

# Assignment 1

CSC 2512—Winter 2012

Out: Feb 10th, 2012

Due: Thursday March 1st (email me your write-up)

Worth 20% of your final mark.

## 1 Overview

SAT solvers offer a very useful and effective technology but require a very primitive form of input. In fact, it is almost impossible to manually specify a problem in the input format required by SAT solvers.

In this assignment you will create a tool that allows the user to specify a SAT problem in a higher level language (first-order logic). The tool will then automatically convert this first-order specification into the format utilized by most SAT solvers. The resultant converted specification can then be feed into a SAT solver and the satisfiability of the original problem determined.

Any formula of first-order logic (i.e., formulas with quantification  $\exists$  and  $\forall$ ) can be converted to propositional logic when the domain over which each quantified variable ranges is finite. In that case every quantified formula can be “grounded out” by replacing each variable by the finite set of values it ranges over. The result is a formula that contains logical operators (and ‘ $\wedge$ ’, or ‘ $\vee$ ’, not ‘ $\neg$ ’, etc.), along with ground atomic formulas (i.e., predicates applied to constants). By treating each different ground atomic formula as a new propositional variable, we obtain a propositional formula that can then be converted into CNF using the technique of introducing new variables described in class.<sup>1</sup>

## 2 First-Order Logic

To specify the assignment more precisely we start off with a (mostly) standard specification of first-order logical languages that is the kind of input language our tool will accept.

A first-order logical language  $\mathcal{L}$  with **bounded quantification** can be defined as follows. We start off with the following sets of symbols.

1. A set of **predicate** symbols  $\{p_1, \dots, p_n\}$ . Some of these predicates will be **types**.
2. A set of **constant** symbols  $\{c_1, \dots, c_m\}$ . These can include numeric constants.
3. A set of **function** symbols  $\{f_1, \dots, f_\ell\}$ .
4. An unlimited set of **variable** symbols  $\{x_1, \dots\}$

---

<sup>1</sup>This technique is described in more detail in the paper “A Structure-preserving Clause Form Translation” by David A. Plaisted and Steven Greenbaum. (See <http://dl.acm.org/citation.cfm?id=7244> and follow the doi link while on a UofT network to get access to the paper).

The function  $f_i$  and predicate symbols  $p_j$  have associated arities,  $arity(f_i)$  and  $arity(p_j)$ . The arity specifies the number of arguments each symbol takes.

Among the set of predicate symbols we distinguish a special subset of symbols all of which have arity one (unary predicates). Elements of this special subset are called **types**.

From the variables, constants and function symbols **terms** can be constructed.

1. Any constant,  $c$ , is a **term**.
2. Any variable,  $x$ , is a **term**.
3. If  $f$  is a function with  $arity(f) = k$ , and  $t_1, \dots, t_k$  are  $k$  terms, then  $f(t_1, t_2, \dots, t_k)$  is a **term**.

From the terms, the predicate symbols, and logical connectives **formulas** can be constructed.

1. If  $p$  is a predicate with  $arity(p) = k$  and  $t_1, \dots, t_k$  are  $k$  terms then  $p(t_1, t_2, \dots, t_k)$  is a formula. These formulas are called **atomic formulas**.
2. if  $t_1$  and  $t_2$  are terms then  $t_1 = t_2$  is a formula. These formulas are called **equality formulas**. They are also considered to be atomic formulas.
3. If  $\phi$  and  $\psi$  are formulas then  $\neg\phi$ ,  $\phi \wedge \psi$ , and  $\phi \vee \psi$  are all formulas.
4. If  $x$  is a variable,  $\phi$  is a formula, and  $t$  is a **type** predicate, then  $\exists(x; t).\phi$  and  $\forall(x; t).\phi$  are both formulas.

Generally, to make the language more useful the following abbreviations are used.

1.  $\phi \rightarrow \psi$  abbreviates  $\neg\phi \vee \psi$ .
2.  $\phi \equiv \psi$  abbreviates  $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ .
3.  $\exists!(x; t).\phi$  abbreviates  $\exists(x; t)(\phi \wedge \forall(y; t).y \neq x \rightarrow \neg\phi)$ .

### 3 Grounding

Given that each type predicate is satisfied by a pre-specified finite set of constants any formula of the language  $\mathcal{L}$  can be **grounded**. That is, all quantification and variables can be removed to yield a formula in which all terms are constants.<sup>2</sup> In particular, given a formula  $\phi$  we can generate its grounding by recursively applying the following two transformations to all of its sub-formulas.

1.  $\exists(x; t).\psi$  is transformed to  $\bigvee_{c \in t} \psi[x/c]$ .
2.  $\forall(x; t).\psi$  is transformed to  $\bigwedge_{c \in t} \psi[x/c]$ .

Here every type  $t$  has a specified set of constants  $t = \{c_1, \dots, c_k\}$  such that  $t(x)$  is true only when  $x$  is equal to some constant in this set. That is,  $t$  specifies a bounded range for the quantified variable  $x$ . Also the notation  $\psi[x/c]$  indicates that in the formula  $\psi$  every instance of  $x$  is replaced by the constant  $c$ .<sup>3</sup>

<sup>2</sup>We assume that the formula has no free variables (variables that are not bound by some quantifier).

<sup>3</sup>For simplicity, we assume that in any formula each quantified variable is unique.

## 4 Example

For example, the 4-Queens CSP problem can be specified using a first-order language defined by the following symbols.

1. Constants:  $\{1, 2, 3\}$
2. Functions:  $\{-, abs\}$
3. Predicate symbols:  $\{queen, pos, on\}$
4. Types:  $queen = \{1, 2, 3\}, pos = \{1, 2, 3\}$ .

Then the formula specifying 4-Queens is

$$\begin{aligned} & \forall(q; queen).(\exists!(p; pos).on(q, p)) \\ & \wedge \forall(p; pos).(\exists!(q; queen).on(q, p)) \\ & \wedge \forall(q_1; queen).\forall(p_1; pos).\forall(q_2; queen).\forall(p_2; pos). \\ & \quad \neg(q_1 = q_2) \wedge on(q_1, p_1) \wedge on(q_2, p_2) \rightarrow \neg(abs(-(q_1, q_2)) = abs(-(p_1, p_2))) \end{aligned}$$

## 5 Restrictions

Dealing with equality and functions in general is difficult. For example, if we have that  $c = b$  is true then any truth assignment that makes  $p(c)$  true must make  $p(b)$  true. To avoid such problems you should make the following assumptions in your tool.

- All functions in the language must have the property that they can be evaluated by your tool. Hence once grounding has been completed, all terms involving functions will be nested applications of functions applied to constants. Such terms can then be evaluated by your tool and replaced by simple constants. For example, using the language specified in the Queens example, if we have the formula

$$\exists(q; queen).abs(-(3, q)) = 1$$

After grounding we would obtain

$$abs(-(3, 1)) = 1 \vee abs(-(3, 2)) = 1 \vee abs(-(3, 3)) = 1$$

Which should be simplified by your tool to the formula

$$2 = 1 \vee 1 = 1 \vee 0 = 1$$

A reasonable policy for this is either (a) don't allow functions in your tool (although numeric functions are very useful), or (b) restrict the functions to be numeric functions and require such functions only apply to numbers on grounding. (Note that you don't need to check this in the syntax, just have the tool reject the formula if it encounters a problem in evaluating the function terms).

- Two constants are equal if and only if they are the same symbol. This means that after grounding one is left with all equality predicates involving constant terms. Therefore, your tool can evaluate these equality formulas to true/false, and then recursively simplify the entire formula.

For example, the formula

$$2 = 1 \vee 1 = 1 \vee 0 = 1$$

would then be evaluated by your tool to

$$\text{false} \vee \text{true} \vee \text{false}$$

which would then be simplified to `true`. As another example, your tool would always evaluate the formula  $a = b$  as `false`.

By applying these two rules, your tool will be able to determine that some input formulas like  $\exists(q; \text{queen}). \text{abs}(-(3, q)) = 1$  are true even without invoking a SAT solver.

You should simplify your grounded formula in this manner at some point before generating the final output CNF.<sup>4</sup>

## 6 To Do

You need to define some parsable input format for specifying input problems expressed in first-order logic with bounded quantification.

Once a problem is specified as a first-order formula in your input format, your tool should build a recursive data structure representing the formula and its sub-formulas.

Then, your tool should build a grounded version of the formula, simplify the formula by evaluating the terms and equality formulas, convert the formula to CNF by adding extra variables,<sup>5</sup> assign a mapping between ground atomic formulas and numeric propositional variables, and finally output a CNF in DIMACS format (the format used by SAT solvers) which uses numbers to specify the different propositional variables. It should also output a key included in the comment section of outputted DIMACS file. The key should give the mapping from the numbered propositional variables used in the DIMACS file, and the ground atomic formulas of your input.

## 7 To hand In

Hand in (email) a short document (max 6 pages including any references, but it can be shorter than that). The document should include a description of the input language your tool takes, a description of some of the key ideas of behind your implementation. For example, if you did any special optimizations you should mention them.

You should test your tool on a few different problems. One simple problem you should apply it to is to express the Zebra problem (given in the paper "A Filtering Algorithm for constraints of difference in CSPs" by Jean-Charles Regin on the course website) in your input language, convert it to CNF and solve it with

<sup>4</sup>Note that predicates over numbers like  $\leq$  could potentially also be included in your language and then evaluated after grounding.

<sup>5</sup>Note that the previously referenced paper by Plaisted and Greenbaum gives an algorithm that could be applied prior to grounding. So you can if choose different ways of sequencing these operations.

a SAT solver (I recommend MiniSat <http://minisat.se/>). In the write up show how this problem is expressed in your input format, and give the results obtained by your tool (e.g., size of the outputted CNF, time to do the translation, speed of the SAT solver on the resultant CNF, etc.). Also describe at least one other problem (you don't necessarily have to give the input specification as you did with the Zebra problem, but do mention some of the performance metrics.). Finally, you should draw some conclusions about whether or not your tool can scale well enough to tackle interesting problems.

## 8 Hints

Feel free to impose any reasonable restrictions on the syntax in order to make parsing easy. Also don't worry too much about checking for syntactic errors (e.g., you don't need to check that every predicate is used with the right arity, or even declare the predicate symbols). Take the approach instead of trying to process the input and failing if the input is incorrectly specified.

Choose an implementation language (and input format) that makes parsing easy.

For example, if you use a list processing language like scheme, your input language could require that the formula be specified as a recursive list. This would allow the language interpreter to automatically generate a recursive list structure that you could then process to generate the right output. Other languages have similar functionality.

You can implement as many features as you wish. In general, converting to CNF by grounding in the most efficient way is a difficult problem. You do not need to solve these hard problems, you are only required to implement a bare-bones tool.