The Heuristic Theorem Prover: Yet Another SMT Modulo Theorem Prover
(Tool Paper)

Kenneth Roe

kendroe@fordocsys.com

1 Introduction

HTP is an SMT Modulo theorem prover similar to many others.[2,3,4,5,6,9,11] As input, HTP accepts problems using the SMT-LIB format[8]. As output, HTP will answer either *SAT*, *UNSAT* or *UNKNOWN*. Alternatively, HTP can be run in a preprocessing mode in which the output is the simplified problem in SMT-LIB format. An evidence file showing the derivation in a human readable form can be produced. There is a *Treeview* application which shows this derivation in a tree widget making it convenient to navigate a complex proof.

The main contribution of HTP is the introduction of a preprocessor that includes algorithms for detecting unate predicates, eliminating variables, symmetry breaking and boolean encoding. The other algorithms of HTP are similar to other systems. HTP implements a DPLL(T) similar to BarcelogicTools[6]. There are domain theories for equality of uninterpreted function symbols, real difference logic, linear inequality and array logic.

2 The Preprocessor

HTP has a preprocessor that applies a number of algorithms to incrementally simplify problems before handing them off to the DPLL(T) solver. There are command line switches to turn on and off some of the algorithms.

2.1 Rewriting

Before anything else is done with a problem, HTP rewrites it using a number of algebraic simplification rules for arithmetic and boolean equations. As an example, the system will rewrite `a+b=2*a+1` to `b=a+1` and it will rewrite `1+1+1+1+1+a<1+1+b` to `a+3<b`. Boolean expressions are also simplified. For example, `a and a` is rewritten to `a`.

HTP also implements a simple contextual rewriting mechanism. The idea is that certain subterms can be assumed to be true or false while rewriting others. For example, for the expression `a<b and not(a=b)`, when simplifying `not(a=b)`, the system can assume `a<b` is true. This reduces `not(a=b)` to `true` and hence the whole expression reduces to `a<b`.

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 467–470, 2006.

2.2 The Unate Detection Algorithm

For any boolean expression E (which HTP is trying to satisfy), if asserting a predicate P is guaranteed to make E false, then we know that for any assignment to the predicates satisfying E, we know that P has to be false. P is thus a *unate* predicate. The theorem can be simplified by making `not(P)` an assumption and then simplifying E. HTP can detect unate predicates efficiently.

The algorithm is most easily described through the use of an example, `(a<b) and (if b=c then a+1=b else a<b+1)`. This expression has four atomic predicates, `a<b`, `b=c`, `a+1=b` and `a<b+1`. The goal is to figure out which are unate.

The system creates a table with all pairwise implications between the atomic predicates. Then the system annotates each boolean subterm of the expression with four sets, the set of atomic predicates that when asserted make that subterm true, the set of atomic predicates which when asserted make that subterm false, the set of atomic predicates which when denied make that subterm true and the set of atomic predicates which when denied make that subterm false.

The system starts by computing these sets for each of the atomic predicate subterms in the theorem. Then these sets are combined with simple set operations (union or intersection) to create the sets for each of the non-atomic terms. The table below shows these computations for the example above.

	Assert makes true	Deny makes false
`b=c`	`b=c`	`b=c`
`a<b`	`a<b,a+1=b`	`a<b,a<b+1`
`a+1=b`	`a+1=b`	`a+1=b,a<b,a<b+1`
`a<b+1`	`a<b,a<b+1,a+1=b`	`a<b+1`
`if b=c then a+1=b else a<b+1`	`a+1=b`	`a<b+1`
`(a<b) and (if b=c (then a+1=b else a<b+1)`	`a+1=b`	`a<b,a<b+1`

HTP stores all expressions as DAGs. Hence, the work in computing these sets only needs to be done once for each unique subterm. Also, the sets are represented as bit vectors making the computations very efficient.

Variable elimination is a special case of unate detection. If it is found that denying a predicate of the form `v=e`, where `v` is a variable, makes the theorem false, then we know any satisfying assignment must have `v=e` as true. Hence, within our theorem, we can replace all instances of `v` with `e` and simplify. We do not need to enter `v=e` as an assumption.

2.3 Symmetry Breaking

Symmetry breaking in HTP is an extension of the idea studied in the context of SAT problem solving.[7]. The algorithm works through the following steps. First, all symmetric pairs of variables are detected. A pair of variables (a,b) are said to be a symmetric pair if in the theorem T (the theorem HTP is trying to prove) when replacing all instances of a with b and b with a, the resulting theorem is exactly the same as T. The current algorithm for finding pairs simply tests all

possible pairs of variables of the same type to see if swapping them produces the same theorem. From this set of symmetric pairs of variables, symmetric pairs of atomic predicates are identified. P1 and P2 are said to be symmetric if there is a set of symmetric variable pairs $\{(a_1, b_1) \ldots (a_n, b_n)\}$ such that P1 can be transformed to P2 by simply replacing each a_i with b_i and each b_i with a_i for each of the pairs in the set of symmetric variables. Next we calculate groups of symmetric predicates. A group of symmetric predicates is a set of two or more atomic predicates such that any two predicates in the group are symmetric. Finally, symmetry breaking disjuncts are added in a manner similar to [7].

2.4 Boolean Encoding

HTP implements an algorithm for doing boolean encoding of difference logic built on ideas from [10]. More information on this algorithm is available on the author's website at **www.fordocsys.com/htp.htm**.

3 Current State of the Implementation

The system is implemented in C and compiled both on Windows and Linux. The Treeview program for viewing outputs is only available on Windows. The author's website contains detailed tables with results and a downloadable executable. HTP has been run in stand alone mode on the entire QF_UF problem set as well as the scheduling problems in QF_RDL[1] giving results which are competitive with other top systems. The preprocessing mode has been run on all SMT-COMP'05 problems except the QF_UFIDL and QF_AUFLIA sections.

The preprocessor was evaluated by running the output in MathSat[2], YICES[3], Simplics[4] and BarcelogicTools[6]. Using symmetry breaking, the preprocessor substantially improved the performance of problems in the QF_UF section. However, the current algorithm cannot be applied to other sections. Combining the preprocessor with YICES yields a combined theorem proving tool that can solve 40 problems from SMT-COMP'05. BarcelogicTools, the top system from the competition, only solved 39 problems. Unate detection and rewriting improved the performance of the QF_LRA/spider_benchmarks, the QF_LIA/sal and the QF_LRA/sal sections. For many of these problems, rewriting and unate detection were sufficient to solve the problems. Rewriting also improved the performance of the QF_LIA/wisa benchmarks. This was due to many expressions of the form `1+1+1+...+x=y` being simplified to `n+x=y`. The unate detection and rewriting had little impact for the other sections. In some cases applying the preprocessor changed the performance of other tools by as much as a factor of five either way. This variance has also been seen with minor permutations of problems in many SMT Modulo solvers. The difference logic encoding was quite effective in improving performance when it was applicable.

4 Conclusion and Future Work

Good preprocessing techniques are the most promising direction for finding performance improvements. Future work will also include expansion of the stand

alone mode to handle problems from all divisions of SMT-COMP'05[1] as well as adding bit vector and quantifier logic. The two most promising preprocessing directions for creating additional performance enhancements are that of developing boolean encoding algorithms and symmetry breaking. Boolean encoding routines are being extended beyond difference logic. Symmetry breaking is being extended to handle problems outside the QF_UF division.

Acknowledgement. The author thanks Leonardo de Moura from SRI for his feedback on this paper.

References

1. Clark Barrett, Leonardo de Moura, and Aaron Stump. Smt-comp: Satisfiability modulo theories competition. In *CAV*, volume 3576 of *LNCS*, pages 20–23. Springer, 2005.
2. Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. The mathsat 3 system. In *CADE-20, Int. Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*. Springer, July 2005.
3. Leonardo de Moura. System description: Yices 0.1. Technical report, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94062, July 2005. http://fm.csl.sri.com/yices.
4. Bruno Dutertre and Leonardo de Moura. Simplics: Tool description. Technical report, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94062, July 2005. http://fm.csl.sri.com/simplics.
5. Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *CAV*, volume 2102 of *LNCS*, 2001.
6. Robert Nieuwenhuis and Albert Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *CAV*, volume 3576, pages 321–334, 2005.
7. Arathi Ramani, Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Dynamic symmetry-breaking for improved boolean optimization. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Januray 2005.
8. Silvio Ranise and Cesare Tinelli. The smt-lib standard: Version 1.1. Technical report, Department of Computer Science, The University of Iowa, 2005. Available at `goedel.cs.uiowa.edu/smtlib`.
9. Sanjit A. Seshia, Shuvendu K. Lahiri, , and Randal E. Bryant. A hybrid sat-based decision procedure for separation logic with uninterpreted functions. In *Proc. 40th Design Automation Conference (DAC)*, pages 425–430, June 2003.
10. Ofer Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with sat. In *Proc. of Computer Aided Verification*, 2002.
11. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.