SAT solver of Howe & King as a logic program

Włodzimierz Drabent*

June 6, 2011

Howe and King [HK11b, HK11a] presented a SAT solver which is an elegant and concise Prolog program of 22 lines. It is not a logic program, as it includes some extra-logical features of Prolog; it was constructed as an implementation of the DPLL algorithm, using logical variables and coroutining. Here we construct this program using the Logic + Control principle of Kowalski. We show how the program can be constructed by adding control to an initial logic program. We discuss correctness, completeness, termination and non-floundering of the program. In particular, we outline a formal proof of correctness and completeness of the underlying logic program. The presented proof methods may be of separate interest.

Sections 1 and 2 present an introductory SAT solver, which is a simple logic program. Section 3 presents methods of proving correctness and completeness of logic programs, and applies them to the program from the previous section. Section 4 presents the actual construction of the program of [HK11b]. To facilitate the intended control flow, the program from Section 2 is first converted into a more sophisticated logic program. Then the control is added by modifying the Prolog selection rule and pruning some redundant fragments of the search space. Properties of the constructed program are discussed in Section 5. Section 3 and the fragments of Section 5 concerning correctness and completeness may be skipped at the first reading.

For examples, further explanations, and the text of the program see [HK11b, HK11a].

*Institute of Computer Science, Polish Academy of Sciences, and IDA, Linköpings universitet, Sweden; `drabent` *at* `ipipan` *dot* `waw` *dot* `pl`.

1 Representation of propositional formulae

Here we describe how formulae in CNF are represented in the program of [HK11b]. Propositional variables are represented as logical variables. A literal of a clause is represented as a pair of a variable and of `true` or `false`; a positive literal, say x, as `true-X` and a negative one, say $\neg x$, as `false-X`. A clause is represented as a list of (representations of) literals, and a conjunction of clauses as a list of their representations. For instance a formula $(x \vee \neg y \vee z) \wedge (\neg x \vee v)$ is represented as `[[true-X,false-Y,true-Z],[false-X,true-V]]`. Thus a clause is satisfiable iff its representation has an instance containing a pair of the form t-t, i.e. `false-false` or `true-true`. A formula in CNF is satisfiable iff its representation has an instance whose each element (is a list which) contains a t-t.

To avoid confusion, the clauses of programs will be called *rules*. We will use a small font to mark intermediary versions of rules, not included in the final program.

2 Satisfiability – logic program

We first construct a logic program P_1 checking satisfiability of CNF formulae represented as above. We will say "formula f" for a formula in CNF represented as a term f. We use the predicate names from [HK11b] (which may be not adequate for our declarative view of the program).

Let

> L_1 be the set of those lists of ground terms that contain an element of the form t-t,
> L_2 be the set of lists, whose all elements are from L_1.

We construct a program defining L_2. Following [HK11b], the predicate defining this set will be called *problem_setup*. Thus, for a formula f, a query *problem_setup*(f) will fail for an unsatisfiable f and succeed when f is satisfiable. In this way the predicate checks the satisfiability of f. Moreover, the computed answer substitutions provide bindings of truth values to variables of f, under which f is true.

To represent the binding as a list of truth values, we introduce the main predicate *sat*/2 of the program. It defines the relation in which the first argument is from L_2 and the second argument is a list of truth values (i.e.

of `true` or `false`).[1] The intended query is $sat(f, l)$ where l is the list of variables of a formula f. Such query succeeds iff f is satisfiable. At success l is instantiated to a list of truth values representing a valuation satisfying f. Predicate *sat* is defined by an obvious rule

$$sat(Clauses, Vars) \leftarrow problem\_setup(Clauses), elim\_var(Vars).$$

where *elim_var* defines the set of lists of truth values. We follow [HK11b]:

$$\begin{array}{l} elim\_var([\,]). \\ elim\_var([Var|Vars]) \leftarrow elim\_var(Vars), assign(Var). \\ assign(true). \\ assign(false). \end{array}$$

It remains to construct a definition of predicate *problem_setup*. We do this in a rather obvious way, using a predicate *clause_setup*, which defines the set L_1.

$$\begin{array}{l} problem\_setup([\,]). \\ problem\_setup([Clause|Clauses]) \leftarrow \\ \quad clause\_setup(Clause), \\ \quad problem\_setup(Clauses). \\ clause\_setup([Pol\text{-}Var|Pairs]) \leftarrow Pol = Var. \\ clause\_setup([Pol\text{-}Var|Pairs]) \leftarrow clause\_setup(Pairs). \end{array}$$

This completes the construction of the logic program P_1.[2]

3 Correctness considerations

It may be obvious for the reader that the constructed program indeed defines the required relations. However we discuss now how to formally prove this fact. The reader may prefer to skip this section at the first reading, and proceed to Section 4. We employ the approach of [DM05, Chapters 3.1 and 3.3]. We present simpler (and less general) versions of the correctness and

[1] Note that the arguments are not related, the relation is the Cartesian product of L_2 and the set of truth value lists.

[2] Following the style of [HK11b], we used a Prolog built-in `=`, which defines the relation of term equality. Formally, it should be assumed that P_1 contains also a unary rule $=(X, X)$, defining the built-in.

completeness criteria from [DM05]. We consider definite clause programs; for programs with negation see [DM05].

We provided a **specification** for the program P_1: for each its predicate a corresponding relation has been given; the predicate should define this relation. Below we introduce some notions which depend on the considered specification. We will however often skip a phrase "with respect to specification ...", as our specification is fixed for this section.

Let us call a ground atom $p(t_1, \ldots, t_n)$ **specified** if the tuple $(t_1, \ldots, t_n)$ is in the relation corresponding to p. So in our case, the specified atoms are of the form $sat(t, u)$, $elim\_var(u)$, $problem\_setup(t)$, $clause\_setup(s)$, $x = x$, $assign(true)$, $assign(false)$, where $s \in L_1$, $t \in L_2$, u is a list whose elements are `true` or `false`, and x is an arbitrary ground term. Let S denote the set of specified atoms. S can be seen as a Herbrand interpretation representing the specification.

In imperative programming, correctness usually means that the program results are as specified. In logic programming, due to its non-deterministic nature, we have actually two issues: **correctness** (all the results are compatible with the specification) and **completeness** (all the results required by the specification are produced). In other words, correctness means that the relation defined by the program is a subset of the specified one, and completeness means inclusion in the opposite direction. In terms of specified atoms and the least Herbrand model M_P of a program P we have: P is correct iff $M_P \subseteq S$; it is complete iff $M_P \supseteq S$.

To show that a program is correct it is sufficient to check that [Cla79, DM05]

> for each ground instance $H \leftarrow B_1, \ldots, B_n$ of a rule of the program, if $B_1, \ldots, B_n$ are specified atoms then H is a specified atom.

The reader is encouraged to check that P_1 satisfies this condition.[3] Thus P_1 is correct.

Our criterion for proving completeness is less general. It will show that for a given query (or a class of queries) the program will produce all the answers required by the specification. Let us say that a program P is *complete for* an atomic query A if, for any specified ground instance $A\theta$ of A, $A\theta$ is in

[3] For instance consider the last rule of the program, and its arbitrary ground instance $clause\_setup([p\text{-}v|s]) \leftarrow clause\_setup(s)$. If $clause\_setup(s)$ is specified then $s \in L_1$, hence $[p\text{-}v|s] \in L_1$ and $clause\_setup([p\text{-}v|s])$ is specified.

M_P. Generally, the program is **complete for a query** $Q = A_1, \ldots, A_n$ if, for any ground instance $Q\theta$ of Q where $A_1\theta, \ldots, A_n\theta$ are specified atoms, $A_1\theta, \ldots, A_n\theta \in M_P$.

> A ground atom H is called **covered** [Sha83] if it is the head of a ground instance $H \leftarrow B_1, \ldots, B_n$ of a rule of the program, such that all the atoms $B_1, \ldots, B_n$ are specified.
>
> If all the specified atoms are covered, and there exists a finite SLD-tree for a query Q then the program is complete for Q [DM05].[4]

Let us apply this criterion to our program. Consider a query $Q = sat(t, l)$ where t, l are (possibly non-ground) lists of a fixed length (i.e. terms of the form $[t_1, \ldots, t_n]$). The intended queries to the program are of this form. For such queries the program terminates, under any selection rule.[5] Thus there exists a finite SLD-tree for each such query. The reader is encouraged to check that each specified atom is covered.[6] Hence the program is complete for the intended initial queries, and it terminates for such queries.[7]

As a final comment, we point out that our specification describes exactly the least Herbrand model M_{P_1} of the program. This is often not the case, M_P is specified approximately, by giving separate specifications S_{compl}, S_{corr} for completeness and correctness; it is required that $S_{compl} \subseteq M_P \subseteq S_{corr}$. The specifications describe, respectively, which atoms have to be computed, and which are allowed to be computed. A standard example is the usual definition of *append*, where it is difficult (and unnecessary) to specify the exact defined relation [DM05].[8]

[4] The notion of completeness used here is weaker than that of [DM05]. Also, this sufficient condition for completeness is, strictly speaking, not an instance of that given in [DM05]. Its correctness follows from [Dra99, Prop. 5.1.1].

[5] Informally: The predicates are invoked with fixed length lists as arguments, each recursive call employs a shorter list. Formally: The program is recurrent [Apt97] under a suitable level mapping (based on the length of the lists from L_1, and the sum of the lengths of the element lists for the lists from L_2).

[6] For instance consider a specified atom $A = problem\_setup(t)$. Thus t is a ground list of elements from L_1. If t is nonempty then $t = [s|t']$, where $s \in L_1$, $t' \in L_2$. Thus a ground instance $A \leftarrow clause\_setup(s), problem\_setup(t')$ of a clause of P_1 has all its body atoms specified, so A is covered. If t is empty then A is covered as it is the head of the rule $problem\_setup([\,])$.

[7] Moreover, this reasoning applies to any atom Q whose arguments are finite length lists. Each specified atom is an instance of such query, hence the program is complete.

[8] Notice that in our case we are not interested in any answers where the argument is

4 Adding control

In this section we modify the program P_1 to improve its efficiency. To be able to influence its control in the intended way, we first construct a more sophisticated logic program P_2. We modify the definition of *clause_setup*/1, introducing some new predicates.

Program P_1 performs inefficient search by means of backtracking. We improve it by delaying unification of pairs *Pol*-*Var* in *clause_setup*. The idea is to perform such unification if *Var* is the only unbound variable of the clause.[9] Otherwise, *clause_setup* is to be delayed until one of the first two variables of the clause is bound to `true` or `false`. The actual binding may be performed by other invocation of *clause_setup*, or by *elim_var*.

This idea will be implemented by separating two cases; the clause has one literal, or it has more literals. For efficiency reasons we want to distinguish these two cases by means of indexing the main symbol of the first argument. So the argument should be the tail of the list. (The main symbol is [ ] for a one element list, and [|] for longer lists.) We redefine *clause_setup*, introducing an auxiliary predicate *set_watch*/3. It defines the same set L_1 as *clause_setup* does, but a clause $[Pol\text{-}Var|Pairs]$ is represented as three arguments $Pairs, Var, Pol$ of *set_watch*.

$$clause\_setup([Pol\text{-}Var|Pairs]) \leftarrow set\_watch(Pairs, Var, Pol).$$

$$set\_watch([\,], Var, Pol) \leftarrow Var = Pol.$$
$$set\_watch([Pol2\text{-}Var2|Pairs], Var1, Pol1) \leftarrow$$
$$\quad watch(Var1, Pol1, Var2, Pol2, Pairs).$$

The first rule of *set_watch* expresses the fact that a clause $[Pol\text{-}Var]$ is in L_1 iff $Pol = Var$; the clause is represented as three arguments $[\,], Var, Pol$ of *set_watch*. We now explain the second rule.

In *set_watch*, delaying is to be controlled by the variables of the first two literals of the clause; so the variables should be separate arguments of a predicate. Thus we introduce an auxiliary predicate *watch*/5. It defines the set of

a list, or a list of lists, with an element which is not a pair of truth values (for instance an answer like $clause\_setup([a, true\text{-}true])$). So it may be considered natural to require completeness w.r.t. a specification which instead of L_1 uses the set L_1' of those elements of L_1 which are lists of pairs of truth values, and instead of L_2 uses the set of those lists whose elements are from L_1'.

[9] The clause which is (represented as) the argument of *clause_setup* in the rule for *problem_setup*.

lists from L_1 of the length > 1; however a list $[Pol1\text{-}Var1, Pol2\text{-}Var2 \mid Pairs]$ is represented as the five arguments $Var1, Pol1, Var2, Pol2, Pairs$ of *watch*. Executing $watch(Var1, Pol1, Var2, Pol2, Pairs)$ is to be delayed until $Var1$ or $Var2$ is bound. This is achieved by a declaration

```
:- block watch(-, ?, -, ?, ?).
```

A list of length > 1 is in L_1 iff its first element is of the form $t\text{-}t$ or its tail is in L_1. So a definition of *watch* could be

$watch(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1.$
$watch(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow set\_watch(Pairs, Var2, Pol2).$

However the first rule may bind $Var1$, which we want to avoid. We know that *watch* will be selected with its first or third argument bound. Thus we introduce an auxiliary predicate *update_watch*/5. Declaratively, it defines the same relation as *watch*, it can be defined by the two rules above (with *watch* replaced by *update_watch*). The intention is to call it with the first argument bound. Predicate *watch* can be defined by a rule with the head

$$watch(Var1, Pol1, Var2, Pol2, Pairs)$$

and the body

$$update\_watch(Var1, Pol1, Var2, Pol2, Pairs)$$

or

$$update\_watch(Var2, Pol2, Var1, Pol1, Pairs).$$

Both define the required relation. We want to choose the body dynamically, to assure that *update_watch* is called with its first argument bound. In other words, the logic program P_2 contains two rules for *watch*; the control has to choose one of them and abandon the other. We do not see a way of doing this by adding control to a logic program. So we use extra-logical features of Prolog.

$$\begin{array}{l} watch(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow \\ \quad nonvar(Var1) \rightarrow \\ \quad\quad update\_watch(Var1, Pol1, Var2, Pol2, Pairs); \\ \quad\quad update\_watch(Var2, Pol2, Var1, Pol1, Pairs). \end{array}$$

Notice that the program containing such rule is not a logic program, due to the built-in *nonvar* and the if-then-else construct.

Our logic program contains the following rules defining *update_watch*.

$update\_watch(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow Var1 = Pol1.$
$update\_watch(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow set\_watch(Pairs, Var2, Pol2).$

If the first argument of the initial query $sat(f, l)$ is a (representation of a) propositional formula then *update_watch* is called with its second argument `true` or `false`. As the first argument is bound, the unification $Var1 = Pol1$ (in the first rule above) does not bind any variable. Thus if the first rule succeeds then no variables are bound and there is no point in invoking the second rule;[10] the search space should be pruned accordingly. We do this by converting the two rules into

$$update\_watch(Var1, Pol1, Var2, Pol2, Pairs) \leftarrow$$
$$Var1 = Pol1 \rightarrow true; \quad set\_watch(Pairs, Var2, Pol2).$$

The unification $Var1 = Pol1$ can be replaced by `==` of Prolog, because – as explained above – it works here only as a test. The program in [HK11b] uses `==`.

This completes the construction of the Prolog program from [HK11b]. The program consists of the rules for predicates *sat*, *elim_vars*, *assign*, *problem_setup*, *clause_setup*, *set_watch*, *watch*, *update_watch*, written with a normal size font. It differs from its declarative version P_2 by the rules for *watch* and *update_watch*. The control has been added to P_2 by (1) changing the default Prolog selection rule (by the `block` declaration), and (2) pruning some redundant parts of the search space (by the if-then-else constructs in the rules for *watch* and *update_watch*).

5 Discussion

The final Prolog program can be seen as the logic program P_2 with a specific control. The default selection rule of Prolog is modified by a `block` declaration. The search space is pruned by removing some redundant parts of SLD-trees. The pruning could be done by employing the cut; here a more elegant solution was used, with the if-then-else construct of Prolog.

P_2 differs from the logic program P_1 by the fragment related to predicate *clause_setup*, defining the set L_1. We divided the set of lists L_1 into the

[10] Because the success of the first rule produces the most general answer for $update\_watch(\ldots)$, which subsumes any other answer.

the subset $L_{1,1}$ of those of length 1, and $L_{1,2}$ of those of length > 1; the two sets are defined separately. To facilitate the intended control flow we introduced a few predicates defining the same set $L_{1,2}$, however they use different representation of the elements of $L_{1,2}$.

We gave a specification for both programs, providing for each predicate the relation it defines. So the specified atoms are those of the form $sat(t, u)$, $elim\_var(u)$, $assign(true)$, $assign(false)$, $problem\_setup(t)$, $clause\_setup(s)$, $set\_watch(s_0, v, p)$, $watch(v_1, p_1, v_2, p_2, s_0)$, $update\_watch(v_1, p_1, v_2, p_2, s_0)$, $x = x$, where $t \in L_2$, u is a list of truth values, $s \in L_1$, $[p\text{-}v|s_0] \in L_1$, $[p_1\text{-}v_1, p_2\text{-}v_2|s_0] \in L_1$, and x is an arbitrary ground term. Program P_2 is correct with respect to the specification; this can be proved by applying the correctness criterion from Section 3. The reader is encouraged to construct the proof.

We do not include here the details of the correctness and completeness proofs for P_1 and P_2, as they are simple and rather obvious (conf. Footnote 3 presenting one of more sophisticated cases). Let us remark that an error in one of the rules in an earlier draft has been found while constructing a correctness proof, as the correctness criterion was violated. This illustrates practicality of the proof methods.

The reader is also encouraged to prove that all the specified atoms are covered by P_2. Hence P_2 is complete for any terminating query, by the sufficient condition from Section 3. The completeness is not violated when one (the first or the second) rule for *watch* is removed from P_2 (as all the specified atoms are covered by P_2 without the clause).

Program P_2 terminates for any query $Q = sat(t, l)$ where t, l are lists of a fixed length (i.e. possibly non-ground terms of the form $[t_1, \ldots, t_n]$). This can be justified in a similar way as termination of P_1 (conf. Footnote 5). The termination holds for any selection rule, in particular for that of Prolog with (arbitrary) delay declarations.

The employed methods of proving correctness, completeness, and termination of logic programs are not applicable to the final Prolog program, as it contains the Prolog conditional and *nonvar*. We informally justified that the two procedures, which differ in P_2 and in the Prolog program, will behave in the same way.[11]

[11] Notice that the informal justification for search space pruning in procedure *update_watch* is based on the assumption that the first argument of the initial query $sat(f, l)$ is a representation of a propositional formula. So we do not know if the Prolog program is complete for initial queries with the first argument not of this form.

It remains to show that the Prolog program does not flounder, i.e. that each delayed atom is eventually selected. (The same holds for P_2 with the `block` declaration.) Assume that the initial query is $sat(f, l)$ where f is a representation of a propositional formula, l is a fixed length list, and that each variable occurring in f occurs in l. Notice that the intended initial goals are of this form. In each non failed derivation, $elim\_var/1$ will eventually bind all the variables of l, and hence all the variables of f. Thus all the delayed atoms will be selected.

6 Conclusions

This paper presents an example of constructing a Prolog program using the Logic + Control approach of Kowalski, and shows that part of the related reasoning can be formalized in a rather simple and natural way. We constructed the SAT solver of Howe and King [HK11b, HK11a]. The initial simple logic program P_1 was first transformed to another logic program P_2, in order to facilitate modifying the control in the intended way. This step could be seen as adding new representation of data (the formula represented as a single argument of *clause_setup* is represented as a few arguments of the newly introduced predicates). Then control was added to P_2, by fixing the selection rule and pruning the search space.

We discussed correctness, completeness, and termination of the three programs, and non-floundering of P_2 and the final program. In particular, we outlined formal proofs of correctness and completeness of P_1 and P_2. The presented sufficient conditions for correctness and completeness of logic programs may be of separate interest. They can be seen as formalizing commonsense ways of reasoning about programs. The condition for correctness is known since 1979 but seems neglected. A stronger form of both conditions is discussed in [DM05]. The author believes that such proof methods, possibly treated informally, are a useful tool for practical reasoning about actual programs. The formal proofs outlined in this paper support this claim.

References

[Apt97] K. R. Apt. *From Logic Programming to Prolog*. International Series in Computer Science. Prentice-Hall, 1997.

[Cla79] K. L. Clark. Predicate logic as computational formalism. Technical Report 79/59, Imperial College, London, December 1979.

[DM05] W. Drabent and M. Miłkowska. Proving correctness and completeness of normal programs – a declarative approach. *Theory and Practice of Logic Programming*, 5(6):669–711, 2005.

[Dra99] W. Drabent. It is declarative: On reasoning about logic programs. Technical report, 1999. `http://www.ipipan.waw.pl/~drabent/itsdeclarative3.ps.gz`. Poster at 1999 International Conference on Logic Programming.

[HK11a] J. M. Howe and A. King. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science*, 2011. To appear. Special Issue on FLOPS 2010.

[HK11b] J. M. Howe and A. King. A pearl on SAT solving in Prolog (extended abstract). *Logic Programming Newsletter*, 24(1), March 31 2011. `http://www.cs.nmsu.edu/ALP/2011/03/a-pearl-on-sat-solving-in-prolog-extended-abstract/`.

[Sha83] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.