Equivalence Checking for Orthocomplemented Bisemilattices in Log-Linear Time[⋆]

Simon Guilloud(✉) and Viktor Kunčak

EPFL IC LARA, Station 14, CH-1015 Lausanne, Switzerland
{Simon.Guilloud,Viktor.Kuncak}@epfl.ch

Abstract. Motivated by proof checking, we consider the problem of efficiently establishing equivalence of propositional formulas by relaxing the completeness requirements while still providing certain guarantees. We present a quasilinear time algorithm to decide the word problem on a natural algebraic structures we call orthocomplemented bisemilattices, a subtheory of Boolean algebra. The starting point for our procedure is a variation of Aho, Hopcroft, Ullman algorithm for isomorphism of trees, which we generalize to directed acyclic graphs. We combine this algorithm with a term rewriting system we introduce to decide equivalence of terms. We prove that our rewriting system is terminating and confluent, implying the existence of a normal form. We then show that our algorithm computes this normal form in log linear (and thus sub-quadratic) time. We provide pseudocode and a minimal working implementation in Scala.

1 Introduction

Reasoning about propositional logic and its extensions is a basis of many verification algorithms [19]. Propositional variables may correspond to, for example, sub-formulas in first-order logic theories of SMT solvers [2,5,26], hypotheses and lemmas inside proof assistants [13,27,32], or abstractions of sets of states. In particular, it is often of interest to establish that *two propositional formulas are equivalent*. The equivalence problem for propositional logic is coNP-complete as a negation of propositional satisfiability [8]. From proof complexity point of view [18] many known proof systems, including (non-extended) resolution [31] and cutting planes [29] have exponential-sized shortest proofs for certain propositional formulas. SAT and SMT solvers rely on DPLL-style algorithms [9,10] and do not have polynomial run-time guarantees on equivalence checking, even if formulas are syntactically close. Proof assistants implement such algorithms as tactics, so they have similar difficulties. A consequence of this is that implemented systems may take a very long time (or fail to acknowledge) that a large formula is equivalent to its minor variant differing in, for example, reordering of internal conjuncts or disjuncts. Similar situations also arise in program verifiers [12,21,30,34,35], where assertions act as lemmas in a proof.

[⋆] We acknowledge the financial support of the Swiss National Science Foundation project 200021_197288 “A Foundational Verifier”.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 196–214, 2022.
https://doi.org/10.1007/978-3-030-99527-0_11

It is thus natural to ask for an approximation of the propositional equivalence problem: *can we find an expressive theory supporting many of the algebraic laws of Boolean algebra but for which we can still have a complete and efficient algorithm for formula equivalence?* By efficient, we mean about as fast, up to logarithmic factors, as the simple linear-time syntactic comparison of formula trees.

We can use such an efficient equivalence algorithm to construct more flexible proof systems. Consider any sound proof system for propositional logic and replace the notion of *identical* sub-formulas with our notion of fast equivalence. For example, the axiom schema $p \to (q \to p)$ becomes $p \to (q \to p')$ for all equivalent p and p'. The new system remains sound. It accepts all the previously admissible inference steps, but also some new ones, which makes it more flexible.

L1:	$x \sqcup y = y \sqcup x$	L1':	$x \wedge y = y \wedge x$
L2:	$x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$	L2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
L3:	$x \sqcup x = x$	L3':	$x \wedge x = x$
L4:	$x \sqcup 1 = 1$	L4':	$x \wedge 0 = 0$
L5:	$x \sqcup 0 = x$	L5':	$x \wedge 1 = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \sqcup \neg x = 1$	L7':	$x \wedge \neg x = 0$
L8:	$\neg(x \sqcup y) = \neg x \wedge \neg y$	L8':	$\neg(x \wedge y) = \neg x \sqcup \neg y$

Table 1. Laws of an algebraic structures $(S, \wedge, \sqcup, 0, 1, \neg)$. Our algorithm is complete (and log-linear time) for structures that satisfy laws L1-L8 and L1'-L8'. We call these structures orthocomplemented bisemilattices (OCBSL).

L9:	$x \sqcup (x \wedge y) = x$	L9':	$x \wedge (x \sqcup y) = x$
L10:	$x \sqcup (y \wedge z) = (x \sqcup y) \wedge (x \sqcup z)$	L10':	$x \wedge (y \sqcup z) = (x \wedge y) \sqcup (x \wedge z)$

Table 2. Neither the absorption law L9,L9' nor distributivity L10,L10' hold in OCBSL. Without L9,L9', the operations $\wedge$ and $\sqcup$ induce different partial orders. If an OCBSL satisfies L10,L10' then it also satisfies L9,L9' and is precisely a Boolean algebra.

1.1 Problem Statement

This paper proposes to approximate propositional formula equivalence using a new algorithm that solves exactly the word problem for structures we call orthocomplemented bisemilattices (axiomatized in Table 1), in only log-linear time. In general, the word problem for an algebraic theory with signature S and axioms A is the problem of determining, given two terms t_1 and t_2 in the language of S with free variables, whether $t_1 = t_2$ is a consequence of the axioms. Our main interest in the problem is that orthocomplemented bisemilattices (OCBSL) are a generalisation of Boolean algebra. This structure satisfies a weaker set of axioms that omits the distributivity law as well as its weaker variant, the absorption law (Table 2). Hence, this problem is a relaxation "up to distributivity" of the propositional formula equivalence. A positive answer implies formulas are equivalent in all Boolean algebras, hence also in propositional logic.

Definition 1 (Word Problem for Orthocomplemented Bisemilattices). *Consider the signature with two binary operations* $\wedge, \sqcup$*, unary operation* $\neg$ *and constants,* $0, 1$. *The OCBSL-word problem is the problem of determining, given two terms* t_1 *and* t_2 *in this signature, containing free variables, whether* $t_1 = t_2$ *is a consequence (in the sense of first-order logic with equality) of the universally quantified axioms L1-L8,L1'-L8' in Table 1.*

Contribution. We present an $\mathcal{O}(n \log^2(n))$ algorithm for the word problem of orthocomplemented lattices. In the process, we introduce a confluent and terminating rewriting system for OCBSL on terms modulo commutativity. We analyze the algorithm to show its correctness and complexity. We present its executable description and a Scala implementation at https://github.com/epfl-lara/OCBSL.

1.2 Related Work

The word problem on *lattices* has been studied in the past. The structure we consider is, in general, *not* a lattice. Whitman [33] showed decidability of the word problem on free lattices, essentially by showing that the natural order relation on lattices between two words can be decided by an exhaustive search. The word problem on *orthocomplemented lattices* has been solved typically by defining a suitable sequent calculus for the order relation with a cut rule for transitivity [4, 17]. Because a cut elimination theorem can be proved similarly to the original from Gentzen [11], the proof space is finite and a proof search procedure can decide validity of the implication in the logic, which translates to the original word problem.

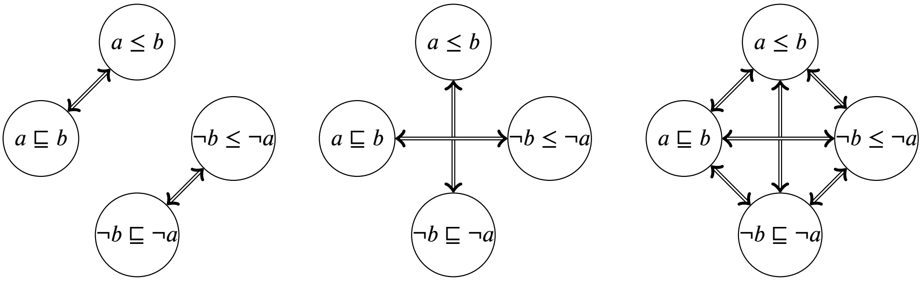
The word problem for free lattices was shown to be in PTIME by Hunt et al. [15] and the word problem for orthocomplemented lattices was shown to be in PTIME by Meinander [25]. Those algorithms essentially rely on similar proof-search methods as the previous ones, but bound the search space. These results make no mention of a specific degree of the polynomial; our analysis suggest that, as described, these algorithms run in $\mathcal{O}(n^4)$. Related techniques of locality have been applied more broadly and also yield polynomial bounds, with the specific exponents depending on local Horn clauses that axiomatize the theory [3, 24].

Aside from the use in equivalence checking, the problem is additionally of independent interest because OCBSL are a natural weakening of Boolean Algebra and orthocomplemented lattices. They are dual to complemented lattices in the sense illustrated by Figure 1. A slight weakening of OCBSL, called de Morgan bisemilattice, has been used to simulate electronic circuits [6, 22]. OCBSL may be applicable in this scenario as well. Moreover, our algorithm can also be adapted to decide, in log-linear time, the word problem for this weaker theory.

To the best of our knowledge, no solution was presented in the past for the word problem for orthocomplemented bisemilattices (OCBSL). Moreover, we are not aware of previous log-linear algorithms for the related previously studied theories either.

1.3 Overview of the Algorithm

It is common to represent a term, like a Boolean formula, as an abstract syntax tree. In such a tree, a node corresponds to either a function symbol, a constant symbol or a variable, and the children of a function node represent the arguments of the function. In general, for a symbol function f, trees $f(x, y)$ and $f(y, x)$ are distinct; the children of a node are stored in a specific order. Commutativity of a function symbol f corresponds to the fact that children of a node labelled by f are instead unordered. Our algorithm thus uses as its starting point a variation of the algorithm of Aho, Hopcroft, and Ullman [14] for tree isomorphism, as it corresponds to deciding equality of two terms modulo commutativity. However, the theory we consider contains many more axioms than merely commutativity. Our approach is to find an equivalent set of reduction rules, themselves understood modulo commutativity, that is suitable to compute a normal form of a given formula with respect to those axioms using the ideas of term rewriting [1]. The interest of tree isomorphism in our approach is two-fold: first, it helps to find application cases of our reduction rules, and second, it compares the two terms of our word problem. In the final algorithm, both aspects are realized simultaneously.



(a) Complemented lattice (b) Orthocomplemented bisemilattice(c) Orthocomplemented lattice

Fig. 1. Bisemilattices satisfying absorption or de Morgan laws.

2 Preliminaries

2.1 Lattices and Bisemilattices

To define and situate our problem, we present a collection of algebraic structures satisfying certain subsets of the laws in tables 1 and 2.

A structure $(S, \wedge)$ that is associative (L1), commutative (L2) and idempotent (L3) is a **semilattice**. A semilattice induces a partial order relation on S defined by $a \le b \iff (a \wedge b) = a$. Indeed, one can verify that $\exists c.(b \wedge c) = a \iff (b \wedge a) = a$, from which transitivity follows. Antisymetry is immediate. In such partially ordered set (poset) S, two elements a and b always have a *greatest lower bound*, or glb, $a \wedge b$. Conversely, a poset such that any two elements have a glb is always a semilattice. A structure $(S, \wedge, 0, 1)$ that satisfies L1, L2, L3, L4, and L5 is a bounded **upper-semilattice**. Equivalently, 1 is the maximum element and 0 the minimum element in the corresponding poset. Similarly, a structure $(S, \sqcup, 0, 1)$ that satisfies L1' to L5' is a bounded **lower-semilattice**. In that case, we write the corresponding ordering relation $\sqsupseteq$. Note that it points in the direction opposite to $\leq$, so that 1 is always the "maximum" element and 0 the "minimum" element. A structure $(S, \land, \sqcup)$ is a **bisemilattice** if $(S, \land)$ is an upper semilattice and $(S, \sqcup)$ a lower semilattice. There are in general no specific laws relating the two semilattices of a bisemilattice. They can be the same semilattice or completely different. If the bisemilattice satisfies the absorption law (L9), then the two semilattices are related in such a way that $a \leq b \iff a \sqsupseteq b$, i.e. the two orders $\leq$ and $\sqsupseteq$ are equal and the structure is called a lattice. A bisemilattice is **consistently bounded** if both semilattices are bounded and if $0_\land = 0_\sqcup = 0$ and $1_\land = 1_\sqcup = 1$, which will be the case in this paper. A structure $(S, \land, \sqcup, \neg, 0, 1)$ that satisfies L1 to L7 and L1' to L7' is called a **complemented bisemilattice**, with complement operation $\neg$. A complemented bisemilattice satisfying de Morgan's Law (L8 and L8') is an **orthocomplemented bisemilattice** and implies $\neg 0 = \neg(\neg 1 \land 0) = \neg\neg 1 \sqcup \neg 0 = 1$. A structure satisfying L1-L9 and L1'-L9' is an **orthocomplemented lattice**. Both de Morgan laws (L8, L8') and absorption laws (L9 and L9') relate the two semilattices, in a way summarised in Figure 1. In bisemilattices, orthocomplementation is (merely) equivalent to $a \leq b \iff \neg b \sqsupseteq \neg a$. Indeed, we have:

$$a \leq b \overset{def}{\iff} a \land b = a \overset{L8'}{\iff} \neg a \sqcup \neg b = \neg a \overset{def}{\iff} \neg b \sqsupseteq \neg a$$

In the presence of L1-L8,L1'-L8', the law of absorption (L9 and L9') is implied by distributivity. In fact, an orthocomplemented bisemilattice with distributivity is a lattice and even a Boolean algebra. In this sense, we can consider orthocomplemented bisemilattices as "Boolean algebra without distributivity".

2.2 Term Rewriting Systems

We next review basics of term rewriting systems. For a more complete treatment, see [1].

Definition 2. *A **term rewriting system** is a list of rewriting rules of the form $e_l = e_r$ with the meaning that the occurence of e_l in a term t can be replaced by e_r. e_l and e_r can contain free variables. To apply the rule, e_l is unified with a subterm of t, and that subterm is replaced by e_r with the same unifier. If applying a rewriting rule to t_1 yields t_2, we say that t_1 reduces to t_2 and write $t_1 \to t_2$. We denote by $\overset{*}{\to}$ the transitive closure of $\to$ and by $\overset{*}{\leftrightarrow}$ its transitive symmetric closure.*

An axiomatic system such as L1-L9, L1'-L9' induces a term rewriting system, interpreting equalities from left to right. In that case $t_1 \overset{*}{\leftrightarrow} t_2$ coincides with the validity of the equality $t_1 = t_2$ in the theory given by the axioms [1, Theorem 3.1.12].

Definition 3. *A term rewriting system is terminating if there exists no infinite chain of reducing terms $t_1 \to t_2 \to t_3 \to \ldots$.*

Fact 1 *If there is a well-founded order $<$ (or, in particular, a measure m) on terms such that $t_1 \to t_2 \implies t_2 < t_1$ (or, in particular $m(t_2) < m(t_1)$) then the term rewriting system is terminating.*

Definition 4. *A term rewriting system is* ***confluent*** *iff: for all* t_1, t_2, t_3, $t_1 \overset{*}{\rightarrow} t_2 \wedge t_1 \rightarrow^* t_3$ *implies* $\exists t_4 . t_2 \overset{*}{\rightarrow} t_4 \wedge t_3 \overset{*}{\rightarrow} t_4$.

Theorem 1 (Church-Rosser Property). *[1, Chapter 2] A term rewriting system is confluent if and only if* $\forall t_1, t_2 . (t_1 \overset{*}{\leftrightarrow} t_2) \implies (\exists t_3 . t_1 \overset{*}{\rightarrow} t_3 \wedge t_2 \overset{*}{\rightarrow} t_3)$.

A terminating and confluent term rewriting system directly implies decidability of the word problem for the underlying structure, as it makes it possible to compute the normal form of two terms to check if they are equivalent. Note that commutativity is not a terminating rewriting rule, but similar results holds if we consider the set of all terms, as well as rewrite rules, modulo commutativity [1, Chapter 11], [28]. To efficiently manipulate terms modulo commutativity and achieve log-linear time, we will employ an algorithm for comparing trees with unordered children.

3 Directed Acyclic Graph Equivalence

The structure of formulas with commutative nodes correspond to the usual mathematical definition of a labelled rooted tree, i.e. an acyclic graph with one distinguished vertex (root) where there is no order on the children of a node. For this reason, we use as our starting point the algorithm of Hopcroft, Ullman and Aho for tree isomorphism [14, Page 84, Example 3.2], which has also been studied subsequently [7, 23].

To account for structure sharing, we further generalize this representation to singly-rooted, labeled, Directed Acyclic Graphs, which we simply call DAGs. Our DAGs generalize rooted directed trees. Any DAG can be transformed into a rooted tree by duplicating subgraphs corresponding to nodes with multiple parent, as in Figure 2. This transformation in general results in an exponential blowup in the number of nodes. Dually, using DAGs instead of trees can exponentially shrink space needed to represent certain terms.

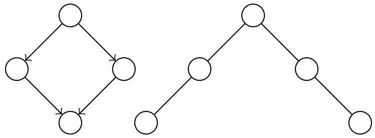


Fig. 2. A DAG and the corresponding Tree

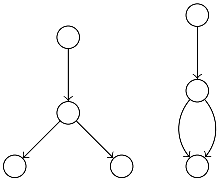


Fig. 3. Two equivalent DAGs with different number of nodes.

Checking for equality between *ordered* trees or DAGs is easy in linear time: we simply recursively check equality between the children of two nodes.

Definition 5. *Two ordered nodes* τ *and* π *with children* $\tau_0, ..., \tau_m$ *and* $\pi_0, ..., \pi_n$ *are equivalent (noted* $\tau \sim \pi$*) iff*

$$label(\tau) = label(\pi),\ m = n \text{ and } \forall i < n, \tau_i \sim \pi_i$$

For unordered trees or DAG, the equivalence checking is less trivial, as the naive algorithm has exponential complexity due to the need to find the adequate permutation.

Definition 6. *Two unordered nodes τ and π with children $\tau_0, ..., \tau_m$ and $\pi_0, ..., \pi_n$ are equivalent (noted $\tau \sim \pi$) iff*

$$label(\tau) = label(\pi),\ m = n \text{ and there exists a permutation } p \text{ s.t. } \forall i < n, \tau_{p(i)} \sim \pi_i$$

For trees, note that this definition of equivalence corresponds exactly to isomorphism. It is known that DAG-isomorphism is GI-complete, so it is conjectured to have complexity greater than PTIME. Fortunately, this does not prevent our solution because our notion of equivalence on DAGs is not the same as isomorphism on DAGs. In particular, two DAGs can be equivalent without having the same number of nodes, i.e. without being isomorphic, as Figure 3 illustrates.

Algorithm 1: Unordered DAG equivalence. The operator ++ is concatenation.

```
   input  : two unordered DAGs $\tau$ and $\pi$
   output: True if $\tau$ and $\pi$ are equivalent, False else.
 1 $codes$ ←HashMap[(String, List[Int]), Int];
 2 $map$ ←HashMap[Node, Int];
 3 $s_\tau : List \leftarrow ReverseTopologicalOrder(\tau)$;
 4 $s_\pi : List \leftarrow ReverseTopologicalOrder(\pi)$;
 5 for $(n{:}Node \text{ in } s_\tau{+}{+}s_\pi)$ do
 6     $l_n \leftarrow [map(c)$ for $c$ in $children(n)]$;
 7     $r_n \leftarrow (label(n), sort(l_n))$;
 8     if $codes$ contains $r$ then
 9         $map(n) \leftarrow codes(r_n)$;
10     else
11         $codes(r_n) \leftarrow codes.size$;
12         $map(n) \leftarrow codes(r_n)$;
13     end
14 end
15 return $map(\tau) == map(\pi)$
```

Algorithm 1 is the generalization of Hopcroft, Ullman and Aho's algorithm. It decides in log-linear time if two labelled (unordered) DAGs are equivalent according to definition 5. The algorithm generalizes straightforwardly to DAGs with a mix of ordered and unordered nodes: if a node is ordered, we skip the sorting operation in line 7.

The algorithm works bottom to top. We first sort the DAG in reverse topological order using, for example, Kahn's algorithm [16]. This way, we explore the DAG starting from a leaf and finishing with the root. It is guaranteed that when we treat a node, all its children have already been treated.

The algorithm recursively assigns codes to the nodes of both DAGs recursively. In the unlabelled case:

- The first node, necessarily a leaf, is assigned the integer 0
- The second node gets assigned 0 if it is a leaf or 1 if it is a parent of the first node
- For any node, the algorithm makes a list of the integer assigned to that node's children and sort it (if the node is commutative). We call this the signature of the node. Then it checks if that list has already been seen. If yes, it assigns to the node the number that has been given to other nodes with the same signature. Otherwise, it assigns a new integer to that node and its signature.

Lemma 1 (Algorithm 1 Correctness). *The codes assigned to any two nodes n and m of s_τ++s_π are equal if and only if $n \sim m$.*

Proof. Let n and m denote any two DAG nodes. By induction on the height of n:

- In the case where n is a leaf, we have $r_n = (label(n), Nil)$. Note that for any node n, $map(n) = codes(r_n)$. Since every time the map *codes* is updated, it is with a completely new number, $codes(r_n) = codes(r_m)$ if and only if $r_n = r_m$, i.e. iff $label(m) = label(n)$ and m has no children (like n).
- In the case where n has children n_i, again $codes(r_n) = codes(r_m)$ if and only if $r_m = r_n$, which is equivalent to $(label(m) = label(n)$ and $sort(l_m) = sort(l_n)$. This means this means there is a permutation of children of n such that $\forall i, codes(n_{p(i)}) = codes(m_i)$. By induction hypothesis, this is equivalent to $\forall i, n_{p(i)} \sim m_i$. Hence we find that $map(n) = map(m)$ if and only if both:

  1. Their labels are equal
  2. There exist a permutation p s.t. $n_{p(i)} \sim m_i$

  i.e n and m have the same code if and only if $n \sim m$.

Corollary 1. *The algorithm returns True if and only if $\tau \sim \pi$.*

Time Complexity. Using Kahn's algorithm, sorting τ and π is done in linear time. Then the loop touches every node a single time. Inside the loop, the first line takes linear time with respect to the number of children of the node and the second line takes log-linear time with respect to the number of children. Since we use HashMaps, the last instructions take effectively constant time (because hash code is computed from the address of the node and not its content).

So for general DAG, the algorithm runs in time at most log-quadratic in the number of nodes. Note however that for DAGs with bounded number of children per node as well as for DAGs with bounded number of parents per nodes, the algorithm is log-linear. In fact, the algorithm is log-linear with respect to the total number of edges in the graph. For this reason, the algorithm is still only log-linear in input size. It also follows that the algorithm is always at most log-linear with respect to the tree or formula underlying the DAG, which may be much larger than the DAG itself. Moreover, there exists cases where the algorithm is log-linear in the number of nodes, but the underlying tree is exponentially larger. The full binary symmetric graph is such an example.

4 Word Problem on Orthocomplemented Bisemilattices

We will use the previous algorithm for DAG equivalence applied to a formula in the language of bisemilattices $(S, \land, \sqcup)$ to account for commutativity (axioms L1, L1'), but we need to combine it with the remaining axioms. From now on we work with axioms L1-L8, L1'-L8' in Table 1. The plan is to express those axioms as reduction rules. Of rules L2-L8 and L2'-L8', all but L8 and L8' reduce the size of the term when applied from left to right, and hence seem suitable as rewrite rules.

It may seem that the simplest way to deal with de Morgan law is to use it (along with double negation elimination) to transform all terms into negation normal form. It happens, however, that doing this causes troubles when trying to detect application cases of rule L7 (complementation). Indeed, consider the following term:

$$f = (a \land b) \sqcup \neg(a \land b)$$

Using complementation it clearly reduces to 1, but pushing into negation-normal form, it would first be transformed to $(a \land b) \sqcup (\neg a \lor \neg b)$. To detect that these two disjuncts are actually opposite requires to recursivly verify that $\neg(a \land b) = (\neg a \lor \neg b)$.

It is actually simpler to apply de Morgan law the following way:

$$x \land y = \neg(\neg x \sqcup \neg y)$$

Instead of removing negations from the formula, we remove one of the binary semilattice operators. (Which one we keep is arbitrary; we chose to keep $\sqcup$.) Now, when we look if rule L7 can be applied to a disjunction node (i.e. two children y and z such that $y = \neg z$), there are two cases: if x is not itself a negation, i.e. it starts with $\sqcup$, we compute $\neg x$ code from the code of x in constant time. If $x = \neg x'$ then $\neg x \sim x'$ so the code of $\neg x$ is simply the code of x', in constant time as well. Hence we obtain the code of all children and their negation and we can sort those codes to look for collisions, all of it in time linear in the number of children.

We now restate the axioms L1-L8 ,L1'-L8' in this updated language in Table 3.

$A1$:	$\bigsqcup(..., x_i, x_j, ...) = \bigsqcup(..., x_j, x_i, ...)$	$A1'$:	$\neg\bigsqcup(\neg x, \neg y) = \neg\bigsqcup(\neg y, \neg x)$
$A2$:	$\bigsqcup(\vec{x}, \bigsqcup(\vec{y})) = \bigsqcup(\vec{x}, \vec{y})$	$A2'$:	$\neg\bigsqcup(\neg\vec{x}, \neg\neg\bigsqcup(\neg\vec{y})) = \neg\bigsqcup(\neg\vec{x}, \neg\vec{y})$
	$\bigsqcup(x) = x$		
$A3$:	$\bigsqcup(x, x, \vec{y}) = \bigsqcup(x, \vec{y})$	$A3'$:	$\neg\bigsqcup(\neg x, \neg x, \neg\vec{y}) = \neg\bigsqcup(\neg x, \neg\vec{y})$
$A4$:	$\bigsqcup(1, \vec{x}) = 1$	$A4'$:	$\neg\bigsqcup(\neg 0, \neg\vec{y}) = 0$
$A5$:	$\bigsqcup(0, \vec{x}) = \bigsqcup(\vec{x})$	$A5'$:	$\neg\bigsqcup(\neg 1, \neg\vec{x}) = \neg\bigsqcup(\neg\vec{x})$
$A6$:	$\neg\neg x = x$		
$A7$:	$\bigsqcup(x, \neg x, \vec{y}) = 1$	$A7'$:	$\neg\bigsqcup(\neg x, \neg\neg x, \neg\vec{y}) = 0$
$A8$:	$\neg\bigsqcup(x_1, ...x_i) = \neg\bigsqcup(\neg\neg x_1, ...\neg\neg x_i)$	$A8'$:	$\neg\neg\bigsqcup(\neg x_1, ...\neg x_i) = \bigsqcup(\neg x_1, ...\neg x_i)$

Table 3. Laws of algebraic structures $(S, \sqcup, 0, 1, \neg)$, equivalent to L1-L8, L1-L8' under de Morgan transformation.

It is straightforward and not surprising that axiom A8 as well as A1'-A8' all follow from axioms A1-A7, so A1-A7 are actually complete for our theory.

4.1 Confluence of the Rewriting System

In our equivalence algorithm, A1 is taken care of by the arbitrary but consistent ordering of the nodes. Axioms A2-A7 form a term rewriting system. Since all those rules reduce the size of the term, the system is terminating in a number of steps linear in the size of the term. We will next show that it is confluent. We will thus obtain the existence of a normal form for every term, and will finally show how our algorithm computes that normal form.

Definition 7. *Consider a pair of reduction rules $l_0 \to r_0$ and $l_1 \to r_1$ with disjoint sets of free variables such that $l_0 = D[s]$, s is not a variable and σ is the most general unifier of $\sigma s = \sigma l_1$. Then $(\sigma r_0, (\sigma D)[\sigma r_2])$ is called a* critical pair.

Informally, a critical pair is a most general pair of term (with respect to unification) (t_1, t_2) such that for some t_0, $t_0 \to t_1$ and $t_0 \to t_2$ via two "overlapping" rules. They are found by matching the left-hand side of a rule with a non-variable subterm of the same or another rule.

Example 1 (Critical Pairs).

1. Matching left-hand side of A6 with the subterm $\neg x$ of rule A7, we obtain the pair

$$(1, \bigsqcup(\neg x, x, \vec{y}))$$

   which arises from reducing the term $t_0 = \bigsqcup(\neg x, \neg\neg x, \vec{y})$ in two different ways.
2. Matching left-hand sides of A2 and A7 gives

$$(\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})), 1)$$

   which arise from reducing $\bigsqcup(\vec{x}, \bigsqcup(\vec{y}), \neg \bigsqcup(\vec{y}))$ using A2 or A7.
3. Matching left-hand sides of A5 and A7 gives

$$(\neg 0, 1)$$

   which arise from reducing $0 \sqcup \neg 0$ in two different ways.

Proposition 1 ([1, Chapter 6]). *A terminating term rewriting system is confluent if and only if all critical pairs (t_1, t_2) are joinable i.e. $\exists t_3.\ t_1 \overset{*}{\to} t_3 \wedge t_2 \overset{*}{\to} t_3$.*

In the first of the previous examples, the pair is clearly joinable by commutativity and a single application of rule A7 itself. The second example is more interesting. Observe that $\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = 1$ is a consequence of our axiom, but the left part cannot be reduced to 1 in general in our system. To solve this problem we need to add the rule A9: $\bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = 1$. Similarly, the third example forces us to add A10: $\neg 0 = 1$ to our set of rules. From A10 and A6 we then find the expected critical pair A11: $\neg 1 = 0$.

$$\begin{aligned}
A1 &: \bigsqcup(..., x_i, x_j, ...) = \bigsqcup(..., x_j, x_i, ...) \\
A2 &: \bigsqcup(\vec{x}, \bigsqcup(\vec{y})) = \bigsqcup(\vec{x}, \vec{y}) \\
&\quad \bigsqcup(x) = x \\
A3 &: \bigsqcup(x, x, \vec{y}) = \bigsqcup(x, \vec{y}) \\
A4 &: \bigsqcup(1, \vec{x}) = 1 \\
A5 &: \bigsqcup(0, \vec{x}) = \bigsqcup(\vec{x}) \\
A6 &: \neg\neg x = x \\
A7 &: \bigsqcup(x, \neg x, \vec{y}) = 1 \\
A9 &: \bigsqcup(\vec{x}, \vec{y}, \neg \bigsqcup(\vec{y})) = 1 \\
A10 &: \neg 0 = 1 \\
A11 &: \neg 1 = 0
\end{aligned}$$

Table 4. Terminating and confluent set of rewrite rules equivalent to L1-L8, L1'-L8'

4.2 Complete Terminating Confluent Rewrite System

The analysis of all possible pairs of rules to find all critical pairs is straightforward. It turns out that the A9, A10 and A11 are the only rules we need to add to our system to obtain confluence. We have checked the complete list of critical pairs for rules A2-A11 (we omit the details due to lack of space). All those pairs are joinable, i.e. reduce to the same term, which implies, by Proposition 1, that the system is confluent. Table 4 shows the complete set of reduction rules (as well as commutativity).

Since the system A2-A11 considered over the language $(S, \bigsqcup, \neg, 0, 1)$ modulo commutativity of $\bigsqcup$ is terminating and confluent, it implies the existence of a normal form reduction. For any term t, we note its normal form $t{\downarrow}$. In particular, for any two terms t_1 and t_2, we have $t_1 = t_2$ in our theory iff $t_1 \overset{*}{\leftrightarrow} t_2$ iff $t_1{\downarrow}$ and $t_2{\downarrow}$ are equivalent terms modulo commutativity. We finally reach our conclusion: an algorithm that computes the normal form (modulo commutativity) of any term gives a decision procedure for the word problem for orthocomplemented bisemilattices.

5 Algorithm and Complexity

The rewriting system readily gives us a quadratic algorithm. Indeed, using our base algorithm for DAG equivalence, we can check, in linear time, for application cases of any one of rewriting rules A2-A11 of Table 4 modulo commutativity. Since a term can only be reduced up to n times, the total time spent before finding the normal form of a term is at most quadratic. It is however possible to find the normal form of a term in a single pass of our equivalence algorithm, resulting in a more efficient algorithm.

5.1 Combining Rewrite Rules and Tree Isomorphism

We give an overview on how to combine rules A2-A7, A9, A10, A11 within the tree isomorphism algorithm, which we present using Scala-like [1] pseudo code in Figure 7.

[1] https://www.scala-lang.org/

For conciseness, we omit the dynamic programming optimizations allowed by structure sharing in DAGs (which would store the normal form and additionally check if a node was already processed.) For each rule, we indicate the most relevant lines of the algorithm in Figure 7.

A2 (Associativity, Lines 10, 20, 32, 42) When analysing a $\bigsqcup$ node, after the recursive call, find all children that are $\bigsqcup$ themselves and replace them by their own children. This is simple enough to implement but there is actually a caveat with this in term of complexity. We will come back to it in section 5.

A3 (Idempotence, Lines 8, 31, 35) This corresponds to the fact that we eliminate duplicate children in disjunctions. When reaching a $\bigsqcup$ node, after having sorted the code of its children, remove all duplicates before computing its own code.

A4, A5 (Bounds, Lines 8, 31, 35, 11, 36) To account for those axioms, we reserve a special code for the nodes 1 and 0. For A4, when we reach some $\bigsqcup$ node, if it has 1 as one of its children, we accordingly replace the whole node by 1. For A5, we just remove nodes with the same codes as 0 from the parent node before computing its own code.

A6 (Involution, Lines 17, 22) When reaching a negation node, if its child is itself a negation node, replace the parent node by its grandchildren before assigning it a code.

A7 (Complement, Lines 11, 36) As explained earlier, our representation of nodes let us do the following to detect cases of A7: First remember that we already applied double negation elimination, so that two "opposite" nodes cannot both start with a negation. Then we can simply separate the children between negated and non-negated (after the recursive call), sort them using their assigned code and look for collisions.

A9 (Also Complement, Lines 11, 36) This rule is slightly more tricky to apply. When analysing a $\bigsqcup$ node x, after computing the code of all children of x, find all children of the form $\neg\bigsqcup$. For every such node, take the set of its own children and verify if it is a subset of the set of all children of x. If yes, then rule A9 applies. Said otherwise, we look for collisions between grandchildren (through a negation) and children of every $\bigsqcup$ node.

A10, A11 (Identities, Lines 17, 26) These rules are simple. In a $\neg$ node, if its child has the same code as 0 (resp 1), assign code 1 (resp 0) to the negated node.

5.2 Case of Quadratic Runtime for the Basic Algorithm

All the rules we introduced in the previous section into Algorithm 1 take time (log)linear in the number of children of a node to apply, which is not more than the time we spent in the DAG/tree isomorphism algorithm. For A3, checking for duplicates is done in linear time in an ordered data structure. A4 and A5 (Bounds) consist in searching for specific values, which take logarithmic time in the size of the list. A6 (Involution) takes constant time. A7 (Complement) is detected by finding a collision between two separate ordered lists, also easily done in (log) linear time. A9 (Also complement) consists in verifying if grandchildren of a node are also children, and since children are sorted this takes log-linear time in the number of grandchildren. Since a node is the grandchild of only one other node, the same computation as in the original algorithm holds. A10 and A11 take constant time. Hence, the total time complexity is $\mathcal{O}(n\log(n))$, as in the algorithm for tree isomorphism.

As stated in Section 3 regarding the algorithm for DAG equivalence whose complexity we aim to preserve, the time complexity analysis crucially relies on the fact that in a tree, a node is never the child (or grandchild) of more than one node during the execution. However, this is generally not true in the presence of associativity. Indeed consider the term represented in Figure 4. The 5th $\bigsqcup$ has 2 children, but after applying A2, the 4th has 3 children, the 3rd has 4 children and so on. On the generalization of such an example, since an x_i is the child of all higher $\bigsqcup$, our key property does not hold and the algorithm runtime would be quadratic. Of course, such a simple counterexample is easily solved by applying a leading pass of associativity reduction before actually running the whole algorithm. It turns out however that it is not sufficient, since cases of associativity can appear after the application of the other A-rules.

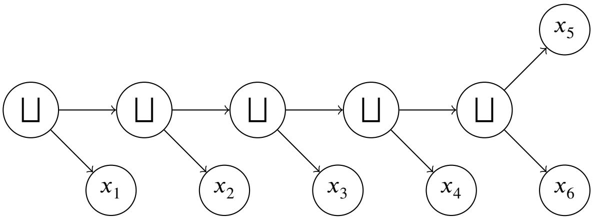


Fig. 4. A term with quadratic runtime

In fact, there is only one rule that can creates case of rule A2, and this rule is A6 (Involution). The remaining rules whose right-hand side can start with a $\bigsqcup$ have their left-hand side already starting with $\bigsqcup$. It may seem simple enough to also apply double negation elimination in a leading pass, but unfortunately, cases of A6 can also be created from other rules. It is easy to see, for similar reasons, that only the application of A2b ($\bigsqcup(x) = x$) can create such cases. And unfortunately, such cases of A2b can arise from rules A3 and A5 which can only be detected using the full algorithm. To summarize, the typical problematic case is depicted in Figure 5. This term is clearly equivalent to $\bigsqcup(x_1, x_2, x_3, x_4)$, but to detect it we must first find that z_1 and z_2 are equivalent to 0, so we cannot simply solve it with an early pass.

5.3 Final Log-Linear Time Algorithm

Fortunately, we can solve this problem at a logarithmic-only price. Observe that if we are able to detect early nodes which would cancel to 0, the problem would not exist: When analysing a node, we would first call the algorithm on all subnodes equivalent to

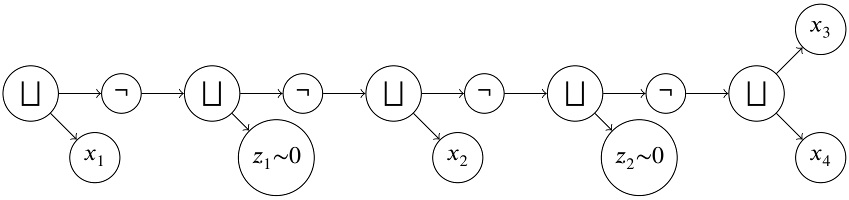


Fig. 5. A non-trivial term with quadratic runtime

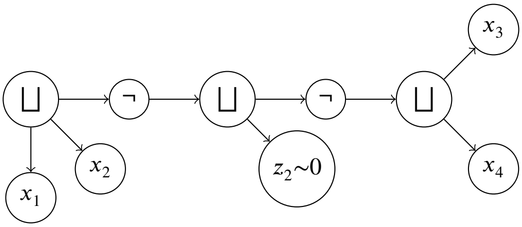


Fig. 6. the term of Figure 5 during the algorithm's execution

0, remove them and then when there is a single children left, remove the trivial disjunct, the double negation and the successive disjunction (as in Figure 5) before doing the recursive call on the unique nontrivial child. However, we of course cannot know in advance which child will be equivalent to 0.

Moreover note (still using Figure 5) that if the z-child is as large as the non-trivial node, then even if we do the "useless" work, we at least obtain that the size a tree is divided by two, and hence the potential depth of the tree as well. By standard complexity analysis, the time penalty would only be a logarithmic factor.

The previous analysis suggests the following solution, reflected in Figure 7 lines 28-29. When analysing a node, make recursive calls on children in order of their size, starting with the smallest up to the second biggest. If any of those children are non-zero, proceed as normal. If all (but possibly the last) children are equivalent to zero, then replace the current node by its biggest (and at this point non-analyzed) child, i.e. apply second half of rule A2 (associativity). If applicable, apply double negation elimination and associativity as well before continuing the recursive call.

We illustrate this on the example of Figure 5. Consider the algorithm when reaching the second $\bigsqcup$ node. There are two cases:

1. Suppose z_1 is a smaller tree than the non-trivial child. In this case the algorithm will compute a code for z_1, find that it is 0 and delete it. Then the non trivial node is a single child so the whole disjunction is removed. Hence, the double negation can be removed and the two consecutive disjunction of x_1 and x_2 merged, obtaining the term illustrated in Figure 6. In particular we did not compute a code for the two deleted $\bigsqcup$ nodes, which is exactly what we wanted for our initial analysis.
2. Suppose z_1 is larger tree than the non-trivial child. In this case, we would first recursively compute the code of the non-trivial child and then detect that $z_1 \sim 0$. We

indeed computed the code of the disjunction that contains x_2 when it was unnecessary since we apply associativity anyway. This "useless" work consists in sorting and applying axioms to the true children of the node (in this case x_2, x_3 and x_4) and takes time quasilinear in the number of such children. In particular, it is bounded by the size of the subtree itself and we know it is the smallest of the two.

Analogous situation can arise from the use of rule A3 (idempotence), but here trivially the two subtrees must have the same number of (real) subnodes, so that the same reasoning holds.

Denote by $|n|$ the size of a node, i.e. the number of descendants of n. We compute the penalty of useless work we incur by computing children of a node n in the wrong order, i.e. by computing a non-0 child n_w when all other are 0. n_w cannot be the largest child of n for otherwise we would have found that all other children are 0 before needing to compute n_w. Hence $|n_w| \leq |n|/2$. It follows that the total amount of useless work is bounded by $\log(|n|) \cdot W(n)$, where

$$W(n) \leq |n|/2 + \sum_i W(n_i) \quad \text{for} \quad \sum_i |n_i| < |n|.$$

It is clear that $W(n)$ is maximized when n has exactly two children of equal size:

$$W(n) \leq |n|/2 + 2 \cdot W(n/2)$$

By observing that we can divide n by 2 only $\log(n)$ times,

$$W(n) \leq \sum_{m=1}^{\log(n)} 2^m \cdot |n|/2^m$$

so we obtain $W(n) = \mathcal{O}(|n| \log(|n|))$ and hence the total runtime is $\mathcal{O}(n(\log n)^2)$.

6 Conclusion

We have described a decision procedure with log-linear time complexity for the word problem on orthocomplemented bisemilattices. This algorithm can also be simplified to apply to weaker theories. Dually, we believe it can be generalized to decide some stronger theories (still weaker than Boolean algebras) efficiently. While the word problem for orthocomplemented *lattices* was known to be in PTIME [15] and as such the membership of orthocomplemented *bisemilattices* in PTIME may not come as a surprise, this is, to the best of our knowledge, the first time that this result has been explicitly stated, and the first time that an algorithm with such low log-linear complexity was proposed for this or a related problem. The algorithm has not only low complexity but, according to our experience, is easy to implement. It can be used as an approximation for Boolean algebra equivalence, and we plan to use it as the basis of a kernel for a proof assistant. We also envision possible uses of the algorithm in SMT and SAT solvers. The algorithm is able to detect many natural and non-trivial cases of equivalence even on formulas that may be too large for existing solvers to deal with, so it may also complement an existing repertoire of subroutines used in more complex reasoning tasks. For a minimal working implementation in Scala closely following Figure 7, see https://github.com/epfl-lara/OCBSL.

```
def equivalentTrees(tau: Term, pi: Term): Boolean =
    val codesSig: HashMap[(String, List[Int]), Int] = Empty
    codesSig.update(("zero", Nil), 0); codesSig.update(("one", Nil), 1)
    val codesNodes: HashMap[Term, Int] = Empty
    def updateCodes(sig: (String, List[Int]), n: Node): Unit = ... // codesSig, codesNodes
    def bool2const(b:Boolean): String = if b then "one" else "zero"
    def rootCode(n: Term): Int =
        val L = pDisj(n, Nil).map(codesNodes).sorted.filter(_≠ 0).distinct
        if L.isEmpty then ("zero", Nil), n)
        else if L.length == 1 then codesNodes.update(n, L.head)
        else if L.contains(1) or checkForContradiction(L) then updateCodes(("one", Nil), n)
        else updateCodes(("or", L), n)
        codesNodes(n)
    def pDisj(n:Node, acc:List[Node]): List[Node] = n match
        case Variable(id) ⇒ updateCodes((id.toString, Nil), n); return n :: acc
        case Literal(b) ⇒ updateCodes((bool2const(b), Nil), n); return n :: acc
        case Negation(child) ⇒ pNeg(child, n, acc)
        case Disjunction(children) ⇒ children.foldleft(acc)(pDisj)
    def pNeg(n:Node, parent:Node, acc:List[Node]): List[Node] = n match // under negation
        case Negation(child) ⇒ pDisj(child, acc)
        case Variable(id) ⇒ updateCodes((id.toString, Nil), n)
                            updateCodes(("neg", List(codesNodes(n))), parent)
                            List(parent)::acc
        case Literal(b) ⇒ updateCodes((bool2const(b), Nil), n)
                          updateCodes((bool2const(!b), Nil), parent)
                          List(parent)::acc
        case Disjunction(children) ⇒
            val r0 = orderBySize(children)
            val r1 = r0.tail.foldLeft(Nil)(pDisj)
            val r2 = r1.map(codesNodes).sorted.filter(_≠ 0).distinct
            if isEmpty(r2) then pNeg(r0.head, parent, acc)
            else val s1 = pDisj(r0.head, r1)
                 val s2 = s1 zip (s1 map codesNodes)
                 val s3 = s2.sorted.filter(_≠ 0).distinct // all wrt. 2nd element
                 if s3.contains(1) or checkForContradiction(s3)
                 then updateCodes(("one", Nil), n); updateCodes(("zero", Nil), parent)
                      List(parent)::acc
                 else if isEmpty(s3) then updateCodes(("zero", Nil), n)
                                          updateCodes(("one", Nil), parent)
                                          List(parent)::acc
                 else if s3.length == 1 then pNeg(s3.head._1, parent, acc)
                 else updateCodes(("or", s3 map (_._2)), n)
                      updateCodes(("neg", List(n)), parent)
                      List(parent)::acc
    return rootCode(tau) == rootCode(pi)
```

Fig. 7. Final algorithm. distinctBy runs in log-linear time. checkForContradiction detects application cases of A7 and A9 (Complement). Maintenance of size field used by orderBySize elided.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998). https://doi.org/10.1017/CBO9781139172752
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 171–177. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
3. Basin, D.A., Ganzinger, H.: Automated complexity analysis based on ordered resolution. J. ACM **48**(1), 70–109 (2001). https://doi.org/10.1145/363647.363681
4. Bruns, G.: Free Ortholattices. Canadian Journal of Mathematics **28**(5), 977–985 (Oct 1976). https://doi.org/10.4153/CJM-1976-095-6
5. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, vol. 6015, pp. 150–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_12
6. Brzozowski, J.: De Morgan bisemilattices. In: Proceedings 30th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2000). pp. 173–178 (May 2000). https://doi.org/10.1109/ISMVL.2000.848616
7. Buss, S.R.: Alogtime algorithms for tree isomorphism, comparison, and canonization. In: Gottlob, G., Leitsch, A., Mundici, D. (eds.) Computational Logic and Proof Theory. pp. 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
8. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. p. 151–158. STOC '71, Association for Computing Machinery, New York, NY, USA (1971). https://doi.org/10.1145/800157.805047
9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (Jul 1962). https://doi.org/10.1145/368273.368557
10. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Alur, R., Peled, D.A. (eds.) Computer Aided Verification, vol. 3114, pp. 175–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_14
11. Gentzen, G.: Untersuchungen über das logische schließen. I. Mathematische Zeitschrift **39**, 176–210 (1935)
12. Hamza, J., Voirol, N., Kunčak, V.: System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang **3** (November 2019). https://doi.org/https://doi.org/10.1145/3360592
13. Harrison, J.: HOL Light: An Overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics, vol. 5674, pp. 60–66. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
14. Hopcroft, J., UIlman, J., Aho, A.: The Design And Analysis Of Computer Algorithms. Addison-Wesley (1974)
15. Hunt, H. B., I., Rosenkrantz, D.J., Bloniarz, P.A.: On the Computational Complexity of Algebra on Lattices. SIAM Journal on Computing **16**(1), 129–148 (Feb 1987). https://doi.org/10.1137/0216011
16. Kahn, A.B.: Topological sorting of large networks. Communications of the ACM **5**(11), 558–562 (Nov 1962). https://doi.org/10.1145/368996.369025

17. Kalmbach, G.: Orthomodular Lattices. Academic Press Inc, London ; New York (Mar 1983)
18. Krajíček, J.: Proof Complexity. Encyclopedia of Mathematics and Its Appplications, Vol.170, Cambridge University Press (2019)
19. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer (2016)
20. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (February 2007), http://hdl.handle.net/1721.1/38533
21. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments. pp. 170–190. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54108-7_9
22. Lewis, D.W.: Hazard detection by a quinary simulation of logic devices with bounded propagation delays. In: Proceedings of the 9th Design Automation Workshop. pp. 157–164. DAC ’72, Association for Computing Machinery, New York, NY, USA (Jun 1972). https://doi.org/10.1145/800153.804941
23. Lindell, S.: A logspace algorithm for tree canonization (extended abstract). In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing. p. 400–404. STOC ’92, Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/129712.129750
24. McAllester, D.A.: Automatic recognition of tractability in inference relations. Journal of the ACM **40**(2), 284–303 (1993). https://doi.org/10.1145/151261.151265
25. Meinander, A.: A solution of the uniform word problem for ortholattices. Mathematical Structures in Computer Science **20**(4), 625–638 (Aug 2010). https://doi.org/10.1017/S0960129510000125
26. Merz, S., Vanzetto, H.: Automatic Verification of TLA + Proof Obligations with SMT Solvers. In: Bjørner, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 289–303. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_23
27. Naumowicz, A., Korniłowicz, A.: A brief overview of mizar. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 67–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_5
28. Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. J. ACM **28**(2), 233–264 (Apr 1981). https://doi.org/10.1145/322248.322251
29. Pudlák, P.: The Lengths of Proofs. In: Studies in Logic and the Foundations of Mathematics, vol. 137, pp. 547–637. Elsevier (1998). https://doi.org/10.1016/S0049-237X(98)80023-2
30. Tschannen, J., Furia, C.A., Nordio, M., Polikarpova, N.: Autoproof: Auto-active functional verification of object-oriented programs. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 566–580. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_53
31. Urquhart, A.: Hard examples for resolution. J. ACM **34**(1), 209–219 (Jan 1987). https://doi.org/10.1145/7531.8928
32. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Theorem Proving in Higher Order Logics. pp. 33–38. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_7
33. Whitman, P.M.: Free Lattices. Annals of Mathematics **42**(1), 325–330 (1941). https://doi.org/10.2307/1969001
34. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI) (2008). https://doi.org/10.1145/1375581.1375624, see also [20]
35. Zee, K., Kuncak, V., Rinard, M.: An integrated proof language for imperative programs. In: ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI) (2009). https://doi.org/10.1145/1543135.1542514