A Coq Formalization of Boolean Unification

Daniel J. Dougherty

Worcester Polytechnic Institute, Worcester, MA, U.S.A.
dd@wpi.edu

Abstract

We report on a verified implementation of two (well-known) algorithms for unification modulo the theory of Boolean rings: Löwenheim's method and the method of Successive Variable Elimination. The implementations and proofs of correctness were done in the Coq proof assistant; we view this contribution as an early step in a larger project of developing a suite of verified implementations of equational unification algorithms.

1 Introduction

There is a significant emerging body of work devoted to formalizing mathematics and to specifying and verifying algorithms and systems using proof assistants such as Coq, Isabelle, HOL, and others. In this paper we report on first steps in a project of developing—in the proof assistant Coq—a library of verified E-unification algorithms. Our long-term goal is to build a resource of reusable data structures, algorithms, and theorems that will be a resource to researchers in mathematics and computer science.

Boolean unification is an attractive choice for a first theory to formalize. Boolean rings arise in many contexts, in connection with satisfiability, with circuit synthesis, and with logic programming, [BS87], and play a role in general topology, algebra, lattice theory. Boolean unification has a well-behaved theory (it is decidable and unitary) and there are interesting algorithms suitable for "native" implementation (as opposed to, for example, AC-unification, which naturally involves passing to reasoning about Diophantine equations).

To date we have implemented and proved correct two well-known algorithms for unification modulo the theory of Boolean rings, in the absence of free function symbols: Löwenheim's method and the method of Successive Variable Elimination. The code comprises about 1600 lines of definitions (of concepts and algorithms) and about 2900 lines of proof. It is available at http://web.cs.wpi.edu/~dd/unif.pdf. We have not made the Coq source available because, as described below, our development is being actively refined.

Our main goal in the present paper is to generate feedback from the unification community about the usefulness of a project like this and a discussion about future directions to pursue.

2 Related Work

There have been several verified implementations of *syntactic* unification, too many to attempt to list here. But we do note that in last year's UNIF (2018) Kasper Fabæch Brandt, Anders Schlichtkrull, and Jøorgen Villadsen present [BSV18] an Isabelle coding of syntactic unification, with a formal proof of termination though not of correctness. The related work section of that paper cites a selection of other verified developments of syntactic unification in various tools. The problem of formalizing unification modulo theories has received much less attention. A notable exception is the Coq formalization of associative-commutative matching by Evelyne Contejean in [Con04].

There are many examples of formal development of algebraic theories, again, far too many to summarize here, but there are relatively few treatments of equational logic in a general, universal-algebra sense.

Two early examples are [Cap99, Dom08], but the example most relevant to the current project is the CoLoR system [BK11], part of a family of tools also including CiME [CCF+11] and Coccinelle [Con]. CoLoR is primarily a (Coq) library for analyzing termination of rewrite relations. Although it does not treat semantic unification, it has an extensive infrastructure for reasoning about term-rewriting generally, and has been a valuable resource for the current project.

During the 2018-19 academic a WPI undergraduate project team (Spyridon Antonatos, Matthew McDonald, Dylan Richardson, and Joseph St. Pierre) worked on Boolean unification in parallel with the author, with some differences in data structures and outcomes.

3 Preliminaries

We assume familiarity with standard notions of equational logic and unification [BS01, BN98].

A *Boolean ring* is a ring in which every element x is idempotent, that is, $x^2 = x$. The specific set $\mathbb{B}$ of axioms we use in our development is the following.

$$\begin{array}{lll} 0+x=x & 1*x=x & (x+y)+z=x+(y+z) \\ (x*y)*z=x*(y*z) & x+y=y+x & x*(y+z)=(x*y)+(x*z) \\ (y+z)*x=(y*x)+(z*x) & x+x=0 & x*x=x \end{array}$$

In what follows we will use $s =_B t$ to mean that s and t are provably equal modulo $\mathbb{B}$; we use $s \stackrel{?}{=}_B t$ to denote a unification problem modulo $\mathbb{B}$.

The axioms other than the last are precisely the axioms for a ring, with the omission of an additive inverse and the inclusion of the axiom $x+x=0$. But in the presence of an additive inverse, the omnipotence axiom $x^2 = x$ entails $x+x=0$, and so the additive inverse operator is in fact the identity. [Proof: expand $x+x$ as $(x+x)(x+x) =_B xx+xx+xx+xx =_B x+x+x+x$. Cancellation yields $0 =_B x+x$.] Presenting the theory as above, with $x+x=0$ as an axiom, rather than using a separate additive inverse operator, thus simplifies a formal development: there is one fewer function symbol in the signature.

Two easy but significant consequences of the axioms are (i) commutativity of multiplication: $xy =_B yx$ [Proof: $x+y =_B (x+y)(x+y) =_B xx+xy+yx+yy =_B x+xy+yx+y$. Cancellation yields $0 =_B xy+yx$] and (ii) the fact that every element is a 0-divisor [since $x(1+x) =_B x+xx =_B x+x =_B 0$].

Since an equation $u =_B v$ is equivalent to the equation $u+v =_B 0$,

As usual in the presence of a "inverse" operator, unification problems $u \stackrel{?}{=}_B v$ reduce to matching problems of the form $t =_B 0$. So in the rest of the paper, instead of speaking of *unifying* two terms u and v, we will often speak of *solving* a term t, meaning finding a substitution σ such that $\sigma t =_B 0$.

There is a well-known translation between Boolean rings and Boolean algebras, under which multiplication corresponds to conjunction and addition to exclusive-or. Any field of sets yields a Boolean ring under intersection and symmetric difference. In fact, by the Stone Representation Theorem (and the relationship between Boolean rings and Boolean algebras) every Boolean ring is isomorphic to a field of sets [Sto36].

Unification Of course unification modulo the theory of rings is undecidable (by reduction to Hilbert's 10th problem). By contrast, there are several well-known algorithms for $\mathbb{B}$-unification. The decision problem for $\mathbb{B}$-unification (without free constants) is essentially the satisfiability problem, and so *NP*-complete, since, as noted in [BN98], $s =_B t$ under $\mathbb{B}$ if and only if s and t are equal terms over the boolean ring $\mathcal{B}_2$. The decision problem for $\mathbb{B}$-unification is Π_2^p-complete for a signature with free constants and is PSPACE-complete for a signature with free non-constant function symbols [Baa98] (by reduction to the validity of sentences of quantified boolean formulas).

Space considerations preclude a full discussion here of the history of algorithms for $\mathbb{B}$-unification (see [BN98] for an introduction) but we do note that $\mathbb{B}$-unification over the basic signature even in the presence of free constants is unitary; it is finitary when arbitrary free function symbols are allowed. Algorithms for constructing unifiers in the basic case can be found in [Löw08, BS87]; Martin and Nipkow [MN89] treat the case of free constants. Our formalizations treat the methods of Löwenheim [Löw08] and of Variable Elimination [Boo47, BS87]. It is not hard to show that simultaneous $\mathbb{B}$-unification reduces to $\mathbb{B}$-unification of single equation, even in the absence of auxiliary function symbols.

4 The Development

Coq is simultaneously a functional programming language and an environment for constructing proofs. The programming language of Coq closely resembles OCaml; the chief differences are the richer type system and the requirement that every function be terminating.

We do not have space here to discuss both algorithms in detail, and even so we can only hint at what the implementation and verification look like.

We represent terms by the following inductive data type. T0, T1, A, and M denote 0, 1, addition, and multiplication, respectively.

```
Inductive bterm : Type :=
| T0 : bterm | T1 : bterm | V : var → bterm
| A : bterm → bterm → bterm | M : bterm → bterm → bterm
```

Using Coq's Notation facility we use x +' y and x ∗' y as syntactic sugar for A x y and M x y and (the quotes are added because symbols + and ∗ have standard-library meanings in Coq).

The equational theory $\mathbb{B}$ is captured by the inductive relation eqv below; we introduce a Coq Notation (s == t) as infix syntactic sugar for (eqv s t). To save space, only a selection of the axioms are given here.

```
Inductive eqv : bterm → bterm → Prop :=
| assocA : forall x y z, x +' y == y +' x
...
|eqv_ref : forall x , x == x
...
| A_compat : forall x x', x == x' → forall y y', y == y' → x +' y == x' +' y'
```

The cases not shown capture the other rules of $\mathbb{B}$ and a set of equations expressing that eqv is an equivalence relation compatible with respect to the ring operations.

4.1 Löwenheim's Method

The algorithm has two stages. To solve $t \stackrel{?}{=}_B 0$, (i) first search for any substitution γ with $\gamma(t) = 0$; (ii) if such a γ is found, return the substitution

$$\sigma \stackrel{\text{def}}{=} \{v := (t+1) * v + (t) * \gamma(v) \mid v \in Vars(t)\}.$$

Remarkably, this will be a most general solution. To ensure that part (i) terminates, we search through the finitely many substitutions γ defined on $Vars(t)$ that only take on values $T0$ or $T1$. Of course, a key part of the verification is that this restriction is complete. If no such γ is found we report that t is not solvable.

For example, to solve $xy \stackrel{?}{=}_B 0$, we can start with $\gamma = [x := 0; y := 1]$; then σ is, after simplification,

$$x := xy + x \qquad y := y$$

Note that σ is more general than γ, indeed, $\gamma \circ \sigma = \gamma$. It is, perhaps, not obvious that σ is actually most general.

Löwenheim's method in Coq The most we can do in this limited space is to show three functions that correspond to the algorithm outlined above, and the corresponding shape of the correctness proof. The functions below, respectively (i) compute an initial solution (returning "None" if none found); (ii) build a new substitution, the "lowenheim lift," out of a given substitution and a given term, and (iii) bundle these together as a constructive decision procedure.

```
Definition ground_soln (t: bterm) : option sub :=
find (fun s ⇒ eqvb (s ' t) T0) (all_01subs_bterm t).

Definition lowenheim_lift (t: bterm) (tau: sub) (x: var) : bterm :=
  if inb x (vars_bterm t)
  then ((t +' T1) *' (V x) +' t *' (apply_sub tau (V x)) )
  else (V x).

Definition solve_lowenheim (t: bterm) :=
  option_map (lowenheim_lift t) (ground_soln t).
```

To prove correctness we must show that if ground_soln t returns None then t is not solvable, and otherwise, lowenheim_lift constructs a most general solution from the substitution returned by ground_soln t. The structure of the correctness claim is the same as the structure of solve_lowenheim itself.

```
Theorem lowenheim_correct (t: bterm) :
match (solve_lowenheim t) with
| None ⇒ ∼(solvable t)
| Some sigma ⇒ mgu_strong sigma t
end .
```

4.2 Successive Variable Elimination

This algorithm depends the fact that for any bterm t and any variable x we can compute quotient term q and remainder term r such that $(t =_B qx + r)$. Using this, the variable elimination algorithm for solvability of a term t is as follows:

1. choose a variable x from among the variables of t, and compute q and r such that t == q x + r.
2. set t' to be $(q+1)r$
3. compute a most general substitution σ' solving t'
4. then the following substitution is a most general solution for t

$$\sigma \stackrel{\text{def}}{=} \{x \mapsto x * ((\sigma' q) + 1) + (\sigma r) \mid x \in vars\, t\}$$

Correctness Argument The correctness argument proceeds by induction on the number of variables in the original term t. Then each of the claims below is captured by a lemma in the formalization.

- Any ground term can be reduced to either 1 or 0, and so is either unsolvable or has the identity as a most general solution.
- As we go from t down through the recursive calls, if t is solvable then each derived bterm $t' \stackrel{\text{def}}{=} (q+1)r$ is solvable.
- If σ' is a most general solution for a derived bterm t', then the updated substitution computed from σ' is a most general solution for the original bterm .

A few words about the factorization of a term t into the form $qx+r$ give some glimpse into the kind of considerations that arise in a formalization in a proof assistant. The construction as described in [BN98] relies on the fact that any term can be represented in polynomial form, as a set of monomials, each of which is a set a variables. Extracting q and r from t is very simple under this representation. But as described in the next section, polynomial form is not the most convenient representation for us. So we construct recursive functions to compute q and r given x and the standard representation of t. For example when t is t_1t_2 (the most interesting case) and is recursively factored as q_1x+r_1 and q_2x+r_2, then

$$\begin{aligned} t_1t_2 &=_B (q_1x+r_1)(q_2x+r_2) \\ &=_B q_1q_2xx+q_1r_2x+q_2r_1x+r_1r_2 \\ &=_B (q_1q_2+q_1+q_2)x+(r_1r_2) \end{aligned}$$

This is mathematically straightforward, and as a programming task it is easy to code a pair of mutually recursive functions handling cases on the form of t. But there is a wrinkle: in Coq all functions must provably terminate, and Coq's automatic techniques for verifying termination do not suffice for the mutual recursion suggested by the above. There are techniques for supplying Coq with a termination justification developed by the user, though this is sometimes a non-trivial task. But luckily in the present case, we can disentangle the calculations: a function computing the remainder r of a term t after division by x can be computed directly, first, and then used in computing the quotient q.

5 Design Choices and Lessons Learned

$\mathbb{B}$-equality as an Inductive Predicate We might have, naively, captured each of the $\mathbb{B}$-equations as a Coq Axiom, such as

```
Axiom invA : forall x, x +' x = T0
```

where we have used the primitive Coq equality predicate = instead of eqv. But this would be a mistake: the resulting theory would be logically inconsistent. A fundamental principle of inductively defined data (such as terms) is that if C_1 and C_2 are distinct constructors (such as +' and T0), then any equation $C_1\vec{s} = C_2\vec{t}$ entails falsehood.

This is why we represent equality under $\mathbb{B}$ as a *Inductive Predicate*. When eqv is defined inductively it means that (eqv s t) holds *precisely* when it follows from the rules in the inductive definition. (And Coq provides tactics, such as inversion, to exploit this fact.) It follows that eqv really does precisely capture provable equality under $\mathbb{B}$.

By the way, this inductive-predicate approach differs from systems such as CoLoR or treatments of algebraic structure in, e.g., the Mathematical Component Library. The difference is that we are not interested in deriving just those theorems that hold in all $\mathbb{B}$-structures, we are interested in one particular structure, the term model for $\mathbb{B}$. So for example, we can *prove* the proposition $\sim$(T0 == T1), which is not a theorem of $\mathbb{B}$.

Terms and Polynomials (The discussion in this paragraph is specific to the theory $\mathbb{B}$.) For any term t there is a $\mathbb{B}$-equivalent term p in polynomial form: monomials are sets of variables and polynomials are set of monomials. If we totally order variables and—then, lexicographically—monomials, every term can be reduced to a unique polynomial normal form modulo $\mathbb{B}$. This is very convenient theoretically, and is exploited in several places in [BN98], and so it suggests a representation for terms in the formalization. But as my undergraduate team grew to realize, this is not a particularly easy representation to work with. Arithmetic operations are not easily *directly* described as operations on these sets-of-sets; there is post-processing to be done to restore the required invariants. In turn, these repairs must be proven correct, for each operation, and the verification effort grows burdensome out of proportion with the intellectual convenience of having unique normal forms. Eschewing this approach meant that certain techniques and results, such as (i) the fact that $s =_B t$ is $\mathbb{B}$-provable if and only if s and t are equal polynomials over 2-element Boolean rings, and (ii) the factoring lemma crucial to the Variable Elimination algorithm, had to be defined and verified without recourse to the most mathematically-natural data structure, which is polynomial form.

Terms as a Signature-specific Inductive Data Type The data structure shown in Section 4 is what the current development uses, and it has much to recommend it. It is direct, it supports programming by pattern-matching, and Coq automatically computes an induction principle that makes tactic-based reasoning convenient. But a significant drawback to this approach is the fact that it is not generic across signatures. Much of the infrastructure of the verification effort comprises standard results about variable occurrences, substitutions, and so forth. Conceptually the definitions of, and proofs about, these notions are the same for any signature, but with the signature-specific approach this infrastructure would, in a formal development, have to be constructed anew for each signature. To see a very simple example of this lack of robustness, note that for a theory E we often want to explore E-unification between terms that may have free function symbols, symbols not occurring in the signature of E. Specifying terms as a signature-specific type obviously precludes anything but an ad-hoc approach to such an analysis.

A Generic Inductive Data Type for Terms A better approach, the object of the work on the project at the time of this writing, is to work with a single definition of term that can be *parameterized* by signatures. Indeed this is the approach taken by Coccinelle and CoLoR.

Given a definition of Signature (a set of function symbols with arities) and a declaration of Sig, a variable of type Signature, the following (taken from the CoLoR distribution) is the definition of term:

```
Inductive term : Type :=
| Var : variable → term
| Fun : forall f : Sig, vector term (arity f) → term .
```

A term is given a function symbol f from Sig and a vector of terms whose length is the arity of f. In fact there is a design choice reflected here: Coccinelle, for example, uses a *list* of terms to hold the arguments to f. Lists have better library support than do vectors in Coq, but we then incur either the stain of accepting that ill-formed terms can be built or the burden of maintaining and verifying well-formedness after each operation, reminiscent of the polynomial-form discussion above.

One small cost to this choice is that, because term occurs recursively inside the vector data structure, the induction principle automatically generated by Coq for these terms is not as strong as it could be. (This drawback applies to the list-based approach as well.) This is a remark about the current strength of Coq's algorithm for constructing induction principles, not about the existence of a useful induction principle. In fact it is straightforward to generate by hand the proper induction principles and tell Coq to use these: this is exactly what Coccinelle and CoLoR do. A slightly greater cost to this approach is

simply that dependent types such as vectors can be awkward to work with and reason about compared to direct inductive types.

6 Future Work

The current development is rather naive, from a Coq perspective: there are doubtless many unrealized opportunities for proof automation (including an adaptation of Coq's built-in ring tactic to incorporate Boolean-ring specific simplifications). The most immediate future, indeed ongoing, work is to address the challenge described in Section 5 of finding the best data structure for terms. This will be crucial to maintaining a flexible set of libraries that admit free function symbols and work across a variety of equational theories. Coq has facilities for extraction of functional programs; we expect this to be straightforward to incorporate into our libraries. Of course the most significant goal for the future is to treat other equational theories. Any number of specific theories E are natural candidates but it will perhaps be even more interesting to formalize "generic" E-unification approaches such as those based on transformations, combining algorithms, and narrowing (see [Mes18] for a discussion)

Acknowledgments It was very helpful to do this work in parallel with the work of the WPI undergraduate project team of Spyridon Antonatos, Matthew McDonald, Dylan Richardson, and Joseph St. Pierre. Although the development described here differs in many ways from theirs, their project led to many useful discussions about design choices for implementation and verification. It also generated a useful succession of deadlines to be met. The extremely clear introduction to Boolean unification in chapter 4 of [BN98] was a crucial element in making this topic suitable for introducing Coq to an undergraduate project team.

We benefited greatly from utility code in the library "Base Library for ICL" written by Gert Smolka. https://www.ps.uni-saarland.de/courses/cl-ss15/coq/ICL.Base.html

References

[Baa98] Franz Baader. On the complexity of boolean unification. *Information Processing Letters*, 67(4):215–220, 1998.

[BK11] Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[Boo47] George Boole. *The mathematical analysis of logic*. Philosophical Library, 1847.

[BS87] Wolfram Buttner and Helmut Simonis. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation*, 4(2):191–205, 1987.

[BS01] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[BSV18] Kasper Fabæch Brandt, Anders Schlichtkrull, and Jørgen Villadsen. Formalization of first-order syntactic unification. In *32nd International Workshop on Unification (UNIF), Informal Proceedings*, 2018.

[Cap99] V. Capretta. Universal algebra in type theory. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *TPHOLs 1999*, volume 1690 of *LNCS*, pages 131–148. Springer, 1999.

[CCF+11] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Automated certified proofs with CiME3. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 21–30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[Con] Evelyne Contejean. The Coccinelle library for rewriting. http://www.lri.fr/~contejea/Coccinelle/coccinelle.html.

[Con04] Evelyne Contejean. A certified AC matching algorithm. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[Dom08] C. Domınguez. Formalizing in Coq Hidden Algebras to Specify Symbolic Computation Systems. In *AISC*, pages 270–284. Springer, 2008.

[Löw08] Leopold Löwenheim. Über das Auflösungsproblem im logischen Klassenkalkul. *Sitzungsberichte Berliner Math. Gesell.*, 7:89–94, 1908.

[Mes18] José Meseguer. Variant-based satisfiability in initial algebras. *Science of Computer Programming*, 154:3–41, 2018.

[MN89] Urusula Martin and Tobias Nipkow. Boolean unification—the story so far. *Journal of Symbolic Computation*, 7(3-4):275–293, 1989.

[Sto36] Marshall H Stone. The theory of representation for boolean algebras. *Transactions of the American Mathematical Society*, 40(1):37–111, 1936.