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

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, manipulating, managing, and generating 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 [11] (a long-term project to produce a framework integrating formal deduction and symbolic computation).

1 Motivation

Type `2 * 3` into your favourite computer algebra system, press enter, and you will receive (unsurprisingly) `6`. But what if you want to go in the opposite direction? Easy: you ask `ifactors(6)` in Maple or `FactorInteger[6]` in Mathematica.[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 the prime's multiplicity in the second. Mathematica's `FactorInteger` is similar, except that it omits the unit (and thus does not document what happens for negative integers).

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 in these systems. So that factoring `6` cannot just return `2^1 * 3^1`, as that simply evaluates to `6`. Thus it is inevitable that

[⋆] This research is supported by NSERC.

[1] Other computer algebra systems have similar commands.

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, in this case a feature to not immediately evaluate an expression but leave it as a representation of a future computation. Maple calls this feature an *inert form*, while in Mathematica it is a *hold form*. Nevertheless, the need for representing future computations was recognized early on: even in the earliest days of Maple, one could do `5 &^256 mod 379` to compute the answer without ever computing 5^{256} over the integers. In summary, this example shows that in some cases we are interested in `2 * 3` for its value and in other cases we are interested in it for its syntactic structure.

A legitimate question would be: Is this an isolated occurrence, 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. Nevertheless, `Arith` and `AA` both represent (trivial) embedded *domain specific languages* (DSLs), which are pervasively used in computing. Being able to know that a function defined over a DSL is correct is an important problem.

In general, both computer algebra systems (CASs) and theorem proving systems (TPSs) 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 factor it into a product of irreducible polynomials [46]. The algorithms to do this have gotten extremely sophisticated over the years [35]. In a TPS, it is more natural to prove that such a polynomial p is equal to a particular factorization, and perhaps also prove 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 CASs 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 CASs and TPSs "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 TPSs (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* performed 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, and 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.

In the next section we will introduce the notion of a *biform theory* that is a combination of an axiomatic theory and an algorithmic theory so that we can define and reason about symbolic computation in the same setting.

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. The expressions serve as data that can be manipulated. Different kinds of expressions correspond to different data representations. 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)* [27]. 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 $(L, \mathit{\Pi}, \mathit{\Gamma})$ 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 [6,25,31]. L includes, for each transformer $\pi \in \mathit{\Pi}$, a name for the function implemented by π that serves as a name for 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 $\mathit{\Pi}$. A formula in $\mathit{\Gamma}$ that refers to the name of a transformer $\pi \in \mathit{\Pi}$ is called a *meaning formula* for π. The transformers in $\mathit{\Pi}$ may be written in the underlying logic or in an programming language external to the underlying logic. We say T is an *axiomatic theory* if $\mathit{\Pi}$ is empty and an *algorithmic theory* if $\mathit{\Gamma}$ is empty.

Example 1. Let $R_{\rm ax} = (L, \mathit{\Gamma})$ be a first-order axiomatic theory of a ring with identity. The language L contains the usual constants (0 and 1), function symbols ($+$ and $*$), and predicate symbols ($=$), and $\mathit{\Gamma}$ contains the usual axioms. The terms of L, which are built from 0, 1, and variables by applying $+$ and $*$, have the form of multivariate polynomials. Thus algorithms that manipulate polynomials — that normalize a polynomial, factor a polynomial, find the greatest common divisor of two polynomials, etc. — would naturally be useful for reasoning about the expressions of $R_{\rm ax}$. Let $\mathit{\Pi}$ be a set of such transformers on the terms in L, L' be an extension of L that includes vocabulary for naming and specifying the transformers in $\mathit{\Pi}$, and $\mathit{\Gamma}'$ contain meaning formulas for the transformers in $\mathit{\Pi}$ expressed in L'. Then $R_{\rm bt} = (L', \mathit{\Pi}, \mathit{\Gamma} \cup \mathit{\Gamma}')$ is a biform theory for rings with identity. It would be very challenging to express $R_{\rm bt}$ in ordinary first-order logic; the meaning formulas in $\mathit{\Gamma}'$ would be especially difficult to express. Notice that $R_{\rm alg} = (L', \mathit{\Pi})$ is algorithmic theory of multivariate polynomials with constants 0 and 1.

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* [29]. There are two main approaches to build such an infrastructure [27]. The *local approach* is to produce a deep embedding of a sublanguage L' of L that includes

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

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 $\textsc{ctt}_{\mathrm{qe}}$ [29] 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 [15], 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* [30] (or even the *tiny theories method* [18]) as a *theory graph* [36] 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 graphs are well suited for representing mathematical knowledge that is expressed both axiomatically and algorithmically.

Our previous work on mechanized mathematics systems and on related technologies has taught us that such a graph of biform theories really should be a central component of any future systems for mathematics. We will expand on the objectives of the project and its current state. At the same time, additional pieces of the project beyond what is motivated above (but is motivated by previous and related work) will be weaved in as appropriate.

3 Project Objectives

The primary objective of the Biform Theories project is:

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

Our strategy for achieving this is to break down the problem into the following subprojects:

> **Logic** Design a logic Log which is a version of simple type theory [26] 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, proved, and instantiated using Log's proof system.

Implementation 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. Some transformers can be automatically generated — investigate the scope of this, and implement those which are feasible.

Theory Graphs Enable theory graphs of biform theories to be defined in Impl. Use combinators to ease the construction of large, structured biform theory graphs. 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'. Some theories (such as theories of homomorphisms and term languages) can be and thus should be automatically generated.

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 subprojects 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 subproject is largely complete. We have developed $\textsc{ctt}_{\mathrm{qe}}$ [29], a version of Church's type theory [22] 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 [34]. The proof system for $\textsc{ctt}_{\mathrm{qe}}$ is an extension of the proof system for $\mathcal{Q}_0$.

We show in [29] that $\textsc{ctt}_{\mathrm{qe}}$ is suitable for defining SBMAs and stating, proving, and instantiating their mathematical meanings. In particular, we prove within the proof system for $\textsc{ctt}_{\mathrm{qe}}$ the mathematical meaning of a symbolic differentiation algorithm for polynomials.

We have also defined $\textsc{ctt}_{\mathrm{uqe}}$ [28], a variant of $\textsc{ctt}_{\mathrm{qe}}$ in which undefinedness is incorporated in $\textsc{ctt}_{\mathrm{qe}}$ according to the traditional approach to undefinedness [24]. 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 [28].

Implementation of the Logic

We have produced an implementation of $\textsc{ctt}_{\mathrm{qe}}$ called HOL Light QE [10] by modifying HOL Light [33], an implementation of the HOL proof assistant [32]. 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.

Building on the experience we gain in the development of HOL Light QE, we would like to create an implementation of $\textsc{ctt}_{\mathrm{qe}}$ in Mmt [44] that is well suited for general use and has strong support for building theory graphs. We will transfer to this Mmt implementation the most successful of the ideas and mechanisms we develop on the three subprojects that follow using HOL Light QE.

Biform Theories, Transformers, and Generation

Implementation of biform theories in HOL Light QE has not yet started, but we expect that it will be straightforward, as will the application of external transformers. 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 subproject will be adequate renderings of *meaning formulas* that express the mathematical meaning of transformers. We do have some experience [7,12] creating biform theories. The exploration and implementation of automatic generation of transformers has started.

Biform Theory Graphs

In [7], 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 [7] in $\textsc{ctt}_{\mathrm{uqe}}$ [28] using the global approach for metareasoning with reflection, and in Agda [42,43] using the local approach. After we have finished with the previous two subprojects, we intend to formalize this in HOL Light QE as well.

In [18], we developed combinators for combining theory presentations. There is no significant difference between axiomatic and biform theories with respect to the semantics of these combinators, and we expect that these will continue to work as well as they did in [8]. There, we also experimented with some small-scale theory generation, which worked well. This subproject will also encompass the implementation of *realms* [9]. We also hope to make some inroads on *high level theories* [6].

Generic, Specializable Transformers

Through substantial previous work [15,4,5,8,13,14,16,17,19,38,39,41] on code generation and code manipulation, it has become quite clear that quite a lot of mathematical code can be automatically generated. One of the most successful techniques is *instantiation*, whereby a single, generic algorithm exposes a series of *design choices* that must be explicitly instantiated to produce specialized code. By clever choices of design parameters, and through the use of partial evaluation, one can thus produce highly optimized code without having to hand-write such code.

5 Related Work

Directly related is [37] whose authors also work with biform theory graphs. Michael Kohlhase and Florian Rabe and their students are actively working on related topics. As a natural progression, we (the authors of this paper) have started actively collaborating with them, under the name of the *Tetrapod Project*.

One of the crucial features for supporting the interplay between syntax and semantics is *reflection*, which has a long history and a deep literature. The interested reader should read the thorough related work section in [29] for more details.

There are substantial developments happening in some systems, most notably Agda [42,43], Idris [3] and Lean [40] that we are paying particularly close attention to. This includes quite a lot of work on making reflection practical [20,21,23,47].

On the more theoretical side, *homotopy type theory* [45] is rather promising. However quite a bit of research still needs to be done to make these results practical. Of particular note is the issue that theories that deal directly with syntax seem to clash with the notion of a *univalent universe*, which is central to homotopy type theory.

6 Conclusion

Building mechanized mathematics systems is a rather complex engineering task. It involves creating new science — principally through the creation of logics which can support reasoning about syntax. It also involves significant new engineering — both on the systems side, where *knowledge management* is crucial to reduce the *information duplication* inherent in a naive implementation of mathematics, and on the usability front, where users do not, and should not, care about all the infrastructure that developers need to create their system. Current systems tend to expose this infrastructure, thus creating an additional burden for casual users who may well have a simple task to perform.

The *Biform Theories* project is indeed about infrastructure that we believe is essential to building large-scale mechanized mathematics systems. And yes, we do believe that eventual success would imply that casual users of such a system never hear of “biform theories”.

Acknowledgments

This research was supported by NSERC. The authors would like to thank the referees for their comments and suggestions.

References

1. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth through Proof, Second Edition. Kluwer (2002)
2. Barwise, J., Seligman, J.: Information Flow: The Logic of Distributed Systems, Tracts in Computer Science, vol. 44. Cambridge University Press (1997)
3. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**, 552–593 (2013), `https://doi.org/10.1017/S095679681300018X`
4. Carette, J.: Gaussian Elimination: a case study in efficient genericity with MetaOCaml. Science of Computer Programming **62**, 3–24 (2006), Special Issue on the First MetaOCaml Workshop 2004
5. Carette, J., Elsheikh, M., Smith, S.: A generative geometric kernel. In: Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation. pp. 53–62. PEPM '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1929501.1929510`
6. Carette, J., Farmer, W.M.: High-level theories. In: Autexier, A., Campbell, J., Rubio, J., Suzuki, M., Wiedijk, F. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 5144, pp. 232–245. Springer (2008)
7. Carette, J., Farmer, W.M.: Formalizing mathematical knowledge as a biform theory graph: A case study. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 10383, pp. 9–24. Springer (2017)
8. Carette, J., Farmer, W.M., Jeremic, F., Maccio, V., O'Connor, R., Tran, Q.M.: The mathscheme library: Some preliminary experiments. Tech. rep., University of Bologna, Italy (2011), uBLCS-2011-04
9. Carette, J., Farmer, W.M., Kohlhase, M.: Realms: A structure for consolidating knowledge about mathematical theories. In: Watt, S., Davenport, J., Sexton, A., Sojka, P., Urban, J. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 8543, pp. 252–266. Springer (2014)
10. Carette, J., Farmer, W.M., Laskowski, P.: HOL Light QE. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving. Lecture Notes in Computer Science, Springer (2018), forthcoming
11. Carette, J., Farmer, W.M., O'Connor, R.: Mathscheme: Project description. In: Davenport, J.H., Farmer, W.M., Rabe, F., Urban, J. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 6824, pp. 287–288. Springer (2011)
12. Carette, J., Farmer, W.M., Sorge, V.: A rational reconstruction of a system for experimental mathematics. In: Proceedings of MKM/Calculemus 2007. LNCS, vol. 4573, pp. 13–26. Springer Verlag (2007)
13. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In: Glück, R., Lowry, M.R. (eds.) GPCE. Lecture Notes in Computer Science, vol. 3676, pp. 256–274. Springer (2005)
14. Carette, J., Kiselyov, O.: Multi-stage programming with Functors and Monads: Eliminating abstraction overhead from generic code. Sci. Comput. Program. **76**, 349–375 (2011)
15. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**, 509–543 (2009), `https://doi.org/10.1017/S0956796809007205`

16. Carette, J., Kucera, M.: Partial Evaluation for Maple. In: ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation. pp. 41–50 (2007)
17. Carette, J., Kucera, M.: Partial evaluation of Maple. Sci. Comput. Program. **76**, 469–491 (2011)
18. Carette, J., O'Connor, R.: Theory presentation combinators. In: Jeuring, J., Campbell, J.A., Carette, J., Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) Intelligent Computer Mathematics, Lecture Notes in Computer Science, vol. 7362, pp. 202–215. Springer Berlin Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-31374-5_14`
19. Carette, J., Shan, C.C.: Simplifying probabilistic programs using computer algebra. In: International Symposium on Practical Aspects of Declarative Languages. pp. 135–152. Springer, Cham (2016)
20. Christiansen, D., Brady, E.: Elaborator reflection: Extending Idris in Idris. SIGPLAN Not. **51**, 284–297 (Sep 2016), `http://doi.acm.org/10.1145/3022670.2951932`
21. Christiansen, D.R.: Type-directed elaboration of quasiquotations: A high-level syntax for low-level reflection. In: Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages. pp. 1:1–1:9. IFL '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2746325.2746326`
22. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**, 56–68 (1940)
23. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proceedings of the ACM on Programming Languages **1**(ICFP), 34 (2017)
24. Farmer, W.M.: Formalizing undefinedness arising in calculus. In: Basin, D., Rusinowitch, M. (eds.) Automated Reasoning—IJCAR 2004. Lecture Notes in Computer Science, vol. 3097, pp. 475–489. Springer (2004)
25. Farmer, W.M.: Biform theories in Chiron. In: Kauers, M., Kerber, M., Miner, R.R., Windsteiger, W. (eds.) Towards Mechanized Mathematical Assistants. Lecture Notes in Computer Science, vol. 4573, pp. 66–79. Springer (2007)
26. Farmer, W.M.: The seven virtues of simple type theory. Journal of Applied Logic **6**, 267–286 (2008)
27. Farmer, W.M.: The formalization of syntax-based mathematical algorithms using quotation and evaluation. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 7961, pp. 35–50. Springer (2013)
28. Farmer, W.M.: Theory morphisms in Church's type theory with quotation and evaluation. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 10383, pp. 147–162. Springer (2017)
29. Farmer, W.M.: Incorporating quotation and evaluation into Church's type theory. Information and Computation **260**, 9–50 (2018)
30. Farmer, W.M., Guttman, J.D., Thayer, F.J.: Little theories. In: Kapur, D. (ed.) Automated Deduction—CADE-11. Lecture Notes in Computer Science, vol. 607, pp. 567–581. Springer (1992)
31. Farmer, W.M., von Mohrenschildt, M.: An overview of a Formal Framework for Managing Mathematics. Annals of Mathematics and Artificial Intelligence **38**, 165–191 (2003)
32. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)

33. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 5674, pp. 60–66. Springer (2009)
34. Henkin, L.: Completeness in the theory of types. Journal of Symbolic Logic **15**, 81–91 (1950)
35. van Hoeij, M.: Factoring polynomials and the knapsack problem. Journal of Number Theory **95**, 167 – 189 (2002), `http://www.sciencedirect.com/science/article/pii/S0022314X01927635`
36. Kohlhase, M.: Mathematical knowledge management: Transcending the one-brain-barrier with theory graphs. European Mathematical Society (EMS) Newsletter **92**, 22–27 (June 2014)
37. Kohlhase, M., Mance, F., Rabe, F.: A universal machine for biform theory graphs. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) Intelligent Computer Mathematics. Lecture Notes in Computer Science, vol. 7961, pp. 82–97. Springer (2013)
38. Kucera, M., Carette, J.: Partial evaluation and residual theorems in computer algebra. In: Ranise, S., Bigatti, A. (eds.) Proceedings of Calculemus 2006. Electronic Notes in Theoretical Computer Science, Elsevier (2006)
39. Larjani, P.: Software Specialization as Applied to Computational Algebra. Ph.D. thesis, McMaster University (2013)
40. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover. In: Automated Deduction - CADE-25, 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (2015)
41. Narayanan, P., Carette, J., Romano, W., Shan, C.C.S., Zinkov, R.: Probabilistic inference by program transformation in hakaru (system description). In: Functional and Logic Programming. vol. 9613, pp. 62–79. Springer-Verlag (2016)
42. Norell, U.: Towards a Practical Programming Language based on Dependent Type Theory. Ph.D. thesis, Chalmers University of Technology (2007)
43. Norell, U.: Dependently typed programming in Agda. In: Kennedy, A., Ahmed, A. (eds.) Proceedings of TLDI'09. pp. 1–2. ACM (2009)
44. Rabe, F., Kohlhase, M.: A scalable model system. Information and Computation **230**, 1–54 (2013)
45. Univalent Foundations Program, T.: Homotopy Type Theory: Univalent Foundations of Mathematics. `https://homotopytypetheory.org/book`, Institute for Advanced Study (2013)
46. Von Zur Gathen, J., Gerhard, J.: Modern computer algebra. Cambridge university press (2003)
47. van der Walt, P.: Reflection in Agda. Master's thesis, Universiteit Utrecht (2012)