Biform Theories: Project Description[⋆]

Jacques Carette, William M. Farmer, and Yasmine Sharoda

Computing and Software, McMaster University, Canada
`http://www.cas.mcmaster.ca/~carette`
`http://imps.mcmaster.ca/wmfarmer`

April 27, 2018

Abstract. A *biform theory* is a combination of an axiomatic theory and an algorithmic theory that supports the integration of reasoning and computation. These are ideal for specifying and reasoning about algorithms that manipulate mathematical expressions. However, formalizing biform theories is challenging as it requires the means to express statements about the interplay of what these algorithms do and what their actions mean mathematically. This paper describes a project to develop a methodology for expressing and managing mathematical knowledge as a network of biform theories. It is a subproject of MathScheme, a long-term project at McMaster University to produce a framework for integrating formal deduction and symbolic computation.

We present the *Biform Theories* project, a subproject of MathScheme [6] (a long-term project to produce a framework integrating formal deduction and symbolic computation).

1 Motivation

Type $2 * 3$ into your favourite CAS, press enter, and you'll receive (unsurprisingly) 6. But what if you want to go in the opposite direction? Easy: you ask `ifactors`(6) (or `FactorInteger[6]`)[1] The Maple command `ifactors` returns a 2-element list, with the first element the unit (1 or -1), and the second element a list of pairs (encoded as two-element lists), with (distinct) primes in the first component, and multiplicity in the second. Mathematica's `FactorInteger` is similar, except that it omits the unit (and does not document what happens for negative integers). On top of that, Maple also offers `ifactor` (without the 's') that returns an abomination that displays nicely: a product of a unit and the function with name *the empty string* applied to a prime, raised to its multiplicity. This horrendous hack has the advantage of displaying the result in a way similar to what appears in elementary textbooks.

⋆ This research is supported by NSERC.

[1] depending on whether you prefer Maple or Mathematica, other CASes have similar commands.

This simple example illustrates the difference between a simple computation ($2 * 3$) and a more complex *symbolic* query, factoring. The reason for using lists-of-lists in both systems is that multiplication and powering are both functions that evaluate immediately. So that factoring 6 cannot just return $2^1 * 3^1$, as that is simply equal to 6. Thus it is inevitable that both systems must *represent* multiplication and powering in some other manner. Because `ifactors` and `FactorInteger` are so old, they are unable to take advantage of newer developments in both systems. Maple calls this feature an *inert form*, while in Mathematica it is a *hold form*, but the idea is the same: representing a computation without actually performing it. Such features are very old: even in the earliest days of Maple, one could do `5 &^256 mod 379` to very compute the answer without ever computing 5^{256} over the integers.

At this point, do I go for more examples, or do I go into biform theories?

In general, both computer algebra (CAS) and theorem proving systems (TPS) manipulate *syntactic representations* of mathematical knowledge. But they tackle the same problems in different ways. In a CAS, it is a natural question to take a polynomial p (in some representation that the system recognizes as being a polynomial) and ask to factorize it into a product of irreducible polynomials [25]. The algorithms to do this have gotten extremely sophisticated over the years [24]. In a TPS, it is more natural to prove that such a polynomial p is equal to a particular factorization, and perhaps also proving that each such factor is irreducible. Verifying that a given factorization is correct is, of course, easy. Proving that factors are irreducible can be quite hard. And even though CASes obtain information that would be helpful to a TPS towards such a proof, that information is usually not part of the output. Thus while some algorithms for factoring do produce irreducibility *certificates*, which makes proofs straightforward, these are usually not available. And the complexity of the algorithms (from an engineering point of view) is sufficiently daunting that, as far as we know, no TPS has re-implemented them.

Given that both CASes and TPSes "do mathematics", why are they so different? Basically because a CAS is based around *algorithmic theories*, which are collections of symbolic computation algorithms whose correctness has been established using pen-and-paper mathematics, while a TPS is based around *axiomatic theories*, comprised of signatures and axioms, but nevertheless representing the "same" mathematics.

continue from here

On the other hand, a computer algebra system (CAS) would use an algebraic theory to represent factorization of polynomials. It describes procedures to calculate the factorization and produce results. Yet, there is no guarantee the results produced are correct. The algorithmic approach is not concerned with specifying the procedures described. Neither representations is capable of producing results that are formally specified.

We used factoring polynomials as an example, but all syntax-based mathematical algorithms (SBMAs) suffer from the same problem. SBMAs are algorithms that produces results by manipulating the syntax of the input. Examples of SBMAs include factoring polynomials, manipulating matrices, computing derivatives and others. They are frequently used in mathematics, that it makes sense to consider reasoning about the behavior of their implementations are crucial task. To be able to perform this reasoning, we need a way to connect a piece of code to its value. In other words, we need to be able to connect syntax and semantics. Biform theories make this possible.

After presenting a piece of mathematical knowledge, there is a need to connect it to other existing pieces. We believe the best way to do that is by presenting mathematical knowledge as a biform theory graph. For example, we know that polynomials are rings. Therefore, the biform theory of polynomials should be an extension of the biform theory of rings. Adding a new operation to manipulate polynomials boils down to extending the theory of polynomials with the new operation. This way, modularity is forced as well as leveraging the information presented by adding the theory morphisms that enable transportation of results among the graph nodes (the biform theories).

More about theory graph, eg: talk about the little theories approach.

2 Background Ideas

A *transformer* is an algorithm that implements a function $\mathcal{E}^n \to \mathcal{E}$ where $\mathcal{E}$ is a set of expressions. Transformers can manipulate expressions in various ways. Simple transformers, for example, build bigger expressions from pieces, select components of expressions, or check whether a given expression satisfies some syntactic property. More sophisticated transformers manipulate expressions in mathematically meaningful ways. We call these kinds of transformers *syntax-based mathematical algorithms (SBMAs)* [11]. Examples include algorithms that apply arithmetic operations to numerals, factor polynomials, transpose matrices, and symbolically differentiate expressions with variables. The *computational behavior* of a transformer is the relationship between its input and output expressions. When the transformer is an SBMA, its *mathematical meaning* is the relationship between the mathematical meanings of its input and output expressions.

A *biform theory* T is a triple (L, Π, Γ) where L is a language of some underlying logic, Π is a set of transformers that implement functions on expressions of L, and Γ is a set of formulas of L [3,9,16]. L includes, for each transformer $\pi \in \Pi$, a name for the function implemented by π. The members of Γ are the *axioms* of T. They specify the meaning of the nonlogical symbols in L including the names of the transformers of T. In particular, Γ may contain specifications of the computational behavior of the transformers in Π and of the mathematical meaning of the SBMAs in Π. We say T is an *axiomatic theory* if Π is empty and an *algorithmic theory* if Γ is empty.

Formalizing a biform theory in the underlying logic requires an infrastructure for reasoning about the expressions manipulated by the transformers as syntactic entities. The infrastructure provides a basis for *metareasoning with reflection* [12]. There are two main approaches for obtaining this infrastructure [11]. The *local approach* is to produce a deep embedding of a sublanguage L' of L that include all the expressions manipulated by the transformers of Π. The *global approach* is to replace the underlying logic of L with a logic such as [12] that has an inductive type of *syntactic values* that represent the expressions in L and global quotation and evaluation operators.

A complex body of mathematical knowledge can be represented in accordance with the *little theories method* [15] as a *theory graph* [20] consisting of axiomatic theories as nodes and theory morphisms as directed edges. A *theory morphism* is a meaning-preserving mapping from the formulas of one axiomatic theory to the formulas of another. The theories — may have different underlying logics — serve as abstract mathematical models and the morphisms serve as information conduits that enable theory components such as definitions and theorems to be transported from one theory to another [2]. A theory graph enables mathematical knowledge to be formalized in the most convenient underlying logic at the most convenient level of abstraction using the most convenient vocabulary. The connections made by the theory morphisms in a theory graph then provide the means to find this knowledge and apply it in other contexts.

A *biform theory graph* is a theory graph whose nodes are biform theories. Having the same benefits as theory graphs of axiomatic theories, biform theory graphs are well suited for representing mathematical knowledge that is expressed both axiomatically and algorithmically.

3 Project Objectives

The general objective of the Biform Theories project is:

> **GenObj.** Develop a methodology for expressing and managing mathematical knowledge as a biform theory graph.

Our strategy for achieving this general objective is to pursue the following specific objectives:

> **SpecOBj 1.** Design a logic Log that is version of simple type theory [10] with an inductive type of syntactic values, a global quotation operator, and a global evaluation operator. In addition to a syntax and semantics, define a proof system for Log and a notion of a theory morphism from one axiomatic theory of Log to another. Demonstrate that SBMAs can be defined in Log and that their mathematical meanings can be stated and proved in using Log's proof system.
>
> **SpecOBj 2.** Produce an implementation Impl of Log. Demonstrate that SBMAs can be defined in Impl and that their mathematical meanings can be stated and proved in Impl.
>
> **SpecOBj 3.** Enable biform theories to be defined in Impl. Introduce a mechanism for applying transformers defined outside of Impl to expressions of Log.
>
> **SpecOBj 4.** Enable theory graphs of biform theories to be defined in Impl. Introduce mechanisms for transporting definitions, theorems, and transformers from a biform theory T to an instance T' of T via a theory morphism from T to T'.

SpecOBj 5. Design and implement in Impl a scheme for defining generic transformers in a theory graph T that can be specialized, when transported to an instance T' of T, using the properties exhibited in T'.

4 Work Plan Status

The work plan for the project is to achieve the five specific objectives described above more or less in the order of their presentation. This section describes the parts of the work plan that have been completed as well as the parts that remain to be done.

SpecObj 1: Logic with Quotation and Evaluation

This objective has been largely been achieved. We have developed $\textsc{ctt}_{\mathrm{qe}}$ [14], a version of Church's type theory [7] with global quotation and evaluation operators. (Church's type theory is a popular form of simple type theory with lambda notation.) The syntax of $\textsc{ctt}_{\mathrm{qe}}$ has the machinery of $\mathcal{Q}_0$ [1], Andrews' version of Church's type theory plus an inductive type ϵ of syntactic values, a partial quotation operator, and a typed evaluation operator. The semantics of $\textsc{ctt}_{\mathrm{qe}}$ is based on Henkin-style general models [19]. The proof system for $\textsc{ctt}_{\mathrm{qe}}$ is an extension of the proof system for $\mathcal{Q}_0$.

We show in [14] that $\textsc{ctt}_{\mathrm{qe}}$ is suitable for defining SBMAs and stating and proving their mathematical meanings. In particular, we prove within the proof system for $\textsc{ctt}_{\mathrm{qe}}$ the mathematical meaning of an symbolic differentiation algorithm for polynomials.

We have also defined $\textsc{ctt}_{\mathrm{uqe}}$ [13], a variant of $\textsc{ctt}_{\mathrm{qe}}$ in which undefinedness is incorporated in $\textsc{ctt}_{\mathrm{qe}}$ according to the traditional approach to undefinedness [8]. Better suited than $\textsc{ctt}_{\mathrm{qe}}$ as a logic for interconnecting axiomatic theories, we have defined in $\textsc{ctt}_{\mathrm{uqe}}$ a notion of a theory morphism [13].

SpecObj 2: Implementation of the Logic

We have produced an implementation of $\textsc{ctt}_{\mathrm{qe}}$ called HOL Light QE [5] by modifying HOL Light [18], a simple implementation of the HOL proof assistant [17]. HOL Light QE provides a built-in global infrastructure for metareasoning with reflection. Over the next couple years we plan to test this infrastructure by formalizing a variety of SBMAs in HOL Light QE.

SpecObj 3: Biform Theories

No work on this objective has been done, but we expect it will be a straightforward task to implement biform theories and the application of external transformer in HOL Light QE.

SpecObj 4: Biform Theory Graphs

We proposed in [4] a biform theory graph test case consisting of eight biform theories encoding natural number arithmetic. We produced partial formalizations of this test case [4] in CTT_{uqe} [13] using the global approach for metareasoning with reflection and in Agda [22,23] using the local approach. After we have finished with SpecObj 2 and SpecObj 3, we intend to formalize this test case in HOL Light QE.

SpecObj 5: Specializable Transformers

Jacques should complete this subsection. Please reference Pouya's thesis [21].

5 Related Work

Jacques should complete this section, referring as necessary to the related work section in [14].

6 Conclusion

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof, Second Edition*. Kluwer, 2002.
2. J. Barwise and J. Seligman. *Information Flow: The Logic of Distributed Systems*, volume 44 of *Tracts in Computer Science*. Cambridge University Press, 1997.
3. J. Carette and W. M. Farmer. High-level theories. In A. Autexier, J. Campbell, J. Rubio, M. Suzuki, and F. Wiedijk, editors, *Intelligent Computer Mathematics*, volume 5144 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2008.
4. J. Carette and W. M. Farmer. Formalizing mathematical knowledge as a biform theory graph: A case study. In H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, editors, *Intelligent Computer Mathematics*, volume 10383 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2017.
5. J. Carette, W. M. Farmer, and P. Laskowski. HOL Light QE. In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science. Springer, 2018. Forthcoming.
6. J. Carette, W. M. Farmer, and R. O'Connor. Mathscheme: Project description. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Computer Science*, pages 287–288. Springer, 2011.
7. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
8. W. M. Farmer. Formalizing undefinedness arising in calculus. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning—IJCAR 2004*, volume 3097 of *Lecture Notes in Computer Science*, pages 475–489. Springer, 2004.
9. W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2007.

10. W. M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6:267–286, 2008.
11. W. M. Farmer. The formalization of syntax-based mathematical algorithms using quotation and evaluation. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, volume 7961 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2013.
12. W. M. Farmer. Incorporating quotation and evaluation into Church's type theory. *Computing Research Repository (CoRR)*, abs/1612.02785 (72 pp.), 2016.
13. W. M. Farmer. Theory morphisms in Church's type theory with quotation and evaluation. In H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, editors, *Intelligent Computer Mathematics*, volume 10383 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2017.
14. W. M. Farmer. Incorporating quotation and evaluation into Church's type theory. *Information and Computation*, 2018. Forthcoming.
15. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer, 1992.
16. W. M. Farmer and M. von Mohrenschildt. An overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.
17. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
18. J. Harrison. HOL Light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.
19. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
20. M. Kohlhase. Mathematical knowledge management: Transcending the one-brain-barrier with theory graphs. *European Mathematical Society (EMS) Newsletter*, 92:22–27, June 2014.
21. P. Larjani. *Software Specialization as Applied to Computational Algebra*. PhD thesis, McMaster University, 2013.
22. U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.
23. U. Norell. Dependently typed programming in Agda. In A. Kennedy and A. Ahmed, editors, *Proceedings of TLDI'09*, pages 1–2. ACM, 2009.
24. Mark van Hoeij. Factoring polynomials and the knapsack problem. *Journal of Number Theory*, 95(2):167 – 189, 2002.
25. Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press.

Todo list