Combining Higher-Order Model Checking with Refinement Type Inference

Ryosuke Sato
Kyushu University
Fukuoka, Japan
ryosuke@ait.kyushu-u.ac.jp

Naoki Iwayama
The University of Tokyo
Tokyo, Japan
iwayama@kb.is.s.u-tokyo.ac.jp

Naoki Kobayashi
The University of Tokyo
Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

Abstract

There have been two major approaches to fully automated verification of higher-order functional programs: higher-order model checking and refinement type inference. The former approach is precise, but suffers from a bottleneck in the predicate discovery phase. The latter approach is generally faster than the former, thanks to the recent advances in constrained Horn clause (CHC) solving, but is imprecise, in that it rejects some valid programs. To take the best of the two approaches, we refine the higher-order model checking approach, by employing CHC solving in the predicate discovery phase. We have implemented the new approach and confirmed that the new system can verify more programs than those based on the previous two approaches.

CCS Concepts • **Theory of computation → Program verification**; *Program analysis*; *Invariants*;

Keywords Automatic Verification, Higher-Order Programs, Higher-order Model Checking, Constrained Horn Clauses, Refinement Types

ACM Reference Format:
Ryosuke Sato, Naoki Iwayama, and Naoki Kobayashi. 2019. Combining Higher-Order Model Checking with Refinement Type Inference. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3294032.3294081

1 Introduction

There have recently been active studies on automated techniques for higher-order program verification [17, 24–26, 29]. Among others, the two major approaches have been higher-order model checking [14, 17, 22, 24] and refinement type inference [10, 26, 27, 29]. Both the approaches have advantages and disadvantages. In the present paper, therefore, we aim to take the best of both the approaches, by extending the former approach with CHC (constrained Horn clause) solving, the key ingredient used in the latter approach.

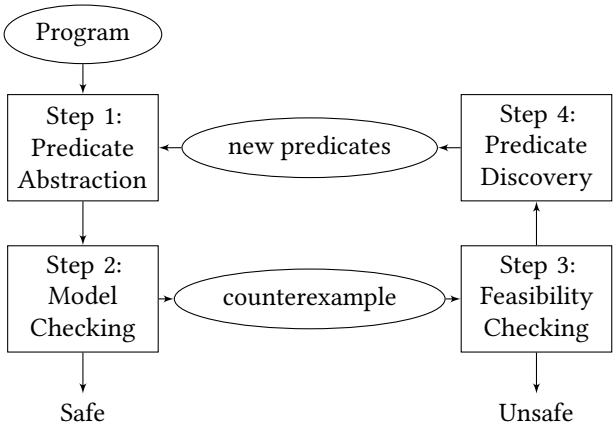


Figure 1. Higher-order model checking with predicate abstraction and CEGAR

Before explaining our new approach, let us summarize the two approaches and their advantages and disadvantages.

Higher-Order Model Checking (HOMC) Approach Figure 1 shows a typical architecture of the higher-order model checking (HOMC, for short) approach [14]. Given a higher-order functional program as an input, *predicate abstraction* is first applied to obtain a higher-order *Boolean* program as an overapproximation of the original program (Step 1). For example, consider the following program.

```
let rec id x =
  if x <= 0 then x else 1 + id (x-1)
let main n = assert (id n = n)
```

Suppose that we wish to verify that the assertion never fails for any integer input `n` of the function `main`. By abstracting the return value of `id` by the predicate $\lambda r.\, r = x$ (where x is the argument of `id`), we obtain the following Boolean program:

```
let rec id' () =
  if * then true else id'()
let main' () = assert (id'())
```


PEPM '19, January 14–15, 2019, Cascais, Portugal

ACM ISBN 978-1-4503-6226-9/19/01…$15.00
https://doi.org/10.1145/3294032.3294081

Here, `*` represents a non-deterministic Boolean value, and `id'` is an abstract version of `id`, which takes the unit value and returns a Boolean value that represents whether the return value r of the original function `id` satisfies $r = x$. (How to obtain `id'` above does not concern us here; an interested reader may wish to consult [14].) Higher-order model checking [13, 16, 24] can then be used to check whether the abstract program is safe (Step 2). In this case, the abstract program is indeed safe (i.e. the assertion never fails); thus the verification succeeds. If we use the predicate $\lambda r.r > 0$ instead of $\lambda r.r = x$, we obtain the following abstract program:

```
let rec id'' () =
  if * then false
  else if id''() then true else *
let main'' () = assert(let r = id''() in * )
```

Here, `id''` returns a Boolean value of whether `id` returns a positive integer. Since that information does not help us determine whether `id n=n`, the argument of the `assert` command may evaluate to `false`. A higher-order model checker thus yields the following error path as a counterexample.

```
main''() ⟶D assert(let r = id''() in * )
⟶D assert(let r = if true then ... in * )
⟶D assert(let r = false in * )
⟶D assert(false) ⟶D fail
```

In Step 3 of Figure 1, we check whether the corresponding path of the original program (i.e., a path that takes the then-branch in `id`, and the argument of `assert` evaluates to false) is feasible. Since it is infeasible (because `id n` must evaluate to `n` if the then-branch is chosen in `id`), the counterexample is spurious for the original program. We then proceed to Step 4, and find a new predicate from a proof that the path is infeasible (this predicate discovery phase will be discussed in more detail later). If the new predicate is $\lambda r.r = x$, the verification succeeds in the next iteration.

Kobayashi et al. [14] have implemented an automated verification tool MoCHi for a subset of OCaml. Advantages of the verification method are that it is precise (in particular, it never reports false positives), and that it can generate a counterexample if a program is unsafe. The tool, however, suffers from a serious bottleneck in the predicate discovery phase: for some valid programs, too specific predicates are repeatedly found and, as a result, the loop of Steps 1–4 (called the CEGAR loop) never terminates.

Refinement Type Inference (RTI) Approach In the refinement type inference (RTI, for short) approach [10, 26, 27, 29], the safety problem (of checking whether a given program reaches an error state) is reduced (in a sound but incomplete manner) to the typability problem in a refinement type system [8, 18, 28]. Then the typability problem is further reduced to the satisfiability problem for constrained Horn clauses (CHC), and CHC solvers [4] are used to check the satisfiability.

For the example of `id` program above, we first prepare the following template for the type of `id`:

$$x : \{\nu : \mathbf{int} \mid P(\nu)\} \to \{r : \mathbf{int} \mid Q(x, r)\}\,.$$

Here, P and Q are predicate variables that represent the pre- and post-conditions of the function: if an integer argument x of `id` satisfies $P(x)$, then a return value r satisfies $Q(x, r)$. Based on a standard refinement type system (such as the one used in Liquid types [18]), the typability of the whole program is expressed by the following constraints on P and Q:

$$\begin{array}{l}\forall n.P(n) \qquad \forall n, r.(Q(n, r) \Rightarrow n = r)\\ \forall x, r.(P(x) \wedge x \leq 0 \Rightarrow Q(x, x))\\ \forall x, r.(P(x) \wedge x > 0 \Rightarrow P(x - 1))\\ \forall x, r.(P(x) \wedge x > 0 \wedge Q(x - 1, r) \Rightarrow Q(x, 1 + r))\end{array}$$

Here, the constraints on the first and second lines respectively come from the `main` function, and the then-clause of the `id` function. Those on the third and fourth lines come from the else-clause. (We do not discuss the detail on how these are generated; an interested reader may wish to consult, e.g., [26].) The above constraints are satisfied by $P(x) \equiv \mathbf{true}$ and $Q(x, r) \equiv x = r$. Thus, we can conclude that the program is typable, and hence also safe.

This approach is often faster than the HOMC approach, partially thanks to the recent advances in CHC solvers [4, 6, 27]. A disadvantage is that it is sometimes imprecise, rejecting some valid programs due to the incompleteness of the underlying refinement type system.

Our New Approach To take the best of both approaches, we refine the HOMC approach by employing the RTI approach for predicate discovery (Step 4 in Figure 1). More precisely, from a spurious counterexample obtained in Step 3, we construct a program slice of the original program, which consists of only the program points visited by the spurious counterexample. We then apply refinement type inference for the slice instead of the original program. If the inference succeeds, we collect predicates that occur in the inferred refinement types and add them as predicates used in the predicate abstraction phase (Step 1) of the next iteration.

For the example of `id` program above and the spurious counterexample obtained from `main''`, the program slice is:

```
let rec id x =
  if x <= 0 then x else _
let main n = assert (id n = n)
```

Here, the part `_` is ignored by RTI; in an implementation, it can be implemented as an infinite loop that never fails. From the slice, we obtain the following CHCs:

$$\begin{array}{l}\forall n.P(n) \qquad \forall n, r.(Q(n, r) \Rightarrow n = r)\\ \forall x, r.(P(x) \wedge x \leq 0 \Rightarrow Q(x, x))\end{array}$$

This set of CHCs is a subset of those given in the section on the RTI approach, obtained by removing the constraints from the else-clause of the original program. We can obtain

$P(x) \equiv \mathbf{true}$ and $Q(x, r) \equiv x = r$ as a solution. Thus, $\lambda r.x = r$ is added as a predicate used for abstracting the return value of `id`. The verification succeeds in the next iteration of the CEGAR loop.

The RTI for a program slice may fail due to the incompleteness of the underlying refinement type system. In that case, we fall back to the previous approach [14] for predicate discovery. Actually, the previous approach [14] uses a kind of RTI, but for a different notion of program slice called a straightline higher-order program (SHP). The main difference is that an SHP unfolds recursion, and replicates a function definition for each function call occurring in a (spurious) counterexample, so that the SHP does not contain any recursion. An advantage of using an SHP is that the resulting CHCs are acyclic and hence always guaranteed to have a solution. The main disadvantage is, as already mentioned, that inferred predicates are often too specific (e.g., $r = 0 \wedge x = 0$, instead of $r = x$) to be used for predicate abstraction.

We have implemented the new approach and confirmed through experiments that the new method can prove more programs to be safe than the previous two approaches.

The rest of this paper is structured as follows. Section 2 explains our new method in a little more detail. Section 3 reports an implementation and experimental results. Section 4 discusses related work, and Section 5 concludes the paper.

2 Our Method

This section explains our method and discusses its properties. We avoid a boring reformalization of the HOMC approach, by focusing on the difference from [14] below.

2.1 The Target Language and Verification Goal

The target language of verification is essentially the same as that of [14]: a simply-typed, call-by-value, higher-order functional language with recursion, Booleans, and integers. The syntax of the core language is given by:

$$\begin{aligned} D \text{ (programs)} &::= \{f_1\,\widetilde{x_1} = e_1, \ldots, f_n\,\widetilde{x_n} = e_n\} \\ e \text{ (expressions)} &::= x \mid c \mid *_B \mid e_1\,e_2 \\ &\quad \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \mid \mathbf{fail} \\ \kappa \text{ (simple types)} &::= B \mid \kappa_1 \to \kappa_2 \\ B \text{ (base types)} &::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} \end{aligned}$$

Here, $*_B$ evaluates to a value of base type B in a non-deterministic manner. The term **fail** aborts the evaluation of the whole program. The assert expression $\mathbf{assert}(e)$ used in Section 1 can be expressed as $\mathbf{if}\ e\ \mathbf{then}\ ()\ \mathbf{else}\ \mathbf{fail}$. We assume that every function in D has a non-zero arity, and that D contains a distinguished function symbol $\mathtt{main} \in \{f_1, \ldots, f_n\}$ whose simple type is $\mathbf{unit} \to \mathbf{unit}$.

We write $\longrightarrow_D$ for the small-step reduction relation on terms (which can be defined in a standard manner [14]), and $\Longrightarrow_D$ for its reflexive and transitive closure. The goal of the verification is, given a program D, to check whether $\mathtt{main}() \not\Longrightarrow_D \mathbf{fail}$.

2.2 The New Method

We summarize the difference from the original method [14] below, and show that the new method satisfies desired properties like the progress property (stated as Theorem 1 below). As mentioned in Section 1, the main change is in Step 4. The other steps remain almost the same, except on the following points:

- In the predicate abstraction phase, we embed program point labels of the original program into an abstract program, so that a counterexample generated by a higher-order model checker contains enough information for constructing the corresponding program slice.
- In Step 2, there can be infinitely many counterexamples for an abstract program, and which counterexample is chosen affects the quality of the predicates discovered in Step 4. We have thus modified Step 2 so that we have more control over the choice of counterexamples; this effect will be reported in Section 3.

As already sketched in Section 1, Step 4 proceeds as follows.

4-1 Given the original program D and a counterexample π (which is a sequence consisting of program point labels), construct the program slice D_π, obtained from D by replacing all the subterms whose labels do not occur in π with an infinite loop "*loop*()", defined by $loop\,x = loop\,x$ (_ in Section 1).

4-2 Construct a set C of CHCs, such that C is satisfiable if and only if D_π is typable in the refinement type system, following [6].

4-3 If C has a solution θ (which is an assignment of a predicate to each predicate variable), then add predicates occurring in θ to the abstraction type environment [14], which specifies what predicates should be used for abstracting each subterm of t. For the example in Section 1, the abstraction type environment used for obtaining `main''` is $\mathtt{id} : (x : \mathbf{int}[\,]) \to \mathbf{int}[\lambda v.v > 0]$, and the one after the predicate discovery is $\mathtt{id} : (x : \mathbf{int}[\,]) \to \mathbf{int}[\lambda v.v > 0, \lambda v.v = x]$, which means that the argument of `id` should be abstracted to a unit value, whereas the return value v should be abstracted to a pair of Boolean values that represent whether $v > 0$ and $v = x$ respectively (see [14] for details on abstraction types).

4-4 If C does not have a solution, we fall back to the previous method: construct an SHP and find new predicates from it [14].

As in the previous method [14], the new method satisfies the following "progress property", that the same counterexample is not found twice. (Some familiarity with [14] is required to understand the proof sketch.)

Theorem 1. *Let D be a program, and let Γ be an abstraction type constructed based on a (spurious) counterexample π. Let D' be the most precise abstraction D obtained (based on the predicate abstraction rules in [14]) by using Γ. Then, D' does not have π as a counterexample.*

Proof sketch. It suffices to consider the case where Γ is constructed in Step 4-3 above (since the case for Step 4.4 has been proved in [14]). By the assumption that Γ is constructed in Step 4-3, the slice D_π is typable under a refinement type environment Δ such that $\Delta \in \mathit{DepTy}(\Gamma)$ (where $\mathit{DepTy}(\Gamma)$ denotes the set of refinement type environments constructed using the predicates in Γ; see [14] for the precise definition). By Theorem 4.4 of [14], $\texttt{main}() \not\Rightarrow_{D'_\pi}$ **fail** holds for the most precise abstraction D'_π of D_π obtained by using Γ, Let D' be the most precise abstraction of D obtained by using Γ. By the compositionality of the predicate abstraction transformation of [14], we may assume that D'_π is a slice of D'. Thus, if D' has an error path as a counterexample, then it must contain a program point that does not belong to D_π (thus, it cannot be π). □

We can also prove that our approach is strictly more powerful than the RTI approach, under a reasonable assumption.

Theorem 2. *Suppose that the CHC solver used in Step 4-2 satisfies the "monotonicity property", that if the solver can find a solution for a set C of CHCs, then it can also find a solution for C' for any subset $C' \subseteq C$. Then, any program D that is proved safe in the RTI approach can also be proved safe in our new approach.*

Proof sketch. Suppose that a program D can be proved safe in the RTI approach. Let C be the set of CHCs generated from D (such that D is typable if and only if C is satisfiable). Then, since any set C' of CHCs generated in Step 4-2 is a subset of C, Step 4-3 never fails by the monotonicity assumption. By the proof of the progress property above, the CEGAR loop must eventually terminate (since there are only finitely many slices and the slice used in a previous predicate discovery phase does not occur again). □

To see that the new approach is *strictly* more powerful than RTI (even without using Step 4-4), consider the following program:

```
let twice f x = f (f x) in
let id x = x in
let neg x = -x in
let main n = if * then assert(twice id n = n)
                  else assert(twice neg n = n)
```

The program is not typable in the usual refinement type system (without intersection types) [6, 26], as we need to assign different types for the two calls of `twice`. Since the slices obtained by removing any one of the assert expressions are typable, however, Step 4-3 above can successfully find appropriate predicates, and the verification succeeds.

3 Experiments

We have implemented our method as an extension of MoCHi[1], a software model checker for higher-order programs. We use HoIce[2] [6] as the underlying CHC solver; as reported in [6], Spacer[3] [15] is generally faster, HoIce tends to infer simpler solutions, which is important for our use of CHC solving in predicate discovery. We have conducted experiments on a machine with Intel Core i7-3930K 3.20 GHz and 16 GB of memory with a timeout of 120 seconds.

We evaluated our implementation against:

- MoCHi: The original MoCHi
- RCaml: Refinement type checking and inference system based on CHC solving [27]
- RT-HoIce: Refinement type checking with HoIce base on the method described in Champion et al.'s work [6]
- RT-Spacer: Same as RT-HoIce but using Spacer as a CHC solver

The latter three are fully automated refinement type checkers for OCaml; we do not compare our tool with Liquid types [18], as it requires users to declare qualifiers. We use 262 benchmark programs written in OCaml, which consist of those taken from the benchmark sets of [6, 21, 22] and 24 programs added by ourselves[4]. Since the refinement type checkers cannot prove the unsafety of a program, (unlike the original MoCHi and our implementation, which can prove the unsafety of a given program by reporting a concrete counterexample), we use only safe programs for evaluation. All the benchmark programs are available at http://posl.ait.kyushu-u.ac.jp/~sato/.

We first compare our system with the verifiers except RCaml [27] and then compare ours with RCaml separately, because RCaml fails with errors before verification (probably due to unsupported features of OCaml) for many of our benchmark programs.

Comparison with MoCHi, RT-HoIce and RT-Spacer Figure 2 shows the result of the comparison with MoCHi, RT-HoIce, and RT-Spacer, using cactus plots. The vertical axis shows an elapsed time (measured in seconds), and the horizontal axis shows the number of instances solved in the given time. As shown in the figure, our implementation outperforms the original MoCHi and the others in terms of the number of solved instances. Within the given timeout of 120 seconds, our tool, MoCHi, RT-HoIce, and RT-Spacer

[1] http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi/
[2] https://github.com/hopv/hoice
[3] https://github.com/Z3Prover/z3
[4] They have been added in another research context to expand the benchmark set. They are not necessarily favorable for our new approach.

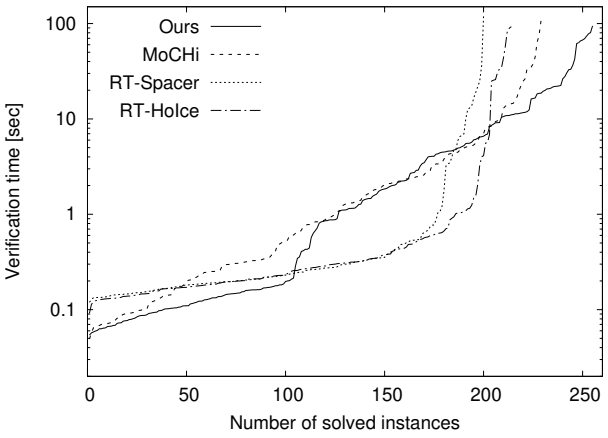


Figure 2. Comparison of our tool with MoCHi, RT-HoIce, and RT-Spacer

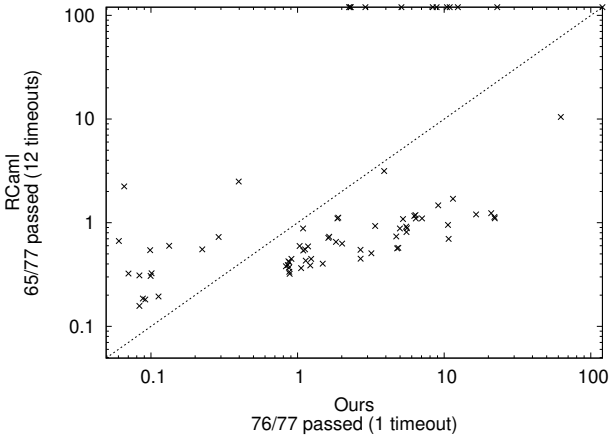


Figure 3. Comparison with RCaml

respectively could verify 255, 229, 200, and 214 programs, out of the 262 programs. More than half of the failures of RT-HoIce and RT-Spacer are due to the incompleteness of the underlying refinement type system.

For easy instances, the refinement type checkers are much faster than MoCHi and ours. For example, if we shorten the timeout from 120 seconds to one second, then our tool could verify only 126 programs, while RT-HoIce verified 185 programs.

Comparison with RCaml We next compare our tool with RCaml. The result is shown in Figure 3. We used only 76 programs, a subset of the benchmark programs above that RCaml can take as input; RCaml fails with errors for the other programs. As in the comparison with RT-HoIce and RT-Spacer, our tool could verify more programs than RCaml, although RCaml was faster than ours for many of the programs.

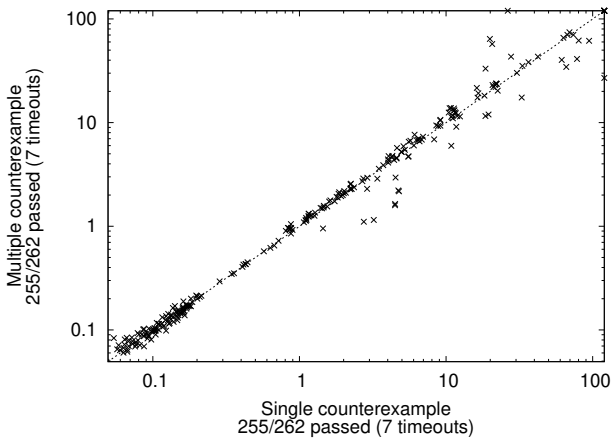


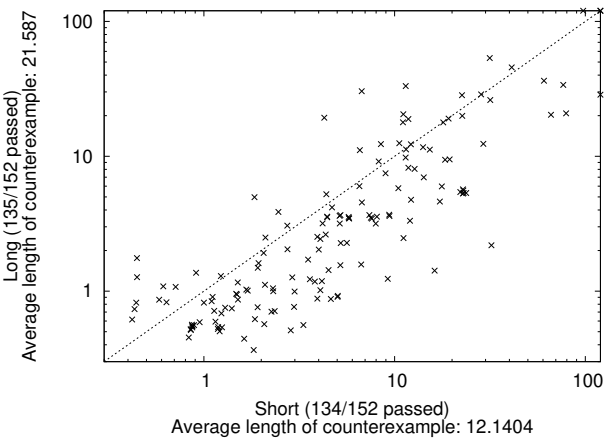
Figure 4. Comparison with the variation that uses multiple counterexamples

Single vs multiple counterexamples We have also tested two kinds of variations of our method. The first variation generates CHCs from multiple counterexamples instead of a single counterexample. More precisely, given a set of counterexamples, we generate a program slice by removing only the subterms that are not used in any of the counterexamples. This is based on the previous work on MoCHi [22], which reported that the predicate discovery was improved by using multiple counterexamples.

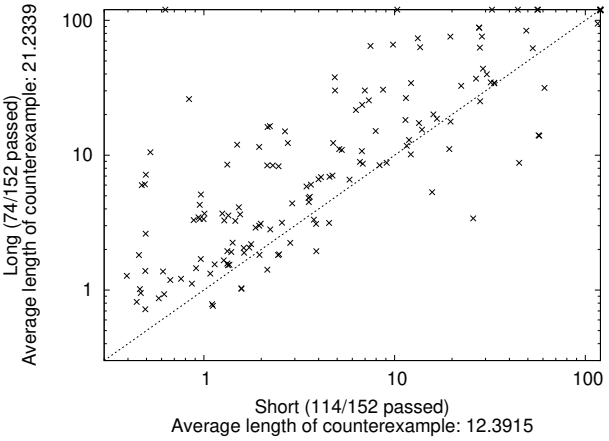
Figure 4 shows a comparison between the default version (that uses a single counterexample in the predicate discovery phase of each CEGAR loop) and the variation that uses multiple counterexamples. Contrary to our expectation, we did not observe a significant difference between them.

Short vs long counterexamples The second kind of variation is about the choice of a counterexample. There is an obvious trade-off on the length of counterexamples. Whilst a longer counterexample should be more useful for finding more general predicates (since it covers more program points), it may cost more time for CHC solving. To evaluate how the length of a counterexample affects the overall performance, we have modified Step 2 in Figure 1, so that 10 counterexamples are generated (if there are any), and created two variations that pick the shortest/longest counterexample among the 10 counterexamples. We have tested the two variations, using 152 programs, obtained from the benchmark suite above by removing (i) the programs for which the counterexample search does not terminate in 180 seconds, and (ii) the programs that are verified in the first CEGAR loop (i.e., we use only programs that need predicate discovery step at least once). The timeout has been set to 120 seconds, excluding the time for searching counterexamples.

Figure 5a shows the result of the comparison between the variations that use the shortest and longest counterexamples. Whilst the numbers of the solved instances of them are the



(a) Comparison of the variations of our method



(b) Comparison of the variations of the original method

Figure 5. Comparisons between the variations that use short and long counterexamples

same, the latter is faster than the former for 77.6 percent of the programs.

Interestingly, the effect of the lengths of counterexamples is opposite for the original MoCHi. Figure 5b shows the result of the comparison between the variations of MoCHi that use the shortest and longest counterexamples. The variation using the shortest counterexample is generally faster than the one using the longest counterexample.

4 Related Work

There has been a lot of work on verification that uses CHCs [1, 4–6, 8, 9, 12, 18–20, 26]. These methods first translate verification problems into CHC satisfiability problems (some of them via refinement type inference problems [6, 12, 18, 26, 27]), and solve them by their original methods or by off-the-shelf solvers. In contrast, our method uses CHC solving (via refinement type inference for a program slice) only for the purpose of predicate discovery, which has been the main bottleneck of the higher-order model checking approach. For verification of higher-order functional programs [6, 18], our method is strictly more powerful than CHC-based RTI approaches, as discussed in Section 2.

Terauchi [25] also uses refinement (intersection) type inference in an interesting manner. His method applies refinement intersection type inference to a recursion-free program, obtained from the original program by unfolding recursive function definitions a finite number of times. His method then checks whether the original program is typable using the inferred types. Since only recursion-free CHCs are used in his method, his method suffers from a similar problem to MoCHi; inferred types tend to be too specific to be used for typing the whole program. As in other RTI-approaches (and unlike in the HOMC approach), his method cannot disprove the safety of programs.

In the context of verification of imperative programs, there are some studies to improve a predicate discovery method, by manipulating counterexamples and controlling interpolation [2, 3, 7, 11]. For example, Beyer et al. [3] have proposed a method that generates multiple counterexamples from a single counterexample by using path slicing [11]. These methods are not directly applicable to the context of higher-order program verification; it is left for future work to combine them with our method.

Terao [23] has also proposed another refinement of the higher-order model checking approach. His proposal of lazy abstraction aims to improve bottleneck in the predicate abstraction phase (Step 1 in Figure 1). Thus, his method is orthogonal to ours; it would be useful to combine his method with ours.

5 Conclusion

We have a refinement of the higher-order model checking approach to program verification, which incorporates CHC-based refinement type inference into the predicate discovery phrase. We have implemented the proposed method and confirmed its effectiveness through experiments. As reported in the last experiment, the choice of counterexamples significantly affects the overall performance of our method; how to find good counterexamples is left for future work. A tighter integration between higher-order model checking and refinement type inference (or CHC solving) is also left for future work.

Acknowledgment

We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP15H05706 and JP18K18030.

References

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

Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. 869–882.

[2] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2015. Refinement Selection. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 20–38.

[3] Dirk Beyer, Stefan Löwe, and Philipp Wendler. 2015. Sliced Path Prefixes: An Effective Method to Enable Refinement Selection. In *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*. 228–243.

[4] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. 24–51.

[5] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 105–125.

[6] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. 365–384.

[7] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. 2010. Interpolant Strength. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*. 129–145.

[8] Cormac Flanagan. 2003. Automatic Software Model Checking Using CLP. In *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. 189–203.

[9] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 405–416.

[10] Kodai Hashimoto and Hiroshi Unno. 2015. Refinement Type Inference via Horn Constraint Optimization. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. 199–216.

[11] Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 38–47.

[12] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying Functional Programs Using Abstract Interpreters. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 470–485.

[13] Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), 20:1–20:62.

[14] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 222–233.

[15] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 846–862.

[16] C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. 81–90.

[17] C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 587–598.

[18] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 159–169.

[19] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Classifying and Solving Horn Clauses for Verification. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*. 1–21.

[20] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive Interpolants for Horn-Clause Verification. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 347–363.

[21] Ryosuke Sato and Naoki Kobayashi. 2017. Modular Verification of Higher-Order Functional Programs. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 831–854.

[22] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2013. Towards a scalable software model checker for higher-order programs. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, Rome, Italy, January 21-22, 2013*. 53–62.

[23] Taku Terao. 2018. Lazy Abstraction for Higher-Order Program Verification. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*. 23:1–23:13.

[24] Taku Terao, Takeshi Tsukada, and Naoki Kobayashi. 2016. Higher-Order Model Checking in Direct Style. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*. 295–313.

[25] Tachio Terauchi. 2010. Dependent types from counterexamples. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 119–130.

[26] Hiroshi Unno and Naoki Kobayashi. 2009. Dependent type inference with interpolants. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*. 277–288.

[27] Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating Induction for Solving Horn Clauses. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 571–591.

[28] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. 214–227.

[29] He Zhu and Suresh Jagannathan. 2013. Compositional and Lightweight Dependent Type Inference for ML. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. 295–314.