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.

A legitimate question would be: is this an isolated occurence, or a more pervasive pattern? It is pervasive. It arises from the dichotomy of being able to *perform* computations and being able to *talk about* (usually to prove the correctness of) computations. For example, we could represent (in Haskell) a tiny language of arithmetic as

```
data Arith = Int Integer | Plus Arith Arith
  | Times Arith Arith
```

and an evaluator as

```
eval :: Arith -> Integer
eval (Int x) = x
eval (Plus a b) = eval a + eval b
eval (Times a b) = eval a * eval b
```

whose "correctness" seems self-evident. But what if we had instead written

```
data AA = TTT Integer | XXX AA AA | YYY AA AA

eval' :: AA -> Integer
eval' (TTT x) = x
eval' (XXX a b) = eval' a * eval' b
eval' (YYY a b) = eval' a + eval' b
```

how would we know if this implementation of `eval'` is correct or not? The two languages are readily seen to be isomorphic. In fact, there are clearly *two* different isomorphisms. As the symbols used are no longer mnemonic, we have no means to (informally!) decide whether `eval'` is correct. Neverthess, `Arith` and `AA` both represent (trivial) embedded Domain Specific Languages (DSL), which are pervasively used in computing. Being able to know that functions defined over a DSL is correct is an important problem.

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 [27]. The algorithms to do this have gotten extremely sophisticated over the years [26]. 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. In a TPS, one typically proves theorems, formally. There is some cross-over: some TPS (notably Agda and Idris) are closer to programming languages, and thus offer the real possibility of mixing computation and deduction. Nevertheless, the problem still exists: how does one verify that a particular function implemented over a representation language, carries out the desired computation?

What is needed is a means to *link* together axiomatic theories and algorithmic theories such that one can state that some "symbol manipulation" corresponds to a (semantic) function defined axiomatically? In other words, we want to know that a *symbolic computation* perfomed on representations performs the same computation as an abstract function defined on the *denotation* of those representations. For example, if we ask to integrate a particular expression e, we would like to know that the system's response will in fact be an expression representing an integral of e — even if the formal definition of integration uses an infinitary process.

These kinds of problems are pervasive: not just closed-form symbolic manipulations, but also SAT solving, SMT solving, model checking, type-checking of programs, most manipulations of DSL terms, are all of this sort. They all involve a mixture of computation and deduction that entwine syntactic representations with semantic conditions.

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)* [13]. 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*[2] 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 [3,11,18]. 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. The logical formulas of Γ which refer to the names of transformers $\pi \in \Pi$ as *meaning formulas*.

Formalizing a biform theory in the underlying logic requires infrastructure for reasoning about the expressions manipulated by the transformers as syntactic entities. This infrastructure provides a basis for *metareasoning with reflection* [14]. There are two main approaches to building such infrastructure [13]. The *local approach* is to produce a deep embedding of a sublanguage L' of L that includes 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 CTT_{qe} [14] that has an inductive type of *syntactic values* that represent the expressions in L and global quotation and evaluation operators. A third approach, based on "final tagless" embeddings [7] has not yet been attempted as most logics do not have the necessary infrastructure to abstract over type constructors.

A complex body of mathematical knowledge can be represented in accordance with the *little theories method* [17] (or even the *tiny theories method* [8]) as a *theory graph* [22] 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 — which 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

[2] computer scientists would call this *denotational semantics* rather than mathematical meaning

graphs are well suited for representing mathematical knowledge that is expressed both axiomatically and algorithmically.

3 Project Objectives

The primary objective of the Biform Theories project is:

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

Our strategy for achieving this is to break down the problem into the following sub-projects:

> **Logic** Design a logic Log which is a version of simple type theory [12] 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 using Log's proof system.
>
> **Implement** 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.
>
> **Transformers** Enable biform theories to be defined in Impl. Introduce a mechanism for applying transformers defined outside of Impl to expressions of Log. Ensure that we know how to write meaning formulas for such transformers.
>
> **Theory Graphs** 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'.
>
> **Generic Transformers** 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 is to pursue the five objectives described above more or less in the order of their presentation. Here we describe the parts of the work plan that have been completed as well as the parts that remain to be done.

Logic with Quotation and Evaluation

This sub-project is largely complete. We have developed CTT_{qe} [16], a version of Church's type theory [9] with global quotation and evaluation operators. (Church's type theory is a popular form of simple type theory with lambda notation.) The syntax of CTT_{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 CTT_{qe} is based on Henkin-style general models [21]. The proof system for CTT_{qe} is an extension of the proof system for $\mathcal{Q}_0$.

We show in [16] that CTT_{qe} is suitable for defining SBMAs and stating and proving their mathematical meanings. In particular, we prove within the proof system for CTT_{qe} the mathematical meaning of an symbolic differentiation algorithm for polynomials.

We have also defined CTT_{uqe} [15], a variant of CTT_{qe} in which undefinedness is incorporated in CTT_{qe} according to the traditional approach to undefinedness [10]. Better suited than CTT_{qe} as a logic for interconnecting axiomatic theories, we have defined in CTT_{uqe} a notion of a theory morphism [15].

Implementation of the Logic

We have produced an implementation of CTT_{qe} called HOL Light QE [5] by modifying HOL Light [20], an implementation of the HOL proof assistant [19]. 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.

Biform Theories, including external Transformers

No work on this has yet 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. External transformers implemented in OCaml (or in languages reachable via OCaml's foreign function interface) can be linked in as well.

The most difficult part of this sub-project will be adequate renderings of *meaning functions* that express the mathematical meaning of transformers.

Biform Theory Graphs

In [**?**], we developed a case study of a biform theory graph consisting of eight biform theories encoding natural number arithmetic. We produced partial formalizations of this test case [4] in CTT_{uqe} [15] using the global approach for metareasoning with reflection, and in Agda [24,25] using the local approach. After we have finished with the previous two sub-projects, we intend to formalize this in HOL Light QE as well.

Generic, Specializable Transformers

Jacques should complete this subsection. Please reference Pouya's thesis [23].

5 Related Work

Jacques should complete this section, referring as necessary to the related work section in [16].

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. J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.
8. Jacques Carette and Russell O'Connor. Theory presentation combinators. In Johan Jeuring, John a. Campbell, Jacques Carette, Gabriel Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, volume 7362 of *Lecture Notes in Computer Science*, pages 202–215. Springer Berlin Heidelberg, 2012.
9. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
10. 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.
11. 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.
12. W. M. Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6:267–286, 2008.

13. 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.
14. W. M. Farmer. Incorporating quotation and evaluation into Church's type theory. *Computing Research Repository (CoRR)*, abs/1612.02785 (72 pp.), 2016.
15. 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.
16. W. M. Farmer. Incorporating quotation and evaluation into Church's type theory. *Information and Computation*, 2018. Forthcoming.
17. 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.
18. 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.
19. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
20. 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.
21. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
22. M. Kohlhase. Mathematical knowledge management: Transcending the one-brain-barrier with theory graphs. *European Mathematical Society (EMS) Newsletter*, 92:22–27, June 2014.
23. P. Larjani. *Software Specialization as Applied to Computational Algebra*. PhD thesis, McMaster University, 2013.
24. U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.
25. U. Norell. Dependently typed programming in Agda. In A. Kennedy and A. Ahmed, editors, *Proceedings of TLDI'09*, pages 1–2. ACM, 2009.
26. Mark van Hoeij. Factoring polynomials and the knapsack problem. *Journal of Number Theory*, 95(2):167 – 189, 2002.
27. Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2003.

Todo list