Extending a Theorem Prover for Deductive Program Verification

Evgenii Kotelnikov[1], Laura Kovács[1], and Andrei Voronkov[2]

[1] Chalmers University of Technology, Gothenburg, Sweden
evgenyk@chalmers.se, laura.kovacs@chalmers.se
[2] The University of Manchester, Manchester, UK
andrei@voronkov.com

1 Introduction

Automated program analysis and verification requires discovering and proving program properties. Typical examples of such properties are loop invariants or Craig interpolants. These properties usually are expressed in combined theories of various data structures, such as integers and arrays, and hence require reasoning with both theories and quantifiers. Recent approaches in interpolation and loop invariant generation [9, 7, 4] present initial results of using first-order theorem provers for generating quantified program properties. First-order theorem provers can also be used to generate program properties with quantifier alternations [7]; such properties could not be generated fully automatically by any previously known method. Using first-order theorem prover to generate, and not only prove program properties, opens new directions in analysis and verification of real-life programs.

First-order theorem provers, such as iProver [5], E [10], and Vampire [8], lack however various features that are crucial for program analysis. For example, first-order theorem provers do not yet efficiently handle (combinations of) theories; nevertheless, sound but incomplete theory axiomatisations can be used in a first-order prover even for theories having no finite axiomatisation. Another difficulty in modelling properties arising in program analysis using theorem provers is the gap between the semantics of expressions used in programming languages and expressiveness of the logic used by the theorem prover. For example, a standard way to capture assignment in program analysis is to use a `let-in` expression, which introduces a local binding of a variable or a function, to a value. There is no local binding expression in first-order logic, which means that any modelling of imperative programs using first-order theorem provers at the backend, should implement a translation of `let-in` expressions.

Efficiency of reasoning-based program analysis largely depends on how programs are translated into a collection of logical formulas capturing the program semantics. The boolean structure of a program property that can be efficiently treated by a theorem prover is however very sensitive to the architecture of the reasoning engine of the prover. Deriving and expressing program properties in the "right" format therefore requires solid knowledge about how theorem provers work and are implemented — something that a user of a verification tool might not have. Moreover, it can be hard to efficiently reason about certain classes of program properties, unless special inference rules and heuristics are added to the theorem prover, see e.g. [3] when it comes to prove properties of data collections with extensionality axioms.

In order to increase the expressiveness of program properties generated by reasoning-based program analysis, the language of logical formulas accepted by a theorem prover needs to be extended with constructs of programming languages. This way, a straightforward translation of programs into first-order logic can be achieved, thus relieving users from designing translations which can be efficiently treated by the theorem prover. One example of such an extension is recently added to the TPTP language [11] of first-order theorem provers, resembling `if-then-else` and `let-in` expressions that are common in programming languages. Namely, special functions `$ite_t` and `$ite_f` can respectively be used to express a conditional statement on the level of logical terms and formulas, and `$let_tt`, `$let_tf`, `$let_ff` and `$let_ft` can be used to express local variable bindings for all four possible combinations of logical terms (`t`) and formulas (`f`). While satisfiability modulo theory (SMT) solvers integrate `if-then-else` and `let-in` expressions, in the first-order theorem proving community so far only Vampire supports such expressions.

We aim for facilitation of reasoning-based program analysis by developing new theories for first-order theorem provers and extending them with features of programming languages. Our recent result in this direction is formalisation of a first-order logic with first class boolean sort that can be efficiently treated by a theorem prover and allows for simpler translation of some program properties, compared to an ordinary many-sorted first-order logic. Our current work is an implementation of this logic in Vampire. Both the theoretical result and the progress on the implementation are summarised in the following section.

2 First Class Boolean Sort

Our recent work [6] presented a modification of many-sorted first-order logic that implements a first class boolean sort. We called this logic FOOL, standing for first-order logic (FOL) + boolean sort. FOOL extends ordinary many-sorted FOL with (i) the boolean sort such that terms of this sort are indistinguishable from formulas and (ii) `if-else-else` and `let-in` expressions. FOOL formulas can be translated to ordinary FOL formulas while preserving models and can hence be treated by a first-order theorem prover.

FOOL is more expressive than FOL. Function and predicate symbols in FOOL can take boolean arguments, formulas in FOOL can be quantified over boolean variables and occur as arguments when the sorts coincide.

We argue that these extensions are useful in reasoning about problems coming from program analysis. A boolean expression in a programming language can be treated both as a value (for example, in a form of a boolean flag, passed as an argument to a function) and as a formula (for example, a loop condition). A translation of a program property into FOOL would not distinguish these cases, whereas a translation into an ordinary FOL would. In the later case, a possible solution would be to map the boolean type of programs to a user-defined boolean sort, postulate axioms about its semantics, and manually convert boolean terms into formulas where needed. This approach however is cumbersome and possibly inefficient due to the way a theorem prover might treat one of the boolean theory axioms.

In contrast to that, for a translation from FOOL to FOL given in [6] we described a modification of superposition calculus that can reason efficiently about the FOL formulas obtained with the translation. This modification involves replacement of the problematic theory axiom by an extra inference rule.

At this point we have an initial implementation of FOOL in Vampire that closely follows [6]. We were able to test the implementation on a set of 490 properties about (co)algebraic datatypes, generated by the Isabelle proof assistant. All of these properties feature quantification over booleans and `if-then-else` expressions and were not previously directly expressible in TPTP. The preliminary results of our experiments show that native implementation of FOOL shows better performance than a naive translation of the boolean type described earlier. However, more experiments are still needed in order to make a conclusion.

Implementation of FOOL in Vampire allowed us to simplify the syntax of monomorphically sorted TPTP, called TFF0 [12]. The lack of distinction between boolean terms and formulas made the distinction between `$ite_t` and `$ite_f` obsolete. They were replaced by a single `$ite` expression. Similarly, four variations of `let-in` expression were replaced by a single `$let`.

3 Future Work

Whereas we proposed a technique for efficient treatment of the boolean sort by a superposition-based theorem prover, no efficient translation of `if-then-else` and `let-in` expressions has been presented. This is left for future work.

Treatment of boolean terms as formulas was implemented in some other logics used in the automated deduction community. The core language of SMT-LIB [1], the collection of benchmarks for SMT-solvers, is a language of first-order logic with this property. The language of higher-order logic supported by theorem provers such as Isabelle is another example. FOOL is a novel result in the area of first-order reasoning and it bridges the semantic gap between logics. Particularly, FOOL is the smallest superset of the core language of SMT-LIB and monomorphically typed subset of TPTP. It means that a first-order theorem prover that supports FOOL can understand SMT-LIB problems without a special translation and we are planning to conduct experiments in reasoning about the corpus of SMT-LIB problems in Vampire.

FOOL is monomorphically typed. It is not hard however to extend it to a polymorphic case. TPTP defines a language of many-sorted FOL with rank-1 polymorphism called TFF1 [2], and implementation of a combination of FOOL and TFF1 is an interesting future work.

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[2] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Proc. of CADE-24*, pages 414–420. Springer, 2013.

[3] A. Gupta, L. Kovács, B. Kragl, and A. Voronkov. Extensionality Crisis and Proving Identity. In *Proc. of ATVA*, pages 185–200, 2014.

[4] K. Hoder, L. Kovács, and A. Voronkov. Playing in the grey area of proofs. In *Proc. of POPL*, pages 259–272, 2012.

[5] K. Korovin. iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proc. of IJCAR*, pages 292–298, 2008.

[6] E. Kotelnikov, L. Kovács, and A. Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In *CICM*, pages 71–86, 2015.

[7] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.

[8] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV*, pages 1–35, 2013.

[9] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS*, pages 413–427, 2008.

[10] S. Schulz. System Description: E 1.8. In *Proc. of LPAR*, pages 735–743, 2013.

[11] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[12] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In *LPAR*, pages 406–419, 2012.