Adding Equivalence Classes to Stålmarck's Method in First Order Logic

Magnus Björk
mab@cs.chalmers.se

Department of Computing Science
Chalmers University of Technology, Gothenburg, Sweden

Abstract. Stålmarck's method is a theorem proving method for propositional logic that has been known for more than a decade, and which has been successful in many industrial applications. During the last few years, a first order logic lifting of the method has been developed. This extended abstract presents the major features of the extension, and describes work in progress on how to lift the equivalence relations known from the propositional version of the method.

1 Introduction to Stålmarck's Method

Stålmarck's method [5,6] is a tautology checking method for propositional logic that has turned out to be efficient in many industrial applications. Stålmarck's method resembles many tableaux calculi. Its most distinguishing feature is the *dilemma rule* which is a branch-and-merge rule, which can be seen as an extension of the analytic cut rule [3]. It creates two branches and assumes that some formula (called the *dilemma formula*) is true in one of the branches and false in the other. If both branches lead to contradictions, then the original problem is unsatisfiable. Hence, if a contradiction is found in one branch, that branch can be removed, and the only problem left is to find a contradiction in the other branch. The special feature of the dilemma rule is that if a contradiction is not found in either of the branches with a limited amount of work, then the two branches can be recombined into one branch again. In that case, the knowledge in the combined branch consists of *the intersection* of the knowledge in the two branches.

Descriptions of Stålmarck's method usually define a notion of hardness; a derivation has *depth 0* if it contains no applications of the dilemma rule, and *depth n* if it contains applications of the dilemma rule where the branches have depth $n-1$ or less. A formula is n*-hard* if its minimal refutation has depth n. The complexity of finding a refutation increases drastically with the hardness — the upper bound for the proof search is $O(m^{2k+1})$, where k is the proof depth and m the size of the formula [5]. Therefore it is important to minimize the hardness by making the 0-saturation as powerful as possible. It has turned out that many problems that arise in industrial applications are 1-hard or easier [5].

In a privately circulated draft, Stålmarck sketched an extension of the method to first order logic (FOL). Since then, it has been refined and extended [4,1,2].

Two prototypes have been implemented by Lundgren and Björk respectively, and these have shown encouraging results. A more efficient implementation is currently being written by Björk, and benchmarks from that implementation are expected to be available during the spring of 2004. A thorough introduction to the method can be found in [2], where also a soundness proof is presented. Proving completeness is future work.

Stålmarck's method in FOL uses formulas that contain two kinds of free variables: universal and rigid ones. Universal variables are implicitly universally quantified ($P(x, c)$ thus means that P holds for any x together with c[1]), while rigid variables act as placeholders for arbitrary terms. Rigid variables are introduced in the dilemma branches when the dilemma formula contains free variables: if the dilemma formula is $P(X)$, we cannot assume that either $P(X)$ or $\neg P(X)$ holds for a universally quantified X. But for any instance of X, either $P(X)$ or $\neg P(X)$ holds. After the two branches have been merged, the rigid variables become universal. Universal variables are thus introduced at the end of dilemmas, and are also introduced and propagated by various non-branching rules.

Note that the rigid variables are not instantiated in the branches of a dilemma rule. While other tableaux-like calculi substitute all occurrences of a rigid variable in an entire tableau, the proof procedure of Stålmarck's method memorizes the possible substitutions, and later redoes the same dilemma, but with the substitution performed already in the dilemma formula. In this way, it is easy to detect relevant dilemma formulas, and the conclusions of some applications of the dilemma rule become more general.

Figure 1 shows a short proof that uses the dilemma rule. The goal of the proof is to derive a contradiction from the formulas labelled 1, 2, and 3. The formulas 4 through 6 are obtained simply by applying $\forall$- and $\exists$-elimination rules to the previous formulas. Formula 7 arises from an application of an implication elimination rule to formulas 4 and 6. After that it is not possible to get further with non branching rules. The rules in Stålmarck's method only produce formulas that are instances of subformulas of the original problem. Therefore it is not possible to conclude, for instance, $Q(c_0) \leftrightarrow \neg Q(c_0)$ from formulas 5 and 7.

In this proof, we select $Q(Y_0)$ as the first dilemma formula. This choice could be made automatically with a strategy that starts with all predicate symbols with only fresh variables as arguments, and then continues with the subformulas of the formula to be proved. The two branches in this example are similar, and do not contain any important conclusions. However, when looking for subformulas to unify $Q(Y_0)$ with, the proof procedure finds that if we could unify Y_0 and c_0 (which we are not allowed to do in the branch), we might get further, since formulas 1.1 and 7 would then together yield the fact $R(c_0)$. This instance is memorized, and after the dilemma is closed (in this case with an empty intersection), the same dilemma is performed again, but with a more specific argument.

[1] In this paper we use x to denote universal variables, X and Y for rigid variables, c for constants, f for functions, and P, Q, and R for predicates. All of these symbols may be indexed.

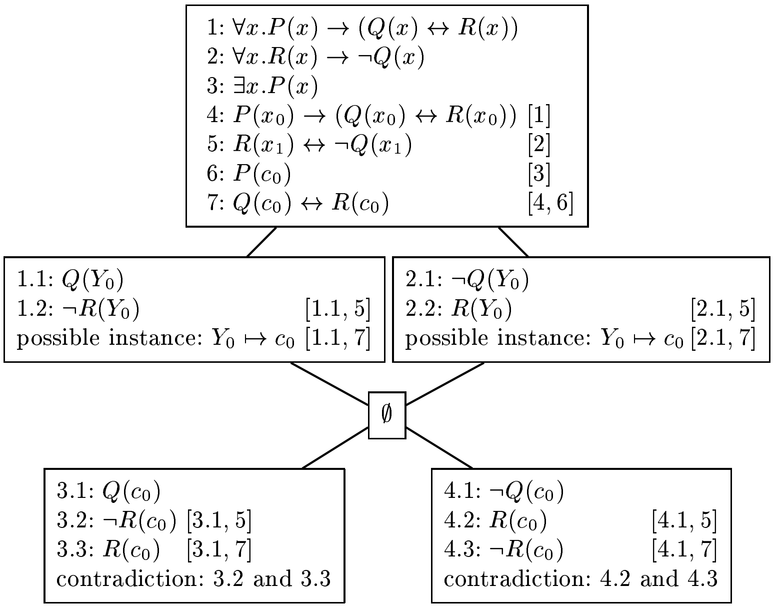

Fig. 1. A proof which shows how to find instances of formulas to use as dilemma formulas.

This time, it is possible to unify the dilemma formula with formula 7, and contradictions are easily found in both branches.

There are several reasons for not performing the instantiation destructively during the first dilemma rule. One is that often the intersection at the end of the first dilemma is not empty. In those cases, the conclusions from the dilemma will be more general, since they contain free variables, which become universal at the end of the dilemma. Another reason is that we do not have to worry about fairness. If there are several ways of instantiating the dilemma formula, we simply memorize all of them and do them all in turn, instead of selecting one.

2 Equivalence Relations

The propositional logic version of Stålmarck's method is often described using equivalence relations (sometimes called *partial equivalence relations*) on formulas. This means that instead of only deriving whether certain formulas are true or false (as shown in the previous section), it is possible to derive that two formulas have the same or the opposite truth value. In the example proof in Figure 1, the intersection ending the first dilemma would not be empty if we used equivalence relations. The equivalence $Q(Y_0) \equiv \neg R(Y_0)$ is a consequence of both branches, and after the dilemma, the rigid variable Y_0 can be replaced by a fresh variable, yielding the equivalence $Q(x_2) \equiv \neg R(x_2)$. This new fact would immediately contradict formula 7, and hence the proof would be finished.

Indeed, with equivalences it is possible to make the proof even shorter. Formulas 5 and 7 can easily be turned into equivalences that contradict each other. Hence, introducing equivalences, the problem presented in Figure 1 becomes 0-hard instead of 1-hard. The problem is to find an efficient way of representing equivalences of first order formulas.

For propositional logic, the equivalence relation is usually represented as a set of disjoint sets, that correspond to the different equivalence classes. The elements of the equivalence classes are subformulas of the formula to be proved. When two subformulas are shown to be equivalent, their respective equivalence classes are merged. If the formulas have different signs, all elements of one of the equivalence classes are negated before the merger. Initially, all classes but one are singleton sets, consisting only of the respective unique subformulas of the formula to be proved. The only class containing two elements is the one containing the entire formula to be proved, which also contain the value false. A contradiction is found when the same formula occurs with different signs in the same equivalence class. The equivalence relations are viewed as sets of pairs (representing all possible pairs of equivalent formulas) when the intersection is calculated.

This idea cannot be lifted to FOL in a straightforward way. Indeed, the complexity of representing the equivalence classes in FOL may turn out to cancel out the benefits. It would, however, be interesting to try this out. The rest of this paper describes some thoughts about how it might be done. This is work in progress, and we have no results about completeness, or even decidability or soundness. The version of the proof method described in [2] does not use equivalence classes.

A problem that arises in FOL is that the number of equivalence classes is unbounded — subformulas may have any number of instances. It is also not clear what to do with instances of existing formulas. Say, for instance, that the equivalence class $\{P(x), Q(x)\}$ is known, and that we derive that $P(c) \equiv R$. We could then create a completely new class for this equivalence: $\{P(c), R\}$. But, then we are not using the known fact that $P(x) \equiv Q(x)$. A better idea may be to create an instance of the known equivalence class first, and then add the new fact to this class, yielding $\{P(c), Q(c), R\}$.

It would be possible to continue on like this; every time we derive a new equivalence we can find all equivalence classes that contain unifiable literals and add the literals of the equivalence (or instances of them) to the classes (or instances of them). The problem is that many equivalence classes will have many instances. Some of them may even subsume each other, and we expect that checking for subsumption is an expensive operation.

Perhaps it can be more efficient to share elements of the classes. We can give names to each class, and refer to that name instead of copying the class. Using the same classes as in the previous example, introduce $C_0(x) = \{P(x), Q(x)\}$. Making the free variables explicit makes the instantiation easier. Now, instead of copying that class when we introduce the equality $P(c) \equiv R$, we can simply refer to the name: $C_2() = \{C_0(c), R\}$. This means that R is equivalent to all elements of $C_0(c)$. This eliminates the copies, and in some cases it is easy to identify subsumption. For example, if $C_i(\ldots) = \{C_j(t_1, \ldots, t_n)\}$, then C_j subsumes C_i, and if $C_i(x_1, x_2, x_3) = \{\ldots, C_j(x_1, x_2), \ldots\}$, then C_i subsumes C_j.

An effect of this representation of knowledge is that we can describe infinite equivalence classes. For instance, the equivalence $P(x) \leftrightarrow P(f(f(x)))$ results in the equivalence class $C_1(x) = \{P(x), C_1(f(f(x)))\}$. This class really contains all literals of the shape $P(f^{2n}(x))$, for any n and x. As an example, we will add the consequences of $P(f(c)) \equiv \neg P(f^5(c))$ to the set. It is easy to detect that $P(f(c))$ is equivalent to $C_1(f(c))$ (by noticing that $P(x)$ is a generalization of $P(f(c))$. In the same way, it is easy to detect that $\neg P(f^5(c))$ is equivalent to $\neg C_1(f^5(c))$. However, we can also easily detect that this is equivalent to $\neg C_1(f^3(c))$ (since $C_1(f^2(x))$ is a generalization of $C_1(f^5(c))$). Finally, $\neg C_1(f^3(c))$ is equivalent to $\neg C_1(f(c))$. But this contradicts $C_1(f(c))$, which we found to be equivalent to $P(f(c))$.

We believe that this method can be used, given a literal, to find a name for it which reuses as much as possible of existing classes. Since the algorithm hinted at above looks for generalizations of given literals, it must terminate eventually — a term can only be generalized in a finite number of steps. The biggest question now is whether there exists an efficient algorithm for finding contradictions. It is not always the case that contradictions turn up so easily as they did in the example. Another important question is how to calculate the intersection of two equivalence relations. The probable solution is that the new equivalences of each branch should also be stored separately as a set of pairs.

References

1. Magnus Björk, *Extending Stålmarck's Method to First Order Logic*, TABLEAUX 2003 Position Papers and Tutorials (Marta Cialdea Mayer and Fiora Pirri, eds.), Technical Report RT-DIA-80-2003, Dipartimento di Informatica e Automazione, Università degli Studi di Roma Tre, September 2003, pp. 23–36.
2. Magnus Björk, *Stålmarck's Method for Automated Theorem Proving in First Order Logic*, Licentiate thesis, Chalmers University of Technology, Department of Computing Science, Technical Report 28L, ISSN 1651-4963, http://www.cs.chalmers.se/~mab, November 2003.
3. Marcello D'Agostino and Marco Mondadori, *The Taming of the Cut*, Journal of Logic and Computation **4** (1994), no. 3, 285–319.
4. Lars Lundgren, *Stålmarck's Method in First Order Logic*, Master's thesis, Chalmers University of Technology, 1999.
5. Mary Sheeran and Gunnar Stålmarck, *A Tutorial on Stålmarck's Proof Procedure for Propositional Logic*, Formal Methods in System Design **16:1** (2000), 23–58.
6. Gunnar Stålmarck, *A system for determining propositional logic theorem by applying values and rules to triplets that are generated from a formula*, Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 907 (1994), European Patent No. 0403 454 (1995), 1992.