Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 286 (2012) 243–256

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Extracting a DPLL Algorithm

Andrew Lawrence [1,2] Ulrich Berger [3] Monika Seisenberger [4]

Department of Computer Science
Swansea University
Swansea, UK

Abstract

We formalize a completeness proof for the DPLL proof system and extract a DPLL 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 which shows that it is unsatisfiable. We use non-computational quantifiers to remove redundant computational content from the extracted program and improve its performance. The formalization is carried out in the Minlog system.

Keywords: DPLL, Program Extraction, Interactive Theorem Proving, SAT.

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 correct certifiable decision procedures. SAT-solvers are such decision procedures which attempt to solve the Boolean satisfiability problem. They are an important component in many contemporary verification tools. The majority of SAT-solvers used in an industrial context are based on the DPLL proof system. We have formalized a correctness and completeness proof of the DPLL proof system in the interactive theorem prover Minlog [20,2,4]. Using the program extraction facilities of Minlog we have been able to obtain a formally verified SAT-solving algorithm. When run on a CNF formula this algorithm produces a model satisfying the formula or a DPLL derivation showing its unsatisfiability. Computational redundancy was then removed from the algorithm by labelling certain universal quantifiers in the proof as non-computational,

[1] We would like to thank Invensys Rail UK for their support.

[2] Email: `csal@swansea.ac.uk`

[3] Email: `u.berger@swansea.ac.uk`

[4] Email: `m.seisenberger@swansea.ac.uk`

doi:10.1016/j.entcs.2012.08.016

an optimisation available in the Minlog system. The performance of the resulting program was tested with a number of pigeon hole formulae.

Program extraction aims at producing formally verified programs from a constructive proof. An example of early work with program extraction is that done in the Nuprl system [10]. Examples of program extraction in Minlog can be found in [5,3]. Other mature interactive theorem provers that support program extraction are Coq [6], which is based on the calculus of inductive constructions, and Isabelle [22,21], a generic theorem prover with extensions for many logics. More recently, other interactive theorem provers based on dependent types [PM80], such as Agda [9] and Epigram [18], have emerged which realise the Curry-Howard correspondence [12,13] and therefore can also be viewed as supporting program extraction.

Several attempts have been made to integrate an automatic theorem prover into Coq. Most of this work has made use of Coq's program extraction facilities to extract programs from proofs of decision procedures. A SAT-solver based on the DPLL algorithm has been formalized and its soundness and completeness are verified in Coq [15] and code has been extracted from the proof. Finally, the extracted system has been instantiated on the propositional fragment of Coq's logic creating a user friendly proof tactic. Binary decision diagrams have been formalized in Coq [26]. Then their correctness has been proven, and certified BDD algorithms have been extracted in Caml. 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 having been verified and integrated into the system. The DPLL algorithm has been formalized in [16]. The automatic theorem prover Metis [23] was formally verified inside Isabelle and is now used to reconstruct proofs from faster external procedures such as the ones used in Sledgehammer [8].

The approaches [15,16] to formalizing a DPLL SAT-solver in both Coq and Isabelle involve explicitly stating the algorithm to be verified. In contrast, we prove a theorem that just states that each formula in CNF is either unsatisfiable or has a model, and synthesise the program from the proof. In the long run we would like to integrate automatic verification techniques into Minlog. Extracting a SAT-solver in Minlog is one step towards our end goal.

2 Preliminaries

We begin with some basic definitions, following [15,16].

Definition 2.1

(i) A *literal* l is either a positive variable $+v$ or a negative variable $-v$, i.e. a variable v with a label $+$ or $-$ attached.

(ii) We define a bar operation which computes the *opposite* value of a literal as follows; $\overline{+v} = -v$, $\overline{-v} = +v$.

(iii) 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.

(iv) A *clause* C is a finite set of literals $\{l_1, \dots, l_k\}$, to be viewed as the disjunction of the literals.

(v) A formula is in *conjunctive normal form* (CNF) if it 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.

(vi) A *valuation* Γ is a finite set of literals $\{l_1, \dots, l_k\}$ to be viewed as the conjunction of the elements.

(vii) A valuation Γ is *consistent* ($\mathrm{cons}(\Gamma)$) if $\forall l\,(l \in \Gamma \to \bar{l} \notin \Gamma)$.

(viii) A *model* is a total function M which maps literals to booleans and satisfies the property $\forall l\,(M\ l \leftrightarrow \neg M\ \bar{l})$.

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* ($\mathrm{compatible}(\Gamma, \Delta)$) 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* ($\mathrm{incompatible}(\Gamma, \Delta)$).

Definition 2.2 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 $X, a := \{x \mid x \in X \vee x = a\}$ (adding a to the set X) and $X \setminus a := \{x \mid x \in X \wedge x \neq a\}$ (removing a from X).

Definition 2.3 [DPLL Proof System] The DPLL proof system consists of five rules:

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

$$\frac{}{\Gamma \vdash \Delta, \emptyset}\ \textbf{Conflict} \qquad \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}\ \textbf{Split}$$

3 Soundness and Completeness

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 doesn't carry computational content and a similar proof is carried out in [15,16]. 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".

Definition 3.1 The set of derivable 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	$l \in \Gamma \to \bar{l} \in C \to C \in \Delta \to \Gamma \vdash (\Delta \setminus C), (C \setminus \bar{l}) \to \Gamma \vdash \Delta$
Split	$\Gamma, l \vdash \Delta \to \ \Gamma, \bar{l} \vdash \Delta \to \Gamma \vdash \Delta$

Theorem 3.2 (Soundness) *If* $\Gamma \vdash \Delta$ *is derivable, 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:

$$\forall \Gamma, \Delta\, (\text{incompatible}(\Gamma, \Delta) \to \Gamma \vdash \Delta).$$

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)

$$\forall \Gamma, \Delta\, (\text{compatible}(\Gamma, \Delta) \vee \Gamma \vdash \Delta)$$

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

(i) 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 when it is absolutely necessary.

(ii) We perform an optimisation 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 formulas Δ, Θ,

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

The proof is by main induction on the measure

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

where

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

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

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

Case 1 $\Delta = \emptyset$.

Case 1.1 $\Theta = \emptyset$.
We define a model M by $M(l) = \text{True} \leftrightarrow l \in \Gamma$. Then $M \models \Gamma \land M \models \emptyset$ 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 $\mu((l, \Gamma); \Theta; \emptyset) < \mu(\Gamma; \emptyset; \Theta)$ since $|\Theta \setminus\setminus \mathrm{Var}(l, \Gamma)| < |\Theta \setminus\setminus \mathrm{Var}(\Gamma)|$. Furthermore, for the values (l, Γ), Θ, $\emptyset$ the hypotheses of the theorem are clearly satisfied. Hence the induction hypothesis for these values yields

$$(\Gamma, l \vdash \Theta) \lor \exists M(M \models \Gamma, l \land M \models \Theta) \tag{1}$$

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

$$(\Gamma, \bar{l} \vdash \Theta) \lor \exists M(M \models \Gamma, \bar{l} \land M \models \Theta) \tag{2}$$

The disjunctions (1) and (2) result in 4 cases: In the case that $\Gamma, l \vdash \Theta$ and $\Gamma, \bar{l} \vdash \Theta$ hold the **Split** rule is applied and we obtain $\Gamma \vdash \Theta$. In all of the 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 $\bar{l} \in \Gamma$, then $\Gamma \vdash (\Delta', \{l\}) \cup \Theta$ can be derived by applying (in backwards fashion) the **Red** rule followed by the **Conflict** rule. If $\bar{l} \notin \Gamma$, 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 $|(\Delta' \cup (\{l\}, \Theta)) \setminus\setminus \mathrm{Var}(\Gamma)| < |(\Delta' \cup \Theta) \setminus\setminus \mathrm{Var}(\Gamma, l)|$ 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 $\bar{l} \notin \Gamma$), 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 $\exists l\,(l \in C \wedge \bar{l} \in \Gamma) \vee \neg\exists l(l \in C \wedge \bar{l} \in \Gamma)$. This disjunction can be proven constructively, since the sets involved are finite.

Case 2.1.3.1 $\bar{l} \in \Gamma$ *for some* $l \in C$.
Then we have $\mu((\Gamma, l); (\Delta', C\backslash l); \Theta) < \mu(\Gamma; (\Delta', C); \Theta)$ since $\#(\Delta', C\backslash l) < \#(\Delta', C)$. The hypotheses of the theorem are satisfied for the chosen values. Hence we obtain, by induction hypothesis, $(\Gamma \vdash (\Delta', (C\backslash l')) \cup \Theta) \vee \exists M(M \models \Gamma \wedge M \models (\Delta', (C\backslash l')) \cup \Theta)$. In the case that $\Gamma \vdash (\Delta', (C\backslash l')) \cup \Theta$ 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 $\neg\exists l\,(l \in C \wedge \bar{l} \in \Gamma)$.
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 the hypotheses of the theorem are satisfied, we obtain $\Gamma \vdash (\Delta', (C\backslash l)) \cup \Theta$ or $\exists M(M \models \Gamma \wedge M \models (\Delta', (C\backslash l)) \cup \Theta$. In the first case we apply the **Elim** rule, in the second case we use the model provided.
□

4 Program Extraction

Program extraction in Minlog is based on modified realizability [14]. 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 [25].

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 $\exists_{\mathrm{nc}}$. 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 quantifiers are:

$$\tau(\forall x^\rho A) = \rho \to \tau(A)$$
$$\tau(\exists x^\rho A) = \rho \times \tau(A)$$

$$f\,\mathbf{r}\,\forall x^\rho A = \forall x^\rho (f(x)\,\mathbf{r}\,A)$$
$$(a, y)\,\mathbf{r}\,\exists x^\rho A = a\,\mathbf{r}\,A[y/x]$$

Here, the notation x^ρ means x has type ρ. For the nc-quantifiers the realizers do not depend on the quantified variables:

$$\tau(\forall_{\mathrm{nc}} x^\rho A) = \tau(A)$$
$$\tau(\exists_{\mathrm{nc}} x^\rho A) = \tau(A)$$

$$a\,\mathbf{r}\,\forall x^\rho A = \forall x^\rho (a\,\mathbf{r}\,A)$$
$$a\,\mathbf{r}\,\exists x^\rho A = \exists x^\rho (a\,\mathbf{r}\,A)$$

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 [24].

5 The Extracted Program

The extracted program has a similar structure as the proof. It takes a formula Δ in CNF as input and produces either a model of Δ or a derivation of unsatisfiability. The Minlog system automatically generates algebraic data-types for realizers of inductively defined predicates. For instance, the inductive definition of derivable is extracted into a data structure representing derivations in the DPLL proof system and induction proofs are translated into structurally recursive procedures. The control structure of the program closely follows the inductions and case distinctions we performed in the proof. The measure induction at the start of the proof turns into guarded recursion using the same measure in the program. The separate lemmas which we have used in the proof are invoked like procedures. The main benefit of the extraction process is that the code is generated automatically and hence no errors occur during coding. Furthermore, a formal correctness proof for the extracted program is automatically generated as well.

The main body of the program as extracted by Minlog is as follows:

```
cgRec([cs3](Rec list cla=>list cla=>valu=>
                (list cla=>list
                 cla=>valu=>algsuccess)=>algsuccess)
                cs3 cbase cstep)
```

The first thing we see in the extracted program is a constant `cgRec` representing the guarded recursion. The latter corresponds to the definition:

$$\begin{aligned}
&(\mathrm{GRecGuard}\,\rho_1\,\rho_2\,\rho_3\,\tau)\mu\,z_1\,z_2\,z_3\,G\,t = f\,z_1\,z_2\,z_3\,t \text{ where}\\
&\quad f : \rho_1 \Rightarrow \rho_2 \Rightarrow \rho_3 \Rightarrow \mathrm{Boole} \Rightarrow \tau\\
&\quad f\,x_1\,x_2\,x_3\,\mathrm{True} = G\,x_1\,x_2\,x_3\,([y_1,y_2,y_3].f\,y_1\,y_2\,y_3(\mu\,y_1\,y_2\,y_3 < \mu\,x_1\,x_2\,x_3))\\
&\quad f\,x_1\,x_2\,x_3\,\mathrm{False} = \mathrm{Inhab}^{5}
\end{aligned}$$

Similarly, it is possible to write the operation Rec $list\,\rho \Rightarrow \tau$ using a function $f : \mathrm{list}\,\rho \Rightarrow \tau$ which gives a more intuitive view of its behaviour.

$$\begin{aligned}
&(\mathrm{Rec}\ \mathrm{list}\,\rho \Rightarrow \tau)\,ys\,z\,G = f\,ys \quad \text{where}\\
&\quad f\,\mathrm{Nil} = z\\
&\quad f\,(x :: xs) = G\,x\,xs\,(f\,xs)
\end{aligned}$$

6 Execution of the Extracted Program

We have extracted two programs, one from the proof above ($\forall$ solver) and one from the proof involving nc-quantifiers ($\forall_{\mathrm{nc}}$ solver). The nc-quantifiers were inserted at strategic places in the main proof, namely in inductive definitions and in the lemmas which are outside of the main proof. 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 [11]. 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] $\mathbf{PHP}(n,m) := \{\{l_{i,1}, \ldots, l_{i,m}\} | 1 \le i \le n\} \cup \{\{\overline{l_{i,k}}, \overline{l_{j,k}}\} | 1 \le i < j \le n, 1 \le k \le m\}$

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 $n \le m$. 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:

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

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

[5] Inhab is a dummy variable indicating a use of the axiom scheme efq: $\bot \to A$.

in the function M : literals $\to \mathbb{B}$ where $\mathrm{M}(l) = \mathrm{True}$ iff $l \in \{l_{12}, \overline{l_{11}}, l_{21}, \overline{l_{22}}\}$.

6.1 Comparison of Program Performance with and without nc-Quantifiers

We have performed two comparisons between our extracted solvers with and without the $\forall_{\mathrm{nc}}$ quantifiers. The first comparison is on unsatisfiable pigeon hole formulae $\mathbf{PHP}(n+1,n)$ which will demonstrate the relative efficiency of the solvers in constructing a derivation. The second comparison is on satisfiable formulae $\mathbf{PHP}(n,n)$. The programs have been run on the term level in the Minlog system using the built in term rewriting system. Therefore, the running times can not be expected to be competitive, but it is interesting to observe the relative difference between the two solvers.

Solver	$\mathbf{PHP}(2,1)$	$\mathbf{PHP}(3,2)$	$\mathbf{PHP}(4,3)$	$\mathbf{PHP}(5,4)$	$\mathbf{PHP}(6,5)$
$\forall$	<1 Sec	1.17	33.62	13:54	5:35:41
$\forall_{nc}$	<1 Sec	<1 Sec	11.61	2:41	37:25.27

Solver	$\mathbf{PHP}(2,2)$	$\mathbf{PHP}(3,3)$	$\mathbf{PHP}(4,4)$	$\mathbf{PHP}(5,5)$	$\mathbf{PHP}(6,6)$
$\forall$	<1 Sec	<1 Sec	5.45	26.09	1:34.11
$\forall_{nc}$	<1 Sec	<1 Sec	5.25	25.03	1:24.88

These results demonstrate that the solver extracted from the proof containing the $\forall_{\mathrm{nc}}$ quantifiers is significantly faster on unsatisfiable formulae than the solver extracted from the original proof. This is due to the number of non-computational quantifiers added to the definition of derivable. When applied to the pigeon hole formula $\mathbf{PHP}(6,5)$ the $\forall$ solver takes 5 and a half hours where as the $\forall_{\mathrm{nc}}$ solvers takes only 37 minutes.

7 Conclusion

In this paper we presented a new application of program extraction to decision procedures. The DPLL proof system was formalized and then a constructive proof of completeness was performed from which we extracted a program. The extracted program attempts to show the (un)satisfiability of a propositional formula in conjunctive normal form (CNF). If the CNF formula is satisfiable it produces a model of the formula; otherwise it produces a derivation showing the unsatisfiability of the formula. We strategically inserted $\forall_{\mathrm{nc}}$ quantifiers into the proof to reduce the complexity of the extracted program and increase its performance. The performance of the original solver was then compared with this improved solver using pigeon hole formulae.

Overall, the case study 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 'looses 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. The big challenge, however, will be to extract from proofs also qualitative information about quantitative aspects of the extracted programs (e.g. computational complexity), for example, by combining approaches to implicit complexity with program extraction (see e.g. [1]).

Future Work

In order to apply the solver practically we need to translate the extracted Minlog term into a functional programming language such as Scheme or Haskell. Currently a translation mechanism from Minlog into Scheme is available, however it does not extract inductive definitions and general recursion. We would like to extend this translation to cover these definitions. Having our DPLL solver as a Haskell program would allow us to observe how lazy evaluation affects performance.

We further want to prove the equivalence of the DPLL proof system and the tree resolution proof system. This will allow us to extract a resolution solver based on the DPLL algorithm.

Extracting efficient data structures for our DPLL solver has the potential of greatly improving the efficiency of the solver and will provide another interesting example of program extraction. Since Haskell is based on lazy data structures such as tries, we would like to know whether a solver in Haskell would make use of these structures and gain something in efficiency. The solver would also benefit from a heuristics to select a splitting literal, currently it just selects the first literal in the formula Θ.

Our current solver could be further improved by adding optimisation techniques such as clause learning and conflict analysis [7,19,17]. This would require a modification of the current completeness theorem so that the derivation is performed with respect to an added implication graph.

References

[1] Aehlig, K., U. Berger, M. Hofmann and H. Schwichtenberg, *An arithmetic for non-size-increasing polynomial-time computation*, Theoretical Computer Science **318** (2004), pp. 3–27.

[2] Benl, H., 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* (1998).

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

[4] Berger, U., 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, *CALCO 2011*, Lecture Notes in Computer Science **6859** (2011), pp. 393–399.

[5] Berger, U., H. Schwichtenberg and M. Seisenberger, *The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction*, Journal of Automated Reasoning **26** (2001), pp. 205–221.

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

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

[8] Böhme, S. and T. Nipkow, *Sledgehammer: Judgement day*, in: J. Giesl and R. Hähnle, editors, *Automated Reasoning*, Lecture Notes in Computer Science **6173** (2010), pp. 107–121.

[9] Bove, A., P. Dybjer and U. Norell, *A brief overview of Agda — A Functional Language with Dependent Types*, in: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science **5674** (2009).

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

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

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

[13] Howard, W. A., *The formulae-as-types notion of construction*, To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (1980).

[14] Kreisel, G., *Interpretation of analysis by means of constructive functionals of finite types*, Constructivity in Mathematics (1959), pp. 101–128.

[15] Lescuyer, S. and S. Conchon, *A Reflexive Formalization of a SAT Solver in Coq*, in: O. A. Mohamed, C. Muñoz and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, Lecture Notes in Computer Science **5170**, 2008.

[16] Marić, F. and P. Janičić, *Formal correctness proof for DPLL procedure*, Informatica **21** (2010), pp. 57–78.

[17] Marques-Silva, J. P. and K. A. Sakallah, *GRASP: A Search Algorithm for Propositional Satisfiability*, IEEE Trans. Computers **48** (1999), pp. 506–521.

[18] McBride, C., *Epigram: Practical Programming with Dependent Types*, Lecture Notes in Computer Science **3622** (2004), pp. 130–170.

[19] Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, *Chaff: Engineering an efficient SAT solver*, in: *Annual ACM IEEE Design Automation Conference* (2001), pp. 530–535.

[20] Minlog Proof Assistant, http://www.minlog-system.de.

[21] Nipkow, T., L. C. Paulson and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science **2283** (1999).

[22] Paulson, L. C., *Isabelle: A Generic Theorem Prover*, Lecture Notes in Computer Science **828** (1994).

[23] Paulson, L. C. and K. W. Susanto, *Source-Level Proof Reconstruction for Interactive Theorem Proving*, in: *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'07 (2007), pp. 232–245.

[24] Ratiu, D. and H. Schwichtenberg, *Decorating proofs*, in: S. Feferman and W. Sieg, editors, *Proofs, Categories and Computations. Essays in honor of Grigori Mints* (2010), pp. 171–188.

[25] Schwichtenberg, H. and S. Wainer, "Proofs and Computations," Cambridge University Press, 2012, 1st edition.

[26] Verma, K. N., J. Goubault-Larrecq, S. Parsad and S. Arun-Kumar, *Reflecting BDDs in Coq*, in: *ASIAN*, Lecture Notes in Computer Science **1961** (2000), pp. 162–181.

Appendix: The Extracted Program

The appendix lists the full extracted program as it was produced by Minlog.

```
Main Program =
cgRec ([cs3](Rec list cla=>list cla=>valu=> (list cla=>list
cla=>valu=>algsuccess)=>algsuccess) cs3 cbase cstep)

cgRec =
[(list cla=>list cla=>valu=>(list cla=>list
cla=>valu=>algsuccess)=>algsuccess)_0,cs1,cs2,val3]
(list cla=>list cla=>valu=>(list cla=>list cla=>valu=>algsuccess)=>algsuccess)_0
cs1
cs2 val3
(
 [cs4,cs5,val6]
 (GRecGuard list cla list cla valu algsuccess)
 (
 [cs7,cs8,val9]
 Lh(toLit cs7)+Lh(toLit cs8)+Lh(setminus(varSetClaList(cs7++cs8))(varv val9))
 )
 cs4
 cs5
 val6
 (list cla=>list cla=>valu=>(list cla=>list cla=>valu=>algsuccess)=>algsuccess)_0
 (Lh(toLit cs4)+Lh(toLit cs5)+Lh(setminus(varSetClaList(cs4++cs5))(varv
val6))<Lh(toLit cs1)+Lh(toLit cs2)+Lh(setminus(varSetClaList(cs1++cs2))(varv
val3)))
)

cbase =
[cs0,val1,(list cla=>list cla=>valu=>algsuccess)_2]
        [if cs0
            (cModelCase val1)
            (
             [c3,cs4]cSplitCase cs0 val1 c3 cs4(list cla=>list cla=>valu=>algsuccess)_2
            )
        ]

cstep =
[c0,cs1,(list cla=>valu=>(list cla=>list
cla=>valu=>algsuccess)=>algsuccess)_2,cs3,val4,(list cla=>list
cla=>valu=>algsuccess)_5]
[if (vcIntersection val4 c0=(Nil lit))
    [if c0
        (
        [ls6]
        [if ls6
            (cConflictCase cs1 cs3 val4)
            (
             [l7,ls8]
             [if ls8
                 [if (memlv(opposite l7)val4)
                     (cElimConflictCase l7 cs1 cs3 val4)
                     (cUnitCase l7 c0 ls6 ls8 cs1 cs3 val4(list cla=>list
cla=>valu=>algsuccess)_5)
                ]
                (
                 [l9,ls10]
                 [if (cindmemlem(l7::l9::ls10)val4)
                     (cReduceCase cs1 c0 cs3 val4 ls6 l7 ls8 l9 ls10(list
cla=>list cla=>valu=>algsuccess)_5)
                     (cCleanCase cs1 c0 cs3 val4 ls6 l7 ls8 l9 ls10(list
cla=>valu=>(list cla=>list
cla=>valu=>algsuccess)=>algsuccess)_2(list cla=>list
cla=>valu=>algsuccess)_5)
                 ]
               )
          ]
         )
     ]
    )
  ]
  (cElimCase c0 cs1 cs3 val4(list cla=>list cla=>valu=>algsuccess)_5)
]

cConflictCase =
[cs0,cs1,val2]csuccessZero cderivableZero
```

```
cElimConflictCase =
[l0,cs1,cs2,val3]csuccessZero(cderivableThree(CC l0:)(opposite l0)cderivableZero

UnitCase =
[l0,c1,ls2,ls3,cs4,cs5,val6,(list cla=>list cla=>valu=>algsuccess)_7]
[if ((list cla=>list cla=>valu=>algsuccess)_7(remccl(CC l0:)cs4++remccl(CC
l0:)cs5)(Nil cla)(conclv l0 val6))
    ([algderivable8]csuccessZero(cderivableTwo l0 algderivable8))
    csuccessOne
]

cReduceCase =
[cs0,c1,cs2,val3,ls4,l5,ls6,l7,ls8,(list cla=>list cla=>valu=>algsuccess)_9,l10]
[if ( (list cla=>list cla=>valu=>algsuccess)_9
       ([if (l10=l5)
             (CC(l7::ls8))
             (conclc l5 [if (l10=l7)
                                 (CC ls8)
                                 (conclc l7(remlc l10(CC ls8)))
                             ]
             )
          ]::remccl(CC(l5::l7::ls8))cs0)
       cs2
        val3
   )
   ([algderivable11]csuccessZero(cderivableThree(CC(l5::l7::ls8))(opposite
l10)algderivable11))
   csuccessOne
]

cCleanCase =
[cs0,c1,cs2,val3,ls4,l5,ls6,l7,ls8,(list cla=>valu=>(list cla=>list
cla=>valu=>algsuccess)=>algsuccess)_9,(list cla=>list cla=>valu=>algsuccess)_10]
[if ((list cla=>valu=>(list cla=>list
cla=>valu=>algsuccess)=>algsuccess)_9(CC(l5::l7::ls8)::cs2)val3(list cla=>list
cla=>valu=>algsuccess)_10)
    ([algderivable11]csuccessZero(cderivableLemma(CF(cs0++(CC(l5::l7::ls8)::cs2)))
algderivable11))
    csuccessOne
]

cElimCase =
[c0,cs1,cs2,val3,(list cla=>list cla=>valu=>algsuccess)_4]
[if ((list cla=>list cla=>valu=>algsuccess)_4(remccl c0 cs1)cs2 val3)
    ([algderivable5]csuccessZero(cderivableOne c0(cmemlemmaOne val3
c0)algderivable5))
    csuccessOne
]

cSplitCase =
[cs0,val1,c2,cs3,(list cla=>list cla=>valu=>algsuccess)_4]
[if ((list cla=>list cla=>valu=>algsuccess)_4(c2::cs3)(Nil cla)(conclv(cSelectLemma
c2 cs3)val1))
    (
     [algderivable5]
     [if ((list cla=>list cla=>valu=>algsuccess)_4(c2::cs3)(Nil
cla)(conclv(opposite(cSelectLemma c2 cs3))val1))
         ([algderivable6]csuccessZero(cderivableFour(cSelectLemma c2
cs3)algderivable5 algderivable6))
         csuccessOne
     ]
    )
    csuccessOne
]

cModelCase =
[val0]csuccessOne([l1]memlv l1 val0)

cSelectLemma = [c0,cs1]SelectLit c0

cmemlemmaOne = [val0,c1][if (vcIntersection val0 c1) (Inhab lit) ([l2,ls3]l2)]

cderivableLemma =
[f0,algderivable1]
(Rec algderivable=>algderivable)
algderivable1
cderivableZero
([c2,l3,algderivable4]cderivableOne c2 l3)
([l2,algderivable3]cderivableTwo l2)
([c2,l3,algderivable4]cderivableThree c2 l3)
```

```
([l2,algderivable3,algderivable4]cderivableFour l2)

cemptyval =
[algderivable0]
(Rec algderivable=>algderivable)
algderivable0
cderivableZero
([c1,l2,algderivable3]cderivableOne c1 l2)
([l1,algderivable2]cderivableTwo l1)
([c1,l2,algderivable3]cderivableThree c1 l2)
([l1,algderivable2,algderivable3]cderivableFour l1)
cemptyvalTwo = [val0,(valu=>algderivable)_1](valu=>algderivable)_1 val0

cDeriveRemoveLemma =
[l0,l1,l2,ls3,cs4,cs5,algderivable6]
(Rec algderivable=>algderivable)
algderivable6
cderivableZero
([c7,l8,algderivable9] cderivableOne c7 l8)
([l7,algderivable8] cderivableTwo l7)
([c7,l8,algderivable9]cderivableThree c7 l8)
([l7,algderivable8,algderivable9]cderivableFour l7)

cindmemlem =
[ls0]
(Rec list lit=>valu=>algindmem)
ls0
([val2]cindmemOne)
(
 [l2,ls3,(valu=>algindmem)_4,val5]
[if ((valu=>algindmem)_4 val5)
    (
     [l6]
     [if (memlv(opposite l2)val5)
         (cindmemZero l2)
         (cindmemZero l6)
     ]
    )
[if (memlv(opposite l2)val5)
    (cindmemZero l2)
    cindmemOne
]
]
)
```