Interactive algebraic manipulation

Geoffrey Irving*

April 13, 2014

1 Introduction

Our goal is a system for manipulating algebraic formulas interactively. Here interactively means that the user will be directing most of the changes, with assistance from the computer in making the changes quickly and verifying their correctness. In contrast, typical computer algebra systems such as Mathematica have quite powerful automatic tools for simplification, transformation, etc., but these tools do not always achieve the desired goal. A system oriented towards interactivity has two key strengths:

1. **Understanding:** A system which presents an entire derivation as a single step provides no insight as to how this transformation was achieved. Besides the educational advantages, presenting the derivation as a transparent chain of simple steps provides insight into generalizability, numerical stability, physical meaning, etc.

2. **Flexibility:** Taking simplification as an example, there are a wide range of goals that the user may wish to achieve in manipulating a formula, including efficient computability, numerical stability, or suitability for some followup transform. Although it may be possible to program the `Simplify` routine of a computer algebra system to achieve one of these goals, an interactive system can achieve any of them even if the user only has a vague idea of the goal in advance.

2 Plan

The user starts with a formula (or possibly several formulae), and makes a series of changes or *moves*. Each move is both checked for validity by the program, and additionally hinted so that it can be made quicker. For example, the user can drag a term from one place to another; once the term is in motion, only locations which produce a correct result will be available for selection. Similarly, moves will be disambiguated based on correctness: if the user drags an additive term, it will switch to subtractive if that is required for validity.

An earlier version of this plan relied on an algorithm for predicates of the form

$$f(x) = 0 \implies g(x) = 0$$

Although this would have simplified the outer algorithm levels, the required algorithmic complexity seemed dangerously high on further reflection: although implication is easy for polynomial formulas, the complexity would likely grow by an order of magnitude if divisions are

*Otherlab, `irving@naml.us`

included, and by several more with square roots. As we will see below, all the moves required for our example derivation can be performed using only predicates of the form

$$f(x) = 0$$

That is, by taking advantage of interactivity, it suffices to use identity testing rather than implication testing. The advantage of pure identity testing is huge, both in terms of speed and flexibility. Speed can be spent on simple programming or more powerful outer level algorithms to reduce the required input from the user. Identity testing amounts to plugging in numbers and checking whether the formula holds, and is therefore easily generalizable to more complicated expressions or programs. For details on randomized testing of identities, implications, etc., see [3, 4, 1].

Rather than guess the set of moves required to make the program useful, we will work through an example derivation first (arising from work on Othermill), and then discuss how each type of move can be implemented using identity testing.

3 Example

As a concrete example of a derivation, consider a ball mill of radius $R > 0$ sweeping along a segment from (u, z) to $(u + v, z + w)$, where $u, v \in \mathbb{R}^2$ and $z, w \in \mathbb{R}$. At time $t \in [0, 1]$, the tip of the mill is at position $(u + tv, z + tw)$. What time t maximizes the height of the mill over the origin?

We will split the derivation into several parts. First, we evaluate the height of the mill at a horizontal distance r away from the tip. Since the mill is a sphere centered at $(0, R)$, this height $g(r)$ satisfies

$$\begin{aligned}
r^2 + (g - R)^2 &= R^2 \\
(g - R)^2 &= R^2 - r^2 && \text{(Move } +r^2 \text{ to RHS)} \\
g - R &= \pm\sqrt{R^2 - r^2} && \text{(Move } \sqrt{\cdot} \text{ to RHS)} \\
g &= R \pm \sqrt{R^2 - r^2} && \text{(Move } -R \text{ to RHS)} \\
g &= R - \sqrt{R^2 - r^2} && \text{(Choose positive sign)} \\
g_r &= \frac{r}{\sqrt{R^2 - r^2}} && \text{(Differentiate w.r.t. } r\text{)}
\end{aligned}$$

Second, treating $g(r)$ temporarily as an unknown function of r, the height above the origin at time t satisfies

$$\begin{aligned}
h &= g(|u + tv|) + z + tw \\
h_t &= g_r(|u + tv|)\frac{v \cdot (u + tv)}{|u + tv|} + w && \text{(Differentiate w.r.t. } t\text{)}
\end{aligned}$$

Finally, we substitute in our equation for g_r and set $h_t = 0$ to get

$$\frac{|u+tv|}{\sqrt{R^2-|u+tv|^2}}\frac{v\cdot(u+tv)}{|u+tv|}+w=0 \qquad \text{(Substitute for } g_r\text{)}$$

$$\frac{v\cdot(u+tv)}{\sqrt{R^2-|u+tv|^2}}+w=0 \qquad \text{(Simplify)}$$

$$\frac{v\cdot(u+tv)}{\sqrt{R^2-|u+tv|^2}}=-w \qquad \text{(Move } +w \text{ to RHS)}$$

$$v\cdot(u+tv)=-w\sqrt{R^2-|u+tv|^2} \qquad \text{(Move } /\sqrt{\cdot} \text{ to RHS)}$$

$$u\cdot v+t|v|^2=-w\sqrt{R^2-|u+tv|^2} \qquad \text{(Distribute } v\cdot \text{ on RHS)}$$

$$(u\cdot v+t|v|^2)^2=w^2(R^2-|u+tv|^2) \qquad \text{(Square both sides)}$$

$$(u\cdot v+t|v|^2)^2=w^2(R^2-|u|^2-2tu\cdot v-t^2|v|^2) \qquad \text{(Expand square on RHS)}$$

$$(u\cdot v)^2+2t|v|^2u\cdot v+t^2|v|^4=w^2(R^2-|u|^2-2tu\cdot v-t^2|v|^2) \qquad \text{(Expand square on LHS)}$$

$$(u\cdot v)^2-w^2R^2+w^2|u|^2+2t|v|^2u\cdot v+t^2|v|^4=w^2(-2tu\cdot v-t^2|v|^2) \qquad \text{(Move term to LHS)}$$

$$(u\cdot v)^2-w^2R^2+w^2|u|^2+2t|v|^2u\cdot v+2tw^2u\cdot v+t^2|v|^4+t^2w^2|v|^2=0 \qquad \text{(Same)}$$

$$(u\cdot v)^2-w^2R^2+w^2|u|^2+2tu\cdot v(|v|^2+w^2)+t^2|v|^2(|v|^2+w^2)=0 \qquad \text{(Same)}$$

which is a quadratic equation for t.

4 Operations

I've labeled each step in the example with the operation performed, so we must consider whether each operation can be implemented efficiency. Computational efficiency shouldn't be a problem; the real question is how quickly the user can specify each of the required operations.

4.1 Move an operation to a different place

Here the user grabs a portion of the equation acting on another part of the equation and moves it to act on a different part of the equation. Thinking in terms of action rather than the portion itself may be useful since it includes *how* the equation should act, whether additively, multiplicatively, etc. Once moved, the action might change, such as from addition to subtraction. The critical quantities are how long it takes the user to select a term to move, and how long it takes them to specify where to put it. The latter can be dramatically reduced if the only selectable places are those which form an equivalent equation, which is doable as long as the primitive implication questions can be resolved quickly enough.

Let's assume that a typical equation has 100 nodes in its expression tree. If the user selects a term and begins dragging, we need to check in real time which of these places the term can be correctly inserted to provide dragging hints. For each place, we need to try several different kinds of operations: let's say that 10 variants need be checked. Most variants are wrong, and can be rigorously proven wrong using a single evaluation. If square roots or other multivalued functions appear, several evaluations might be required to find the incorrect branch; let's say 10 evaluations are required. Thus, checking whether the term can be correctly placed in one location takes around

$$100 \text{ terms} \cdot 10 \text{ variants} \cdot 10 \text{ evaluations} \cdot 10 \text{ overhead} = 10^5 \text{ operations}$$

If Javascript adds an additional 10× overhead, and pessimism adds a final 10×, we're at around 10 ms per location check, which means we can check dozens of locations at once without the user noticing any lag.

The above time estimate was made assuming that identity testing suffices, which remains to be established. All of the moves in our example have the form

$$f(x) = y \quad \mapsto \quad x = g(y)$$

where g is the inverse of f. Even if we don't know the details of f and g, the move can be checked via the identities

$$f(g(x)) = g(f(x)) = x$$

Another kind of move that might show up is a within-formula move, such as rearranging some terms. Those kinds of moves typically don't change the value of the ancestor node, so the before and after values can be checked with identity testing. Indeed, if such a move is entirely within the left or right hand side of the equation, we can simply check whether the side that changed is the same.

4.2 Choose positive sign

Here the user either edits the formula directly or selects the sign from a dropdown menu. The main cognitive and computational load comes from the multivalued nature of square roots, which will often mean that the implications between formulas flow in only one direction. Computationally $\pm$ can be handled by simply trying both alternatives. Usually there's only one or two of these, so the slowdown is minor.

4.3 Substitute

Trivial.

4.4 Simplify

We do want some automatic simplification support, but it can be far simpler than in traditional systems. Most simplications can be achieved by trying all pairs of terms and checking whether we can eliminate both of them. Only identity testing is required, so one such simplification pass would take roughly $100^2 \cdot 100 \cdot 10 = 10^7$ operations (cheap).

4.5 Distribute

The general distributivity principle has the form

$$f(g(x, y)) = g'(f'(x), f'(y))$$

That is, a unary function commutes through a binary function, possibly with the functions changing in the process. There's no need to hard code the set of principles: if the set of functions is small, it's easy to learn all the small principles by brute force search. Missing principles could be added by the user.

4.6 Differentiate

Basic chain rule differentiation followed by a simplification pass to remove unnecessary clutter.

5 Some details

5.1 Vector math

The beauty of Schwartz-Zippel style testing is its easy extension to "more complicated" operations such as tensor algebra; indeed, the core identity testing layer doesn't see any difference. A bit of work is required to make the distributivity and differentiation layers flexible, but it should be quite practical, and any user input is easily unit tested numerically. The same goes for other operations on $\mathbb{R}^k$, in particular truncated infinitesimal series such as $x + \epsilon y$ with automatic differentiation. Ideally, we'd have a small layer allowing the user to define new types (all set-isomorphic to some $\mathbb{R}^k$) together with possibly overloaded operations and simple type inference.

5.2 Algorithmic requirements

Polynomial and rational equations are both straightforward to test for correctness, and not too much harder to test for implication. Unfortunately, the cost of implication testing jumps dramaticatically even for rational functions since the degree bound grows exponentially even through additions. For identity testing this isn't a problem, since the degree only has to be much less than the available precision, but implication testing requires operations on unary polynomials and thus degree matters more. In either case we can either use evaluation modulo a randomly chosen prime if rigorous probabilities are required, or interval arithmetic if we're lazier.

Unfortunately, square roots are significantly more complicated. If we work over a finite field, they can still be done exactly, so identity testing is still reasonably easy. However, implication testing is significantly harder, and might require transforming away from black box form. This is the main motivating for formulating the internals in terms of identity testing only.

5.3 Arithmetic

Arithmetic can be performed either in a randomly chosen finite field or over using floating point. Finite fields are great for rational functions, and as noted can be extended to small roots. However, the root extension already has a significant cost: the square root itself is at least an order of magnitude slower, and each one taken doubles the number of evaluations required since only half the field elements have square roots.

Interval arithmetic is probably a better choice. It isn't perfect: an interval evaluation can only prove that an equation is false, not true, so unlike finite fields we would lack rigorous probability estimates. However, the benefit in flexibility and generality is huge, and wins over rigor for our purposes. Rigor can be restored piecemeal later on if required.

5.4 User interface

The main challenge in the user interface is to make the initial selection of terms fast and easy. Once a term is selected, identity testing-based hinting turns onx, and choosing where the term

goes is easier. Standard drag-based selection would work, but is a bit time consuming for the user; we may be able to do better by taking advantage of knowledge of the recursive structure. For example, selections could be made by clicking on a symbol inside a term, then dragging up or down to indicate how large a term to select. Whether the tool is pleasant to use will dependent critically on the ease of selection, so it will be worth playing around with different methods.

5.5 Platform and language

All of the required computation should be fast enough to handle at interactive rates in Javascript, so the main choice is between client-only Javascript and iOS/Android. The main reason for mobile would be if touch allows faster or more intuitive selection. The ideal case would be if we can access intuitive touchscreen gestures from pure Javascript, but I'm not sure if that is possible.

6 Handling arbitrary code?

The background motivation here is verified, interactive programming. Algebraic manipulation alone is already interesting, but it's worth asking how much of this could be generalized to other types of code.

Boring answer: it depends on the code. The underlying trick that makes identity testing-based algebra work is that we know all the moves, and can design essentially perfect unit tests *for each move*. We never test the validity of the entire formula; usually the formula isn't an identity so testing it is meaningless. Instead, the specification of correctness is encoded in the string of derivations, and in our autogenerated unit tests that each move is correct.

For codebases that have simple specifications, and grow in complexity from there to handle new features, improve parallelism, or distribution over different machines, a similar style of interactive gradual programming would likely work. More importantly, it could work on part of a codebase even if the rest is out of reach: as long as a move is local, the automatically generated unit test which verifies it is also local, and need not depend on unrelated code that we don't understand.

How practical such a system would actually be could be estimated by mapping it out in the same way we worked through the small example above. The pentago codebase could likely be made to work this way, as most of the unit tests are already structured as identity tests between simple slow routines and complex fast routines [2]. A more useful but also likely example would be fast multipole-based gravity codes such as [5]. Here again the serial, slow version could be written in a few dozen lines of code, and the fast version must be gradually bootstrapped into existence with numerous tests.

References

[1] Chen, X., Kayal, N., and Wigderson, A. *Partial derivatives in arithmetic complexity and beyond*. Now Publishers Inc, 2011.

[2] Irving, G. Pentago is a first player win: Strongly solving a game using parallel in-core retrograde analysis. *arXiv preprint arXiv:1404.0743* (2014).

[3] Schwartz, J. T. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM) 27*, 4 (1980), 701–717.

[4] Tulone, D., Yap, C., and Li, C. Randomized xero testing of radical expressions and elementary geometry theorem proving. In *Automated Deduction in Geometry*. Springer, 2001, pp. 58–82.

[5] Warren, M. S. 2HOT: an improved parallel hashed oct-tree n-body algorithm for cosmological simulation. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* (2013), ACM, p. 72.