ETH zürich

Alexander J. Summers

Program Verification

Exercise Solutions 2: Encoding Problems to SAT

Assignment 1 (Eliminating Equality)

The variation of Ackermannization should in the lectures to eliminate equality potentially generates many variables and extra conjuncts to express instances of the reflexivity, symmetry and transitivity properties of equality.

1. We generate 2^n new boolean variables; one for each pair of original term variables.
2. The number of possible instances is n instances of reflexivity, n^2 of symmetry and n^3 of transitivity.
3. Here are a few ideas; if you have others, I'd be interested to hear them.
   - We don't need boolean variables to represent reflexive cases (equating a term with itself); we could just translate such equalities to $\top$. In this way, we also avoid the need for adding instances of the reflexivity property entirely.
   - For transitivity, we can avoid instances which only differ by symmetry with one of the three axioms already chosen (however, this idea is subsumed by the next one).
   - We can "build in" symmetry properties as part of our Ackermannization process in the following way: choose any total ordering $<$ on the original term variables in the problem. Now, only introduce boolean variables $eq_{x,y}$ for pairs of variables x and y such that $x < y$. When translating the original formula, translate equalities between x and y to this, regardless of the order of the terms in the equality (i.e. $y = x$ also gets translated to $eq_{x,y}$). Note that the case of reflexive equalities is already dealt with above. In this way, we avoid the need for symmetry axioms entirely.
   - For transitivity, we need to consider triples of term variables x,y and z. But there is no need to consider choices in which any of these three variables are the same (the property is then logically trivial). Similarly (by symmetry), once we instantiate the property for x,y and z, there is no need to instantiate it also for z,y and x. We could implement this by only instantiating for $x < z$ combinations (according to the ordering above).

- We can further optimize transitivity with the following idea: let $\equiv_{eq}$ be the smallest equivalence relation on the original term variables, such that $x \equiv_{eq} y$ if x and y form the two sides of some equality in the original formula. We only need generate transitivity instances for triples of variables all in the same $\equiv_{eq}$-equivalence class. The intuition behind this idea is that some variables might never (even via transitivity) get compared with one another in constructing models for the original formula.

Assignment 2 (Eliminating Uninterpreted Functions)

We work on the inner terms first. Starting from:

$$f(g(x)) = x \wedge f(y) = x \wedge \neg(y = g(x))$$

we generate a new term variable g_x to replace $g(x)$, and a new term variable f_y to replace $f(y)$. Since, so far there are no pairs of newly-introduced terms regarding the same top-level function (just one for f and one for g), we don't need to add any congruence constraints. The resulting formula is:

$$f(g_x) = x \wedge f_y = x \wedge \neg(y = g_x)$$

Now, we generate a new term variable f_{g_x} to replace the function application $f(g_x)$. Since we have already introduced the term variable f_y (for the same function f), we have to add the congruence constraint: $y = g_x \Rightarrow f_y = f_{g_x}$, conjoining this to the new formula. We obtain:

$$f_{g_x} = x \wedge f_y = x \wedge \neg(y = g_x) \wedge (y = g_x \Rightarrow f_y = f_{g_x})$$

This formula is satisfiable (in fact, we can almost "read off" a model from the conjunction of equalities and inequalities): in any model M for which $M(f_{g_x}) = M(x) = M(f_y)$ and where $M(y)$ and $M(g_x)$ are interpreted as two different values. Note that the congruence conjunct is satisfied, since the left-hand-side of the implication is false in such a model.

This also tells us how to build models for the original formula: take any model in which $M(g)$ is a mathematical function which does *not* map $M(x)$ to $M(y)$; i.e. $M(g)(M(x))$ should be some value v in the model different from $M(y)$. Furthermore, the interpretation of f in the model (i.e. $M(f)$) must be a function which maps v to the same value as $M(x)$, and which also maps $M(y)$ to this same value.

Assignment 3 (Sudoku Encodings)

In both encodings, we employ 9^3 variables $val_{i,j,n}$ to represent that the cell (i, j) has the value n. To express that each cell has at least one value, we have to write 9^2 clauses, each of 9 disjuncts. To express that each cell has at most one value requires $\frac{9 \cdot 8}{2} \cdot 9^2$ clauses, each of size 2 (one for each pair of different possible values, for each cell).

The clauses so far express the constraints necessary for our modelling of a grid as boolean variables to make sense. On top of these, we require the clauses to express the "rules", as discussed in the lectures. These require 27 instances of the `eachValue` "macro"; the difference between the two encodings is in how this macro is defined.

In the first encoding, each instance of the macro generates 9 clauses, each of size 9. In the second, the macro generates $\frac{8 \cdot 7}{2} \cdot 9 = 784$ clauses, each of size 2. Despite the high number of clauses, this latter encoding is likely to perform significantly better: the many clauses of size 2 will cause many immediate unit propagation steps whenever a literal is chosen, while in the former encoding, many literals must be chosen before making any "deductions" (conceptually, we have to fill in all but one numbers of, e.g., a given row, before deducing the value of the last one).

Assignment 4 (Bit-blasting)

Suppose the first input is represented by a 2-length bit-vector $b_1 b_0$, and the second by $c_1 c_0$. Then, we can write out the result via "long multiplication". In evaluating additions, the result of adding two bits is true when exactly one of the two is true (and causes a carry if both are true). To conveniently represent "exactly one is true" we use the "xor" ($\oplus$) connective (which has the same precedence as $\Leftrightarrow$), defined by e.g. $A \oplus B \equiv A \wedge \neg B \vee \neg A \wedge B$. The arithmetic then works out as follows:

			b_1	b_0
$*$			c_1	c_0
			$(b_1 \wedge c_0)$	$(b_0 \wedge c_0)$
$+$		$(c_1 \wedge b_1)$	$(c_1 \wedge b_0)$	
		$(c_1 \wedge b_1)$	$(b_1 \wedge c_0 \oplus c_1 \wedge b_0)$	$(b_0 \wedge c_0)$
$+$	$(b_1 \wedge c_0 \wedge c_1 \wedge b_0)$			
	$(b_1 \wedge c_0 \wedge c_1 \wedge b_0)$	$(c_1 \wedge b_1 \wedge (\neg c_0 \vee \neg b_0))$	$(b_1 \wedge c_0 \oplus c_1 \wedge b_0)$	$(b_0 \wedge c_0)$

These resulting four formulas define the respective output bits of our multiplier.