Logical Methods in Computer Science
Vol. 11(1:6)2015, pp. 1–18
www.lmcs-online.org

Submitted Nov. 11, 2013
Published Mar. 10, 2015

EXTRACTING VERIFIED DECISION PROCEDURES: DPLL AND RESOLUTION

ULRICH BERGER [a], ANDREW LAWRENCE [b], FREDRIK NORDVALL FORSBERG [c], AND MONIKA SEISENBERGER [d]

[a,b,d] Swansea University, UK
e-mail address: {u.berger,csal,m.seisenberger}@swansea.ac.uk

[c] University of Strathclyde, UK
e-mail address: fredrik.nordvall-forsberg@strath.ac.uk

Abstract. This article is concerned with the application of the program extraction technique to a new class of problems: the synthesis of decision procedures for the classical satisfiability problem that are correct by construction. To this end, we formalize a completeness proof for the DPLL proof system and extract a SAT solver from it. When applied to a propositional formula in conjunctive normal form the program produces either a satisfying assignment or a DPLL derivation showing its unsatisfiability. We use non-computational quantifiers to remove redundant computational content from the extracted program and translate it into Haskell to improve performance. We also prove the equivalence between the resolution proof system and the DPLL proof system with a bound on the size of the resulting resolution proof. This demonstrates that it is possible to capture quantitative information about the extracted program on the proof level. The formalization is carried out in the interactive proof assistant Minlog.

1. Introduction

In order for verification tools to be used in an industrial context they have to be trusted to a high degree and in many cases are required to be certified. We present a new application of program extraction to develop a formally verified decision procedure for the satisfiability problem for propositional formulae in conjunctive normal form. The procedure is based on the DPLL proof system [17, 16] which is also the basis of most contemporary SAT solvers that are used in an industrial context.

The need for verified SAT solvers is obvious; they are part of safety critical software, and also used for the verification and certification thereof. SAT solvers are nowadays highly optimized for speed, which makes the introduction of errors (in the process of optimization) more likely, and their verification more difficult. Besides the correctness also totality (or universality) of SAT solvers is an issue. For example, in the 2012 SAT competition

2012 ACM CCS: [**Theory of computation**]: Logic—Proof theory / Logic and verification.
Key words and phrases: DPLL, Program Extraction, Interactive Theorem Proving, SAT.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.2168/LMCS-11(1:6)2015

(`www.smtcomp.org`) many systems were not total in the sense that they returned "Unknown" for certain inputs signifying that they could not deal with the given problem.

In this paper we report about the extraction of a SAT solver that is both correct and total by construction. In addition, it produces in the unsatisfiable case a formal proof of this fact, which is recognized in the SAT community as a highly desirable feature of SAT solvers. To be more precise, we formalize a correctness and completeness proof of the DPLL proof system in the interactive theorem prover Minlog, and use Minlog's program extraction facilities to obtain a formally verified SAT solving algorithm. When run on a CNF formula it produces a model satisfying the formula or a DPLL derivation showing its unsatisfiability. We also prove the equivalence of DPLL and resolution and extract a program translating DPLL proofs into resolution proofs of smaller or equal size.

Minlog [31, 2, 4] is an interactive proof assistant based on a first-order natural deduction calculus. It implements various methods of program extraction such as realizability [23] (which can be viewed as a technical rendering of the Curry-Howard correspondence [15, 20]) and the Dialectica interpretation. It also extends program extraction to classical proofs via the Friedman/Dragalin A-translation. All these techniques are refined and optimized in order to improve usability and to obtain simpler programs. In addition to extracting a program from a proof, Minlog also automatically extracts a proof that the program meets its specification; see for instance [42] for an overview on program extraction and its underlying theory. A number of substantial case studies on program extraction have been carried out reaching from the extraction of a normalization-by-evaluation algorithm [3] to the extraction of programs in constructive analysis [41]. Recent developments concentrate on program extraction for induction and coinduction, including applications in the context of exact real number computation [5].

An optimization in Minlog that is particularly important for this paper is the use of so-called non-computational quantifiers, which flag certain information in the proof as computationally irrelevant, and therefore allow for the removal of computational redundancy in the extracted program. In case of the extracted SAT solver, this leads to a significant improvement.

We also applied an automatic translation of Minlog terms into Haskell code to the extracted program and observed a further dramatic improvement of performance. We evaluate the performance of our extracted solver by comparing it 1) with another verified SAT solver, Versat [36], using Pigeon hole formulae and 2) with an industrial tool, SCADE [1], by means of an example from the railway domain.

An earlier version of this article, containing partial results, was reported at the MFPS 2012 [25] conference.

1.1. **Related Work.** There are several other systems supporting program extraction from proofs for the purpose of producing formally verified programs. An early example is the Nuprl system [13]; other mature interactive theorem provers that implement program extraction are Coq [8], which is based on the Calculus of Inductive Constructions, and Isabelle [35], a generic theorem prover with extensions for many logics (see [7] for code generation and [6] for program extraction from proofs in Isabelle). More recently, other interactive theorem provers based on dependent types [30], such as Agda [11] and Idris [12], have emerged which realize the Curry-Howard correspondence and therefore can also be viewed as supporting program extraction.

The Coq system has been used in several approaches to formalize automatic theorem proving. Lescuyer and Conchon [26] program a SAT solver based on the DPLL algorithm as a recursive function in Coq, and verify its soundness and completeness formally in the system. The solver is then instantiated on the propositional fragment of Coq's logic, creating a user friendly proof tactic. Similarly, Verma et al. [46] formalize Binary Decision Diagrams in Coq, prove their correctness, and extract certified BDD algorithms in OCaml. The main reason for their formalization was to integrate symbolic model checking in Coq. Significant work has also been performed in Isabelle with several decision procedures verified and integrated into the system. The DPLL algorithm has been formalized by Marić and Janičić [28]. This approach was extended to formalize a SAT solver including optimizations such as clause learning and the lazy two-watched-literal data structure [27]. The authors investigated automatic code generation, but in the end the verified algorithm was manually translated into C code. The automatic theorem prover Metis [37] is used inside Isabelle to reconstruct proofs from faster external procedures such as the ones used in Sledgehammer [10]. A different direction to deal with the correctness of SAT solvers has been to verify a proof checker for resolution proofs [48]. This will check and guarantee that the output from a solver for a particular SAT problem is correct.

The DPLL solver Versat [36], mentioned earlier, was formalized and verified in the dependently typed programming language Guru [45] and then translated into imperative C code. This translation is possible because Guru contains mutable arrays. Since Guru allows for the verification of low level optimizations involving such arrays and Versat implements clause learning, the resulting solver is quite efficient. However, this approach differs from ours in that only soundness has been proven for Versat, whilst we have the possibility to deliver a proof in the case of unsatisfiability. This means that while every satisfiable assignment produced by Versat can be trusted, it is not guaranteed that Versat can solve every solvable problem.

A program extraction project related to ours was carried out by Weich [47] who gave two constructive proofs of the decidability of intuitionistic propositional logic and extracted two different programs that, for a given formula, either produce a derivation in intuitionistic sequent calculus, or a Kripke counter-model. The second proof and program extraction were formalized in Minlog for the implicational fragment.

The articles [26, 28] verifying a DPLL SAT solver (in both Coq and Isabelle) were the main motivation for our work. Their approaches involve a formalization of the algorithm to be verified. In contrast, we work in a system that does not require any formalization of algorithms. It is enough to prove that each CNF-formula is either unsatisfiable or has a model. The desired SAT solving algorithm and its correctness proof are then extracted fully automatically.

2. Preliminaries

We begin with some basic definitions, following [26, 28].

Definition 2.1.

(1) A *literal* l is either a positive variable $+v$ or a negative variable $-v$, i.e. a variable v with a label $+$ or $-$ attached.
(2) For every literal l we define the opposite literal $\overline{l}$ by $\overline{+v} = -v$, $\overline{-v} = +v$.

(3) We set $\mathrm{Var}(+v) = \mathrm{Var}(-v) = v$, $\mathrm{Var}(L) = \{\mathrm{Var}(l) \mid l \in L\}$ for a set of literals L, and $\mathrm{Var}(\Delta) = \bigcup\{\mathrm{Var}(L) \mid L \in \Delta\}$ for a set of sets of literals Δ.
(4) A *clause* C is a finite set of literals to be viewed as their disjunction.
(5) A formula in *conjunctive normal form* (CNF) is a finite conjunction of clauses. By a *formula* Δ we will always mean a formula in CNF, and we will identify it with a finite set of clauses $\{C_1, \dots, C_k\}$, representing the conjunction of the C_i.
(6) A *valuation* Γ is a finite set of literals to be viewed as their conjunction.
(7) A valuation Γ is *consistent* if $\forall l\,(l \in \Gamma \to \overline{l} \notin \Gamma)$. We let Cons denote the set of all consistent valuations.
(8) A *model* is a total function M which maps literals[1] to booleans and satisfies the property $\forall l\,(M\ l \leftrightarrow \neg M\ \overline{l})$.

We shall use the abbreviations

- $M \models \Gamma$, for $\forall l \in \Gamma\,(M\ l)$ ('M is a model of Γ'),
- $M \models \Delta$, for $\forall C \in \Delta\, \exists l \in C\,(M\ l)$ ('M is a model of Δ').

We call a valuation Γ and a formula Δ *compatible* if there exists a model satisfying both, i.e. $\exists M\,(M \models \Gamma \wedge M \models \Delta)$; otherwise Γ and Δ are called *incompatible*.

A *sequent* $\Gamma \vdash \Delta$ is a pair consisting of a valuation and a formula. The intended meaning of a sequent $\Gamma \vdash \Delta$ is that Γ and Δ are incompatible. As a special case, when Γ is empty, $\vdash \Delta$ means that Δ is unsatisfiable. In the following we use the notations $X, a := \{x \mid x \in X \vee x = a\}$ and $X \setminus a := \{x \mid x \in X \wedge x \neq a\}$.

Definition 2.2 (DPLL Proof System)**.** The DPLL proof system consists of five rules:

$$\frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, \{l\}}\,(\mathbf{Unit}) \qquad \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, (C, \overline{l})}\,(\mathbf{Red}) \qquad \frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, (C, l)}\,(\mathbf{Elim})$$

$$\frac{}{\Gamma \vdash \Delta, \emptyset}\,(\mathbf{Conflict}) \qquad \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \overline{l} \vdash \Delta}{\Gamma \vdash \Delta}\,(\mathbf{Split})$$

Several variants of the DPLL proof system have featured in the literature. The above definition is closest to the Coq formalisation [26], other formalisations such as [28] and [19] combine the **Unit**, **Red** and **Elim** rules to form a single rule called the "1-literal rule" or "unit propagation".

3. Soundness and Completeness

3.1. **Soundness and Completeness of DPLL.** In this section we sketch the formal proof of soundness and completeness of the DPLL proof system. We will be very brief with the Soundness Theorem since its proof does not carry computational content and a similar proof is carried out in [26, 28]. On the other hand, we will describe the proof of the Completeness Theorem in some detail since we extract our SAT solver from it.

We first reformulate the DPLL proof system as an inductive definition that can be immediately formalized in the Minlog system. The definition has a clause for each rule. We notationally identify a sequent $\Gamma \vdash \Delta$ with the statement '$\Gamma \vdash \Delta$ is derivable'.

[1]We map literals instead of variables as a model is constructed from a set of literals in the form of a valuation.

Remark 3.1. The proof system described in Definition 2.2 has been reformulated for our theorem prover. The set of sequents $\Gamma \vdash \Delta$ is defined inductively by the following (universally quantified) inductive clauses:

Conflict	$\emptyset \in \Delta \to \Gamma \vdash \Delta$
Unit	$\{l\} \in \Delta \to \Gamma, l \vdash \Delta \setminus \{l\} \to \Gamma \vdash \Delta$
Elim	$l \in \Gamma \to l \in C \to C \in \Delta \to \Gamma \vdash \Delta \setminus C \to \Gamma \vdash \Delta$
Red	$l \in \Gamma \to \overline{l} \in C \to C \in \Delta \to \Gamma \vdash (\Delta \setminus C), (C \setminus \overline{l}) \to \Gamma \vdash \Delta$
Split	$\Gamma, l \vdash \Delta \to\ \Gamma, \overline{l} \vdash \Delta \to \Gamma \vdash \Delta$

Theorem 3.2 (Soundness)**.** *If $\Gamma \vdash \Delta$, then Γ and Δ are incompatible.*

The proof proceeds by structural induction on the given derivation of the sequent $\Gamma \vdash \Delta$. We omit further details.

We now turn our attention to the Completeness Theorem for the DPLL proof system. The expected statement of completeness is:

$$\forall \Gamma \in \mathrm{Cons}, \forall \Delta\, (\mathrm{incompatible}(\Gamma, \Delta) \to \Gamma \vdash \Delta).$$

A constructive proof of this statement would yield a program that computes a DPLL proof for incompatible Γ, Δ. We reformulate the statement by replacing the implication 'incompatible$(\Gamma, \Delta) \to \Gamma \vdash \Delta$' with the classically equivalent but constructively stronger disjunction 'compatible$(\Gamma, \Delta) \vee \Gamma \vdash \Delta$'. In this way, we obtain an enhanced program that still computes a DPLL proof for incompatible Γ, Δ, but in addition produces a model if Γ and Δ are compatible.

Theorem 3.3 (Completeness of DPLL)**.**

$$\forall \Gamma \in \mathrm{Cons}, \forall \Delta\, (\mathrm{compatible}(\Gamma, \Delta) \vee \Gamma \vdash \Delta)$$

Proof. We aim to perform the proof in such a way that an efficient program is extracted. Therefore, we adopt the following strategy:

(1) Since performing a **Split** rule is the only computational expensive operation – it is the only rule forcing the proof search to branch – we only apply it if absolutely necessary.
(2) We perform an optimization on the proof level by partitioning the clauses into 'clean' and 'unclean' clauses, where a clause is called clean if we cannot apply **Elim**, **Red** or **Unit** to that clause. This increases the efficiency of the algorithm by reducing the number of comparisons needed.

To this end we show that for all valuations Γ, and formulae Δ, Θ,

$$\emptyset \notin \Theta \wedge \Gamma \in \mathrm{Cons} \wedge \mathrm{Var}(\Gamma) \cap \mathrm{Var}(\Theta) = \emptyset \to (\Gamma \vdash \Delta \cup \Theta) \vee \exists M (M \models \Gamma \wedge M \models \Delta \cup \Theta).$$

The proof is by main induction on the measure

$$\mu(\Gamma; \Delta; \Theta) := |(\Delta \cup \Theta) \setminus\setminus \mathrm{Var}(\Gamma)| + \#(\Delta) + \#(\Theta)$$

where

$$\begin{array}{lcl} |X| & := & \text{the cardinality of } X \\ \Delta \setminus\setminus V & := & \{l | \exists C \in \Delta (l \in C) \wedge \mathrm{Var}(l) \notin V\} \\ \#(\Delta) & := & \sum_{C \in \Delta} |C| \end{array}$$

and a side induction on $|\Delta|$ (i.e. the number of clauses in Δ).

Let Γ, Δ, Θ be given such that $\emptyset \notin \Theta$, $\Gamma \in \mathrm{Cons}$, and $\mathrm{Var}(\Gamma) \cap \mathrm{Var}(\Theta) = \emptyset$.

Case 1 $\Delta = \emptyset$.

Case 1.1 $\Theta = \emptyset$.
We define a model M by $M(l) = \mathrm{True} \leftrightarrow l \in \Gamma$. Then $M \models \Gamma \wedge M \models \emptyset$ holds.

Case 1.2 $\Theta \neq \emptyset$.
Let C be a clause in Θ and let $l \in C$ ($C \neq \emptyset$, by the assumption on Θ). Then $\mu((\Gamma, l); \Theta; \emptyset) < \mu(\Gamma; \emptyset; \Theta)$ since $|\Theta \setminus\setminus \mathrm{Var}(\Gamma, l)| < |\Theta \setminus\setminus \mathrm{Var}(\Gamma)|$ and $\#(\Theta)+\#(\emptyset) = \#(\emptyset)+\#(\Theta)$. Furthermore, for the values (Γ, l), Θ, $\emptyset$ the hypotheses of the theorem are clearly satisfied. Hence the induction hypothesis for these values yields

$$(\Gamma, l \vdash \Theta) \vee \exists M(M \models \Gamma, l \wedge M \models \Theta) \tag{3.1}$$

Similarly, we can apply the induction hypothesis to $(\Gamma, \overline{l})$, Θ, and $\emptyset$ yielding

$$(\Gamma, \overline{l} \vdash \Theta) \vee \exists M(M \models \Gamma, \overline{l} \wedge M \models \Theta) \tag{3.2}$$

The disjunctions (3.1) and (3.2) result in 4 cases: In the case that $\Gamma, l \vdash \Theta$ and $\Gamma, \overline{l} \vdash \Theta$ hold the **Split** rule is applied and we obtain $\Gamma \vdash \Theta$. In all other cases we use one of the models obtained from the induction hypotheses.

Case 2 $\Delta = \Delta', C$.
We perform a case distinction on whether the valuation Γ has a literal in common with C.

Case 2.1 $\Gamma \cap C = \emptyset$.
We perform a further case distinction on the cardinality of the clause C.

Case 2.1.1 $C = \emptyset$.
It suffices to show $\Gamma \vdash (\Delta', \emptyset) \cup \Theta$. This follows from the **Conflict** rule.

Case 2.1.2 $C = \{l\}$.
If $\overline{l} \in \Gamma$, then $\Gamma \vdash (\Delta', \{l\}) \cup \Theta$ can be derived by applying (in backwards fashion) the **Red** rule followed by the **Conflict** rule. If $\overline{l} \notin \Gamma$, then we use the induction hypothesis with (Γ, l), $\Delta' \cup \Theta$, $\emptyset$. This is possible since $\mu((\Gamma, l); \Delta' \cup \Theta; \emptyset) < \mu(\Gamma; (\Delta', \{l\}); \Theta)$ because $|(\Delta' \cup \Theta) \setminus\setminus \mathrm{Var}(\Gamma, l)| < |(\Delta' \cup (\{l\}, \Theta)) \setminus\setminus \mathrm{Var}(\Gamma)|$ and $\#(\Delta' \cup \Theta) < \#(\Delta', \{l\})+\#(\Theta)$. Since for the values (Γ, l), $\Delta' \cup \Theta$, $\emptyset$ the hypotheses of the theorem are satisfied (i.p. Γ, l is consistent since $\overline{l} \notin \Gamma$), we obtain the disjunction $(\Gamma, l \vdash \Delta' \cup \Theta) \vee \exists M(M \models \Gamma, l \wedge M \models (\Delta' \cup \Theta))$. In the case that $\Gamma, l \vdash \Delta' \cup \Theta$ holds we apply the **Unit** rule resulting in $\Gamma \vdash \Delta \cup \Theta$. In the other case we have a model of Γ, l and $\Delta' \cup \Theta$ which clearly also models Γ and $\Delta \cup \Theta$.

Case 2.1.3 $|C| \geq 2$.
We perform a case distinction on $\exists l\, (l \in C \wedge \overline{l} \in \Gamma) \vee \neg\exists l (l \in C \wedge \overline{l} \in \Gamma)$. This disjunction can be proven constructively, since the sets involved are finite.

Case 2.1.3.1 $\overline{l} \in \Gamma$ *for some* $l \in C$.
Then we have $\mu(\Gamma; (\Delta', C \setminus l); \Theta) < \mu(\Gamma; (\Delta', C); \Theta)$ since $\#(\Delta', C \setminus l) < \#(\Delta', C)$ and $|(\Delta', C \setminus l) \setminus\setminus \mathrm{Var}(\Gamma)| = |(\Delta', C) \setminus\setminus \mathrm{Var}(\Gamma)|$. The hypotheses of the theorem are satisfied for the

chosen values. Hence we obtain, by induction hypothesis, $(\Gamma \vdash (\Delta', (C \setminus l)) \cup \Theta) \vee \exists M(M \models \Gamma \wedge M \models (\Delta', (C \setminus l)) \cup \Theta)$. In the case that $\Gamma \vdash (\Delta', (C \setminus l)) \cup \Theta$ holds, we apply the **Red** rule. In the other case we have a model of Γ and $(\Delta', (C \setminus l)) \cup \Theta$ which also models the weaker formula $(\Delta', C) \cup \Theta$.

Case 2.1.3.2 $\neg\exists l\,(l \in C \wedge \overline{l} \in \Gamma)$.
In this case we may move C from Δ to Θ: Since $\mu(\Gamma; \Delta'; (\Theta, C)) \leq \mu(\Gamma; (\Delta', C); \Theta)$ we can apply the side induction hypothesis to Γ, Δ', (Θ, C). Since for these values the hypotheses of the theorem are satisfied we obtain $\Gamma \vdash \Delta' \cup (\Theta, C) \vee \exists M(M \models \Gamma \wedge M \models \Delta' \cup (\Theta, C))$ which is the same as the required disjunction $\Gamma \vdash (\Delta', C) \cup \Theta \vee \exists M(M \models \Gamma \wedge M \models (\Delta', C) \cup \Theta)$.

Case 2.2 $\Gamma \cap C \neq \emptyset$.
We can prove constructively that in this case Γ and C have some literal l in common. We apply the induction hypothesis to Γ, $(\Delta', (C \setminus l))$, Θ. Since clearly the measure decreases, $\#(\Delta', (C \setminus l)) < \#(\Delta', C)$ and $|(\Delta', (C \setminus l)) \setminus\setminus \mathrm{Var}(\Gamma)| = |(\Delta', C) \setminus\setminus \mathrm{Var}(\Gamma)|$, and the hypotheses of the theorem are satisfied, we obtain $\Gamma \vdash (\Delta', (C \setminus l)) \cup \Theta$ or $\exists M(M \models \Gamma \wedge M \models (\Delta', (C \setminus l)) \cup \Theta)$. In the first case we apply the **Elim** rule, in the second case we use the model provided. □

3.2. **Resolution.** The resolution proof system [39] is widely used in practical applications, for instance in tools for proof checking and debugging [44] or interchange between different solvers [22]. State-of-the-art SAT solvers such as MiniSAT [18] and zChaff [33] return (extended) resolution proofs for unsatisfiable problems. By formalizing that every DPLL derivation has an equivalent resolution derivation, and combining this result with the completeness proof from the previous section, we can extract a SAT solver which produces resolution derivations. The equivalence of DPLL and resolution was first shown by Robinson [40] who translated between the two proof systems using semantic trees.

By enriching the systems with size information we are able to show that the size of the resulting resolution proof does not exceed the size of the original DPLL proof.

For every valuation Γ we define a clause $\overline{\Gamma}$ representing its negation by $\overline{\{l_1, \ldots, l_k\}} = \{\overline{l_1}, \ldots, \overline{l_k}\}$.

Definition 3.4 (Resolution Proof System)**.** The derivable resolution sequents $\Gamma \overset{n}{\underset{\mathrm{Res}}{\vdash}} C$ with a derivation of size n are conveniently defined by two rules: subsumption (or axiom) and resolution.

$$\frac{}{\Delta, C \overset{0}{\underset{\mathrm{Res}}{\vdash}} C'}\,(\mathbf{Sub})\ C \subseteq C' \qquad \frac{\Delta \overset{n}{\underset{\mathrm{Res}}{\vdash}} C \vee l \qquad \Delta \overset{m}{\underset{\mathrm{Res}}{\vdash}} C' \vee \overline{l}}{\Delta \overset{n+m+1}{\underset{\mathrm{Res}}{\vdash}} C \vee C'}\,(\mathbf{Res})$$

We also need a version of the DPLL proof system with added bounds in order to speak about the sizes of the proofs.

Remark 3.5 (Derivable refined DPLL sequents)**.** The derivable DPLL sequents $\Gamma \overset{n}{\underset{\mathrm{DPLL}}{\vdash}} \Delta$ with a derivation of size n are inductively defined by the following clauses:

$$\textbf{Conflict} \quad \emptyset \in \Delta \to \Gamma \overset{0}{\underset{\text{DPLL}}{\vdash}} \Delta$$

$$\textbf{Unit} \quad \{l\} \in \Delta \to \Gamma, l \overset{n}{\underset{\text{DPLL}}{\vdash}} \Delta \setminus \{l\} \to \Gamma \overset{n+1}{\underset{\text{DPLL}}{\vdash}} \Delta$$

$$\textbf{Elim} \quad l \in \Gamma \to l \in C \to C \in \Delta \to \Gamma \overset{n}{\underset{\text{DPLL}}{\vdash}} \Delta \setminus C \to \Gamma \overset{n+1}{\underset{\text{DPLL}}{\vdash}} \Delta$$

$$\textbf{Red} \quad l \in \Gamma \to \overline{l} \in C \to C \in \Delta \to \Gamma \overset{n}{\underset{\text{DPLL}}{\vdash}} (\Delta \setminus C), (C \setminus \overline{l}) \to \Gamma \overset{n+1}{\underset{\text{DPLL}}{\vdash}} \Delta$$

$$\textbf{Split} \quad \Gamma, l \overset{n}{\underset{\text{DPLL}}{\vdash}} \Delta \to \ \Gamma, \overline{l} \overset{m}{\underset{\text{DPLL}}{\vdash}} \Delta \to \Gamma \overset{n+m+1}{\underset{\text{DPLL}}{\vdash}} \Delta$$

Remark 3.6. The resolution proof system from Definition 3.4 has been reformulated as follows for our theorem prover. The derivable resolution sequents $\Gamma \overset{n}{\underset{\text{Res}}{\vdash}} C$ with a derivation of size n are inductively defined by the following clauses:

$$\textbf{Sub} \quad C_0 \in \Delta \to C_0 \subseteq C \to \Gamma \overset{0}{\underset{\text{Res}}{\vdash}} C$$

$$\textbf{Res} \quad \Delta \overset{n}{\underset{\text{Res}}{\vdash}} (C' \vee \overline{l}) \to \ \Delta \overset{m}{\underset{\text{Res}}{\vdash}} (C \vee l) \to \Delta \overset{n+m+1}{\underset{\text{Res}}{\vdash}} (C \vee C')$$

Theorem 3.7 (DPLL implies Resolution)**.** *For all consistent valuations* Γ*, CNF formulae* Δ *and natural numbers* n*: If* $\Gamma \overset{n}{\underset{DPLL}{\vdash}} \Delta$*, then* $\Delta \overset{m}{\underset{Res}{\vdash}} \bar{\Gamma}$ *for some* $m \leq n$*.*

Proof. The proof is an easy induction on DPLL derivations. We only sketch the overall idea. The **Conflict** and **Split** rule translate into the **Sub** and **Res** rule respectively. Both of these rules have the same cost to perform them as the DPLL rules and so the size of the derivations are less or equal. An application of the **Unit** rule is a special case of the **Res** rule in which one of the branches is obtained via a subsumption of a unit clause. The size of these two proofs is less or equal since the cost of performing the **Sub** rule and **Res** rule together is the same as that of the **Unit** rule. Finally, both the **Elim** and **Red** DPLL rules correspond to a form of weakening in the resolution proof which is done at no cost because the resulting resolution proofs are smaller in size than the DPLL proofs. □

Remark 3.8. One can also easily prove that resolution implies DPLL, more precisely, if $\Delta \underset{\text{Res}}{\vdash} C$, then $\bar{C} \underset{\text{DPLL}}{\vdash} \Delta$. However, as long as the sizes of derivations are measured only in terms of the number of applications of rules (as we do above), no size bound can be given. The reason is that the translation of one instance of the subsumption rule

$$\frac{}{\Delta, C \overset{0}{\underset{\text{Res}}{\vdash}} C'} \, (Sub) \; C \subseteq C'$$

into DPLL requires n applications of the **Red** rule where n is the number of literals in C.

The Completeness Theorem for DPLL (Theorem 3.3), adapted to the DPLL system with size information, and Theorem 3.7 (a) immediately imply:

Theorem 3.9 (Completeness of the Resolution Proof System)**.**

$$\forall \Delta \, ((\exists M \, M \models \Delta) \vee (\exists n \, \Delta \overset{n}{\underset{Res}{\vdash}} \emptyset))$$

The program extracted from Theorem 3.7 translates DPLL derivations into equivalent resolution derivations. This translator and the SAT solver extracted from the Completeness Theorem for DPLL (Theorem 3.3) are combined in the program extracted from Theorem 3.9 to a SAT solver that yields resolution refutations for unsatisfiable formulae. Since the computationally hard and interesting part of this program is entirely contained in the DPLL-based SAT solver, we will restrict our attention to the latter when we discuss the extracted programs in detail in Sect. 5.

4. Program Extraction

4.1. **Theory.** Program extraction in Minlog is based on modified realizability [23]. We highlight a few aspects that are important to understand the optimizations we achieved. For a complete and precise description of program extraction we refer to [42]. A formula is said to have *computational content* if it has at least one occurrence of $\exists$ or $\vee$ at a strictly positive position. To every such formula A one assigns a type $\tau(A)$ of 'potential realizers'. If the formula has no computational content, one sets $\tau(A) = \epsilon$. From a proof of a formula A with computational content one can extract a program M of type $\tau(A)$ that *realizes* A (written $M\,\mathbf{r}\,A$), that is, M solves the computational problem expressed by A. In order to fine-tune the computational content, in particular to remove redundant content, Minlog offers, besides the usual quantifiers $\forall$ and $\exists$, the *non-computational (nc)* quantifiers $\forall_{\mathrm{nc}}$ and $\exists_{\mathrm{nc}}$ (which roughly correspond to quantification in Prop in Coq). These have the same logical meaning as the usual quantifiers, but indicate that the extracted program does not operate on the quantified variable, only on its realizer. The definitions of the type and the realizability relations for the ordinary universal quantifier contrasted with its nc version are:

$$\begin{array}{lcllcl} \tau(\forall x^\rho A) & = & \rho \to \tau(A) & f\,\mathbf{r}\,\forall x^\rho A & = & \forall x^\rho (f(x)\,\mathbf{r}\,A) \\ \tau(\forall_{\mathrm{nc}} x^\rho A) & = & \tau(A) & a\,\mathbf{r}\,\forall_{\mathrm{nc}} x^\rho A & = & \forall x^\rho (a\,\mathbf{r}\,A) \end{array}$$

Similarly for the two versions of the existential quantifier:

$$\begin{array}{ll} \tau(\exists x^\rho A) = \rho \times \tau(A) & (a,y)\,\mathbf{r}\,\exists x^\rho A = a\,\mathbf{r}\,A[y/x] \\ \tau(\exists_{\mathrm{nc}} x^\rho A) = \tau(A) & a\,\mathbf{r}\,\exists_{\mathrm{nc}} x^\rho A = \exists x^\rho (a\,\mathbf{r}\,A) \end{array}$$

One sees that for the nc-quantifiers the realizers do not depend on the quantified variables. The program extraction procedure respects the different kind of quantifiers by omitting in the nc case any information corresponding to the quantified variable. The proof rules for the nc-quantifiers are subject to stricter variable conditions ensuring that the omitted information is indeed not needed in the extracted program. Minlog is able to automatically detect the maximal set of occurrences of quantifiers in a proof that can be made non-computational without compromising the correctness of the proof [38]. This holds for the logical parts of the proof only. In the formalization of inductive definitions one has to manually place $\forall_{nc}$ quantifiers.

4.2. **Extraction to Haskell.** The programs extracted by Minlog are terms in Minlog's internal term language. This has the advantage that extracted programs can be reused for further proofs, and properties of the programs can be formally proven, again inside Minlog. Furthermore, the extracted programs are provably correct, and a (soundness) proof of this fact is automatically generated by Minlog. However, there are also inherent disadvantages: the interoperability of the extracted programs with external libraries or devices is limited, and executing the programs is sometimes slow. For both these reasons, it makes sense to translate the extracted programs into more conventional, general-purpose programming languages. Minlog implements a translation to Haskell (and also a limited translation to Scheme). Extracting to a lazy language such as Haskell makes the treatment of coinduction and corecursion (which is not used in our example) particularly simple [32].

There is a close fit between Haskell and the Minlog term language, and the translation is quite straightforward; basic terms such as variables, lambda abstractions, etc are translated to the corresponding Haskell terms. Standard algebras such as e.g. lists, integers, booleans, sum and product types are translated to their implementation in the Haskell Prelude, while user-defined algebras in general are translated to algebraic data types. Natural numbers are translated to (unbounded) integers for efficiency.[2] Program constants and their computation rules in Minlog correspond to functions defined by pattern matching in Haskell. Some care must be taken for e.g. the natural numbers; in Minlog, pattern matching on natural numbers is possible, but natural numbers are translated to integers, for which no pattern matching is available in Haskell. Instead guard conditions have to be used. Recursion operators, realizing structural induction, are automatically generated as Haskell functions by the translation. Minlog also supports general recursion along a decreasing measure, which makes sure that the program terminates. The Minlog implementation of the general recursion operator ensures that recursive calls are only made on arguments that are smaller than the current argument with respect to the measure:

$$\mathrm{gRec} : (\rho \to \mathbb{N}) \to \rho \to (\rho \to (\rho \to \tau) \to \tau) \to \tau$$
$$\mathrm{gRec}(\mu, x, f) = f(x, (\lambda y \,.\, \text{if } \mu(y) < \mu(x) \text{ then } \mathrm{gRec}(\mu, y, f) \text{ else } \mathrm{inhab}_\tau))$$

(inhab_τ is a canonical inhabitant of τ, justified by the fact that all domains are inhabited in the intended, standard semantics). Note that the (potentially expensive) test $\mu(y) < \mu(x)$ is computationally unnecessary, since at runtime we already know that our extracted program will only use recursive calls on smaller arguments. However, this test is needed because of Minlog's eager evaluation strategy. Omitting the test:

$$\mathrm{gRec}(x, f) = f(x, (\lambda y \,.\, \mathrm{gRec}(y, f))) \tag{4.1}$$

would make Minlog get stuck in an endless loop, forever evaluating the recursive call $\mathrm{gRec}(y, f)$ regardless of whether it is going to be used or not.

However, since Haskell is a lazy language, we can safely implement general recursion using (4.1). This can give large efficiency gains in certain situations (see Section 6.1). In a lazy setting, soundness of this variant of the program extraction process can still be proven, and the Haskell translation supports this optimization. However, there is now a discrepancy between Minlog programs and their Haskell translations: if called in a way that does not respect the measure, the Minlog implementation of gRec will halt with an arbitrary value,

[2]Using bounded `Int`s instead of unbounded `Integer`s would of course not be sound. In the cases where it would be safe to do so, it would also not result in any particular performance gains, since GHC stores small `Integer`s as `Int`s.

while the Haskell version will diverge. For this reason, the optimization can be turned on and off with a switch, if identical behavior is important. Of course, every extracted term will respect the measure.

5. The Extracted Program

The size of the DPLL formalization is approximately 5500 lines of Minlog code. The extracted program comes to 300 lines of code as a Minlog term and 600 lines of Haskell code. In the following we present two versions of our extracted solver: one optimized with $\forall_{\mathrm{nc}}$ quantifiers which we shall refer to as the $\forall_{\mathrm{nc}}$ solver, and the other without these optimizations which we shall refer to as the $\forall$ solver.

The $\forall$ solver takes a CNF formula Δ represented as a list of clauses as input, and produces either a model of Δ or a derivation of its unsatisfiability. Models are represented as functions from literals to booleans. An algebraic data type for DPLL derivations is automatically generated from its inductive definition in Minlog. It has five constructors, one for each of the DPLL rules in Definition 3.1:

```
data Algdpll = CConflict Valu For
             | CElim Valu For Cla Lit Algdpll
             | CUnit Valu For Lit Algdpll
             | CRed Valu For Cla Lit Algdpll
             | CSplit Valu For Lit Algdpll Algdpll
  deriving (Show, Read, Eq, Ord)
```

Each constructor takes a formula and a valuation as arguments. The formula itself never changes during the proof and is only part of the algebra for the purpose of proving correctness and does not play a role in any computation. While the valuation changes during the proof search, these changes can be captured by indicating which literal was added by the **Unit** and **Split** rules, thus making the valuation redundant as well. We added nc-quantifiers to the definition by hand in order to remove redundant computational content, resulting in

```
data Algdpll = CConflict
             | CElim Cla Lit Algdpll
             | CUnit Lit Algdpll
             | CRed Cla Lit Algdpll
             | CSplit Lit Algdpll Algdpll
  deriving (Show, Read, Eq, Ord)
```

The control structure of the program closely follows the structure of the case distinctions and proofs by induction performed in the proof. Lemmas invoked during the proof are extracted separately and called as procedures. Since the proof is by general induction along a measure, the main body of the program is using general recursion along the same measure.

6. Execution of the Extracted Program

In the following we will see how both $\forall$ and $\forall_{\mathrm{nc}}$ solvers behave when they are applied to a number of SAT problems. The extracted decision procedure was run on several instances of the pigeon hole principle [14] in both Minlog and as Haskell programs. The pigeon hole principle states that there is no injective function that maps $\{1, 2 \ldots, n\}$ to $\{1, 2, \ldots, n-1\}$.

Definition 6.1 (Pigeon Hole Formula)**.** $\mathbf{PHP}(n,m) := \{\{l_{i,1},\ldots,l_{i,m}\}|1 \leq i \leq n\} \cup \{\{\overline{l_{i,k}},\overline{l_{j,k}}\}|1 \leq i < j \leq n, 1 \leq k \leq m\}$

Here $l_{i,k}$ represents the statement "pigeon i sits in hole k". The whole formula $\mathbf{PHP}(n,m)$ states that n pigeons sit in m holes such that no two pigeons are in the same hole. Hence, $\mathbf{PHP}(n,m)$ is satisfiable iff $n \leq m$. For example, if we run our DPLL solver with the formula $\mathbf{PHP}(2,1) = \{\{l_{11}\},\{l_{21}\},\{\overline{l_{11}},\overline{l_{21}}\}\}$, the following derivation is produced:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{l_{11},l_{21} \vdash \emptyset}\,\textbf{Conflict}}{l_{11},l_{21} \vdash \{\overline{l_{21}}\}}\,\textbf{Red}}{l_{11},l_{21} \vdash \{\overline{l_{11}},\overline{l_{21}}\}}\,\textbf{Red}}{l_{11} \vdash \{l_{21}\},\{\overline{l_{11}},\overline{l_{21}}\}}\,\textbf{Unit}}{\vdash \{l_{11}\},\{l_{21}\},\{\overline{l_{11}},\overline{l_{21}}\}}\,\textbf{Unit}$$

The following is the Minlog output for the pigeon hole formulae $\mathbf{PHP}(2,1)$. There is a constructor `CsuccessZero` of the algebra `success` which represents the disjunction in the main proof statement. The data type extracted from this algebra can be seen as a union type that contains either a DPLL derivation or a model of the formula. In this case it contains a DPLL derivation showing that the formula is unsatisfiable. The arguments to `CsuccessZero` store how the **Conflict** is derived. The literal l_{11} is represented as `(Pos(Variable 11))` in the Minlog formalization and the clause $\{\overline{l_{11}},\overline{l_{21}}\}$ is represented as `CC(Neg(Variable 21)::(Neg(Variable 11)):)`.

```
CsuccessZero
 (CUnit
  (Pos(Variable 11))
  (CUnit
   (Pos(Variable 21))
   (CRed
    (CC(Neg(Variable 21)::(Neg(Variable 11)):))
    (Pos(Variable 21))
    (CRed
     (CC(Neg(Variable 11)):)
     (Pos(Variable 11))
     CConflict))))
```

Running the DPLL solver on a satisfiable formula results in a function which maps literals to booleans. For example running the solver with $\mathbf{PHP}(2,2)$ results in the function M : literals $\to \mathbb{B}$ where $\mathrm{M}(l) =$ True iff $l \in \{l_{12},\overline{l_{11}},l_{21},\overline{l_{22}}\}$. The Minlog output for the satisfiable formula $\mathbf{PHP}(2,2)$ is as follows. Here the square brackets represent a lambda abstraction for the literal l_0. The model M is written as $\lambda l_0.l_0 \in \{l_{12},\overline{l_{11}},l_{21},\overline{l_{22}}\}$.

```
CsuccessOne
 ([l0]
  [if (l0=Pos(Variable 12))
      True
      [if (l0=Neg(Variable 11))
          True
```

Table 1: Performance in Minlog versus Haskell

Formula	Minlog $\forall$	Minlog $\forall_{\mathrm{nc}}$	Compiled (`ghc -O2`)		Compiled (`ghc -O2 -fllvm`)	
	Witness	Witness	Witness	Yes/No	Witness	Yes/No
PHP(4,3)	33.62s	11.61s	0.019s	0.006s	0.015s	0.004s
PHP(4,4)	5.45s	5.25s	0.019s	0.010s	0.014s	0.007s
PHP(5,4)	13m54s	2m41s	0.055s	0.020s	0.036s	0.012s
PHP(5,5)	26.09s	25.03s	0.024s	0.015s	0.020s	0.010s
PHP(6,5)	5h35m41s	37m25s	0.367s	0.066s	0.279s	0.039s
PHP(6,6)	1m34.11s	1m24.88s	0.035s	0.025	0.025s	0.015s
PHP(8,8)	-	-	0.054s	0.029s	0.040s	0.025s
PHP(9,8)	-	-	-	1m21.915s	-	32.062s
PHP(9,9)	-	-	0.064s	0.042s	0.052s	0.030s
PHP(10,9)	-	-	-	102m 16s	-	15m 5s

Table 2: Performance compared to Versat

Formula	$\forall_{\mathrm{nc}}$ compiled (Yes/No)	Versat
PHP(7,6)	0.226s	0.089s
PHP(8,7)	2.42s	0.794s
PHP(9,8)	32.062s	17.217s
PHP(10,9)	15m 5s	15m 46s

```
[if (l0=Pos(Variable 21))
    True
    (l0=Neg(Variable 22))]]])
```

6.1. **Comparison of Program Performance.** The $\forall$ solver and $\forall_{\mathrm{nc}}$ solver were compared using both unsatisfiable $\mathbf{PHP}(n+1,n)$ and satisfiable $\mathbf{PHP}(n,n)$ pigeon hole formulae. The unsatisfiable pigeon hole formulae are harder than the satisfiable formulae as they have a large search space that must be traversed entirely by the solver in order to construct a derivation. This difficulty can be seen – compare column 2 and 3 in Table 1 – when both the $\forall$ and $\forall_{\mathrm{nc}}$ solver are applied to the unsatisfiable pigeon hole formulae. The solver without the optimization takes considerably longer to construct a derivation of unsatisfiability. This is due to computationally irrelevant data being stored in the unoptimized derivations.

The next two columns of Table 1 present two versions of the $\forall_{\mathrm{nc}}$ solver when extracted to Haskell and compiled by the Glasgow Haskell Compiler (GHC). The first returns a witness of the result i.e. either a model which satisfies the formula or a derivation of its unsatisfiability. The second returns only a Yes or No answer as to whether a formula is satisfiable or not. Due to the inherent laziness of Haskell the two programs differ quite dramatically in their behavior. The solver that returns a Yes/No answer performs considerably faster compared to the solver which produces the witness in addition. By using the Low Level Virtual Machine (LLVM) backend [24] for GHC, a further speed up was achieved, which can be seen in the last two columns of Table 1.

Table 3: Industrial case study: Extracted solver versus Versat

Formula	$\forall_{\mathrm{nc}}$ compiled (Yes/No)	Versat
$\neg R1 \vee \neg R3$	7.028s	0.050s
$\neg R1 \vee \neg R4$	6.961s	0.040s
$\neg R2 \vee \neg R3$	7.105s	0.053s
$\neg R2 \vee \neg R4$	7.059s	0.044s
$\neg R3 \vee \neg R4$	7.015s	0.047s

We also compared the performance of our $\forall_{\mathrm{nc}}$ solver, compiled using the LLVM backend of GHC, with that of Versat [36]. Our solver was run with the option of not computing a witness since Versat does generally not compute a proof. The results in Table 2 show that our solver is comparable with Versat. It is slower on the easier formulae and faster on the hardest pigeon hole formulae. This is because the clause learning optimization of Versat has some overhead and does not increase the performance on pigeon hole formulae. The point of the learned clauses is to reduce the search space for the solver. In this case, they instead consume more memory and time to compute.

6.2. **Industrial Case Study.** The same version of our solver was also applied to the verification of a real world railway control system which was provided by our industrial partner Invensys Rail (now Siemens), via a description in Ladder logic. We adapted [21] to translate Ladder logic programs into Minlog/Haskell and the industrial tool SCADE [1], and also performed a comparison with Versat. The SAT problem is formulated to perform falsification checking, as described in [43], that is, a satisfying assignment represents a counter example, and an unsatisfiable result means the safety property can not be violated in the system. The size of our case study is 14726 clauses and 8166 variables. For comparison, we present the run-times for checking five safety conditions which show that two conflicting routes, out of a set of four routes $R1, \ldots, R4$, can not be active in the railway at the same time. For each of the five conditions our solver produces a proof certifying that the safety property holds in approximately 7s. The SCADE suite can verify that each of the safety properties holds in less than one second (no greater accuracy of run-times provided by the system for this case).

While we cannot expect to compete with an industrial tool on speed and functionality, we have been able to solve a large practical problem in a reasonable amount of time. It is important to note that the solver inside the SCADE suite has not been formally verified whereas our solver has. Interestingly however, also Versat solves these problems in less than one second – see Table 3 for a comparison between our extracted solver and Versat – that is, we may conclude that optimizations such as clause-learning and the use of efficient data structures that enable to efficiently parse and identify (un-)satisfiability of a formula indeed improve the performance for this type of problems (and our extracted solver should be extended by these optimizations as well).

7. Conclusion

We have presented a conceptually new approach to the synthesis and verification of SAT algorithms that, in contrast to similar work in Coq and Isabelle [26, 28] does not require the

formalization of the SAT programs in the formal system, but obtains SAT algorithms purely by program extraction. To this end, we formalized the DPLL proof system and performed a constructive proof from which a correct SAT solving algorithm was extracted automatically. The extracted program attempts to show the (un)satisfiability of a propositional formula in conjunctive normal form. If the CNF formula is satisfiable it produces a model of the formula; otherwise it produces a derivation showing unsatisfiability. We strategically placed $\forall_{\mathrm{nc}}$ quantifiers into the proof to reduce the complexity of the extracted program and increase its performance. The solver containing $\forall_{\mathrm{nc}}$ quantifiers was extracted into the functional programming language Haskell, and the performance of the two solvers was evaluated using pigeon hole formulae. We have also shown how it is possible to extract a program that translates between DPLL and resolution proofs. This was done in such a way that we obtain some qualitative information about quantitative aspects of the extracted program i.e. computational complexity. Using this translation it was possible to extract a resolution solver based on the DPLL proof system.

Overall, our paper shows that the approach of developing verified programs via extraction from proofs is scalable to non-trivial applications. Furthermore, it demonstrates how to include efficiency considerations into this approach. For instance, we have avoided repeated unnecessary look-ups of clauses by the split of clause sets in two sets Δ and Θ. This counters the often heard argument that with program extraction one 'loses the grip' on the program and its efficiency. It is important to note that these efficiency considerations do not compromise the correctness of the extracted program since these are applied at the proof level where correctness is guaranteed by the proof system.

We consider the fact that our approach does not require any formalization of algorithms a major advantage, since it means that program development via extraction can be carried out in a formal system that is much more lightweight than in the verification approach, where the term language must include a programming language, and the meaning of the programming constructs must be specified by axioms and proof rules. This advantage is particularly striking in applications in analysis [4, 5] where corecursive exact real number algorithms (whose formalization and specification is non-trivial and subject of ongoing research) can be automatically extracted from proofs involving only coinductive definitions in the form of largest fixed points of predicate transformers.

7.1. **Future Work.** There are two directions for further work: applying our method to extract a more advanced class of SAT solvers, and applying our approach to a different class of decision problems.

We are in the process of formalizing optimizations such as clause learning and conflict analysis [9, 33, 29]. This requires a modification of the DPLL proof system such that it captures the additional behavior. A completeness theorem has then been proven for the modified calculus. We currently have extracted a prototype clause learning solver from this proof. In order for this solver to be an improvement on the previous one we need to lower the computational overhead resulting from clause learning. Such a solver would also benefit from lazy data structures such as the two-watched-literal scheme. It is unclear whether the inherent laziness of Haskell will provide the same effect as these data structures or if they would have to be formalized as part of the proof.

It is desirable to be able to solve not just propositional formulae but also first-order formulae. This is possible by extending SAT algorithms so that they can apply some background theory for first order formulae. Such algorithms are called Satisfiability Modulo

Theories (SMT) solvers. We would have to formalize a proof system used by SMT solvers such as abstract DPLL [34] and then perform a completeness proof. A solver extracted from such a proof system would be able to solve a broader range of problems described in a language richer than propositional logic.

7.2. **Sources.** The Minlog formalization optimized with $\forall_{nc}$ quantifiers and its extracted program as Haskell code can be found at `http://cs.swan.ac.uk/minlog/dpll/`.

Acknowledgment. We would like to thank the anonymous referees for their constructive comments. The financial support of Invensys Rail/Siemens Rail Automation is gratefully acknowledged.

References

[1] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing Safe, Reliable Systems Using Scade. In *Leveraging Applications of Formal Methods*, volume 4313 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2006.

[2] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction — A Basis for Applications II*. Kluwer, 1998.

[3] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82:25–49, 2006.

[4] U. Berger, K. Miyamoto, H. Schwichtenberg, and M. Seisenberger. Minlog — A Tool for Program Extraction Supporting Algebras and Coalgebras. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Conference on Algebra and Co-Algebra in Computer Science 2011*, volume 6859 of *Lecture Notes in Computer Science*, pages 393–399. Springer, 2011.

[5] U. Berger and M. Seisenberger. Proofs, Programs, Processes. *Theory of Computing*, 51(3):313–329, 2012.

[6] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Munich, 2003.

[7] S. Berghofer and T. Nipkow. Executing Higher Order Logic. In P. Callaghan, L. Zhaohui, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2002.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[9] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, 2009.

[10] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[11] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[12] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[13] R. L. Constable. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

[14] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, 1979.

[15] H. B. Curry, R. Feys, and W. Craig. Combinatory Logic. *Studies in Logic and the Foundations of Mathematics*, 1, 1958.

[16] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5, 1962.

[17] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[18] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
[19] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
[20] W. A. Howard. The formulae-as-types notion of construction. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980.
[21] K. Kanso, F. Moller, and A. Setzer. Automated verification of signalling principles in railway interlocking systems. In *Automated Verification of Critical Systems 2008*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 19–31. Elsevier, 2009.
[22] Chantal Keller. Extended Resolution as Certificates for Propositional Logic. In J. C. Blanchette and J. Urban, editors, *Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series*, pages 96–109, 2013.
[23] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pages 101–128, 1959.
[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization 2004*. IEEE Computer Society, 2004.
[25] A. Lawrence, U. Berger, and M. Seisenberger. Extracting a DPLL Algorithm. In *Mathematical Foundations of Programming Semantics*, volume 486 of *Electronic Notes in Theoretical Computer Science*, pages 243–256, 2012.
[26] S. Lescuyer and S. Conchon. A reflexive formalization of a SAT solver in Coq. In *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics*, 2008.
[27] F. Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.
[28] F. Marić and P. Janičić. Formal correctness proof for DPLL procedure. *Informatica*, 21(1):57–78, 2010.
[29] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
[30] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
[31] Minlog Proof Assistant. `http://www.minlog-system.de`.
[32] K. Miyamoto, F. Nordvall Forsberg, and H. Schwichtenberg. Program Extraction from Nested Definitions. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 370–385. Springer, 2013.
[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Annual ACM IEEE Design Automation Conference*, pages 530–535. ACM, 2001.
[34] R. Nieuwenhuis, A Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *Logic for Programming, Artificial Intelligence and Reasoning 2004*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 36–50, 2005.
[35] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/HOL: A proof assistant for higher-order logic. *Lecture Notes in Computer Science*, 2283, 1999.
[36] D. Oe, A. Stump, C. Oliver, and K. Clancy. Versat: A Verified Modern SAT Solver. In *Verification, Model Checking and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2012.
[37] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In *Theorem Proving in Higher Order Logics 2007*, pages 232–245. Springer, 2007.
[38] D. Ratiu and H. Schwichtenberg. Decorating proofs. In S. Feferman and W. Sieg, editors, *Proofs, Categories and Computations. Essays in honor of Grigori Mints*, pages 171–188. College Publications, 2010.
[39] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
[40] J. A. Robinson. The generalized resolution principle. *Machine Intelligence*, 3:77–94, 1968.
[41] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. *Theory of Computing Systems*, 43(3):229–239, 2008.
[42] H. Schwichtenberg and S. Wainer. *Proofs and Computations*. Cambridge University Press, 1st edition, 2012.
[43] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, pages 108–125. Springer, 2000.

[44] C. Sinz and A. Biere. Extended resolution proofs for conjoining BDDs. In D. Grigoriev, J. Harrison, and E. Hirsch, editors, *CSR 2006*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.

[45] A. Stump, M. Deters, A. Petcher, and T. Schiller, T. Simpson. Verified programming in Guru. In *Programming Languages Meets Program Verification, PLPV'09*, pages 49–58, 2009.

[46] K. N. Verma, J. Goubault-Larrecq, S. Parsad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN*, volume 1961 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 2000.

[47] K. Weich. Decision procedures for intuitionistic propositional logic by program extraction. In H. de Swart, editor, *TABLEAUX'98*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 292–306. Springer, 1998.

[48] N. Wetzler, M. J. H. Heule, and W. A. Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2013.