Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types

Nikolaj Bjørner
Microsoft Research

Ken McMillan
Microsoft Research

Andrey Rybalchenko
Microsoft Research & Technische Universität München

Abstract

This work-in-progress report summarizes and proposes automatic procedures for proving properties of programs written in higher-order languages. The approach we examine is by using encodings of the higher-order languages directly as first-order SMT problems over Horn clauses. Horn clauses offer a direct match when considering classical Hoare-style theorem proving of first-order programs, but the presence of closures offer challenges both theoretically when it comes to the completeness of the proof system; and practically when it comes to effectiveness of search procedures.

1 Background

Automatic verification of programs in higher-order languages has received significant attention in recent years. The Liquid Type systems have been used successfully on a large set of challenges from Ocaml [12] and Haskell [15]. Liquid types rely checking program properties as type checking of refinement types. Not all refinement types need to be provided fully, instead users can supply a space of templates that the type inference engine instantiates and refines the template invariants (using the Houdini approach [4]). A very different approach is taken in HALO [16] where the denotational semantics of Haskell, using types that include error states, is encoded using quantified equalities. HALO relies on quantifier instantiation engines and finite model building capabilities to discharge correctness proof conditions. Unno et.al. [14] develop a custom engine that realizes proof rules for Hoare logic for programs with procedure parameters [5]. There are several other systems that use first-order/SMT technologies for establishing properties of higher-order languages. They typically rely on user-annotated refinement types. The executable subset of the PVS specification language, uses decision procedures for discharging type checking conditions and the F$\star$ system includes verification condition generation from a higher-order language with higher-order refinement types.

These systems have in common that they rely on SMT solvers, CVC4, Yices and Z3. It is tempting to ask why program verification cannot directly be reduced to an SMT problem.

2 Program Verification as Horn Clause Satisfiability

Consider the McCarthy function:

```
let mc x = if x > 100 then x - 10 else mc (mc (x + 11))
assert x <= 100 => mc(x) = 91
```

We can check *partial correctness* of `mc` by representing the summary as a binary relation between input and outputs, and describing the function and assertion equivalently as a conjunction of Horn clauses:

$$\forall x \,.\, x > 100 \;\rightarrow\; \texttt{mc}(x, x-10)$$
$$\forall x, y, z \,.\, x \leq 100 \;\wedge\; \texttt{mc}(x+11, y) \;\wedge\; \texttt{mc}(y, z) \;\rightarrow\; \texttt{mc}(x, y)$$
$$\forall x, y \,.\, x \leq 100 \;\wedge\; \texttt{mc}(x, y) \;\rightarrow\; y = 91$$

These Horn clauses are satisfiable if and only if the original partial correctness assertion holds. A satisfying assignment to the Horn clauses (found by Z3) is:

$$mc(x,y) \quad \equiv \quad (y \leq x-10 \ \vee y \leq 91) \wedge y \geq 91 \wedge (x \leq y+10)$$

The assignment is an *inductive* summary for `mc`.

This example illustrates a central claim from [7] (and illustrated in the context of SMT solving in [3]):

Claim 1. *Satisfiability of Horn clauses is an adequate basis for program correctness.*

Synthesis of ranking functions can also be encoded into Horn clauses [7], so Horn clause solving extends also to establishing *total* correctness.

The challenge is of course to *solve* Horn clauses and a number of tools are being developed for solving Horn clauses at scale. These include the Duality [11], HSF [7], Eldarica [13], PDR implementation in Z3 [8], SPACER [10]. The Horn clause format is also a convenient interchange format for symbolic software verification benchmarks. We have collected benchmarks from symbolic software model checking as Horn clauses in an online repository [1]. There are currently around 10,000 benchmarks from various sources. There are several possible strategies for encoding model checking problems into Horn clauses and consequently some benchmarks use different encodings of the same problems.

The repository also includes problems from the tools for liquid types. The liquid type system leverages the Hindley Milner type system for extracting Horn clauses for closures.

3 Closures as Algebraic Data-types

We will here consider a direct encoding of problems from a simply typed higher-order programming language into Horn clauses. The basic idea is to encode functions as relations and higher-order closures as algebraic data-types. The Horn clauses encode an interpreter, suitably specialized to the programs that are analyzed. We use examples from [14] to illustrate the approach.

To warm up, consider the program

```
let f x y = assert (not (x() > 0 && y() < 0))
let h x y = x
let g n = f (h n) (h n)
```

The curried function `h` is partially applied to the same integer n, and `f` is applied to two functions with signature `unit` $\rightarrow$ `int`. This is the only closure type used in this program, so we introduce algebraic data-types to encode the possible closures that are used in the program:

$$\begin{array}{rcl} \mathit{clo} & ::= & \underline{\texttt{h}}\ \texttt{int} \\ \mathit{unit} & ::= & \underline{\texttt{unit}} \end{array}$$

The closure `clo` has a single constructor $\underline{\texttt{h}}$ that takes an integer. It is evaluated to arguments of type *unit*. We assign meaning to the closure by defining an evaluator. A canonical evaluator can be formulated as a relation over (1) a closure, (2) the argument of the closure, (3) the output value, and (4) a flag to indicate successful termination of the evaluation. For our example, there is only a single relevant rule for the evaluator; it reduces to evaluating the function h

$$\forall x, r, ok \ . \ h(x, \underline{\texttt{unit}}, r, ok) \quad \rightarrow \quad Ev(\underline{\texttt{h}}(x), \underline{\texttt{unit}}, r, ok)$$

[1] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/`

The other functions translate in a straight-forward way to Horn clauses:

$$\begin{array}{rrrcl} \left(\begin{array}{rl} & Ev(x,\underline{\texttt{unit}},r_1,ok_1) \\ \wedge & Ev(y,\underline{\texttt{unit}},r_2,ok_2) \\ \wedge & (ok \equiv \neg(r_1 > 0 \wedge r_2 < 0)) \wedge ok_1 \wedge ok_2 \end{array}\right) & \rightarrow & f(x,y,\underline{\texttt{unit}},ok) \\ & & & h(x,y,x,\underline{\texttt{true}}) \\ & f(\underline{\texttt{h}}(n),\underline{\texttt{h}}(n),r,ok) & \rightarrow & g(n,r,ok) \\ & g(n,r,\underline{\texttt{false}}) & \rightarrow & \underline{\texttt{false}} \end{array}$$

where $x,y,r,r_1,r_2,ok,ok_1,ok_2,n$ are variables.

We used this example to sketch a systematic encoding from functional programs to Horn clauses. It already contains some shortcuts (alternatively, we could have considered defining a generic interpreter). It can still be simplified by specializing the program to the correctness assertion and removing arguments. Since $\underline{\texttt{h}}(x)$ is the *only* closure passed to Ev it can furthermore be removed entirely. The simpler, equi-satisfiable set of clauses is:

$$\begin{array}{rcl} h(x,\underline{\texttt{unit}},r) & \rightarrow & Ev(x,r) \\ Ev(x,r_1) \wedge Ev(y,r_2) \wedge r_1 > 0 \wedge r_2 < 0 & \rightarrow & f(x,y) \\ & & h(x,y,x) \\ f(n,n) & \rightarrow & g(n,r) \\ g(n,r) & \rightarrow & \underline{\texttt{false}} \end{array}$$

where x,y,r_1,r_2,n,r are variables. The Horn clauses are non-recursive and Z3's engines for Horn clauses can easily establish satisfiability (e.g., that the assertion holds).

A significantly more challenging example is suggested in [14] to illustrate their method (that comprises of adding extra parameters to closures).

```
let app1 f g = if * then app1 (succ f) g else g f
let app2 i f = f i
let succ f x = f (x + 1)
let check x y = assert (x <= y)
let main i = app1 (check i) (app2 i)
```

We use the shortcuts used in the previous example to encode the recursive functions in an economical way. The example uses two closures corresponding to the types $\underline{\texttt{check}}(i) : \texttt{int} \rightarrow \texttt{unit}$ and $\underline{\texttt{app2}}(i) : (\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{unit}$.

$$\begin{array}{rcl} clo_1 & ::= & \underline{\texttt{app2}}\ \texttt{int} \\ clo_2 & ::= & \underline{\texttt{check}}\ \texttt{int} \mid \underline{\texttt{succ}}\ clo_2 \end{array}$$

Similar to the previous example, the closure clo_1 is superfluous and does not need to appear in the generated Horn clauses. The second closure is recursive and therefore essential, and we include this in the corresponding Horn clauses. Let x,y,f,i be variables, then safety of the example program is

equivalent to satisfiability of the following set of clauses:

$$\begin{array}{rcl} app_1(\underline{\texttt{succ}}(f),i) & \rightarrow & app_1(f,i) \\ Ev(f,i) & \rightarrow & app_1(f,i) \\ Ev(f,x+1) & \rightarrow & succ(f,x) \\ x > y & \rightarrow & check(x,y) \\ app_1(\underline{\texttt{check}}(i),i) & \rightarrow & main(i) \\ main(i) & \rightarrow & \underline{\texttt{false}} \\ succ(f,x) & \rightarrow & Ev(\underline{\texttt{succ}}(f),x) \\ check(x,y) & \rightarrow & Ev(\underline{\texttt{check}}(x),y) \end{array}$$

Z3 accepts recursive Horn clauses with algebraic data-types as input, but is unable to establish satisfiability of these clauses directly. In a nutshell, it lacks the ability to synthesize properties that select leaves in algebraic data-types. We will here examine a couple of approaches that could be used to solve such problems.

3.1 In-lining by resolution

Our first approach is to inline definitions by resolving Horn clauses and merge predicates. The result is given below where app_1 and Ev are merged into Ev.

$$\begin{array}{rcl} Ev(\underline{\texttt{succ}}(f),i) & \rightarrow & Ev(f,i) \\ Ev(f,i) & \rightarrow & Ev(f,i) \\ Ev(\underline{\texttt{check}}(i),i) & \rightarrow & \underline{\texttt{false}} \\ Ev(f,x+1) & \rightarrow & Ev(\underline{\texttt{succ}}(f),x) \\ x > y & \rightarrow & Ev(\underline{\texttt{check}}(x),y) \end{array}$$

The second clause is a tautology and the two clauses that contain $\underline{\texttt{succ}}(f)$ can be resolved leaving an equi-satisfiable set of Horn clauses:

$$\begin{array}{rcl} Ev(\underline{\texttt{check}}(i),i) & \rightarrow & \underline{\texttt{false}} \\ Ev(f,x+1) & \rightarrow & Ev(f,x) \\ x > y & \rightarrow & Ev(\underline{\texttt{check}}(x),y) \end{array}$$

Z3 can immediately establish satisfiability of this set of clauses.

3.2 Quantified abstraction

In [2] we consider synthesis of intermediary assertions using quantified invariants for array manipulating programs. Our approach from [2] is to search over solutions over a template space of quantified invariants. The idea is to supply additional arguments to recursive predicates and bind these in quantifiers. Using the running example, the modified problem is:

$$\begin{array}{rcl} (\forall \vec{z}\,.\, Ev(\vec{z},\underline{\texttt{succ}}(f),i)) & \rightarrow & (\forall \vec{z}\,.\, Ev(\vec{z},f,i)) \\ (\forall \vec{z}\,.\, Ev(\vec{z},\underline{\texttt{check}}(i),i)) & \rightarrow & \underline{\texttt{false}} \\ (\forall \vec{z}\,.\, Ev(\vec{z},f,x+1)) & \rightarrow & (\forall \vec{z}\,.\, Ev(\vec{z},\underline{\texttt{succ}}(f),x)) \\ x > y & \rightarrow & (\forall \vec{z}\,.\, Ev(\vec{z},\underline{\texttt{check}}(x),y)) \end{array}$$

where $\vec{z}$ is a tuple of integer variables. The number of variables in $\vec{z}$ is a parameter to the abstraction procedure. The instantiation heuristic suggested in [2] is very simple and will not produce useful instantiations for this case. Instantiation heuristics that are more powerful for arithmetic are used in [1], and of course the template method of [14].

Let us for the sake of illustrating the idea (but not the practicality) pretend an instantiation procedure produces the following set of clauses:

$$\begin{array}{rcl} Ev(u,v,\underline{\texttt{succ}}(f),i) & \rightarrow & Ev(u,v,f,i) \\ Ev(i,i,\underline{\texttt{check}}(i),i) & \rightarrow & \underline{\texttt{false}} \\ Ev(u,v,f,x+1) & \rightarrow & Ev(u,v+1,\underline{\texttt{succ}}(f),x) \\ x > y & \rightarrow & Ev(u,v,\underline{\texttt{check}}(x),y) \end{array}$$

With some struggle, Z3 can produce the inductive invariant:

$$Ev(u,v,\_,i) \quad \equiv \quad 2 \cdot u = v + i$$

More specifically, the invariant is obtained by analyzing the reverse program:

$$\begin{array}{rcl} Ev(u,v,\underline{\texttt{succ}}(f),i) & \leftarrow & Ev(u,v,f,i) \\ Ev(i,i,\underline{\texttt{check}}(i),i) & \leftarrow & \underline{\texttt{false}} \\ Ev(u,v,f,x+1) & \leftarrow & Ev(u,v+1,\underline{\texttt{succ}}(f),x) \\ x > y & \leftarrow & Ev(u,v,\underline{\texttt{check}}(x),y) \end{array}$$

Karr's algorithm [9] for computing all affine relations for Horn clauses produces the useful congruence.

4 Summary

We have examined using algebraic data-types to encode closures directly in Horn clauses. They provide a simple encoding of higher-order programs into Horn clauses. Existing Horn clause solvers currently emphasize solving clauses over integers, reals, and more lately arrays. Solving Horn clauses with algebraic data-types (Herbrand terms) pose a set of new challenges.

References

[1] Tewodros Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving Existentially Quantified Horn Clauses. In *CAV*, 2013.

[2] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. On Solving Quantified Horn Clauses. In *SAS*, 2013.

[3] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Program verification as Satisfiability Modulo Theories. In *SMT*, 2012.

[4] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[5] Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. Reasoning about procedures as parameters in the language l4. *Inf. Comput.*, 83(3):265–359, 1989.

[6] Roberto Giacobazzi and Radhia Cousot, editors. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013.

[7] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[8] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[9] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[10] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, 2013.

[11] Kenneth L. McMillan and Andrey Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013. `http://research.microsoft.com/apps/pubs/?id=180055`.

[12] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

[13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, 2013.

[14] Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. Automating relatively complete verification of higher-order functional programs. In Giacobazzi and Cousot [6], pages 75–86.

[15] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2013.

[16] Dimitrios Vytiniotis, Simon L. Peyton Jones, Koen Claessen, and Dan Rosén. Halo: haskell to logic through denotational semantics. In Giacobazzi and Cousot [6], pages 431–442.