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 25, 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 since 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.

This paper presents a project named *Biform Theories* that is a subproject of MathScheme [6], a long-term project at McMaster University to produce a framework for integrating formal deduction and symbolic computation. The motivation, background ideas, objectives, work plan status, and related work for the project are described in the sections below.

1 Motivation

Jacques should complete this section with help from Yasmine with examples.

Computer algebra and theorem proving systems manipulate syntactic representation of mathematical knowledge. However, their way of manipulating them is very different from each other. Computer algebra systems manipulate expressions via algorithms. Knowledge is represented in the form of algorithmic theories that computes new expressions according to algorithms operating on the input. CAS suffer from lack of formalization of many of the algorithms used. On the other hand, theorem provers make use of axiomatic theories and proof systems to represent and proof properties of mathematical knowledge. Despite their usefulness, theorem provers are hard to use and are not being widely utilized. Biform theories aim to combine the benefits of both. This is better explained by an example. Consider the problem of representing and factoring polynomials. An axiomatic representation of a polynomial would extend the representation of the

⋆ This research was supported by NSERC.

theory of rings by adding an axiom `is-poly` to represent the acceptable forms of polynomials

```
Axiomatic representation of theories (Two possible representations)
- flattened theory as in the paper
   (http://imps.mcmaster.ca/doc/qe-in-church.pdf) P.18
- using the extend relation
Maybe we can present the flattened version then argue about
how to enhance it using theory graphs
```

An algorithmic theory would have the same constants as in the axiomatic one presented above. Instead of the axioms, algorithms for manipulating polynomials and performing factoring are defined. These algorithms are well defined in the literatures

Add references

A biform theory of polynomials would be described as a tuple (L, Π, Γ) where

- L defines constants describing the language of polynomials
- Π defines the algorithms (transformers) that manipulate the syntax of the polynomial to perform different operations - as in the algorithmic theory.
- Γ is a set of axioms. We differentiate between two types of axioms. Some axioms describe the behavior of the constants of the language - as in axiomatic theory. Other axioms, called the meaning formula, which describes the behavior of the transformer by describing how the mathematical meaning of the output expression relates to that of the input expression.

The meaning formula that would be added to Γ are

meaning formula for factor

We have described how biform theories are useful in representing mathematical knowledge. Now we discuss why a biform theory graph is useful. We build up on the example of factoring polynomials. Instead of describing all the components of a polynomial manipulation in one theory as we did above, the biform theory graph provides a more modular representation where

- Represent the basic theory of polynomial as an extension of the theory of ring by adding is-poly axiom.
- Represent the theory of factoring polynomial as an extension of the theory of polynomials

Describe the theory graph

1.1 Domain Specific Language (DSL)

A domain specific language is a programming language that focuses on solving problems related to a specific field. The target is to provide solutions that favor the aspects important to that specific domain. It also makes it easier for domain experts to use the system, as it uses terminology from the domain. One way of implementing a DSL is by extending a base language. In this case, the DSL uses the compiler or interpreter of the base language. Also, the semantics of the DSL is defined in terms of the semantics of the base language. Therefore, defining an internal DSL is more of building connections between two languages. In this case, it is essential that the axioms of the theory of DSL reason about how the syntax and semantics of the two languages are connected. This is where biform theories strength appear. To ensure the correctness of the Also, the semantics of expressions of the DSL are determined in terms of the semantics of the base language. Therefore, a theory of DSLs cannot be formulated that are able to talk about the semantics of expression. On a very abstract level, we need an axiom like

$$\forall e \in L_{DS} \cdot \exists e' \in L_B \ . \ [\![e]\!] = [\![e']\!]$$

that ensures all valid expressions in DSL has a meaning-preserving mapping to expressions in the base language.

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 $(L, \mathit{\Pi}, \mathit{\Gamma})$ where L is a language of some underlying logic, $\mathit{\Pi}$ is a set of transformers that implement functions on expressions of L, and $\mathit{\Gamma}$ is a set of formulas of L [3,9,16]. L includes, for each transformer $\pi \in \mathit{\Pi}$, a name for the function implemented by π. The members of $\mathit{\Gamma}$ 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, $\mathit{\Gamma}$ may contain specifications of the computational behavior of the transformers in $\mathit{\Pi}$ and of the mathematical meaning of the SBMAs in $\mathit{\Pi}$. We say T is an *axiomatic theory* if $\mathit{\Pi}$ is empty and an *algorithmic theory* if $\mathit{\Gamma}$ 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 $\mathit{\Pi}$. 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 CTT_{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 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 [19]. The proof system for CTT_{qe} is an extension of the proof system for $\mathcal{Q}_0$.

We show in [14] 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} [13], a variant of CTT_{qe} in which undefinedness is incorporated in CTT_{qe} according to the traditional approach to undefinedness [8]. Better suited than CTT_{qe} as a logic for interconnecting axiomatic theories, we have defined in CTT_{uqe} a notion of a theory morphism [13].

SpecObj 2: Implementation of the Logic

We have produced an implementation of CTT_{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.

Todo list