2.
5. $\mathsf{Quasinorm}_{\epsilon\to o}$ is the predicate representing the subtype of ϵ that denotes the subset of the subtype $\mathsf{RatExpr}_{\epsilon\to o}$ whose members are quasinormal forms. So, for example, $\mathsf{Quasinorm}_{\epsilon\to o}{}^{\ulcorner} x_q/x_q {}^{\urcorner}$ and $\neg(\mathsf{Quasinorm}_{\epsilon\to o}{}^{\ulcorner}\mathbf{A}_q/\mathbf{A}_q{}^{\urcorner})$, where $\mathbf{A}_q$ is $x_q^2 +_{q\to q\to q} 1_q$, are valid in T_2.
6. $\mathsf{body}_{\epsilon\to\epsilon}$ is a partial function that maps each member of ϵ denoting an expression of the form $(\lambda\, x_\alpha\ .\ \mathbf{B}_\beta)$ to the member of ϵ that denotes ${}^{\ulcorner}\mathbf{B}_\beta{}^{\urcorner}$ and is undefined on the rest of ϵ. Note that there is no *scope extrusion* here as, in syntactic expressions, the x_α is visible.

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

$$
\begin{array}{ll}
\forall\, u_\epsilon\ . & (1)\\
\quad \mathsf{if}\ (\mathsf{RatExpr}_{\epsilon\to o}\, u_\epsilon) & (2)\\
\quad\quad (\mathsf{Norm}_{\epsilon\to\epsilon}(\mathsf{normRatExpr}_{\epsilon\to\epsilon}\, u_\epsilon) \wedge & (3)\\
\quad\quad\quad \mathsf{val\text{-}in\text{-}}f_{\epsilon\to f}\, u_\epsilon \simeq \mathsf{val\text{-}in\text{-}}f_{\epsilon\to f}(\mathsf{normRatExpr}_{\epsilon\to\epsilon}\, u_\epsilon)) & (4)\\
\quad\quad (\mathsf{normRatExpr}_{\epsilon\to\epsilon}\, u_\epsilon)\!\uparrow & (5)
\end{array}
$$

(3) says that, if the input to $\mathsf{RatExpr}_{\epsilon\to o}$ represents a rational expression in x_q, then the output represents a rational expression in x_q in normal form (the syntactic component). (4) says that, if the input represents a rational expression in x_q, 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_q, then the output is undefined.

$\mathsf{specNormRatFun}_o$ is the formula

$$
\begin{array}{ll}
\forall\, u_\epsilon\ . & (1)\\
\quad \mathsf{if}\ (\mathsf{RatFun}_{\epsilon\to o}\, u_\epsilon) & (2)\\
\quad\quad (\mathsf{RatFun}_{\epsilon\to o}\, (\mathsf{normRatFun}_{\epsilon\to\epsilon}\, u_\epsilon) \wedge & (3)\\
\quad\quad\quad \mathsf{Quasinorm}_{\epsilon\to\epsilon}(\mathsf{body}_{\epsilon\to\epsilon}(\mathsf{normRatExpr}_{\epsilon\to\epsilon}\, u_\epsilon)) \wedge & (4)\\
\quad\quad\quad [\![u_\epsilon]\!]_{q\to q} = [\![\mathsf{normRatExpr}_{\epsilon\to o}\, u_\epsilon]\!]_{q\to q}) & (5)\\
\quad\quad (\mathsf{normRatFun}_{\epsilon\to\epsilon}\, u_\epsilon)\!\uparrow & (6)
\end{array}
$$

(3–4) say that, if the input to $\mathsf{RatFun}_{\epsilon\to o}$ 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 normRatExpr and normRatFun in $\mathrm{CTT}_{\mathrm{uqe}}$, it is also possible to define the functions that these algorithms implement. Then applications of these functions can be evaluated in $\mathrm{CTT}_{\mathrm{uqe}}$ using a proof system for $\mathrm{CTT}_{\mathrm{uqe}}$.

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[4], these are done respectively via `eval(e, x = 0)` and `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 a 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 normRatExpr, 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 normRatExpr and normRatFun can be sensitive to definedness considerations. We can, however, specify these algorithm, as we have shown, in a logic like CTT_{qe}.

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 $f : \mathbb{R} \to \mathbb{R}$ 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,

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

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

$$\lambda\, x : \mathbb{R} \,.\, \mathsf{sin}(x^2 + x)$$

is

$$\lambda\, x : \mathbb{R} \,.\, (2x + 1)\mathsf{cos}(x^2 + x).$$

[4] Mathematica has similar commands.

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

5.2 Problem

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

$$\frac{d}{dx}\mathsf{ln}(x^2-1) = \frac{2x}{x^2-1}$$

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. Over $\mathbb{C}$ there are even more egregious examples where infinitely many singularities are "forgotten". Hence symbolic differentiation does not reliably produce derivatives.

5.3 A 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: $+$, $*$, $-$, $^{-1}$, the power function, the natural exponential and logarithm functions, and the trigonometric functions. Let 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 diff is that, for all $e \in \mathcal{L}$, (A) $\mathsf{diff}(e) \in \mathcal{L}$ 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 $(\lambda\, x : \mathbb{R}\ .\ \mathsf{diff}(e))(a)$. (A) is the syntactic component and (B) is the semantic component.

5.4 Specification in $\mathbf{CTT_{uqe}}$

We specify diff in a theory T of $\textsc{ctt}_{\textsc{uqe}}$ once again using $\textsc{ctt}_{\textsc{uqe}}$'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. $\mathsf{DiffExpr}_{\epsilon \to o}$ is a predicate representing the subtype of ϵ 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, $\mathsf{DiffExpr}_{\epsilon\to o}$ is the subtype of expressions that can be symbolically differentiated and, for example, $\mathsf{DiffExpr}_{\epsilon\to o}\ulcorner\mathsf{ln}(x_r^2-1)\urcorner$ (where parentheses and types have been dropped) is valid in T_1.

2. $\mathsf{deriv}_{(r\to r)\to r\to r}$ is a function such that, if f and a are expressions of type $r\to r$ and r, respectively, then $\mathsf{deriv}_{(r\to r)\to r\to r}\, f\, a$ 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 $\mathsf{diff}_{\epsilon\to\epsilon}$ and Γ contains the following axiom $\mathsf{specDiff}_o$:

$$\forall\, u_\epsilon\ . \tag{1}$$
$$\quad \mathsf{if}\ (\mathsf{DiffExpr}_{\epsilon\to o}\, u_\epsilon) \tag{2}$$
$$\quad\quad (\mathsf{DiffExpr}_{\epsilon\to\epsilon}(\mathsf{diff}_{\epsilon\to\epsilon}\, u_\epsilon)\wedge \tag{3}$$
$$\quad\quad\ \forall\, a_r\ . \tag{4}$$
$$\quad\quad\quad (\mathsf{deriv}_{(r\to r)\to r\to r}\,(\lambda\, x_r\ .\ [\![u_e]\!]_r)\, a_r)\!\downarrow \supset \tag{5}$$
$$\quad\quad\quad\quad \mathsf{deriv}_{(r\to r)\to r\to r}\,(\lambda\, x_r\ .\ [\![u_e]\!]_r)\, a_r = (\lambda\, x_r\ .\ [\![\mathsf{diff}_{\epsilon\to\epsilon}\, u_e]\!]_r)\, a_r \tag{6}$$
$$\quad\quad (\mathsf{diff}_{\epsilon\to\epsilon}\, u_\epsilon)\!\uparrow \tag{7}$$

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

5.5 Discussion

Merely applying the rules of symbolic differentiation does not always produce the derivative of function. The problem is that symbolic differentiation does not actually analyze the regions of differentiability of a function. A specification of differentiation as a symbolic algorithm, to merit the name of *differentiation*, must not just perform rewrite rules on the syntactic expression, but also compute the corresponding validity region. This is a mistake common to essentially all symbolic differentiation engines that we have been able to find.

A better solution then is to have syntactic representations of functions have an explicit syntactic component marking their *domain of definition*, so that a symbolic differentiation algorithm would be forced to produce such a domain on output as well.

In other words, we should regard the "specification" $f=\lambda\, x:\mathbb{R}\ .\ \mathsf{ln}(x^2-1)$ itself as incorrect, and replace it instead with $f=\lambda\, x:\{y\in\mathbb{R}\mid y<-1 \text{ or } y>1\}\ .\ \mathsf{ln}(x^2-1)$.

6 Related Work

The literature on the formal specification of symbolic computation algorithms is fairly modest; it includes the papers [7,17,18,19]. One of the first systems to implement SBMAs in a formal setting is MATHPERT [2] (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 [14] by Cezary Kaliszyk and Freek Wiedijk [16]. Both systems deal in a careful way with the interplay of syntax and semantics that characterize SBMAs. Kaliszyk addresses in [15] 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 [22] 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 [13] 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}. For work on developing infrastructures in proof assistants for global reflection, see [1,3,4,6,8,20,21], which covers, amongst others, the recent work in Agda, Coq, Idris, and Lean in this direction. Note that this infrastructure is all quite recent, and has not yet been used to deal with the kinds of examples in this paper — thus we do not yet know how *adequate* these features are for the task.

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 CTT_{uqe} [12], 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 CTT_{uqe} [12], CTT_{qe} [13], and other logics. Second by formally defining SBMAs in CTT_{uqe} and CTT_{qe}. Third by formally proving in CTT_{uqe} [12] and

CTT_{qe} [13] the mathematical meanings of SBMAs from their formal definitions. And fourth by further developing HOL Light QE [5] 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 [13, subsections 4.4 and 9.3].

Acknowledgments

This research was supported by NSERC. The authors would like to thank the referees for their comments and suggestions.

References

1. Anand, A., Boulier, S., Cohen, C., Sozeau, M., Tabareau, N.: Towards certified meta-programming with typed Template-Coq. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. pp. 20–39. Lecture Notes in Computer Science, Springer (2018)
2. 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)
3. Boyer, R., Moore, J.: Metafunctions: Proving them correct and using them efficiently as new proof procedures. In: Boyer, R., Moore, J. (eds.) The Correctness Problem in Computer Science, pp. 103–185. Academic Press (1981)
4. Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards computer-aided mathematical theory exploration. Journal of Applied Logic **4**, 470–504 (2006)
5. Carette, J., Farmer, W.M., Laskowski, P.: HOL Light QE. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. pp. 215–234. Lecture Notes in Computer Science, Springer (2018)
6. Christiansen, D.R.: Type-directed elaboration of quasiquotations: A high-level syntax for low-level reflection. In: Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages. pp. 1:1–1:9. IFL '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2746325.2746326
7. 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)
8. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proceedings of the ACM on Programming Languages **1**(ICFP), 34 (2017)
9. Farmer, W.M.: A simple type theory with partial functions and subtypes. Annals of Pure and Applied Logic **64**, 211–240 (1993)
10. 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)

11. 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)
12. Farmer, W.M.: Theory morphisms in Church's type theory with quotation and evaluation. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 10383, pp. 147–162. Springer (2017)
13. Farmer, W.M.: Incorporating quotation and evaluation into Church's type theory. Information and Computation **260**, 9–50 (2018)
14. 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)
15. 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)
16. 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)
17. Khan, M.T.: Formal Specification and Verification of Computer Algebra Software. Ph.D. thesis, RISC, Johannes Kepler Universität Linz (2014)
18. 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)
19. Limongelli, C., M.Temperini: Abstract specification of structures and methods in symbolic mathematical computation. Theoretical Computer Science **104**, 89–107 (1992)
20. Melham, T., Cohn, R., Childs, I.: On the semantics of ReFLect as a basis for a reflective theorem prover. Computing Research Repository (CoRR) **abs/1309.5742** (2013), `http://arxiv.org/abs/1309.5742`
21. van der Walt, P., Swierstra, W.: Engineering proof by reflection in Agda. In: Hinze, R. (ed.) Implementation and Application of Functional Languages. Lecture Notes in Computer Science, vol. 8241, pp. 157–173. Springer (2012)
22. Watt, S.M.: Making computer algebra more symbolic. In: Proceedings of Transgressive Computing 2006. pp. 43–49. Granada, Spain (2006)