Formula Normalizations in Verification

Simon Guilloud(✉), Mario Bucev, Dragana Milovančević, and Viktor Kunčak

School of Computer and Communication Sciences, EPFL, Station 14, 1015 Lausanne, Switzerland
{simon.guilloud,mario.bucev,dragana.milovancevic,viktor.kuncak}@epfl.ch

Abstract. We apply and evaluate polynomial-time algorithms to compute two different normal forms of propositional formulas arising in verification. One of the normal form algorithms is presented for the first time. The algorithms compute normal forms and solve the word problem for two different subtheories of Boolean algebra: orthocomplemented bisemilattice (OCBSL) and ortholattice (OL). Equality of normal forms decides the word problem and is a sufficient (but not necessary) check for equivalence of propositional formulas. Our first contribution is a quadratic-time OL normal form algorithm, which induces a coarser equivalence than the OCBSL normal form and is thus a more precise approximation of propositional equivalence. The algorithm is efficient even when the input formula is represented as a directed acyclic graph. Our second contribution is the evaluation of OCBSL and OL normal forms as part of a verification condition cache of the Stainless verifier for Scala. The results show that both normalization algorithms substantially increase the cache hit ratio and improve the ability to prove verification conditions by simplification alone. To gain further insights, we also compare the algorithms on hardware circuit benchmarks, showing that normalization reduces circuit size and works well in the presence of sharing.

1 Introduction

Algorithms and techniques to solve and reduce formulas in propositional logic (and its generalizations) are a major field of study. They have prime relevance in SAT and SMT solving algorithms [2,8,31], in optimization of logical circuit size in hardware [25], in interactive theorem proving where propositional variables can represent assumptions and conclusions of theorems [23,35,43], for decision procedures in automated theorem proving [13,26,37,41,42], and in every subfield of formal verification in general [27]. The propositional problem of satisfiability is NP-complete, whereas validity and equivalence are coNP-complete. While heuristic techniques give useful results in practice, in this paper we investigate guaranteed worst-case polynomial-time deterministic algorithms. Such algorithms can serve as building blocks of more complex functionality, without creating an unpredictable dependency.

Recently, researchers proposed the use of certain non-distributive complemented lattice-like structures to compute normal forms of formulas [20]. These results appear to have a practical potential, but they have not been experimentally evaluated. Moreover, the proposed completeness characterization is in

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13966, pp. 398–422, 2023.
https://doi.org/10.1007/978-3-031-37709-9_19

terms of "orthocomplemented bisemilattices" (OCBSL), which have a number of counterintuitive properties. For example, the structure is not a lattice and does not satisfy the absorption laws $x \wedge (x \vee y) = x$ and $x \vee (x \wedge y) = x$. As a consequence, there is no natural semantic ordering on formulas corresponding to implication, with $x \wedge y = x$ and $x \vee y = y$ inducing two different relations.

Inspired by these limitations, we revisit results on *lattices*, which are much better behaving structures. We strengthen the OCBSL structure with the absorption law to consider the class of *ortholattices*, as summarized in Table 1. Ortholattices (OL) have a natural partial order for which $\wedge, \vee$ act as the greatest lower bound and the least upper bound. They also satisfy de Morgan's law, allowing the elimination of one of the connectives in terms of the other two. On the other hand, ortholattices do not, in general, satisfy the distributivity law, which sets them apart from Boolean algebras.

We present a new algorithm that computes a normal form for OL in quadratic time. The normal form is strictly stronger than the one for OCBSL: there are terms in the language $\{\wedge, \vee, \neg\}$ that are distinct in OCBSL, but are equal in OL. Checking equality of OL normal forms thus more precisely approximates propositional formula equivalence. Both normal forms can be thought of as strengthening of the negation normal form.

Table 1. Laws of algebraic structures with signature $(S, \wedge, \vee, 0, 1, \neg)$. Structures satisfying laws L1–L8 and L1'–L8' were called *orthocomplemented bisemilattices* (OCBSL) in [20]. Those OCBSL that additionally satisfy L9 and L9' are *ortholattices* (OL).

L1:	$x \vee y = y \vee x$	L1':	$x \wedge y = y \wedge x$
L2:	$x \vee (y \vee z) = (x \vee y) \vee z$	L2':	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$
L3:	$x \vee x = x$	L3':	$x \wedge x = x$
L4:	$x \vee 1 = 1$	L4':	$x \wedge 0 = 0$
L5:	$x \vee 0 = x$	L5':	$x \wedge 1 = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \vee \neg x = 1$	L7':	$x \wedge \neg x = 0$
L8:	$\neg(x \vee y) = \neg x \wedge \neg y$	L8':	$\neg(x \wedge y) = \neg x \vee \neg y$
L9:	$x \vee (x \wedge y) = x$	L9':	$x \wedge (x \vee y) = x$

Example 1. Consider the formula $x \wedge (y \vee z)$. An OCBSL algorithm finds it equivalent to

$$x \wedge \neg(\neg y \wedge \neg z) \wedge x$$

but it will consider these two formulas non-equivalent to

$$x \wedge (u \vee x) \wedge (y \vee z)$$

The OL algorithm will identify the equivalence of all three formulas, thanks to the laws (L9, L9'). It will nonetheless consider them non-equivalent to

$$(x \wedge y) \vee (x \wedge z)$$

which a complete but exponential worst-case time algorithm for Boolean algebra equalities, such as one implemented in SAT solvers, will identify as equivalent.

A major practical question is the usefulness of such $O(n\log(n)^2)$ (OCBSL) and $O(n^2)$ (OL) algorithms in verification. Are they as predictably efficient as the theoretical analysis suggests? What benefits do they provide as a component of verification tools? To answer these questions, we implement both OCBSL and OL algorithms on directed acyclic graph representations of formulas. We deploy the algorithms in tools that manipulate formulas, most notably verification conditions in a program verifier, as well as combinational Boolean circuits.

Contributions. We make the following contributions:

- We present the first algorithm computing a *normal form* of *ortholattice* (OL) terms. The algorithm preserves the quadratic time for the decision problem of equality in free ortholattices [7]. The quadratic time remains even when the formula is given in a shared (DAG) representation.
- We implement and experimentally evaluate both the new algorithm for the OL normal form and a previously known (weaker) OCBSL algorithm (shown to run in quasilinear time). Our evaluation (Sect. 6) includes:
  - behavior on randomly generated formulas;
  - scalability evaluation on normalizing circuits of size up to 10^8 gates;
  - normalization for simplification and caching of verification conditions when using the Stainless verifier, with both hard benchmarks (such as a compression algorithm) and collections of student submissions for programming assignments.

  We show that OCBSL and OL both have notable potential in practice.

1.1 Related Work

The overarching perspective behind our paper is understanding polynomial-time normalization of boolean algebra terms. Given (co)NP-hardness of problems related to Boolean algebras, we look at subtheories given by a subset of Boolean algebra axioms, including structures such as lattices. Lattices themselves have many uses in program abstraction, including abstract interpretation [11] and model checking [14,18]. The theory of the word problem for lattices has been studied already by Whitman [44], who proposed a quadratic solution for the word problem for free *lattices*. Lattices alone do not incorporate the notion of a complement (negation). Whitman's algorithm has been adapted and extended to finitely presented lattices [17] and other variants, and then to free ortholattices by Bruns [7]. We extend this last result to not only decide equality, but also to compute a *normal form* for free ortholattices and to *circuit* (DAG) representation of terms. An efficient normal form does not follow from an efficient equivalence checking, as there are many formulas in the same equivalence class. Normal form is particularly useful in applications such as formula caching, which we evaluate in Sect. 6. For a weaker theory of OCBSL, the normal form algorithm was introduced in [20], without any experimental evaluation. The theory of ortholattices, even if it adds only one more axiom, is notably stronger and better understood. The underlying lattice structure makes it possible to draw on the body of work on using lattices to abstract systems and enable algorithmic verification. The support for graphs (instead of only terms) as a representation is of immense practical relevance, because expanding circuits into trees without the use of auxiliary variables creates structures of astronomical size (Sect. 6).

A notable normal form that decides equality for propositional logic (thus also accounting for the distributivity law) are reduced ordered binary decision diagrams (ROBDDs) [9]. ROBDDs are of great importance in verification, but can be exponential in the size of the initial formula. Circuit synthesis and verification tools such as ABC [6] use SAT solvers to optimize sub-circuits [45], which is an approach to choose a trade-off between the completeness and cost of exponential-time algorithm. Boolean algebras are in correspondence with boolean rings, which replace the least upper bound operation $\vee$ with the symmetric difference $\oplus$ (defined as $(p \wedge \neg q) \vee (\neg p \wedge q)$ and satisfying $x \oplus x = 0$, corresponding to the *exclusive or* in the two-element case). There have been proposals to exploit the boolean ring structure in verification [12]. Polynomials over rings can also be used to obtain a normal form, but the polynomial canonical forms that we are aware of are exponential-sized. SMT solvers [2,34] extend SAT solvers, which makes them worst-case exponential (at best). We expect that our approach and algorithms could be used for preprocessing or representation, especially in non-clausal variants of SMT solvers [24,39]. In our evaluation, we apply formula normal forms to the problem of caching of verification conditions. Caching is often used in verification tools, including Dafny [28] and Stainless [22]. Our caching works on formulas and preserves the API of a constraint solver. It is thus fine grained and can be added to a program verifier or analyzer, regardless of whether it uses any other, domain-specific, forms of caching [29].

2 Preliminaries

We present definitions and results necessary for the presentation of the ortholattice (OL) normal form algorithm. We assume familiarity with term rewriting and representation of terms as trees and directed acyclic graphs [15,20]. We use first-order logic with equality (whose symbol is =). We write $A \models F$ to mean that a first-order logic formula F is a consequence of (thus provable from) the set of formulas A.

Definition 1 (Terms). *Consider an algebraic signature S. We use $\mathcal{T}_S(X)$ to denote the set of terms over S with variables in X (typically an arbitrary countably infinite set, unless specified otherwise). Terms are constructed inductively as trees. Leaves are labeled with constant symbols or variables. Nodes are labeled with function symbols. If the label of a node is a commutative function, the children of the node are considered as a set (non-ordered) and otherwise as a list (ordered). We assume that commutative symbols are denoted as such in the signature.*

Definition 2 (The Word Problem). *Consider an algebraic signature S and a set of equational axioms E on S (for example the theory of lattices or ortholattices). The word problem for E is the problem of determining, given two terms t_1 and $t_2 \in \mathcal{T}_S(X)$, whether $E \models t_1 = t_2$.*

Definition 3 (Normal Form). *Consider an algebraic signature S and a set of equational axioms E on S. A function $f : \mathcal{T}_S(X) \mapsto \mathcal{T}_S(X)$* produces a normal form for *E iff: $\forall t_1, t_2 \in \mathcal{T}_S(X)$, $E \models t_1 = t_2$ is equivalent to $f(t_1) = f(t_2)$.*

For Z an arbitrary non-empty set and $(\sim) \subseteq Z \times Z$ an equivalence relation on X we use a common notation: if $x \in Z$ then $[x]_\sim = \{y \in Z \mid x \sim y\}$. Let $Z_{/\sim} = \{[x]_\sim \mid x \in Z\}$.

We now briefly review key concepts of free algebras. Let $\mathcal{S}$ be a signature and E be an equational theory over this signature. Consider an equivalence relation on terms $p \sim_E q \iff (E \models p = q)$, and note that $\mathcal{T}_S(X)_{/\sim_E}$ is itself an E-algebra. A **freely generated E-algebra**, denoted $F_E(X)$, is an algebra generated by variables in X and isomorphic to $\mathcal{T}_S(X)_{/\sim_E}$, i.e. in which *only* the laws of all E-algebra hold. There is always a homomorphism from a freely generated E-algebra to any other E-algebra over X.

The set of terms $\mathcal{T}_S(X)$ is also called the **term algebra** over S. It is the algebra of all terms that contains no identity other than syntactic equality. Given a (possibly free) algebra A over S and generated by X, there is a natural homomorphism κ_A, in a sense an evaluation function, from $\mathcal{T}_S(X)$ to A. The word problem for a theory E then consists in, given $p, q \in \mathcal{T}_S(X)$, deciding if $E \models p = q$, that is, $\kappa_{F_E}(t_1) = \kappa_{F_E}(t_2)$.

In the sequel, we continue to use $=$ to denote the equality symbol inside formulas as well as the usual identity of mathematical objects. We use $==$ to specifically denote the computer-performed operation of *structural* equality on trees and sets, whereas $===$ denotes *reference* equality of objects, meaning that $a === b$ if and only if a and b denote the same object in memory. The distinction between $==$ and $===$ is relevant because $==$ is a larger relation but may take linear or worse time to compute, whereas we assume $===$ is constant time.

Lattices. Lattices [4] are well-studied structures with signature $(\wedge, \vee)$ satisfying laws L1–L3, L9, L1'–L3' and L9' from Table 1. In particular, they do not have a complement operation, $\neg$, in the signature. Lattices can also be viewed as a special kind of partially ordered sets with an order relation defined by $(a \leq b) \iff (a \wedge b = a)$, where the last condition is also equivalent to $(a \vee b = b)$, given the axioms of lattices. When applied to two-element Boolean algebras, this order relation corresponds to logical implication in propositional logic. A **bounded lattice** is a lattice with maximal and minimal elements 1 and 0. The word problem for *lattices* has been solved by Whitman [44] through an algorithm to decide the $\leq$ relation and is based on the following properties of free lattices:

$$\begin{array}{lll}
(1)\ s_1 \vee \ldots \vee s_m \leq t & \iff & \forall i. s_i \leq t \\
(2)\ s \leq t_1 \wedge \ldots \wedge t_n & \iff & \forall j. s \leq t_j \\
(3)\ s_1 \wedge \ldots \wedge s_m \leq y & \iff & \exists i. s_i \leq y \\
(4)\ x \leq t_1 \vee \ldots \vee t_n & \iff & \exists j. x \leq t_j
\end{array}$$

$$\begin{array}{c}
s \leq t \iff (\exists i. s_i \leq t) \vee (\exists j. s \leq t_j), \\
\text{with } s = (s_1 \wedge \ldots \wedge s_m) \text{ and } t = (t_1 \vee \ldots \vee t_n)
\end{array} \tag{w}$$

where x and y denote variables and s and t terms. The first four properties are direct consequences of the axioms of lattices. (w) above is *Whitman property* and holds in free lattices (not in all lattices). Applying the above rules recursively decides the $\leq$ relation.

Orthocomplemented Bisemilattices (OCBSL). OCBSL [20] are also a weakening of Boolean algebras (and, in fact, a subtheory of ortholattices). They satisfy laws L1–L8, L1'–L8' but not the absorption law (L9, L9'). This implies in particular that OCBSL do not have a canonical order relation as lattices do, but rather have two, in general distinct, relations:

$$a \leq b \iff a \land b = a$$
$$a \sqsubseteq b \iff a \lor b = b$$

If we add absorption axioms, $a \land b = a$ implies $a \lor b = (a \land b) \lor b = b$ (and dually), so the structure becomes a lattice. The algorithm presented in [20] does not rely on lattice properties. Instead, it is proven that the axioms of OCBSL can be extended to a term rewriting system which is confluent and terminating, and hence admits a normal form. Using variants of algorithms on labelled trees to handle commutativity, this normal form can be computed in quasilinear time $\mathcal{O}(n \log^2(n))$. In contrast, in the case of free *lattices*, there exists no confluent and terminating term rewriting system [16].

3 Deriving an Ortholattice Normal Form Algorithm

Ortholattices [3, Chapter II.1] are structures satisfying laws L1–L9, L1'–L9' of Table 1. An ortholattice (OL) need not be a Boolean algebra, nor an orthomodular lattice; the smallest example of such OL is “Benzene” (O6), with elements $\{0, a, b, \neg b, \neg a, 1\}$ where $a \leq b$ [5]. The word problem for free ortholattices, which checks if a given equation is true, has been shown to be solvable in quadratic time by Bruns [7]. In this section, we go further by presenting an efficient computation of *normal forms*, which reduces the word problem to syntactic equality. In addition, normal forms can be efficiently used for formula simplification and caching, unlike equality procedure itself.

Definition 4. *For a set of variables X, we define a disjoint set of the same cardinality X' with a bijective function $(\cdot)' : X \mapsto X'$. Denote by L the theory of bounded lattices and OL the theory of ortholattices. Define F_L, F_{OL} to be their free lattices and $\mathcal{T}_L$ and $\mathcal{T}_{OL}$ to be the sets of terms over their respective signature. Define $\leq_L$ as the relation on $\mathcal{T}_L$ such that $s \leq_L t \iff \kappa_{F_L}(s) \leq \kappa_{F_L}(t)$ and $\leq_{OL}$ analogously by $s \leq_{OL} t \iff \kappa_{F_{OL}}(s) \leq \kappa_{F_{OL}}(t)$, where κ denotes natural homomorphisms as introduced in the previous section.*

Note: $p \leq_{OL} q \iff (E_{OL} \models (p \land q = q))$ where E_{OL} is the set of axioms of Table 1.

3.1 Deciding $\leq_{OL}$ by Reduction to Bounded Lattices

We consider $\mathcal{T}_L(X \cup X')$ as a subset of $\mathcal{T}_{OL}(X)$ via the injective inclusion on variables mapping $x \mapsto x$ and $x' \mapsto \neg x$. We also define a function $\delta : \mathcal{T}_{OL}(X) \to \mathcal{T}_L(X \cup X')$ as transformation into negation normal form, using laws L6 (double negation elimination), L8 and L8' (de Morgan's laws).

We define a set $R \subseteq \mathcal{T}_L(X \cup X')$ of terms reduced with respect to the contradiction laws (L7 and L7'). These imply that, e.g., given a term $a \vee b$, if $\neg b \leq (a \vee b)$, then from as $b \leq a \vee b$, we have $1 = b \vee \neg b \leq (a \vee b)$. The following inductive definition induces an algorithm to check $x \in R$, meaning that such reductions do not apply inside x:

$$\begin{aligned}
&0, 1, x, x' \in R \quad (\text{for } x \in X)\\
&a \vee b \in R \iff a \in R, b \in R,\ \delta(\neg a) \nleq_L a \vee b,\ \delta(\neg b) \nleq_L a \vee b\\
&a \wedge b \in R \iff a \in R, b \in R,\ \delta(\neg a) \ngeq_L a \wedge b,\ \delta(\neg b) \ngeq_L a \wedge b
\end{aligned}$$

Above, $\leq_L$ is the order relation on lattices, $x \geq_L y$ denotes $y \leq_L x$, and $\nleq_L$, $\ngeq_L$ are the negations of those conditions: $x \nleq_L y$ iff not $x \leq_L y$, whereas $x \ngeq_L y$ iff not $y \leq_L x$.

We also define $\beta : \mathcal{T}_L(X \cup X') \to R$ by:

$$\begin{aligned}
&\beta(0) = 0, \beta(1) = 1, \beta(x) = x, \beta(x') = x' \ (\text{for } x \in X)\\
&\beta(a \vee b) = \begin{cases} \beta(a) \vee \beta(b) & \text{if } \beta(a) \vee \beta(b) \in R \\ 1 & \text{otherwise} \end{cases}\\
&\beta(a \wedge b) = \begin{cases} \beta(a) \wedge \beta(b) & \text{if } \beta(a) \wedge \beta(b) \in R \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Example 2. We have $\beta((x \wedge \neg y) \vee (\neg x \vee y)) = 1$ because $\delta(\neg(x \wedge \neg y)) = \neg x \vee y$ and $\neg x \vee y \leq_L (x \wedge \neg y) \vee \neg x \vee y$.

Note that it is generally not sufficient to check only for $\delta(\neg a) \nleq_L b$ for larger examples. In particular, if $\delta(\neg a)$ is itself a conjunction, by Whitman's property, the condition $\delta(\neg a) \nleq (a \vee b)$ is not in general equivalent to having either $\delta(\neg a) \nleq_L b$ or $\delta(\neg a) \nleq_L a$.

We next reformulate the theorem from Bruns [7]. A key construction from the proof is the following Lemma.

Lemma 1. *$R_{/\sim_L}$ is an ortholattice isomorphic to $F_{OL}(X)$.*

Theorem 1. *Let $s, t \in \mathcal{T}_{OL}(X)$. Then, $s \leq_{OL} t \iff \beta(\delta(s)) \leq_L \beta(\delta(t))$.*

Proof. We sketch and adapt the original proof. Intuitively, computing $\beta(\delta(s)) \leq_L \beta(\delta(t))$ should be sufficient to compute the $\leq_{OL}$relation: δ reduces terms to normal forms modulo rules L6 (double negation elimination) and L8, L8' (De Morgan's Law), and then β takes care of rule L7 (contradiction). The only rules left are rules from (bounded) lattices, which should be dealt with by $\leq_L$. From Lemma 1, the fact that β factors in the evaluation function $\kappa_{F_{OL}}$

(i.e. is equivalence preserving) and properties of free algebras, it can be shown that $\kappa_{F_{OL}} = \gamma \circ N_{\sim_L} \circ \beta \circ \delta$, where $N_{\sim_L}(x) = [x]_{\sim_L}$, and $\gamma : R_{/\sim_L} \to F_{OL}(X)$ is an isomorphism. Hence

$$\kappa_{F_{OL}}(s) \leq \kappa_{F_{OL}}(t) \iff \beta(\delta(s))_{/\sim_L} \leq \beta(\delta(t))_{/\sim_L}$$

which is equivalent to $s \leq_{OL} t \iff \beta(\delta(s)) \leq_L \beta(\delta(t))$.

3.2 Reduction to Normal Form

To obtain a normal form for $\mathcal{T}_{OL}(X)$, we will compose δ and β with a normal form function for $\mathcal{T}_L(X \cup X')$. A disjunction $a = a_1 \vee ... \vee a_m$ (and dually for a conjunction) is in normal form for $\leq_L$ if and only if the following two properties hold [15, p. 17]:

1. if $a_i = (a_{i1} \wedge ... \wedge a_{in})$, then for all j, $a_{ij} \not\leq a$
2. $(a_1, ..., a_n)$ forms an antichain (if $i \neq j$ then $a_i \not\leq a_j$)

We now show how to reduce a term in R so that it satisfies both properties using function ζ that enforces property 1, and then η that additionally enforces property 2. The functions operate dually on $\wedge$ and $\vee$; we specify them only on $\vee$ cases for brevity.

Enforcing Property 1. Define $\zeta : R \to R$ recursively such that:

$$\zeta(a_1 \vee ... \vee a_m) = \begin{cases} \zeta(a_1 \vee ... \vee a_{ij} \vee ... \vee a_m) & \text{if } a_i = (a_{i1} \wedge ... \wedge a_{in}) \\ & \text{and } a_{ij} \leq_L a_1 \vee ... \vee a_m \\ \zeta(a_1) \vee ... \vee \zeta(a_m) & \text{otherwise} \end{cases}$$

(dually for $\wedge$). It follows that $s \sim_L \zeta(s)$ for every term s because $a_{ij} \leq_L a_1 \vee ... \vee a_m$ implies $a_1 \vee ... \vee a_m = a_1 \vee ... \vee a_m \vee a_{ij}$ and $a_i \vee a_{ij} = a_{ij}$ by absorption.

Enforcing Property 2 (Antichain). Define $\eta : R \to R$ such that

$$\eta(a_1 \vee ... \vee a_m) = \begin{cases} \eta(a_1 \vee ... \vee a_{i-1} \vee a_{i+1} \vee ... \vee a_m) & \text{if } a_i \leq_L a_j, i \neq j \\ \eta(a_1) \vee ... \vee \eta(a_m) & \text{otherwise} \end{cases}$$

We have $s \sim_L \eta(s)$ for every term s because $a_i \leq_L a_j$ means $a_i \vee a_j = a_j$.

Example 3. We have: $\eta(\zeta(\ [(a \vee b) \wedge (a \vee c)] \vee b\ )) = \eta((a \vee b) \vee b) = a \vee b$. Indeed, the first equality follows from

$$(a \vee b) \leq_L [(a \vee b) \wedge (a \vee c)] \vee b$$

and the second from $b \leq_L (a \vee b)$.

Denote by R' the subset of R containing the terms satisfying property 1 and R'' the subset of R' of terms satisfying property 2. It is easy to see that ζ is actually $R \to R'$ and η can be restricted to $R' \to R''$. Moreover $s, t \in R''$ and $s \sim_L t$ implies $s = t$. Recall that $\forall w \in \mathcal{T}_{OL}(X).\beta(\delta(w)) \in R$. Since β and δ are equivalence preserving, $\forall w_1, w_2 \in \mathcal{T}_{OL}(X)$

$$w_1 \sim_{OL} w_2 \iff \beta(\delta(w_1)) \sim_{OL} \beta(\delta(w_2))$$

Moreover, since (by Lemma 1) $R_{/\sim_L}$ is an ortholattice, we have

$$\beta(\delta(w_1)) \sim_{OL} \beta(\delta(w_2)) \iff \beta(\delta(w_1)) \sim_L \beta(\delta(w_2))$$

i.e. in R, $\sim_{OL} \equiv \sim_L$. Then,

$$\beta(\delta(w_1)) \sim_L \beta(\delta(w_2)) \iff \eta(\zeta(\beta(\delta(w_1))) \sim_L \eta(\zeta(\beta(\delta(w_2))))$$

and since both $\eta(\zeta(\beta(\delta(w_1))) \in R''$ and $\eta(\zeta(\beta(\delta(w_2))) \in R''$

$$\eta(\zeta(\beta(\delta(w_1))) = \eta(\zeta(\beta(\delta(w_2))))$$

We finally conclude:

Theorem 2. *$NF_{OL} = \eta \circ \zeta \circ \beta \circ \delta$ is a computable normal form function for ortholattices.*

3.3 Complexity and Normal Form Size

Before presenting the algorithm in more detail, we argue why the normal form function from the previous section can be computed efficiently. We assume a RAM model and hence that creating new nodes in the tree representation of terms can be done in constant time.

Note that the size of the output of each of δ, β, ζ and η is linearly bounded by the size of the input. Thus, the asymptotic runtime complexity of the composition is the sum of the runtimes of these functions. Recall that δ (negation normal form) is computable in linear time and ζ and η are both computable in worst-case quadratic time, plus the time needed to compute $\leq_L$. Then, β, R and $\leq_L$ are each computable in constant time plus the time needed for the mutually recursive calls. While a direct recursive implementation would be exponential, observe that the computation time of R and β is proportional to the total number of times they get called on. If we store (memoize) the results of the functions for each different input, this time can be bounded by the total number of different sub-nodes that are part of the input or which we create during the algorithm's execution. Similarly, $\leq_L$ needs to be applied to, at worst, every pair of such sub-nodes. Consequently, if we memoize the result of each of these functions at all their calls, we may expect to obtain at most quadratic time to compute them on all the sub-nodes of a formula.

The above argument is, however, not entirely sufficient, because computing $R(a \wedge b)$ requires creating the new nodes $\neg a$ and $\neg b$ and then computing their negation normal form, which again creates new nodes. Indeed, note that, for memoization, we need to rely on *reference* (pointer) equality, as *structural* equality would take a linear amount of time to compute (for a total cubic time). Hence, to obtain quadratic time and space, we need to be able to negate a node in negation normal form without creating new nodes too many new nodes in memory. To do so, define $op : \mathcal{T}_L(X \cup X') \to \mathcal{T}_L(X \cup X')$ by

$$\begin{array}{ll} op(x) = x' & op(a \wedge b) = op(a) \vee op(b) \\ op(x') = x & op(a \vee b) = op(a) \wedge op(b) \end{array}$$

$\mathrm{op}(a)$ is functionally equal to $\delta(\neg a)$, but has the crucial property that

$$\mathrm{children}(\mathrm{op}(\tau)) === \mathrm{op}[\mathrm{children}(\tau)]$$

Where τ denotes a formal conjunction or disjunction and $\mathrm{children}(\tau)$ is the set of children of τ as a tree. op can be efficiently memoized. Moreover, it can be bijectively memoized: if $op(a) = b$ we shall also store $\mathrm{op}(b) = a$. We thus obtain $\mathrm{op}(\mathrm{children}(\mathrm{op}(\tau))) === \mathrm{children}(\tau)$. In this approach we are guaranteed to never instantiate any node beyond the n subnodes of the original formula (in negation normal form) and their opposite for a total of $2n$ nodes. Hence, we only ever needed to call op, R and β on up to $2n$ different inputs and $\leq$ on up to $4n^2$ different inputs, guaranteeing a final quadratic running time.

Minimal Size. Finally, as none of δ, β, ζ and η ever increase the size of the formula (in terms of the number of literals, conjunctions and disjunctions), neither does NF_{OL}. Consequently, for any term w, $\mathrm{NF}_{OL}(w)$ is one of the smallest terms equivalent to w. Indeed, let $w_{\min} = w$ such that $w_{\min}$ is a term of smallest size in the equivalence class of w. In particular, $\mathrm{NF}_{OL}(w_{\min})$ cannot be smaller than $w_{\min}$ (because $w_{\min}$ is minimal in the class) nor larger (because NF_{OL} is size non-increasing). Since $\mathrm{NF}_{OL}(w) = \mathrm{NF}_{OL}(w_{\min})$, $\mathrm{NF}_{OL}(w)$ is of minimal size.

Theorem 3. *The normal form from Theorem 2 can be computed by an algorithm running in time and space $\mathcal{O}(n^2)$. Moreover, the resulting normal form is guaranteed to be smallest in the equivalence class of the input term.*

4 Algorithm with Memoization and Structure Sharing

To obtain a practical realization of Theorem 3, we need to address two main challenges. First, as explained in the previous section, we need to memoize the result of some functions to avoid exponential blowup. Second, we want to make the procedure compatible with structure sharing, since it is an important feature for many applications.

By *memoization* we mean modifying a function so that it saves the result of the calls for each argument, so that they can be found without future recomputations. Results of function calls can be stored in a map. For single-argument functions we find it is typically more efficient to introduce a field in each object to hold the result of calling a function on it. Under *structure sharing* we understand the possibility to reuse subformulas multiple times in the description of a logical expression. In case of signature $\land, \lor, \neg$, such expressions can be viewed as combinational Boolean circuits. We represent such terms using directed acyclic graph (DAG) reference structures instead of tree structures.

Circuits can be exponentially more succinct than equivalent formulas, but not all formula rewrites are efficient in the presence of structure sharing (consider for example, rules with substitution such as $x \land F \rightsquigarrow x \land F[x := 1]$, where F may also be referred to somewhere else). Structure sharing is thus non-trivial to maintain throughout all representations and transformations. Indeed, making a naive recursive modification of a circuit will unfold the DAG into a tree, often causing an exponential increase in space. Doing so optimally also requires the use of memoization. Moreover, the choice of representations and datastructures is critical.

We show that it is possible to make both algorithms fully compatible with structure sharing without ever creating node duplicates. The algorithm ensures that the resulting circuits will contain a smaller number of subnodes, preserve equivalence, and enforce that two circuits have the same representation if and only if they describe the same term (by the laws of OL).

Algorithm 1: Datastructure for Formulas

```
numberOfFormulas ← 0
Datastructure AIGFormula
    val uniqueId: Int ← numberOfFormulas++    // get fresh ID on node creation
    var inverse:AIGFormula ← null
    var normal:AIGFormula ← null
    var smaller: Set[Int] ← ∅                            // sparse bitset
    var notSmaller: Set[Int] ← ∅                         // sparse bitset
case Variable(id:String, polarity:Bool) of AIGFormula
case Literal(polarity:Bool) of AIGFormula
case Conjunction(children:List[AIGFormula], polarity:Bool) of AIGFormula
val Positive: Bool = True; val Negative: Bool = False
```

Algorithm 2: Computing Negations

```
def inverse($\tau$)                                  // AIGFormula -> AIGFormula
    if isDefined($\tau$.inverse) then
        return $\tau$.inverse
    else
        $\bar{\tau}$ ← $\tau$.copy(polarity = !$\tau$.polarity)
        $\tau$.inverse ← $\bar{\tau}$
        $\bar{\tau}$.inverse ← $\tau$
        return $\bar{\tau}$
```

Algorithm 3: Computing $\leq$

```
def $\leq(\tau, \pi)$                                    // AIGFormula -> AIGFormula -> Bool
    if $\tau$.smaller contains $\pi$.uniqueId then return True
    else if $\tau$.notSmaller contains $\pi$.uniqueId then return False
    else
        r ← match ($\tau$, $\pi$) :
            case (lhs, Conjunction(children, Positive)) :
                $\forall c \in$ children. $\tau \leq c$
            case (Conjunction(children, Negative), rhs) :
                $\forall c \in$ children. $inverse(c) \leq \pi$
            case (Variable(id), Conjunction(children, Negative) :
                $\exists c \in$ children. $\tau \leq inverse(c)$
            case (Conjunction(children, Positive), Variable(id)) :
                $\exists c \in$ children. $c \leq \pi$
            case (Conjunction(tauCh, Positive), Conjunction(piCh, Negative)) :
                // would cause exponential explosion without memoization:
                $(\exists c \in$ tauCh. $c \leq \pi) \vee (\exists c \in$ piCh. $\tau \leq inverse(c))$
            case (Variable(id1), Variable(id2)) :
                id1 == id2
        if r then $\tau$.smaller += $\pi$.uniqueId
        else $\tau$.notSmaller += $\pi$.uniqueId
        return r
```

Pseudocode. Algorithms 1, 2, 3, 4 present pseudocode implementation of the normal form function from Theorem 2. To more easily maintain structure sharing and gain performance, we move away from the *negation normal form* representation and prefer to use a representation of formulas similar to AIG (And-Inverter Graph) where a formula is either a Conjunction, a Variable or a Literal and contains a boolean value telling if the formula is positive or negative (see Algorithm 1). This implies that δ needs to transform arbitrary Boolean formulas into AIGFormulas instead of negation normal forms. Fortunately, AIGFormula can be efficiently translated to NNF (and back) so we can view them as an alternative representation of terms in $\mathcal{T}_L(X \cup X')$. For the sake of space, we do not show the reduction from general formula trees on the signature $(\wedge, \vee, \neg)$ and work directly with AIGFormulas, but the implementation needs memoization to avoid exponential duplication in presence of structure sharing.

Recall that computing R requires taking the negation of some formulas, and projecting them back into $\mathcal{T}_L(X \cup X')$ with δ. Using *AIGFormula* makes it possible to always take the negation of a formula in constant time and space. The corresponding function *inverse(τ)* is in Algorithm 2, and corresponds to the *op* function from the previous section. The memoization ensures that for all τ, $inverse(inverse(\tau)) === \tau$, and our choice of data structure ensures that $children(inverse(\tau)) === children(\tau)$. Those two properties guarantee that any sequence of access to children and inverses of τ will always yield a formula object within the original DAG, or its single inverse copy. In particular, regardless of structure sharing in the input structure, we never need to store in memory more than twice the total number of formula nodes in the input. As explained in Sect. 3.3, a similar condition could be made to hold with NNF, but we believe it is more complicated and less efficient when implemented.

Function $\leq$ in Algorithm 3 is based on Whitman's algorithm adapted to AIGFormula. For memoization, because the function takes two arguments, we store in each node the set of nodes it is smaller than or not using two sets. Note that storing and accessing values in a set (even a hash set) is only as efficient as computing the equality relation on two objects is. Because structural equality $==$ takes linear time to compute, we use referential equality with the *uniqueId* of each formula (declared in Algorithm 1). We found that using sparse bit sets yields the best performances.

The *simplify* function in Algorithm 4 makes a one-level simplification of a conjunction node, assuming that its children have already been simplified. We present the case when τ is positive. It works in three steps. The subfunction *zeta* corresponds to the ζ function from the previous section. It both flattens consecutive positive conjunctions and applies a transformation based on a strengthened version of the absorption law. Then at line 13, we filter out the nodes which are smaller than some other node, for example if $c \leq b$ then $a \wedge b \wedge c$ becomes $a \wedge c$. This corresponds to function η. Finally, line 16 applies the contradiction law, i.e. if $a \wedge b \wedge c \leq \neg a$ then $a \wedge b \wedge c$ becomes 0. Note again that checking only if either $b \leq \neg a$ or $c \leq \neg a$ holds is not sufficient (see for example the case $a = (\neg b \vee \neg c)$. This corresponds to the β function. The correspondence with the three functions ζ, η and β is not exact; all computations are done in a single traversal over the structure of the formula, rather than in separate passes as the composition $\circ$ of functions in Theorem 2 might suggest.

Importance of Structure Sharing. As detailed in Sect. 6, our implementation finished in a few tenths of a second on circuits containing approximately 10^5 And gates, but whose expanded formula would have size over 10^{2000}, demonstrating the compatibility of the algorithm with structure sharing. For this, we must ensure at every phase and for every intermediate representation, from parsing of the input to exporting the solution, that no duplicate node is ever created. This is achieved, again, using memoization. The complete and testable implementation of both the OL and OCBSL algorithms in Scala is available at https://github.com/epfl-lara/lattices-algorithms.

5 Application to More Expressive Logics

This section outlines how we use OCBSL and OL algorithms in program verification. Boolean Algebra is not only relevant for pure propositional logic; it is also the coreof more complex logics, such as the ones used for verification of software.

Algorithm 4: Computing normal form

```
def simplify($\tau$)                                          // Conjunction -> AIGFormula
    // Assume $\tau$ is positive
    // (In negative cases, some nodes must be inverted and $\leq$ reversed.)
    newChildren ← List()
    def zeta(child)
        match child :
            case PositiveConjunction :
                newChildren.add(child.Children)
            case child:NegativeConjunction :
                gc ← child.children.find(gc ↦ $\tau \leq$ gc)
                if isDefined(gc) then zeta(gc)
                else newChildren.add(child)
    for child ← $\tau$.children do
        zeta(child)
    children' ← // filter out redundant children smaller than another child
    if children'.size == 0 then return Literal(True)
    else if children'.size == 1 then return children'.head
    else if $\exists$ c $\in$ children'. $\tau \leq$ inverse(c) then return Literal(False)
    else return Conjunction(newChildren)

def $NF_{OL}$($\tau$)                                        // AIGFormula -> AIGFormula
    if isDefined($\tau$.normal) then return $\tau$.normal
    else
        $\tau$.normal ← match $\tau$ :
            case Variable(id, True): $\tau$
            case Variable(id, False): inverse($NF_{OL}$(inverse($\tau$)))
            case Conjunction(children, polarity): simplify(children map $NF_{OL}$
              polarity)
        return $\tau$.normal
```

Propositional terms appear as subexpressions of the program (as members of the Boolean type), but also in verification conditions corresponding to correctness properties. This section highlights key aspects of such a deployment.

We consider programs containing let bindings, pattern matching, algebraic data types, and theories including numbers and arrays. Let bindings typically arise when a variable is set in a program, but is also introduced in program transformations to prevent exponential increase in the size of program trees. Since OCBSL and OL are compatible with a DAG representation—fulfilling a similar role to let bindings—they can similarly "see through" bindings without breaking them or duplicating subexpressions.

If-then-else and pattern matching conditions can be analyzed and used by the algorithms, possibly leading to dead-branch removal or condition simplification. Extending OCBSL and OL to reason about ADT sorts further increases the simplification potential for pattern matching. For instance, given assumptions ϕ, a scrutinee s and an ADT constructor identifier id of sort S, we are interested in determining whether s is an instance of the constructor id. A trivial case includes checking the form of s. Otherwise, we can run OCBSL or OL to check whether $\phi \implies (s \ \mathtt{is} \ id)$ holds. If $\phi \implies (s \ \mathtt{is} \ id)$ fails, we instead test whether $\phi \implies \neg(s \ \mathtt{is} \ id')$ for all $id' \neq id \in S$. We may also negatively answer to the query if $\phi \implies (s \ \mathtt{is} \ id')$ for some $id' \neq id \in S$.

The original OCBSL algorithm presented in [20] achieves quasi-linear time complexity by assigning codes to subnodes such that equivalent nodes (by the laws of OCBSL) have the same codes. This is not required for the OL algorithm as it is quadratic anyway, but can still be done to allow common subexpression elimination. This is similar to hash-consing, but more powerful, as it also eliminates expressions which are equivalent with respect to OCBSL or OL.

Of particular relevance is the inclusion of underlying theories such as numbers or arrays. OL has an advantage over OCBSL in terms of extensibility. Namely, OL makes it possible to implement more properties of theories through expansion of its $\leq_{OL}$ relation (Algorithm 3) with inequalities between syntactically distinct *atomic* formulas. For example, if $<_I$ and $\leq_I$ are relations on mathematical integers in the theory of the SMT solver, our implementation deduces that $(x <_I y) \leq_{OL} (x \leq_I y)$ using the rule $z + a <_I 0 \implies z + b \leq_I 0$ when $b \leq_I a + 1$, instantiated with $z = x - y$ and $a = b = 0$. In one of our benchmarks, this simple rule led OL to simplify a verification condition (VC) of the form $\neg(x <_I y \land \phi_1 \land x >_I y \land \phi_2)$ to true, which was of interest because ϕ_1, ϕ_2 were large. This simplification is performed at line 16 of Algorithm 4 with $\tau = x <_I y \land x >_I y \land \phi$, where we have $c = x >_I y$ because $\tau \leq_{OL} (x \leq_I y) \impliedby (x <_I y) \leq_{OL} (x \leq_I y)$. In contrast, OCBSL was not able to do the simplification because it is not able to systematically check for inequalities of subterms. For arrays, our implementation also checks for the property $i \neq j \leq_{OL} a[i := v](j) = a(j)$. Combined with two other rules, related to congruence, OL performs particularly well for array-intensive benchmarks such as `SortedArray`. Note that in OCBSL we may encode a weak form of implication by specifying (giving the same code to) $\phi \land \psi = \phi$ or $\phi \lor \psi = \psi$, but unlike the OL encoding, this does not even allow simplifying formulas such as $\phi \land \tau \land \neg\psi$ without a specific check, which would require quadratic time in general.

Other Extensions. Beyond program verification, we suspect OL or OCBSL based techniques to be extendable in applications such as type checkers, interactive and automated theorem provers using first order, higher order, temporal and modal logics, SMT solvers or lattice problems in abstract interpretation. Unidirectional rules which may be particularly relevant for automated theorem proving include $[f(x) = f(y)] \leq_{OL} [x = y]$, $[\forall x, P(x)] \leq_{OL} P(t)$, and $P \leq_{OL} Q$ when $P \rightarrow Q$ is a known theorem. In the context of quantified logics and lambda calculus, both algorithms are compatible with de Bruijn index representation of bound variables. Both algorithms can be used as partial simplification before or while applying more powerful but possibly incomplete heuristic simplification methods, such has the simplification rule $x \land F[x] \rightsquigarrow x \land F[x := 1]$ (which, if viewed as an equality axiom, turns OL into Boolean algebra).

6 Evaluation

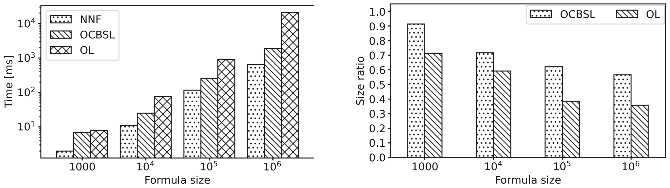
Our experimental evaluation comprises three parts. First, we analyze the behavior of the OL and OCBSL algorithms on large random formulas, to understand the feasibility of using them for normalization. Second, we evaluate the algorithms on combinatorial circuits [1]. Third and most importantly, we show their impact through a new simplifier for verification conditions of the Stainless [22] verifier. The goal of the simplifier is to avoid the need to invoke a solver for some of the formulas by reducing them to True, as well as to normalize them before storing them in a persistent cache file. The cache avoids the need to repeatedly prove previously proven verification conditions. By improving normalization, we improve the cache hit rate. We conduct all experiments on a server with 2× Intel®Xeon®CPU E5-2680 v2 at 2.80 GHz, 40 cores including hyperthreading and 64 GB of memory.

6.1 Randomly Generated Propositional Formulas

We first evaluate the two algorithms on randomly generated formulas. We measure the running time and the reduction in formula size. We build the random formulas as follows.

Definition 5. *A random formula is parameterized by a size s and a set of available variables $X = \{x_1, ..., x_n\}$. Given a size s, if $s \leq 1$ then pick uniformly at random a variable from X or its negation and return it. Otherwise, pick t such that $0 < t < s - 1$ and generate two formulas ϕ_1 and ϕ_2 of sizes t and $s - 1 - t$. Return uniformly at random And(ϕ_1, ϕ_2) or Or(ϕ_1, ϕ_2).*

Running Time. We show in Fig. 1a the approximate running time of both algorithms for various sizes of formulas. We ran the experiment 21 times for each formula size category and took the median. For comparison with a theoretically linear time process, we also give the running time of the corresponding negation normal form transformation. These implementations do not come with low-level optimizations and are intended for demonstrating usability in practice, and do not serve as a competitive indicator.



(a) Running time

(b) Size ratio, showing reduction

Fig. 1. (a) Median running time of NNF and the two algorithms (log-log scale). (b) Median size of the normalized formulas relative to the original in NNF. $|X| = 50$ variables.

Size Reduction. For a fairer comparison, we apply a basic simplification (flattening and transformation into negation normal form) to random formulas before computing their size. We compare the number of connectors before and after the simplification for both algorithms. We show the relative improvements of the OL and OCBSL algorithms compared to the original formulas for various sizes of formulas and 50 variables. We have run both algorithms 21 times and report the median results in Figs. 1b.

It is interesting to note that the OL normal form is consistently and significantly smaller than the OCBSL normal form, i.e. the Absorption law actually allows non-trivial reductions in size. This confirms that, in general, there is a trade-off between the two algorithms between speed and simplification strength.

6.2 Computing Normal Forms for Hardware Circuits

Moving towards more realistic formulas, we assess the scalability of OCBSL and OL on the EPFL Combinatorial Benchmark [1] comprising 10 arithmetic circuits designed to challenge optimization tools, with up to 10^8 gates.

Table 2. Results on the EPFL Combinatorial Benchmark. OL times-out for `hyp` after 1h.

	adder	bar	div	hyp	log2	max	mult	sin	sqrt	square
# of gates	50173	72704	10^7	10^8	10^7	10^7	10^7	10^6	10^7	10^7
OCBSL Ratio	1.00	0.703	0.777	0.961	0.700	0.861	0.867	0.652	0.661	0.927
OL Ratio	1.00	0.703	0.777	–	0.697	0.861	0.865	0.647	0.661	0.927
OCBSL Time [s]	0.142	0.182	0.866	2.06	0.564	0.189	0.442	0.255	0.362	0.365
OL Time [s]	0.276	0.338	706	–	339	0.319	73.8	15.7	256	36.0

We run the experiment five times. We report the median running time and the relative size after optimization in Table 2. We observe that the OCBSL algorithm is close to as good as the OL algorithm in all cases, and, moreover, that it is very time-efficient even for problems with hundreds of millions of gates. The OL algorithm sometimes performs slightly better and is pretty much as time-efficient for not too large inputs, but becomes significantly more time-consuming for inputs with more than approximately 10^6 gates. Those results suggest on one hand that OCBSL may be a more suitable reduction technique on some applications with very large formulas, depending on their internal structures. It also suggests that both algorithms work well in practice with Boolean circuits making heavy use of structure sharing. Indeed, the expanded form of, for example, the adder circuit would have about 2^{2000} nodes.

6.3 Caching Verification Conditions in Stainless

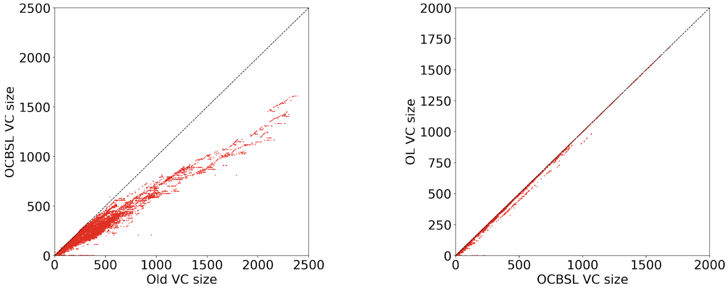
We implement the approach described in Sect. 5 by modifying the Stainless verifier [22,40][1], a publicly available tool for building formally verified Scala programs.

---

[1] https://github.com/epfl-lara/stainless/.

Our implementation adds two new simplifiers to Stainless: OCBSL-backed and OL-backed. They are part of Stainless release v0.9.8[2] and are selectable by the command line options `--simplifier=ocbsl` and `--simplifier=ol` respectively. For the OL simplifier, we have extended the $\leq_{OL}$ relation with 12 simple arithmetic and array rules.

We experimentally compare the two new simplifiers to the existing one (which we denote Old). We use two groups of benchmarks: (1) six Stainless case studies from the Bolts repository[3] that take a significant amount of time to verify, and (2) nine benchmark sets from automated grading of student assignments. Together, this constitutes around 84'000 lines of Scala code, specifications, and auxiliary assertions. We report the following metrics: the size of the VCs after simplification, the number of cache hits, the number of VCs simplified to 1, the wall-clock time and the cumulative solving time. The wall-clock time comprises the full Stainless pipeline, from parsing the program to outputting the result, passing by solver calls and VC simplification.



(a) Old vs OCBSL

(b) OCBSL vs OL

Fig. 2. VCs (tree) size scatter plot from all benchmarks for Old, OCBSL and OL.

Evaluation on Bolts Case Studies. We consider the following case studies from the mentioned Bolts repository:

- `LongMap` (9613 VCs, 7091 LOC), a mutable hash map, 64-bit integer keys, open addressing, formalized by Samuel Chassot (EPFL) and proven to behave equivalently to a list of (key, value) pairs.
- A type checker for `System F` [19] (5040 VCs, 2501 LOC) formalized in Stainless by Andrea Gilot and Noé De Santo (EPFL). Among the key properties proven are type judgment uniqueness, preservation and progress.
- `QOI` (4487 VCs, 2812 LOC), an implementation of the Quite OK Image format. Decoding an encoded image is shown to yield the original image [10].
- `RedBlack`, a red-black tree (764 VCs, 796 LOC).
- `SortedArray` (472 VCs, 429 LOC), a mutable array preserving order on insertion. Developed for use in a simplified model of part of a file system [21].

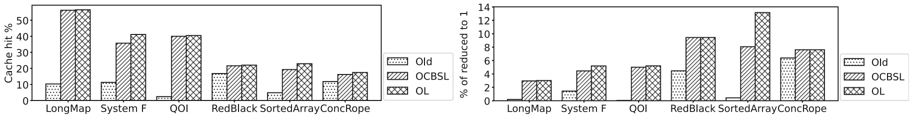
---

[2] https://github.com/epfl-lara/stainless/releases/tag/v0.9.8.

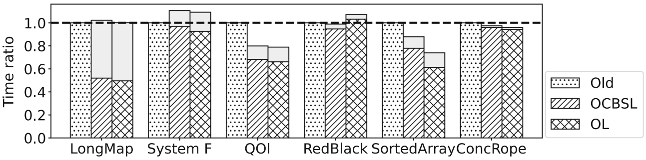
[3] https://github.com/epfl-lara/bolts.

- `ConcRope` (408 VCs, 621 LOC), a Conc-Tree rope [36], supporting amortized constant time append and prepend operation, based on a Leon formalization [30].

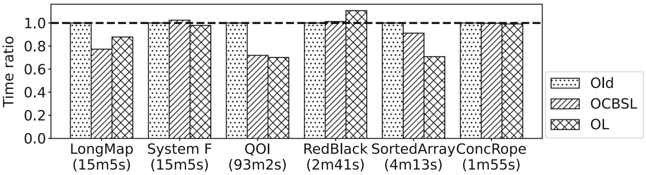
We report the VCs size measurement in Fig. 2, where we aggregate the results from all benchmarks. Figure 2a reveals a couple of VCs with an increased size. Inspection of these VCs shows the reason is due to the new simplifiers always inlining “simple expressions”, such as field projection on free variables, instead of having them bound. On average, OCBSL and OL decrease the size of the VCs by 37% compared to Old. OL reduces the size of the VCs slightly compared to OCBSL (Fig. 2b).



(a) Cache hits in a single run (b) VCs simplified to 1



(c) Cumulative solving time



(d) Wall-clock time

Fig. 3. Old, OCBSL and OL results for cache hits, VCs reduced to 1, solving and running time. (c), (d) are normalized with respect to Old. In (c), the gray boxes represent the time spared due to extra cache hits and VCs reduced to 1 compared to Old.

In Fig. 3a, we report the cache hit ratio. For the new simplifiers, reducing the formula size has the desired effect of noticeably increasing the hit ratio, especially for 4 out of 6 benchmarks. The additional power of OL helps for `System F` and `SortedArray`.

We report in Fig. 3c not only the solving time for the two simplifiers (normalized with respect to Old), but also the solving time saved thanks to additional cache hits and VCs simplified to 1. `ConcRope` and `RedBlack` do not benefit from the new simplifiers, while the other benchmarks do in various degrees. For `LongMap`, adding the two ratios yields a ratio of ≈ 1, implying the reduced solving time is due to extra caching. The solver did not benefit from the new simplifiers for non-cached VCs. The `System F` benchmark shows a ratio exceeding 1, meaning that OCBSL and OL did not help the solver more than the extra time they took to run. For `QOI` and `SortedArray`, the combined ratio is less than 1: the new simplifiers helped the solver for non-cached VCs. OL performs significantly better than OCBSL in the `SortedArray` benchmark, thanks to the extension of the $\leq_{OL}$ relation with array rules. We note that 25% of `QOI` VCs have a size of more than 880, against 480 for the second benchmark (`SortedArray`), and 450 for the third (`LongMap`).

Turning our attention to Fig. 3d, we note that the time spared to solver calls is essentially compensated for more work on the new simplifiers on three of the benchmarks. Moreover, `LongMap`, `SortedArray` and especially `QOI` have a net benefit over Old.

OCBSL and OL simplifiers show the greatest improvement on large VCs. Note that the outcome of a Stainless run highly depends on user-provided assertions, which were hand-tuned under the Old simplifier. It is thus possible that new simplifiers have a disadvantage because they were not used during the verification process. The additional power provided by the new simplifiers may make writing such intermediate assertions easier and faster, so we expect the full advantage of new simplifiers in newly developed verified software.

Table 3. Results on programming assignments

Benchmark		filter	max	mirror	mem	sigma	nat	uniq	formula	lambda
# Submissions		210	216	96	136	734	381	147	677	782
Cumulative LOC		2367	3452	1165	1987	8347	8950	3648	19226	17958
# VCs		820	844	387	560	1528	2653	1352	9865	5922
Solver Calls	Old	28	81	44	77	75	133	264	1037	1115
	OCBSL	19	**79**	43	75	58	133	**251**	1033	1069
	OL	**18**	**79**	**42**	**74**	**50**	**131**	**251**	**1032**	**1066**
# VCs reduced to **1**	Old	211	302	95	151	4	886	381	1322	1320
	OCBSL	211	302	95	151	6	**890**	381	1327	**1322**
	OL	**213**	302	95	151	**794**	**890**	381	**1332**	**1322**
Cache Hits	Old	581	461	248	332	1449	1634	707	7506	3487
	OCBSL	590	463	249	334	1464	1630	720	7505	3531
	OL	589	463	250	335	684	1632	720	7501	3534
VCs (tree) Size	Old	6705	5576	3077	5097	47759	15378	12144	126968	78962
	OCBSL	6479	**5546**	3073	5063	49775	14514	11465	125289	75837
	OL	**6457**	**5546**	**2982**	**5000**	**34173**	**14482**	**11444**	**125037**	**75307**
Solving Time [s]	Old	2.48	5.61	3.72	5.79	4.17	7.97	14.27	118.61	108.42
	OCBSL	1.91	5.22	3.52	5.75	**3.43**	5.73	14.27	**102.48**	**104.27**
	OL	**1.70**	**4.92**	**3.06**	**5.34**	3.66	**7.03**	**13.57**	134.73	104.60
Total Time [m:s]	Old	**0:27**	**0:36**	**0:16**	**0:21**	**0:59**	**14:02**	**1:36**	51:01	**115:39**
	OCBSL	0:29	0:38	0:17	0:22	1:04	14:33	1:37	**50:08**	120:48
	OL	0:29	0:38	**0:16**	0:22	1:10	14:43	1:46	58:05	116:09

Evaluation on Programming Assignments. We additionally evaluate our approach on benchmarks consisting of many student solutions for several programming assignments. We consider benchmarks from [32,33], obtained by translation of student solutions in OCaml [38]. In this evaluation, we only prove termination of all student solutions, which is one of the bottlenecks when proving correctness of students solutions. We annotated all benchmarks with explicit decreasing measures. Stainless generates verification conditions that require the measure to decrease in recursive calls. Caching is particularly desirable in this scenario, with many programs and a high degree of similarity. Table 3 shows our evaluation results, comparing the two new simplifiers (OCBSL and OL) to the old one.

First, we note that moving from Old to OCBSL to OL reduces the number of calls to the solver. Furthermore, many new VCs are proven valid by normalization alone (reduced to 1). The largest benefit of OL is in the `sigma` benchmark, where the subsumption of linear arithmetic literals in the simplifier substantially increases the number of formulas proven by normalization: from 6 (0.4%) in OCBSL to 794 (52%) for OL.

The new simplifiers improve the number of cache hits, even if not as much as for the Bolts case studies. The smaller reduction is because there is a high degree of similarity across the submissions, so the Old simplifier already achieves a large percentage of cache hits. Note also that a smaller number of cache hits in the `sigma` benchmark is because many of the VCs are proven valid by the simplifier, avoiding the need to consult the cache or the solver in first place.

Second, we notice a slight reduction in the overall VC size, with a couple of exceptions where OCBSL resulted in a size increase due to inlining. Thanks to formulas proven by normalization and improved cache hits, the overall solving time decreases in several benchmarks. The wall clock running time is approximately unchanged, but we expect such benefits in the future.

7 Conclusion

We proposed a new approach to simplify and reason about formulas, based on algorithms which are sound and complete for the normal form problem (and the word problem) of two subtheories of Boolean algebra. These algorithms are sound but incomplete for Boolean algebras (and thus for the two-element boolean algebra of propositional logic). We introduced and proved the correctness of a new algorithm to compute normal forms in a theory of *ortholattices*, which do not enforce the distributivity law but only its weaker variation, absorption. Our algorithm runs in time $\mathcal{O}(n^2)$. A weaker subtheory, OCBSL, gives up the absorption law. The disadvantage of OCBSL is a weaker normal form, whereas the advantage is that we know of an algorithm running in subquadratic time, $\mathcal{O}(n\log(n)^2)$. We evaluated both algorithms, using them to reduce the size of large random formulas and combinatorial circuits, showing that they work well with structure sharing. We also implemented the algorithms in the Stainless verifier, where computing normal forms reduced the size of formulas given to the solver and improved the cache hit ratio. Our experimental evaluation confirmed that the tradeoff between normal form strength and the asymptotic complexity remains visible in practice. We found both algorithms useful in practice. OCBSL normalization has excellent running time even for very large circuits, so we believe it can replace the simpler negation normal form and syntactic equality checking at low cost in essentially all applications. The quadratic cost of the OL algorithm is too prohibitive on circuits over 10^7 gates. However, this was not a problem for its application to verification conditions in Stainless, where its added precision and the ability to compare atomic formulas made it more effective in normalizing certain formulas to True and increasing cache hits. In some of the most difficult case studies, such as Quite OK Image Format [10], these improvements translated into substantial reduction of the wall clock time. Such measurable improvements, combined with theoretical guarantees, make the OL and OCBSL algorithms an appealing building block for verification systems.

References

1. Amarù, L., Gaillardon, P.E., De Micheli, G.: The EPFL combinational benchmark suite. In: Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS) (2015). https://github.com/lsils/benchmarks
2. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
3. Beran, L.: Orthomodular Lattices (An Algebraic Approach). Springer, Dordrecht (1985). https://doi.org/10.1007/978-94-009-5215-7
4. Birkhoff, G.: Lattice Theory, AMS Colloquium Publications, 3rd edn., vol. 25. AMS (1973)
5. Bonzio, S., Chajda, I.: A note on orthomodular lattices. Int. J. Theor. Phys. **56**, 3740–3743 (2017). https://doi.org/10.1007/s10773-016-3258-6
6. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
7. Bruns, G.: Free ortholattices. Can. J. Math. **28**(5), 977–985 (1976). https://doi.org/10.4153/CJM-1976-095-6
8. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_12
9. Bryant, R.E.: Binary decision diagrams. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 191–217. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_7
10. Bucev, M., Kunčak, V.: Formally verified quite OK image format. In: Formal Methods in Computer-Aided Design (FMCAD) (2022)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979, pp. 269–282. ACM Press (1979). https://doi.org/10.1145/567752.567778

12. Dershowitz, N., Hsiang, J., Huang, G.-S., Kaiss, D.: Boolean rings for intersection-based satisfiability. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 482–496. Springer, Heidelberg (2006). https://doi.org/10.1007/11916277_33
13. Duarte, A., Korovin, K.: Implementing superposition in iProver (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 388–397. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_24
14. Even-Mendoza, K., Asadi, S., Hyvärinen, A.E.J., Chockler, H., Sharygina, N.: Lattice-based refinement in bounded model checking. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 50–68. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_4
15. Freese, R., Jezek, J., Nation, J.: Free Lattices, Mathematical Surveys and Monographs, vol. 42. American Mathematical Society, Providence (1995). https://doi.org/10.1090/surv/042
16. Freese, R., Jezek, J., Nation, J.B.: Term rewrite systems for lattice theory. J. Symb. Comput. **16**(3), 279–288 (1993). https://doi.org/10.1006/jsco.1993.1046
17. Freese, R., Nation, J.B.: Finitely presented lattices. Proc. Am. Math. Soc. **77**(2), 174–178 (1979). https://doi.org/10.2307/2042634
18. Genet, T., Le Gall, T., Legay, A., Murat, V.: A completion algorithm for lattice tree automata. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 134–145. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39274-0_13
19. Girard, J.Y.: Une extension de L'interpretation de Gödel a L'analyse, et son application a L'elimination des coupures dans L'analyse et la theorie des types. In: Fenstad, J. (ed.) Proceedings of the Second Scandinavian Logic Symposium, Studies in Logic and the Foundations of Mathematics, vol. 63, pp. 63–92. Elsevier (1971). https://doi.org/10.1016/S0049-237X(08)70843-7
20. Guilloud, S., Kunčak, V.: Equivalence checking for orthocomplemented bisemilattices in log-linear time. In: TACAS 2022. LNCS, vol. 13244, pp. 196–214. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_11
21. Hamza, J., Felix, S., Kunčak, V., Nussbaumer, I., Schramka, F.: From verified Scala to STIX file system embedded code using Stainless. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NFM 2022. LNCS, vol. 13260, pp. 393–410. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06773-0_21. http://infoscience.epfl.ch/record/292424
22. Hamza, J., Voirol, N., Kunčak, V.: System FR: formalized foundations for the Stainless verifier. Proc. ACM Program. Lang. **3**, 1–30 (2019). https://doi.org/10.1145/3360592
23. Harrison, J.: HOL light: an overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_4
24. Jain, H., Bartzis, C., Clarke, E.: Satisfiability checking of non-clausal formulas using general matings. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 75–89. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_10
25. Kojevnikov, A., Kulikov, A.S., Yaroslavtsev, G.: Finding efficient circuits using SAT-solvers. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 32–44. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_5
26. Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1

27. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-540-74105-3
28. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014. EPTCS, Grenoble, France, 6 April 2014, vol. 149, pp. 3–15 (2014). https://doi.org/10.4204/EPTCS.149.2
29. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 380–397. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_22
30. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL) (2017). https://doi.org/10.1145/3009837.3009874
31. Merz, S., Vanzetto, H.: Automatic verification of TLA$^+$ proof obligations with SMT solvers. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 289–303. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_23
32. Milovancevic, D., Kuncak, V.: Proving and disproving equivalence of functional programming assignments (artifact) (2023). https://doi.org/10.5281/zenodo.7810840
33. Milovancevic, D., Kunčak, V.: Proving and disproving equivalence of functional programming assignments. In: ACM SIGPLAN Conference Programming Language Design and Implementation (PLDI) (2023). https://doi.org/10.1145/3591258
34. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
35. Naumowicz, A., Korniłowicz, A.: A brief overview of Mizar. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 67–72. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_5
36. Prokopec, A., Odersky, M.: Conc-trees for functional and parallel programming. In: Shen, X., Mueller, F., Tuck, J. (eds.) LCPC 2015. LNCS, vol. 9519, pp. 254–268. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29778-1_16
37. Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 735–743. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_49
38. Song, D., Lee, W., Oh, H.: Context-aware and data-driven feedback generation for programming assignments. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, pp. 328–340. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3468264.3468598
39. Suter, P.: Non-clausal satisfiability modulo theories. Technical report, M.Sc. thesis, EPFL (2008). http://infoscience.epfl.ch/record/126445
40. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Scala Symposium (2015). https://doi.org/10.1145/2774975.2774978
41. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: Platzer, A., Sutcliffe, G. (eds.) CADE 2021. LNCS (LNAI), vol. 12699, pp. 415–432. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_24

42. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 140–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_10
43. Wenzel, M., Paulson, L.C., Nipkow, T.: The isabelle framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_7
44. Whitman, P.M.: Free lattices. Ann. Math. **42**(1), 325–330 (1941). https://doi.org/10.2307/1969001
45. Zhang, H.T., Jiang, J.H.R., Mishchenko, A.: A circuit-based SAT solver for logic synthesis. In: 2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD), pp. 1–6 (2021). https://doi.org/10.1109/ICCAD51958.2021.9643505