Understanding Expression Simplification

Jacques Carette
Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, Ontario L8S 4K1
Canada
carette@mcmaster.ca

ABSTRACT

We give the first formal definition of the concept of *simplification* for general expressions in the context of Computer Algebra Systems. The main mathematical tool is an adaptation of the theory of Minimum Description Length, which is closely related to various theories of complexity, such as Kolmogorov Complexity and Algorithmic Information Theory. In particular, we show how this theory can justify the use of various "magic constants" for deciding between some equivalent representations of an expression, as found in implementations of simplification routines.

Categories and Subject Descriptors

I.1.1 [**Symbolic and Algebraic Manipulation**]: Simplification of expressions

General Terms

Theory

Keywords

Simplification of expressions, computer algebra, Kolmogorov Complexity, model description length

1. INTRODUCTION

It is easy to argue that Maple's `simplify` and Mathematica's `Simplify` and `FullSimplify` are some of the most heavily used commands of either system. A short conversation with end users or a survey of Maple worksheets (or Mathematica notebooks) quickly confirms this impression. But if one instead scours the scientific literature to find papers relating to simplification, a few are easily found: a few early general papers [4, 6, 19] [and the earlier work they reference], some on elementary functions like [3], as well as papers on nested radicals [17, 26], but even dedicated searches found little more. Looking at the standard textbooks on Computer Algebra Systems (CAS) [14, 11, 23] leaves one even more perplexed: it is not even possible to find a proper *definition* of the problem of simplification. There is an extensive discussion of the topic in [11] which largely focuses on heuristics for useful transformations while avoiding a formal definition of the problem. The Handbook of Computer Algebra [15] does not even acknowledge that the problem exists! This is all the more troubling as conversations with system builders quickly convinces one that the code for the simplification routines is as complex as that of a symbolic integrator; integrators on the other hand are amply documented in the scientific literature, where the underlying theory is clearly expounded even if their software design is not. This paper is an attempt to fill this void: we will give a formal definition of what it means for one expression to be simpler than another semantically equivalent expression.

It is worth noting that the concept of *simplification* studied here is the one which is empirically implemented in simplification routines in current systems: in other words, it is a study of *representational* simplification. Issues of computational complexity and of "usefulness" of a representation for further computations are not our concern, as it is not the concern of `simplify` nor `Simplify`.

In some cases, some expressions are universally(?) recognized as "simpler": 0 is simpler than

$$(x+3)^3 - x^3 - 9x^2 - 27x - 27,$$

$1+\sqrt{2}$ is simpler than

$$\sqrt[3]{7+5\sqrt{2}},$$

and 4 is simpler than $2+1+1$. In other cases, the issue is not as clear: is the expression $2^{2^{1024}} - 1$ simpler than the universe-filling equivalent integer? Or consider the 10,000th Chebyshev polynomial: Is $\mathsf{ChebyshevT}(10000, x)$ simpler than the several pages long expanded polynomial? We argue that the former is simpler. Of course, most would agree that x is simpler than $\mathsf{ChebyshevT}(1, x)$, while others would rightfully argue that the latter contains valuable information which might be crucial for further computations, but that is a different issue. Another question to ask is whether 1 is simpler than $\frac{x-1}{x-1}$, and $x+3$ is simpler than

$$\frac{x^2-9}{x-3}.$$

A good overview of these issues is given by Moses [19], who comes closest to defining simplification when he says "Thus

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSAC'04, July 4–7, 2004, Santander, Spain.
Copyright 2004 ACM 1-58113-827-X/04/0007 ...$5.00.

an ideal, but not very helpful, way to describe simplification is that it is the process which transforms expressions into a form in which the remaining steps of a computation can be most efficiently performed". We strenuously disagree with this view of simplification, which puts undue emphasis on the efficiency of uncertain future operations.

The examples in the previous paragraph should be sufficient to convince the reader that the issue of simplification is quite complex, as what is "simpler" does not a priori seem to have a common definition from situation to situation. The main contribution of this paper is to show that this is in fact not the case, that there is a straightforward notion of simplicity that underlies all of the above. An informal definition would read

Definition 1. *An expression A is* simpler *than an expression B if*

- *in all contexts where A and B can be used, they mean the same thing, and*
- *the length of the description of A is shorter than the length of the description of B.*

In other words, we wish to put emphasis on the representational complexity of an expression. However, it should also be clear that the context of an expression matters, and thus the representational complexity has to depend on the context. This is the problem we solve.

It is not our intent to discuss the interpretation of expressions (as functions) within a context - the reader is directed to texts on Logic [2] and on Denotational Semantics [22] for the relevant background. For a good exposition on expression equivalence, see the work of Davenport and co-authors, for example [10, 1]. We instead wish to concentrate of showing how it is possible to properly define the informal notion of the *length of the description* of an expression so as to get a powerful tool to encapsulate the notion of simplification of the representation of an expression.

The main contribution of this paper is to show how to combine the theory of Minimum Description Length (MDL) [21], and that of Biform Theories to give a clear definition of the problem of simplification. Furthermore, as simplification is in general an undecidable problem [6], our theory gives guidelines to system builders on how to architect their simplifier(s) from various transformation heuristics and specialized (semi-)decision procedures.

This paper is organized as follows: the next section gives a quick introduction to Kolmogorov Complexity and MDL, which are the theoretical tools used to define "simplicity". Section 3 defines biform theories, which give the context in which to understand the notion of simplicity. The results in those two section are then used in section 4 to define a coherent theory of simplification of expressions. In section 5 we give an application of this theory to "magic constants" as found in implementations of simplification routines in CASes, followed by a description of part of Maple's implementation of a simplifier augmented with comments relating our theory and the implementation details. We finish with some conclusions and outline further work to be done using these concepts.

The author wishes to thank Bill Farmer for many fruitful conversations on material relating to this paper. Comments by Freek Wiedijk on a previous draft improved the presentation of the material. Further comments by an anonymous referee were also very useful. This paper grew out of the author's desire to build a theoretical framework which could justify the work of (amongst others) Michael Monagan and Edgardo Cheb-Terrab on Maple's simplify command.

2. COMPLEXITY

> "Nulla pluralitas est ponenda nisi per rationem vel experiantiam vel auctoritatem illius, qui non potest falli nec errare, potest convivi."
>
> (A plurality should only be postulated if there is some good reason, experience or infallible authority for it.)
>
> - William of Ockham (c. 1285 - c. 1349)

Out of the desire to define a stable notion of information content as well as universal notions of randomness, several people (Shannon, Kolmogorov, Rissanen, Solomonoff, and Chaitin to name a few, see [18] for a complete treatment) have developed theories of complexity of data. This section will outline the main tenets of these theories, and the next section will show how these apply to the problem of simplification of expressions in Computer Algebra Systems.

Let us first remind the reader that although in CASes we often wish to represent, via expressions, uncomputable functions, we still want to perform computations on those representations. Thus it makes sense to restrict all discussions to computable expressions, even though those expressions frequently represent formally uncomputable functions.

2.1 Kolmogorov Complexity

This subsection follows section 2.1 of [18] very closely, where the interested reader can find a much more thorough discussion of the issues. Let $\langle\cdot\rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a standard recursive bijective pairing function mapping the pair (x, y) to the singleton $\langle x, y\rangle$.

To set the stage, we first need a fundamental result on partial recursive functions.

Definition 2. *Let x, y, p be natural numbers. Any partial recursive function ϕ, together with p and y such that $\phi(\langle y, p\rangle) = x$ is a* description *of x. The* complexity *C_ϕ of x* conditional *to y is defined by*

$$C_\phi(x|y) = \min\{\text{length}(p) : \phi(\langle y, p\rangle) = x\},$$

and $C_\phi(x|y) = \infty$ if there are no such p. We call p a program *to compute x by ϕ given the input y.*

Theorem 1. *There is a universal partial recursive function ϕ_0 for the class of partial recursive functions to compute x given y. Formally this says that $C_{\phi_0}(x|y) \leq C_\phi(x|y) + c_\phi$ for all partial recursive functions ϕ and all x and y, where c_ϕ is a constant depending on ϕ but not on x or y.*

From this theorem, it is easy to derive that, given two such universal functions ψ, ψ', there exists a constant $c_{\psi,\psi'}$ such that

$$|C_\psi(x|y) - C_{\psi'}(x|y)| \leq c_{\psi,\psi'}.$$

In other words, even though neither length is necessarily optimal, they are equal up to a fixed constant, for all x and y. This allows us to make the following definition.

DEFINITION 3. *Fix a universal ϕ_0, and dispense with the subscript by defining the* conditional Kolmogorov complexity $C(\cdot|\cdot)$ *by*

$$C(x|y) = C_{\phi_0}(x|y).$$

This particular ϕ_0 is called the reference function *for C. We also fix a particular Turing machine U that computes ϕ_0 and call U the* reference machine*. The* unconditional Kolmogorov complexity $C(\cdot)$ *is defined by*

$$C(x) = C(x|0).$$

To be precise about our intent, we will regard U as being chosen to be a universal Turing machine given as either the programming languages Maple or Mathematica [as both of these systems are Turing complete!]. In other words, we fix U as a basic programming language, but we explicitly want to allow for conservative extensions, and study their effects. In other words, what effect (if any) does allowing the addition of new definitions and subroutines (new "library" code) have on the representational complexity of expressions in a system?

There is one severe impediment to using $C(x)$: it is not computable! It is however approximable by partial recursive functions (see section 2.3 in [18] for further details on these points, as well as the references therein).

2.2 Minimum Description Length

It is a deep and extremely useful fact that the shortest effective description of an object x can be expressed in terms of a *two-part code*: the first part describing an appropriate Turing machine and the second part describing the program that interpreted by the Turing machine reconstructs x. By examining the proof of theorem 1, it is possible to transform the definition of Kolmogorov complexity into (essentially)

$$C(x) = \min\{\text{length}(T) + \text{length}(p) : T(p) = x\},$$

where we are minimizing over all Turing machines, and we use a standard self-delimiting encoding of a Turing machine program T to compute its length. The above emphasizes the two-part code decomposition of x into what are called its regular part (encoded in T) and its random aspects (encoded in p).

For our purposes however, we wish to regard T as describing the space of *models*, and p as being an index into that model space which corresponds to x. In the works of J.J. Rissanen and of C.S. Wallace and coauthors, this has been developed into the

> **Minimum Description Length Principle.** Given a sample of data and an effective enumeration of the appropriate alternative theories to explain the data, the best theory is the one that minimizes the sum of
>
> - the length, in bits, of the description of the theory;
> - the length, in bits, of the data when encoded with the help of the theory.

In other words, if there are regularities present in the data which can be extracted ("factored out"), then the theory which gives rise to the most overall compression is taken as the one that most likely explains the data. Minimum Description Length (MDL) is based on striking a balance between regularity and randomness in the data.

The crucial aspect of MDL to remember is that it relies on an *effective enumeration of the appropriate alternative theories* rather than on the complete space of partial recursive functions. This makes MDL much more amenable to applications than pure Kolmogorov complexity. For a much more thorough overview of (ideal) MDL, the reader should consult section 5.5 of [18]; for a review of "modern" MDL, Grünwald's thesis [16] is recommended. Figure 1 shows a typical result that one gets when applying this theory to noisy data—the last graph is of a third degree polynomial. It is also worth pointing out that there is a somewhat different theory with similar results: Minimum Message Length [24].

There is one important difference between classical MDL and our own use: MDL tries to find the simplest model that explains a set of inexact data, whereas we have only *one exact* data point. But, as we will see later in section 4, this one data point corresponds to a whole equivalence class of representations, and so it makes sense to understand the data set as varying over this equivalence class. Applying MDL to expressions in context means that we seek to minimize the sum of the size of the representation of an expression in a context and the size of a representation of that context.

3. BIFORM THEORIES

At the heart of this work lies the notion of a "biform theory", which is the basis for FFMM, a Formal Framework for Managing Mathematics [13]. The form of this notion is essentially the one used in [5] for applications to trustable communications between mathematical systems. Informally, a biform theory is simultaneously an axiomatic and an algorithmic theory.

3.1 Logics

A *language* is a set of typed expressions. The types include $*$, which denotes the type of truth values. A *formula* is an expression of type $*$. For a formula A of a language L, $\neg A$, the negation of A, is also a formula of L. A *logic* is a set of languages with a notion of logical consequence. If $\mathbf{K}$ is a logic, L is a language of $\mathbf{K}$, and $\Sigma \cup \{A\}$ is a set of formulas of L, then $\Sigma \models_{\mathbf{K}} A$ means that A is a logical consequence of Σ in $\mathbf{K}$.

3.2 Transformers and Formuloids

Let L_i be a language for $i = 1, 2$. A *transformer* Π from L_1 to L_2 is an algorithm that implements a partial function $\pi : L_1 \rightharpoonup L_2$. For $E \in L_1$, let $\Pi(E)$ mean $\pi(E)$, and let $\mathsf{dom}(\Pi)$ denote the domain of π, i.e., the subset of L_1 on which π is defined. For more on transformers, see [12, 13].

A *formuloid* of a language L is a pair $\theta = (\Pi, M)$ where:

1. Π is a transformer from L to L.
2. M is a function that maps each $E \in \mathsf{dom}(\Pi)$ to a formula of L.

M is intended to give the *meaning* of applying Π to an expression E. $M(E)$ usually relates the input E to the output $\Pi(E)$ in some way; for many transformers, $M(E)$ is the equation $E = \Pi(E)$, which says that Π transforms E into an expression with the same value as E itself.

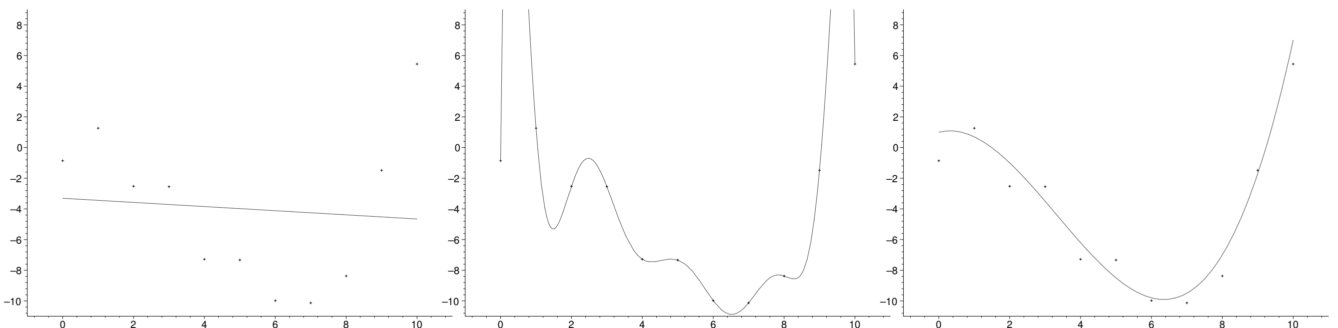

Figure 1: Lowest model complexity fit (line), best polynomial fit and MDL fit for some data

The *span* of θ, written $\mathsf{span}(\theta)$, is the set

$$\{M(E) \mid E \in \mathsf{dom}(\Pi)\}$$

of formulas of L. Thus a formuloid has both an *axiomatic meaning*—its span—and an *algorithmic meaning*—its transformer. The purpose of its span is to assert the truth of a set of formulas, while its transformer is meant to be a deduction or computation rule.

3.3 Biform Theories

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

1. $\mathbf{K}$ is a logic called the *logic* of T.

2. L is a language of $\mathbf{K}$ called the *language* of T.

3. Γ is a set of formuloids of L called the *axiomoids* of T.

The *span* of T, written $\mathsf{span}(T)$, is the union of the spans of the axiomoids of T, i.e.,

$$\bigcup_{\theta \in \Gamma} \mathsf{span}(\theta).$$

A is an *axiom* of T if $A \in \mathsf{span}(T)$. A is a *(semantic) theorem* of T, written $T \models A$, if

$$\mathsf{span}(T) \models_{\mathbf{K}} A.$$

A *theoremoid* of T is a formuloid θ of L such that, for each $A \in \mathsf{span}(\theta)$, $T \models A$. Obviously, each axiomoid of T is also a theoremoid of T. An axiomoid is a generalization of an axiom; an individual axiom A (in the usual sense) can be represented by an axiomoid (Π, M) such that $\mathsf{dom}(\Pi) = \{A\}$ and $M(A) = A$.

T can be viewed as simultaneously both an *axiomatic theory* and an *algorithmic theory.* The axiomatic theory is represented by

$$T_{\rm axm} = (\mathbf{K}, L, \{M(E) \mid \exists \Pi . (\Pi, M) \in \Gamma \text{ and } E \in \mathsf{dom}(\Pi)\}),$$

and the algorithmic theory is represented by

$$T_{\rm alg} = (\mathbf{K}, L, \{\Pi \mid (\Pi, M) \in \Gamma \text{ for some } M\}).$$

Let $T_i = (\mathbf{K}, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. T_2 is an *extension* of T_1, written $T_1 \le T_2$, if $L_1 \subseteq L_2$ and $\Gamma_1 \subseteq \Gamma_2$. T_2 is a *conservative extension* of T_1, written $T_1 \trianglelefteq T_2$, if $T_1 \le T_2$ and, for all formulas A of L_1, if $T_2 \models A$, then $T_1 \models A$. Note that $\le$ and $\trianglelefteq$ are partial orders.

3.4 Translations and Interpretations

Let $\mathbf{K}_i$ be a logic and $T_i = (\mathbf{K}_i, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. A *translation* from T_1 to T_2 is a transformer Φ from L_1 to L_2 that:

1. Respects types, i.e., if E_1 and E_2 are expressions in L_1 of the same type and $\Phi(E_1)$ and $\Phi(E_2)$ are defined, then $\Phi(E_1)$ and $\Phi(E_2)$ are also of the same type.

2. Respects negation, i.e., if A is a formula in L_1 and $\Phi(A)$ is defined, then $\Phi(\neg A) = \neg\Phi(A)$.

T_1 and T_2 are called the *source theory* and the *target theory* of Φ, respectively. Φ is *total* if $\Phi(E)$ is defined for each $E \in L_1$. Φ *fixes* a language L if $\Phi(E) = E$ for each $E \in L$.

An *interpretation* of T_1 in T_2 is a total translation Φ from T_1 to T_2 such that, for all formulas $A \in L_1$, if $T_1 \models A$, then $T_2 \models \Phi(A)$. An interpretation thus maps theorems to theorems. (Since any translation respects negation, an interpretation also maps negated theorems to negated theorems.) A *retraction* from T_2 to T_1 is an interpretation Φ of T_2 in T_1 such that $T_1 \le T_2$ and Φ fixes L_1.

LEMMA 1. *Let Φ_1 be a retraction from T_2 to T_1 and Φ_2 be a retraction from T_3 to T_2. Then $\Phi_1 \circ \Phi_2$ is a retraction from T_3 to T_1.*

PROOF. Let $\Phi = \Phi_1 \circ \Phi_2$. We first need to prove that Φ is an interpretation. Φ is clearly total. Assume $T_3 \models A$. Then $T_2 \models \Phi_2(A)$ since Φ_2 is an interpretation of T_3 in T_2. In turn, $T_1 \models \Phi_1(\Phi_2(A))$, i.e., $T_1 \models \Phi(A)$ since Φ_1 is an interpretation of T_2 in T_1. Hence, Φ is an interpretation of T_3 in T_1.

By transitivity of $\le$, since $T_1 \le T_2$ and $T_2 \le T_3$, $T_1 \le T_3$.

Finally, we need to prove that Φ fixes L_1. Let $E \in L_1 \subseteq L_2 \subseteq L_3$. $\Phi_2(E) = E$ since Φ_2 is a retraction from T_3 to T_2 and $E \in L_2$. Similarly, $\Phi_1(\Phi_2(E)) = \Phi_1(E) = E$ since Φ_1 is a retraction from T_2 to T_1 and $E \in L_1$. Hence $\Phi(E) = E$ and Φ fixes L_1. □

PROPOSITION 1. *If Φ is a retraction from T_2 to T_1, then $T_1 \trianglelefteq T_2$.*

PROOF. Let A be a formula of the language of T_1 such that $T_2 \models A$. We must show that $T_1 \models A$. By definition, (1) Φ is an interpretation of T_2 in T_1 and (2) Φ fixes the language of T_1. (1) implies that $T_1 \models \Phi(A)$, and (2) implies $\Phi(A) = A$. Therefore, $T_1 \models A$. □

Along the same lines, it is possible to define the union and the intersection of theories. One must be careful, as the union of two theories may produce a trivial (inconsistent) theory, but there are no essential technical difficulties involved.

4. SIMPLIFICATION OF EXPRESSIONS

Let $T = (\mathbf{K}, L, \Gamma)$ be a biform theory where

1. the language L contains the syntactic representation of a programming language which is Turing complete,
2. there exists a total length function length : $L \to \mathbb{N}$ compatible with the subexpression relation, in other words if E_1 is a proper subexpression of E then
$$\text{length}(E_1) < \text{length}(E),$$
3. all formuloids $\theta = (\Pi, M)$ are such that the algorithm of Π is expressible in L,
4. Γ is finite, and the domain of the axiomoids of Γ are finite.
5. Γ always contains at least the axiomoid corresponding to the identity transformer.

We will call such a biform theory *reflexive*. Let $\sim$ be a relation on L; we will interpret this relation as being the "means the same thing as" relation. We explicitly refrain from defining this relation. Our notion of simplification will be parametrized by this relation; one could choose $\sim$ to be equality, or such that $1 \sim \frac{x}{x}$ even as denotations of total functions on the reals.

DEFINITION 4. *Let e_1, e_2 be two expressions of the language L of T. We say that $e_1 < e_2$ if* $\text{length}(e_1) < \text{length}(e_2)$ *and $e_1 \sim e_2$. Let c be a positive integer. We say that e_1 and e_2 are c-equivalent, denoted $e_1 \sim_c e_2$ if $e_1 \sim e_2$ and* $|\text{length}(e_1) - \text{length}(e_2)| \leq c$.

Since our theories T are quite powerful, the coding does not make a huge difference. But since it can make a difference for very simple expressions, it is generally better to consider simplification of expressions only up to c-equivalence, as the notion of "simpler" is not stable enough for c-equivalent expressions. Our experience seems to show that taking c between 50 and 100 seems to lead to a meaningful notion of "simpler".

DEFINITION 5. *Let e be an expression of the language L of T. The (absolute) complexity of e is*
$$C(e) = \min\{\text{length}(p) \ : \ p() = e\}$$
where p ranges over all nullary programs in L.

It is important to remark that if e is essentially random, then the program `() -> e` will be the one to achieve this minimum. The previous two definitions are the natural ones coming directly from Kolmogorov complexity. However, although intuitively clear, they are not very helpful in practice, which is why we have to turn to MDL.

From now on, to make the exposition simpler, we will assume that we have a logic $\mathbf{K}$ and a fixed language L. Assume that we have a finite set of reflexive biform theories $T_i = (\mathbf{K}, L, \Gamma_i)$ where the Γ_i form a complete lattice (with union and intersection for join and meet), and that furthermore, if $\Gamma_i \subseteq \Gamma_j$ then Γ_j must be a conservative extension of Γ_i. This is not a very stringent restriction: it simply corresponds to proper modular construction of mathematical software, where adding new modules does not modify the meaning of previously defined notions. Denote by $\mathfrak{T}$ such a lattice of theories.

Let $\langle \Pi_1, \Pi_2, \ldots \rangle$ be a recursively enumerable sequence of transformers from a reflexive biform theory T, which correspond to a sequence $\langle \Theta_1, \Theta_2, \ldots \rangle$ of formuloids of T. Furthermore, suppose that given an expression $e \in L$, not only is $e \sim \Pi_i(e)$ for all i, but in fact that $e = \Pi_i(e)$ is a theorem of some member of $\mathfrak{T}$. Call $e_i = \Pi_i(e)$ a *reachable* expression. It is instructive to think of these transformers as the (composition of) all the basic term rewrites that preserve the meaning of expressions, like $\sin^2(x) + \cos^2(x) = 1$ and so on. It is very important the this sequence be recursively enumerable, otherwise none of the theory of Kolmogorov Complexity applies.

DEFINITION 6. *Let e be an expression of L, and $\Theta = (\Pi, M)$ an axiomoid of some $T_j \in \mathfrak{T}$. Then there exists a smallest reflexive biform theory $T_i \in \mathfrak{T}$ such that $e = \Pi(e)$ is a theorem of T_i. Denote this as* $\text{theory}(e, \Pi) = T_i$. *The theory of e,* $\text{theory}(e)$ *is defined to be* $\text{theory}(e, \mathsf{Identity})$.

Note that an expression like $sin(x) = sin(x)$ is only a theorem of those T_i which have enough machinery to first show the expression in question denotes a valid term in that theory. For example $1/0 = 1/0$ is rarely a theorem since $1/0$ is usually non-denoting.

In the spirit of MDL, we are now ready to define the notion of length we will use:

DEFINITION 7. *Let e be an expression of L. The length of e in $\mathfrak{T}$ is defined to be*
$$\text{length}_{\mathfrak{T}}(e) = \text{length}(e) + \text{length}(\text{theory}(e)),$$
where the length of a theory is defined to be the sum of the length of the representation in L of all the spans of all the axiomoids of $\text{theory}(e)$.

PROPOSITION 2. $\text{length}_{\mathfrak{T}}(e)$ *is well-defined.*

PROOF. First, $\text{length}(e)$ is clearly well-defined. Since $\mathfrak{T}$ is formed from a complete lattice of biform theories $\text{theory}(e)$ is also well-defined. Furthermore, we assumed that our theories have finite Γ_i and the functions M are representable as formulas of L—which means that $\text{length}(\text{theory}(e))$ is well-defined and finite. □

The length of an expression e with respect to a set of theories is essentially the length of the axiomatic description of the theory necessary to completely describe e, plus the length of e, as encoded with the help of that theory. To completely describe e, it is necessary to be able to prove that e denotes a value.

It is important to note that although we *use* the transformers Π constantly, their representation length is not used at all in the definition of the length of an expression. This is because we are not interested in computational complexity issues, and such issues have very significant impact on the size of the representation of the transformers. In other words, the length of expressions only depends on the size of the generators of the axiomatic part of the theory of that expression.

Putting all of these ideas together, this leads naturally to

DEFINITION 8. *Let e be an expression of L, $\langle \Pi_0, \Pi_1, \Pi_2, \ldots \rangle$ (where $\Pi_0 = \texttt{Identity}$) be a recursively enumerable*

sequence of transformers from some reflexive theory family $\mathfrak{T}$. *Let* $e_j = \Pi_j(e)$. *The* simplest *reachable member from this family is the* e_j *which minimizes* $\text{length}_{\mathfrak{T}}$.

If we pick the sequence of transformers as $\langle \texttt{Identity}, \Pi \rangle$ where Π is idempotent, then for an expression e, simplest in this context means choosing between e and $\Pi(e)$ depending on $\text{length}_{\mathfrak{T}}$. Furthermore if $\text{theory}(e) = \text{theory}(\Pi(e))$, then this notion further reduces to that implied by definition 4.

5. APPLICATIONS

We will first go through two example applications of the above theory, to understand what this means in specific cases. We then explain what this means for the architecture of simplification routines in Computer Algebra Systems.

5.1 Examples

Let us first study a rather simple example, but one which can be easily understood, and which in fact displays quite a number of the issues rather well. Suppose we want to know when 2^n, with n an explicit positive integer, should be displayed as is or as an explicit integer. Clearly 4 is simpler than 2^2, yet 2^{10000} is intuitively simpler than the integer it represents.

Fix L to be the language of Maple, and $\mathbf{K}$ an appropriate logic. For T, pick Γ to contain only two axiomoids, the identity and one which evaluates integer expressions built from integers and the operations $+$, $*$, $-$ and ^. We will encode our integers in base 2, and measure length in bits; for technical issues (see [18] for the details), we encode our expressions using self-delimiting bit strings. Note that in this example, we are in the situation described in the last paragraph of section 4 where we have only one idempotent transformer and one fixed theory. The integer 2^n takes $2n + 2$ bits to represent using a self-delimiting encoding (the length of the complete integer, plus its length in unary, plus delimiters). The expression 2^n takes $2\lceil \log_2(n) \rceil + 2 + 9$ bits where we use 9 extra bits to represent the function call $\hat{}(2, n)$. In other words we wish to know when

$$2n + 2 > 2\lceil \log_2(n) \rceil + 11.$$

An easy computation shows that this happens whenever $n \geq 8$. With the particular encoding we have chosen, this says that 2^7 is more complex than 128 but that 2^8 is simpler than 256.

It is also possible to analyze more complex examples fully, in a very parametric fashion:

PROPOSITION 3. *Let* T_1 *be a theory of expanded polynomials, and* T_2 *be a conservative extension of* T_1 *which adds machinery for Chebyshev polynomials. Let* $n \in \mathbb{N}$ *and* x *be a symbol in* T_1, $e_2 = ChebyshevT(n, x)$ *and* e_1 *be the expanded polynomial (in* T_1*) such that* $e_1 \sim e_2$. *Then there exists a (computable) constant* C *such that if* $n > C$ *then* $e_2 < e_1$. C *depends only on* $\text{length}_{\mathfrak{T}}(T_2) - \text{length}_{\mathfrak{T}}(T_1)$, *and the constants appearing in the encodings of* e_1 *in* T_1 *and* e_2 *in* T_2.

PROOF. e_1 can be encoded using at most $a_1 n^2 + a_2 ln(n) + a_3$ bits in T_1 (the coefficients grow exponentially with n, thus their size grows linearly with n); e_2 needs at least $b_1 ln(n) + b_2$ bits in T_2. Let T_1 be encoded using c_1 bits and T_2 using $c_1 + c_2$ bits. Choose C to be the largest positive real root of

$$|a_1 n^2 + (a_2 - b_1) ln(n) + (a_3 - b_2 - c_2)| = 0$$

(if it exists), or 0 otherwise. The above expression is easily seen to be real and increasing for $n > 0$, and negative for $n = 1$ if $a_1 + a_3 - b_2 - c_2 < 0$. In typical encodings, a_1 is small, a_3 and b_2 are of comparable (small) size and c_2 much larger, making the overall expression negative. □

In fact, with $a = a_1, b = a_2 - b_1, c = a_3 - b_2 - c_2$, one can even get a closed form for the above constant C:

$$C = \frac{1}{2}\sqrt{\frac{2b}{a}}\sqrt{W_{-1}(\frac{2a}{b}e^{-\frac{2c}{b}})},$$

where $W_{-1}(z)$ denotes the -1 branch of the Lambert W function [9]. The appearance of Lambert's W function is due to the fact that we are changing scales between a polynomial scale and a (simple) exponentially larger scale.

Using this theory, we can also prove a non-simplification theorem: given two explicit integers n and m, it is never the case that the algebraic expression $n + m$ is simpler than the integer q equal to $n + m$; this result is indendent of the bit representation of the explicit integers. This result does not hold anymore if either of n or m are implicit intgers, or if $+$ is replaced by $*$. In other words, representational issues alone are not sufficient to argue for an inert representation for $+$ as being absolutely necessary in a CAS (much to the author's chagrin).

5.2 Implementations

A very rough description of a simplifier is as an ordered collection of semantics-preserving expression transformations. An expression is first decomposed into its basic components (variables, special functions, operators, etc). To each of these basic components, as well as to some specific combinations of components, is associated a set of applicable transformations. These transformations are ordered, where transformers from more complicated functions (like Gauss's hypergeometric function) to simpler ones (Bessel functions, polynomials, etc) are placed first, followed by transformations that stay in the same class. These transformers are then applied in order. This is repeated, as some transformations can produce new basic components, and thus the list of applicable transformations has to be updated. Some of these transformations are heuristic in nature - in other words they may or may not produce a "simplification". Others, like the work of Monagan and Mulholland [20], could be called *structure revealing* transformations, and are deeply algorithmic; they tend to be intra-theory transformations.

For example, at a particular point in time (for Maple 9.5), `simplify` classified sub-expressions according to the following (ordered) categories:

```
CompSeq, constants, infinity, @@, @, limit, Limit,
max, min, polar, conjugate, D, diff, Diff, int,
Int, sum, Sum, product, Product, RootOf,
hypergeom, pochhammer, Si, Ci, LerchPhi, Ei, erf,
erfc, LambertW, BesselJ, BesselY, BesselK, BesselI,
polylog, dilog, GAMMA, WhittakerM, WhittakerW,
LegendreP, LegendreQ, InverseJacobi, Jacobi,
JacobiTheta, JacobiZeta, Weierstrass, trig,
arctrig, ln, radical, sqrt, power, exp, Dirac,
Heaviside, piecewise, abs, csgn, signum, rtable,
constant
```

Some of the categories contain single items (like `BesselI`), while others contain many (like `trig`). The ordering in

Maple was obtained after a large number of practical experiments [7]. In large part, the ordering is based on the idea that the currently implemented transformations from categories in the earlier parts of the list are more likely to produce results from categories in latter parts of the list; this naturally produces a lattice, which was then flattened to produce the given list. The exceptions are enabling transformations (like the ones in the `constant` and `infinity` classes), which allow many more latter transformations to be performed. Interestingly, the correspondence between this ordering and the one obtained by measuring theory length is a good match. The match at the level of pure axiomatic theories is not so good, but once the theories are augmented with all the valid transformation theorems, as one needs to do with proper biform theories that contain transformers for conversions from one form to another, the match becomes very good indeed. This points to an area where our definitions could be improved to take this effect into account. The only cases where theory and practice do not necessarily agree are in cases where the difference in length between the theories involved is small, so that the expressions involved are frequently c-equivalent. In other words, this decomposition into basic components is, in the context of the mathematical functions that simplify deals with, quite a good proxy for the underlying axiomatic theories involved.

Here and there, there are "magic constants", chosen completely at the whim of the developer, which control whether a particular transformation routine will in fact expand a function (like binomial) or not. For example, the Bessel functions will automatically expand into a trigonometric form (ie $J_{\nu/2}(z)$ can be rewritten using only sin and cos for integer ν). But this is done only if $|\nu/2| \leq 10$; similarly, `simplify` will reduce $J_\nu(z)$ using `BesselJ`'s recurrence relation, but only if $|\nu| < 100$. The author previously did not believe in such magic constants, as there did not seem to be a reasonable way to choose them, although the pragmatism behind the approach was very appealing. At least now it might be possible to objectively choose these constants.

6. CONCLUSIONS AND FURTHER WORK

We have presented a framework for the simplification of representations of expressions which precisely defines when to choose between two particular semantically equivalent representations of an expression. This is fundamentally inspired by the theories of Kolmogorov Complexity and minimum description length. To be able to apply these theories to the mixed computational-axiomatic formalism of expressions in a Computer Algebra System, we have used biform theories, which were invented expressly for this purpose of mixing deduction and computation. We added a certain set of reflexivity axioms to the base biform theories to refine the framework to one immediately applicable to MDL and current CASes. These axioms were needed to insure that we had a uniform language which could express formulas and algorithms, and that these formulas and algorithms could be effectively enumerated in some cases of interest. Effective enumeration is one of the key ingredients which makes the theory of Kolmogorov Complexity as powerful as it is.

An interesting aspect of this work that we have not had a chance to explore is that changes in knowledge affect the axiomatization of theories, which thus affects the length of the expressions associated with those theories. Typically, this serves to reduce the overall complexity of expressions. A leading example is the explosion of work on using holonomy as a unifying theory for special functions [25, 8], which has had a tendency to make hitherto very complex expression seem quite a bit simpler; our theory should help make this intuition somewhat more quantifiable.

Another issue is that of computational complexity. Our approach explicitly avoids such issues, both for computation of the length as well as dealing with the fact that asymptotically computationally efficient algorithms for arithmetic (like polyalgorithms for fast integer multiplication) tend to make implementations much larger. Certainly it does not seem wise to penalize expressions because they are part of a computationally more efficient theory; however we do not yet know how to adjust our framework to properly account for this. A balanced approach, like that of MDL, seems best.

7. REFERENCES

[1] J. Beaumont, R. Bradford, and J. H. Davenport. Better simplification of elementary functions through power series. In *Proceedings of the 2003 international symposium on symbolic and algebraic computation*, pages 30–36. ACM Press, 2003.

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

[3] M. Bronstein. Simplification of real elementary functions. In *Proceedings of ISSAC 1989*, pages 207–211, 1989.

[4] B. Buchberger and R. Loos. Algebraic simplification. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra - Symbolic and Algebraic Computation*, pages 11–44. Springer-Verlag, New York, 1982.

[5] J. Carette, W. Farmer, and J. Wajs. Trustable communication between mathematical systems. In T. Hardin and R. Rioboo, editors, *Proceedings of Calculemus 2003*, pages 58–68, Rome, 2003. Aracne.

[6] B. Caviness. On canonical forms and simplification. *J. ACM*, 17(2):385–396, 1970.

[7] E. Cheb-Terrab. personal communication.

[8] F. Chyzak and B. Salvy. Non-commutative elimination in ore algebras proves multivariate identities. *Journal of Symbolic Computation*, 26(2):187–227, 1998.

[9] R. Corless, G. Gonnet, D. Hare, D. Jeffrey, and D. Knuth. On the Lambert W function. *Advances in Computational Mathematics*, 5:329–359, 1996.

[10] R. M. Corless, J. H. Davenport, David J. Jeffrey, G. Litt, and S. M. Watt. Reasoning about the elementary functions of complex analysis. In *Proceedings AISC Madrid*, volume 1930 of *Lecture Notes in AI*. Springer, 2000.

[11] J. Davenport, Y. Siret, and E.Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, 1988.

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

[13] W. M. Farmer and M. v. Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*,

2003. In the forthcoming special issue: B. Buchberger, G. Gonnet, and M. Hazewinkel, eds., *Mathematical Knowledge Management*.

[14] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer, Boston/Dordrecht/London, 1992.

[15] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook: Foundations, Applications, Systems*. Springer-Verlag, 2003.

[16] P. Grünwald. *The Minimum Description Length Principle and Reasoning under Uncertainty*. PhD thesis, CWI, 1998.

[17] S. Landau. Simplification of nested radicals. *SIAM Journal on Computing*, 14(1):184–195, 1985.

[18] M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Berlin, 1997.

[19] J. Moses. Algebraic simplification: a guide for the perplexed. *Communications of the ACM*, 14(8):548–560, 1971.

[20] J. Mulholland and M. Monagan. Algorithms for trigonometric polynomials. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pages 245–252. ACM Press, 2001.

[21] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[22] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. W. C. Brown, Dubuque, Iowa, 1986.

[23] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003.

[24] C. S. Wallace and D. Dowe. Minimum message length and kolmogorov complexity. *Computer Journal (special issue on Kolmogorov complexity)*, 42(4):270–283, 1999.

[25] D. Zeilberger. A holonomic systems approach to special function identities. *J. Comput. Appl. Math.*, 32:321–368, 1990.

[26] R. Zippel. Simplification of expressions involving radicals. *J. of Symbolic Computation*, 1(1):189–210, 1985.