Implementing a Verified Efficient RUP Checker

어덕기
아이오와대학교 컴퓨터과학과
e-mail : duckki-oe@uiowa.edu

Implementing a Verified Efficient RUP Checker

Duckki Oe
Dept. of Computer Science, The University of Iowa

요 약

To ensure the correctness of high performance satisfiability (SAT) solvers, several proof formats have been proposed. SAT solvers can report a formula being unsatisfiable with a proof, which can be independently verified by a trusted proof checker. Among the proof formats accepted at the SAT competition, the Reverse Unit Propagation (RUP) format is considered the most popular. However, the official proof checker was not efficient and failed to check many of the proofs at the competition. This inefficiency is one of the drawbacks of SAT proof checking. In this paper, I introduce a work-in-progress project, vercheck to implement an efficient RUP checker using modern SAT solving techniques. Even though my implementation is larger and more complex, the level of trust is preserved by statically verifying the correctness of the code. The vercheck program is written in GURU, a dependently typed functional programming language with a low-level resource management feature.

1. Introduction

Satisfiability (SAT) solvers are automated propositional theorem provers and are widely used in several fields such as formal verification and artificial intelligence, due to their high performance. Mainstream SAT solvers are highly optimized and usually written in C/C++. To ensure the correctness of those high performance SAT solvers, it is desirable for SAT solvers to provide certificates, which can be independently verified by a trusted checker. For satisfiable formulas, most SAT solvers can produce candidate models. And, for unsatisfiable formulas, some solvers can produce proofs refuting the input formulas. Several proof formats have been proposed for unsatisfiability certificates. Among the proof formats accepted at the SAT competition, the Reverse Unit Propagation (RUP) format is considered the most popular [1]. However, the official proof checker was not efficient and failed to check many of the proofs within the given timeout at the competition [2]. An efficient RUP checker implementation would require a sophisticated SAT solving feature, called two-literal watch lists [3]. Instead of trusting such complex software, the official proof checker translates proofs into a resolution-based format and checks them using a simple trusted resolution checker. Although this method keeps the size of trusted base small, the translation process wastes time and space. That is the main reason of the inefficiency of the official RUP proof checker.

This paper introduces a work-in-progress project, called vercheck to implement an efficient RUP checker that checks RUP proofs directly without translation. The vercheck program implements the two-literal watched lists data structure and other optimizations for efficiency. Although the complexity of vercheck will be much higher than the existing simple resolution checker, a formal verification technique is used to ensure the correctness of the vercheck program. A verified efficient proof checker can check bigger proofs generated from more difficult formulas and also refreshen the interests in the certified track of the SAT competition, which has not been held since 2007.

The vercheck program is being developed and verified in a dependently typed functional programming language, called GURU [4]. GURU also provides a way to safely manage imperative data structures and generate efficient code [5]. The program reuses some of the code from my previous work, called versat, which is a verified SAT solver [6]. As a specification, the propositional resolution rule is defined as the basic inference rule. And the goal is to prove that whenever the RUP inference checker certifies a clause, there exists a resolution proof of the clause. Note that the proof is not generated at run-time. Instead, ensures that such a proof can be always computable.

2. Background

2.1 The RUP Proof Format

The Reverse Unit Propagation (RUP) proof format has been proposed by Van Gelder as an efficient propositional proof representation scheme [7]. RUP is an inference rule that concludes $F \vdash C$ when $(F \cup \neg C)$ is refutable using only unit resolution, which is similar to standard binary resolution except that one of the two resolved clauses is required to be a unit clause. Unit resolution is not refutation complete in general, but it has been shown that conflict clauses generated from standard conflict-analysis algorithms are indeed RUP inferences [7]. If a clause is a RUP inference, a unit-resolution proof deriving the clause can be calculated from that clause, itself. Potentially, a long resolution proof of a RUP inference can be compressed to the concluded clause. Also, an efficient RUP inference checker can be implemented using the two-literal watch lists, a standard unit propagation algorithm used in most SAT solvers [8]. A complete RUP proof is a sequence of clauses (lemmas) with the last one being the empty clause. And the sequence of clauses are checked incrementally one clause at a time. Each clause C is checked with respect to the RUP inference rule, where F is the original formula and the clauses that have been checked previously. Even though all correct lemmas are logically true in the input formula, RUP inference is so weak that intermediate clauses are necessary as stepping stones leading to the empty clause. For example, here is an unsatisfiable formula in the Conjunctive Normal Form (CNF): $(p \lor q) \land (p \lor \neg q) \land (\neg p \lor q) \land (\neg p \lor \neg q)$. That formula can be encoded in the DIMACS format, which a standard input format used at the SAT competition, as below:

```
 1 2 0    1 -2 0    -1 2 0    -1 -2 0
```

Positive numbers represent propositional variables and negative numbers are negated variables. The variables p and q are renamed as 1 and 2, respectively. A zero indicates the end of each clause. Now, consider a RUP proof of the formula below:

```
1 0
0
```

The RUP proof format has a similar syntax as the DIMCAS format. The proof above has two clauses (RUP inferences). Because the input formula does not have a unit clause, the empty clause cannot be a RUP inference directly from the input formula. So, at least one intermediate clause is necessary. The first proof clause is a unit clause 1. Assume the negation of the clause, which is -1. The assumed clause -1 and the first clause of the formula concludes 2 by unit resolution, and similarly, -1 and the second clause concludes -2. Finally, 2 and -2 are contradictory. So, 1 is a RUP inference. Once a clause is verified, it is kept as a lemma and may be used in the later inferences. Using the clause just verified, the empty clause can be checked in a similar fashion, resolving 1 with the third and forth input clauses ans so on.

2.2 Related Works

The Isabelle theorem prover has been used to verify SAT and Satisfiability Modulo Theories (SMT) proofs [9, 10]. Such a theorem prover with a small kernel has a high assurance, however, proofs from SAT/SMT solvers have to be translated and reconstructed into the theorem prover's proof language. Thus, those systems have the same performance limitation due to proof translation.

More closely related work is Darbari et al.'s TraceCheck proof checker that is verified in the Coq theorem prover [11]. TraceCheck is another SAT proof format supported by PicoSAT, an open source state-of-the-art SAT solver [12]. A TraceCheck proof is a sequence of lemmas and each lemma is a list of clause names. To check a lemma, those clauses mentions are resolved one after another. The conclusions of resolutions are implicit in the proof and it is the checker's responsibility to calculate the resolvent of each resolution. They proved that their resolvent computation is correct, and they extracted an OCaml code from the Coq implementation for faster execution and portable compilation. Compared to TraceCheck, the RUP format is easier to be instrumented in an existing SAT solver, because the solver can simply dump all the deduced lemmas and that will be a RUP proof.

3. The GURU Programming Language

GURU is a functional programming language with dependent types, in which programs can be verified by means of type checking[1]. With dependent types, for example, we can define an indexed data type for lists. A type index is a program value occurring in the type, in this case the length of the list. We define the type <vec A n> to be the type of lists storing elements of type A, and having length n, where n is a Peano (i.e., unary) number:

```
Inductive vec : Fun(A:type)(n:nat).type :=
| vecn : Fun(A:type).<vec A Z>
| vecc : Fun(A:type)(spec n:nat)
          (a:A)(l:<vec A n>). <vec A (S n)>
```

This states that vec is inductively defined with constructors vecn and vecc. The return type of vecc is <vec A (S n)>, where S is the successor function. So the length of the list returned by the constructor vecc is one greater than the length of the sublist l. Note that the argument n (of vecc) is labeled "spec", which means specificational. GURU will enforce that no run-time results will depend on the value of this argument, thus enabling the compiler to erase all values for that parameter in compiled code.

We can now define the type of vec_append function on vectors:

```
vec_append : Fun(A:type)(spec n m:nat)
             (l1:<vec A n>)(l2:<vec A m>).
             <vec A (plus n m)>
```

This type states that append takes in a type A, two specificational natural numbers n and m, and vectors l1 and l2 of the corresponding lengths, and returns a new vector of length (plus n m). This is how the relationship between lengths can be expressed using dependent types. Type-checking code like this may require the programmer to prove that two types are equivalent. For example, a proof of commutativity of addition is needed to prove <vec A (plus n m)> equivalent to <vec A (plus m n)>. Currently, these proofs must mostly be written by the programmer, using special proof syntax, including syntax for inductive proofs.

GURU supports memory-safe programming without full memory garbage collection, using a combination of techniques [5]. Immutable tree-like data structures are handled by reference counting, with some optimizations to avoid unnecessary increments/decrements. Mutable data structures like arrays are handled by statically enforcing a readers/writers discipline: either there is a unique reference available for reading and writing the array, or else there may be multiple read-only references. The one-writer discipline ensures that it is sound to implement array update destructively, while using a pure functional model for formal reasoning. The connection between the efficient

[1] Guru is downloadable from http://www.guru-lang.org/.

implementation and the functional model is not formally verified, and must be trusted. This is reasonable, as it concerns only a small amount of simple C code (less than 50 lines), for a few primitive operations like indexing a C array and managing memory/pointers.

4. Specification

Checking a RUP inference is computationally complex requiring the checker to search for an appropriate sequence of unit resolutions. Instead of formalizing the RUP inference directly, vercheck's specification is based on two simpler inference rules: resolution and hypothesis. Then, I formalized the correctness of the code checking a single RUP inference (a RUP clause) as there exists a resolution proof of the clause. Although the idea is very similar to the proof translation, vercheckdoes not create resolution proofs at run-time. Rather, the existence of such resolution proofs is to be proved statically from the invariants of the vercheck code.

4.1 Inference System

Figure 1 shows the important definitions for the propositional inference system, which is encoded as the data type pf. The word type is the 32-bit machine integer type built in GURU. The negative integer values represent the negated propositional variables, in the same way that mainstream SAT solvers represent literals. The eq_lit function compares two integers and the negated function changes the sign of integer. The standard polymorphic list type and some of related functions such as member and list_subset are defined in the GURU's standard library. The member function checks the membership of an item in a list with respect to the given equality relation, and list_subset similarly checks if all members of a list are in the other list. The cl_erase function removes all the occurrences of a literal from the given clause. The expressions between curly braces {A = B} are equality types meaning A and B are provably equal. And tt is a boolean constant for "true".

The <pf F C> type is a dependent type indexed by a formula F and a clause C, and it represents the judgement F ⊢ C. Any value of the type <pf F C> is meant to be a proof of F ⊢ C. The constructors, pf_asm, pf_res and pf_hyp, encode inference rules with necessary conditions, namely assumption, resolution and hypothesis rules, respectively. The assumption rule can conclude any clause in the input formula, and the resolution rule concludes a resolvent of two proven clauses over a specified pivot literal. The hypothesis rule concludes the clause $l \rightarrow C$ (encoded as $\neg l \vee C$), when C can be proved under the hypothesis of l. Thus, constructing such a data structure of type <pf F C> is essentially proof checking.

```
Define lit := word
Define clause := <list lit>
Define formula := <list clause>

Define cl_subsume := fun(c1:clause)(c2:clause).
  (list_subset lit eq_lit c1 c2)

Define is_resolvent :=
  fun(r:clause)(c1:clause)(c2:clause)(l:lit).
  (and (and (member (negated l) c1 eq_lit)
           (member l c2 eq_lit))
      (and (cl_subsume (cl_erase c1 (negated l))
   r)
           (cl_subsume (cl_erase c2 l) r)))

Inductive pf : Fun(F:formula)(C:clause).type :=
| pf_asm : Fun(F:formula)(C:clause)
             (u:{ (member C F eq_clause) = tt }).
              <pf F C>
| pf_res : Fun(F:formula)(C C1 C2:clause)(l:lit)
             (d1:<pf F C1>)(d2:<pf F C2>)
             (u:{  (is_resolvent  C  C1  C2  l)  =
   tt }).
             <pf F C>
| pf_hyp : Fun(F:formula)(l:lit)(C:clause)
             (d:<pf (cons (cons l nil) F) C>).
             <pf F (cons (negated l) C)>
```

(Figure 1) The pf data type and helper definitions

4.2 Typing RUP Checking Function

Figure 2 shows the typing of the rup_check function, which checks each RUP inference. It has three input arguments: the input formula F, the current checker's state s, and the clause c to check. The state s is a blackbox data structure to store the internal SAT solver's state. The return type of the function is <check_t F C A>, which is indexed by the input formula, the clause to check, and the type of the blackbox. The blackbox does not affect the correctness. It just allows the check function to return the updated internal state as part of the return value. Whenever a RUP inference is checked, the implementation needs to update its state and store the RUP clause as a lemma in its clause database. A check_t value has two cases: check_fail and check_ok. The check_fail case means the checker failed to verify the RUP inference. On the other hand, the check_ok case means the checker verified the RUP inference and a proof data structure for the clause C is provided as the evidence. Note that the proof p is marked as specificational using the spec keyword. So, the proof data structure will not be created at run-time. Instead, GURU compiler guarantees that it is always computable by check that the value is only dependent on the invariants of the program. So, the type of rup_check defines the correct RUP checker. Now, it is all up to the implementation to efficiently implement the checker and prove (in GURU) its correctness.

```
Inductive check_t : Fun(F:formula)(C:clause)
                    (A:type). type :=
| check_fail: Fun(spec F:formula)(spec C:clause)
               (A:type)(s:A).
               <check_t F C A>
| check_ok  : Fun(spec F:formula)(spec C:clause)
               (A:type)(s:A)(spec p:<pf F C>).
               <check_t F C A>

rup_check : Fun(spec F:formula)
              (s:<CheckerState F>)(c:clause).
              <check_t F c <CheckerState F>>
```

(Figure 2) check_t type and rup_check function type

5. Current Status

In this section, I'll give an overview of how vercheck works and the current status. Suppose a clause C is a correct RUP inference, and F is the union of the input formula and the previously checked RUP clauses learned as lemmas. Under this hypothesis $\neg C$, the unit propagation operation should find a contradiction in the input formula F and the hypothesis. Let C be $l_1 \vee l_2 \vee \vee l_n$. Then, the hypothesis is $\neg l_1 \wedge \neg l_2 \wedge \wedge \neg l_n$. First, versat assigns those variables in li's so that all li's are true. Then, versat performs unit propagation, and it should find a conflicting clause D, which is a clause in F that is falsified under the hypothesis. That is an ordinary functionality of versat as a SAT solver. From the fact that D is false under the hypothesis, we can construct a proof of $F \wedge \neg C \vdash \neg D$. Second, in vercheck, we need to deduce $F \wedge \neg C \vdash \bot$, which is true because D is in F . At the time of writing this paper, vercheck has to explicitly perform the resolutions to prove the empty clause ($\bot$). This explicit resolution can be avoided by proving a sophisticated invariant of the program, which tells that every variable hypothetically assigned a truth value has a unit clause supporting that assignment. From that invariant, the empty clause can be immediately proved from any clause conflicting under the current assignments without performing resolutions. Finally, the hypothesis rule is applied to derive $F \vdash \neg C \rightarrow \bot$, which is $F \vdash C$. That process verifies the RUP inference C using the existing efficient unit propagation code and simple inference rules.

6. Conclusion

Formal verification technique is usually used to reduce the size of trusted base and increase the level of confidence in a system. However, the official RUP checker used at the SAT competition is already small and trusted. Here, the challenge is to implement a more efficient checker that is as trustworthy as before. In vercheck, formal verification technique is used to improve the performance over the existing proof checker without increasing the size of trusted base. And it is also an interesting engineering case study in the dependent typed programming field, because the existing versat code is reused in a different context, and the properties of the code are reinterpreted for the new software, vercheck.

Acknowledgements. Many thanks to my advisor Aaron Stump for supporting this research and implementing helpful features in the GURU compiler.

참고문헌

[1] A. Van Gelder. http://users.soe.ucsc.edu/~avg/ProofChecker/ProofChecker-fileformat.txt.

[2] A. Van Gelder. http://users.soe.ucsc.edu/~avg/ProofChecker/Documents/cert-tables-sat07.pdf.

[3] H. Zhang, "Sato: An efficient propositional prover," in CADE (W. McCune, ed.), vol. 1249 of Lecture Notes in Computer Science, pp. 272–275, Springer, 1997.

[4] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson, "Verified Programming in Guru," in Programming Languges meets Program Verification (PLPV) (T. Altenkirch and T. Millstein, eds.), 2009.

[5] A. Stump and E. Austin, "Resource Typing in Guru," in Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010 (J.-C. Filliâtre and C. Flanagan, eds.), pp. 27–38, ACM, 2010.

[6] D. Oe, A. Stump, C. Oliver, and K. Clancy, "versat: A verified modern sat solver," in VMCAI (V. Kuncak and A. Rybalchenko, eds.), vol. 7148 of Lecture Notes in Computer Science, pp. 363–378, Springer, 2012.

[7] A. Van Gelder, "Verifying rup proofs of propositional unsatisfiability," in ISAIM, 2008.

[8] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for cnf formulas," in Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, DATE '03, (Washington, DC, USA), pp. 10886–, IEEE Computer Society, 2003.

[9] T. Weber and H. Amjad, "Efficiently checking propositional refutations in hol theorem provers," J. Applied Logic, vol. 7, no. 1, pp. 26–40, 2009.

[10] S. Böhme and T. Weber, "Fast lcf-style proof reconstruction for z3," in ITP (M. Kaufmann and L. C. Paulson, eds.), vol. 6172 of Lecture Notes in Computer Science, pp. 179–194, Springer, 2010.

[11] A. Darbari, B. Fischer, and J. Marques-Silva, "Industrial-strength certified sat solving through verified sat proof checking," in ICTAC (A. Cavalcanti, D. Déharbe, M.-C. Gaudel, and J. Woodcock, eds.), vol. 6255, pp. 260–274, Springer, 2010.

[12] A. Biere, "Picosat essentials," JSAT, vol. 4, no. 2-4, pp. 75–97, 2008.