Solving Higher-Order Quantified Boolean Satisfiability via Higher-Order Model Checking

Hiroshi Unno[1], **Takeshi Tsukada**[2], **Jie-Hong Roland Jiang**[3]
[1]Tohoku University
[2]Chiba University
[3]National Taiwan University
hiroshi.unno@acm.org, t.tsukada@acm.org, jhjiang@ntu.edu.tw

Abstract

The satisfiability (SAT) problem of higher-order quantified Boolean formula (HOQBF) emerged as a natural generalization of SAT, quantified SAT, and second-order quantified SAT. It allows succinct encoding of k-EXPTIME problems beyond the reach of prior Boolean satisfiability formulations, but its application was hampered by the lack of solvers. In this paper, we present the first HOQBF solver that leverages techniques from the model-checking community. Our HOQBF solver is based on reduction to higher-order model checking, which is a generalization from model checking of while-programs to that of higher-order functional programs. The ability of a higher-order model checker to deal with higher-order functions in a program is used to reason about higher-order quantifiers in HOQBF.

1 Introduction

Higher-order quantified Boolean satisfiability (HOSAT) extends Boolean satisfiability (SAT). It allows quantification over higher-order Boolean function variables, and subsumes all the currently known SAT variants, including the satisfiability of quantified Boolean formula (QBF), dependency QBF (DQBF), and second-order QBF (SOQBF). The HOSAT problem is TOWER-complete and the higher-order quantified Boolean formula (HOQBF) is capable of succinct encoding of k-EXPTIME problems (Chistikov et al. 2022). Its usefulness has been demonstrated in encoding Presburger arithmetic for a theoretical complexity study (Chistikov et al. 2022). Other potential applications may include, e.g., memory consistency verification (Cooksey et al. 2020), planning for multi-agent systems, secure system synthesis (Jiang 2023), and solving quantified bit-vector arithmetic problems (Jonáš and Strejček 2018). Despite the potential broad applications of HOQBF, to date, there are no HOQBF solvers, not even for SOQBF.

In this work, we develop the first HOQBF solver to initiate the study for practical applications. The technical novelty lies in leveraging techniques from *higher-order model checking* for reasoning about higher-order quantifiers in HOQBFs. Higher-order model checking is a generalization from model checking of `while`-programs to that of higher-order functional programs, where functions can be passed as arguments to other functions and returned as results. Since the decidability problem of higher-order model checking being resolved by (Ong 2006), various model checkers (Kobayashi 2009a; Broadbent and Kobayashi 2013; Ramsay, Neatherway, and Ong 2014) have been actively researched and developed, with the aim of making them practical for the formal verification of higher-order functional programs, even with the high computational complexity.

Specifically, we reduce the HOSAT problem to higher-order model checking by constructing a higher-order program that returns true when the HOSAT problem is satisfiable and false otherwise. Here, the higher-order variables in the original HOQBF, introduced by the higher-order quantifiers, naturally correspond to higher-order variables in the program through the transformation, allowing a higher-order model checker to efficiently and precisely reason about higher-order functions and, as a result, higher-order quantifiers. Technically, to achieve this transformation, it is necessary to define recursive functions for enumerating the elements in the domains of higher-order quantifiers. However, since higher-order model checking can only handle finite data domains, such as Boolean or enumerated types as primitive data types, careful design is required. To address this, we inductively define and use what we call *enumeration structures* for Boolean and higher-order function types (for more details, see Section 3).

In some applications, such as synthesis (Jiang 2023), it is required not only to determine the satisfiability of a HOQBF but also to extract assignments to existential quantifiers that make the HOQBF true as Skolem functions. Therefore, this work also provides a method for extracting Skolem functions from the witness output by a higher-order model checker when it answers "yes."

We implemented the proposed reduction and combined it with the state-of-the-art higher-order model checker HorSat2 (Broadbent and Kobayashi 2013; Kobayashi 2016), realizing the first HOQBF solver, which we call HOMCSat. We evaluated the solver using a benchmark set consisting of 21 problem instances and obtained promising results while also identifying the limitations of the backend solver and suggesting directions for improvement.

By bridging these two distinct worlds of higher-order model checking and higher-order quantified Boolean logic satisfiability through our reduction, and by applying higher-

order model checking to such a novel and challenging domain, we have opened the door to mutually exchanging techniques and advancing the solving methods in both fields.

The remainder of this paper is structured as follows. Section 2 provides a brief review of higher-order quantified Boolean formulas and higher-order model checking. In Section 3, we discuss the reduction from HOSAT to higher-order model checking. Section 4 presents a method for extracting Skolem functions from the witness output by a higher-order model checker. Section 5 reports on the implementation and evaluation results. In Section 6, we compare related work, and in Section 7, we conclude the paper with remarks on future work.

2 Preliminaries

This section briefly reviews higher-order quantified boolean formulas and higher-order model checking.

2.1 Higher-Order Quantified Boolean Formula

A *higher-order quantified boolean formula* (HOQBF) is composed of Boolean operators (i.e., conjunction, disjunction and negation) and quantifiers (i.e. $\forall$ and $\exists$), just like an ordinary quantified Boolean formula. The difference is the range of quantifiers: HOQBF may have a quantifier over functions and/or over higher-order functions (i.e. functions on functions).

We use types to appropriately control higher-order functions. The syntax of *(simple) types* is given by $A, B ::= \texttt{bool} \mid A \to B$, so a type is either the boolean type $\texttt{bool}$ or a function type $A \to B$. The meaning of each type should be obvious: $\texttt{bool}$ denotes the set $[\![\texttt{bool}]\!] := \{\texttt{true}, \texttt{false}\}$ and $A \to B$ the set $[\![A \to B]\!] := [\![B]\!]^{[\![A]\!]}$ of all functions from $[\![A]\!]$ to $[\![B]\!]$. Note that $[\![A]\!]$ is finite for every A, since the denotation $[\![\texttt{bool}]\!]$ of the unique atomic type is finite.

A higher-order quantifier is of the form $\forall x^A.\varphi$ or $\exists x^A.\varphi$. These quantifiers introduce the new variable x of type A whose scope is φ. The formula $\forall x^A.\varphi$ means that φ is true no matter which element of $[\![A]\!]$ is assigned to x, and $\exists x^A.\varphi$ is true if φ is true for some assignment to x.

In the context of satisfiability testing, the *order* of a type is defined by $\mathrm{order}(\texttt{bool}) := 1$ and

$$\mathrm{order}(A_1 \to \cdots \to A_k \to \texttt{bool}) := \max_i \{\mathrm{order}(A_i) + 1\}.$$

The *order* of a formula is the maximal order of types A such that a quantifier, $\forall x^A$ or $\exists x^A$, appears in the formula.

2.2 Higher-Order Model Checking

Higher-order model checking is the problem to decide if a given program in a certain programming language satisfies a given property. This subsection defines the model-checking problem by providing the target programming language. The target programming language is a functional programming language with recursion but with two important constraints: the language is simply-typed and has only finite data types.

That the target language is *simply-typed*[1] means that each expression in the language is associated with a type. The syntax of types and their intuitive meaning are the same as in Section 2.1.

The syntax of *expressions* or *terms* is given as follows. We assume infinite sets Var^A indexed by types A such that Var^A and Var^B are disjoint if $A \neq B$. Intuitively, Var^A is the set of variables of type A, and its element is written as x (or x^A to emphasize its type). The terms are constructed by the following rules.

- If $x^A \in \mathrm{Var}^A$, then x^A is a term of type A.
- If t is a term of type A and $x^B \in \mathrm{Var}^B$, then $\lambda x^B.t$ is a term of type $B \to A$. This is a function that takes an argument of type B, evaluates t, and returns the result of the evaluation. The function body t can contain the variable x.
- If t is a term of type A, u is a term of type $B \to C$ and $f^{B\to C} \in \mathrm{Var}^{B\to C}$ is a variable of type $B \to C$, then $\texttt{let rec}\, f = u\, \texttt{in}\, t$ is a term of type A. Here t and u can contain the variable f, so $f = u$ is a recursive definition of the function f. The value of this expression is the result of the evaluation of t, which may refer to the definition $f = u$ of f when the function f is invoked during the evaluation.
- If t is a term of type $A \to B$ and u is a term of type A, then $t\, u$ is a term of type B. This is the function application.
- $\texttt{true}$ and $\texttt{false}$ are terms of type $\texttt{bool}$.
- If t is a term of type $\texttt{bool}$ and s and u are terms of type A, then $\texttt{if}\, t\, \texttt{then}\, s\, \texttt{else}\, u$ is a term of type A.

A term t of type $A_1 \to A_2 \to \cdots \to A_k \to \texttt{bool}$ is often regarded as a function taking k arguments, and we write a series of applications $t\, u_1\, u_2\, \ldots\, u_k$ as $t(u_1, \ldots, u_k)$. The logical conjunction $t\ \texttt{and}\ s$ and disjunction $t\ \texttt{or}\ s$ for $t, s : \texttt{bool}$ are defined by $\texttt{if}\, t\, \texttt{then}\, s\, \texttt{else}\, \texttt{false}$ and $\texttt{if}\, t\, \texttt{then}\, \texttt{true}\, \texttt{else}\, s$, respectively.

The variable x in $\lambda x.t$ (resp. f in $\texttt{let rec}\, f = u\, \texttt{in}\, s$) is a *binding variable* whose scope is t (resp. u and s). An occurrence of a variable x in t (resp. f in u or s) is *bound*. An unbound occurrence of a variable x is *free*. A variable in t is a *free variable* if it has a free occurrence, and we write $\mathrm{FV}(t)$ for the set of free variables. A term t is *closed* if $\mathrm{FV}(t) = \emptyset$.

A *program* is a closed term of type $\texttt{bool}$. The result of the evaluation of a program is $\texttt{true}$, $\texttt{false}$, or divergence.

The *higher-order model-checking* asks whether a given program t is not evaluated to $\texttt{false}$ (i.e. the evaluation diverges or results in $\texttt{true}$).[2] Remarkably, the higher-order model checking is decidable, despite the fact that many program verification problems are known to be undecidable,

[1] This is a technical term in the programming language community, meaning that the language has a type system that has no advanced features such as polymorphism.

[2] The higher-order model-checking studied by Ong (Ong 2006) is more general than the problem considered here. The specification used in this paper concerns only the final outcome of the evaluation of a program, but Ong studied a verification problem against temporal properties. The complexity result (Theorem 1) holds for the subproblem of this paper, as shown in (Kobayashi and Ong 2009). The target languages differ in several respects, but the gap can be filled by program transformations.

and some efficient solvers (Broadbent and Kobayashi 2013; Kobayashi 2009a; Ramsay, Neatherway, and Ong 2014) are available.

The hardness of the higher-order model checking depends on the order of a program. The *order* of a term t is the maximum order of the types of the subterms of t.

Theorem 1 ((Ong 2006; Kobayashi and Ong 2009)). *The higher-order model checking for order-n terms is $(n-1)$-EXPTIME complete.*

Since the second-order QBF belongs to 2-EXPTIME (Jiang 2023), there exists a polynomial-time many-one reduction from the second-order QBF to the order-3 higher-order model checking. This general result motivates us to utilize the algorithms and tools for the higher-order model checking to solve the second-order QBF. Unfortunately, the reduction obtained by the general result looks very inefficient. So, we will discuss a more direct reduction in the following section.

3 Reduction of HOSAT to Higher-Order Model Checking

This section reduces the satisfiability problem of higher-order quantified boolean formulas to the higher-order model checking (Ong 2006; Kobayashi 2009b, 2013). We encode higher-order variables in formulas introduced by the higher-order quantifiers by using the higher-order variables of the functional programming language.

3.1 Basic Idea and Challenge

We reduce the HOQBF solving to higher-order model checking. A given formula φ is mapped to a program P_φ that calculates the truth value of φ.

The translation is straightforward in most cases:

$$\begin{aligned} P_{\varphi\wedge\psi} &:= P_\varphi \ \texttt{and}\ P_\psi \\ P_{\varphi\vee\psi} &:= P_\varphi \ \texttt{or}\ P_\psi \\ P_{f(x_1,\ldots,x_k)} &:= f(x_1,\ldots,x_k) \\ P_{\neg\varphi} &:= \texttt{not}\, P_\varphi. \end{aligned}$$

Here `and`, `or`, and `not` are implementations of logical conjunction, disjunction, and negation. (We use the infix notations for these functions.)

The remaining constructs are quantifiers. A naïve approach is to translate the first order quantifiers $\forall x^{\texttt{bool}}$ and $\exists x^{\texttt{bool}}$ into the following programs:

$$\begin{aligned} P_{\forall x^{\texttt{bool}}.\varphi} &:= ((\lambda x.P_\varphi)\,\texttt{true})\ \texttt{and}\ ((\lambda x.P_\varphi)\,\texttt{false}) \\ P_{\exists x^{\texttt{bool}}.\varphi} &:= ((\lambda x.P_\varphi)\,\texttt{true})\ \texttt{or}\ ((\lambda x.P_\varphi)\,\texttt{false}). \end{aligned}$$

A similar approach is applicable to quantifiers $\forall f^A$ and $\exists f^A$ over arbitrary type A since the set of all elements of the type A is finite, say $\{F_0, F_1, \ldots, F_N\}$:

$$\begin{aligned} P_{\forall f^A.\varphi} &:= ((\lambda f.P_\varphi)\,F_0)\ \texttt{and}\ \cdots\ \texttt{and}\ ((\lambda f.P_\varphi)\,F_N) \\ P_{\exists f^A.\varphi} &:= ((\lambda f.P_\varphi)\,F_0)\ \texttt{or}\ \cdots\ \texttt{or}\ ((\lambda f.P_\varphi)\,F_N). \end{aligned}$$

As expected, the above translation yields a program P_φ that is evaluated to true if and only if φ is true. A problem here is that the size of P_φ can be huge for the following reasons:

1. The number of elements of $\tau = \texttt{bool} \to \cdots \to \texttt{bool}$ is 2^{2^k}, where k is the number of arguments of τ. Hence, even for 2nd-order formulas, the size of the constructed program may be hyper-exponential in size. So the translation itself is not in polynomial time.
2. The program $P_{\forall x^{\texttt{bool}}.\varphi}$ has two copies of P_φ. Hence $P_{\forall x_1^{\texttt{bool}}.\forall x_2^{\texttt{bool}}.\ldots.\forall x_n^{\texttt{bool}}.\varphi}$ has 2^n copies of P_φ, so its size cannot be bounded by a polynomial.

3.2 Compact Implementation of Quantifiers

First, for simplicity, consider quantifiers on a finite subset $A := \{0, 1, \ldots, N\}$ of natural numbers. In other words, suppose we want programs that calculate the truth values of $\forall x \in A.\varphi(x)$ and $\exists x \in A.\varphi(x)$. They can be easily described using loops: for example, a program computing the universal quantifier is given by

```
r := true
for x in 0..N:
    r := r and P_φ
return r
```

Loops and mutable variables are not in the functional programming language for the higher-order model checking, but there are known ways to simulate these features. The above program can be written as the following functional program:

$$P_{\forall x^A.\varphi} := \texttt{let rec}\ f = t\ \texttt{in}\ f(0)$$

where t is

$$\lambda x^A.P_\varphi(x)\ \texttt{and if}\ x = N\ \texttt{then true else}\ f\,(x+1).$$

The program itself can also be understood directly. The function f (or, equivalently, t since $f = t$ by the recursive definition of f) takes an integer i and computes the truth value of $\forall x \in \{i, \ldots, N\}.\varphi(x)$. The above definition of f corresponds to the logical equivalence

$$\begin{aligned} &\forall x \in \{i, \ldots, N\}.\varphi(x) \\ &\quad \Leftrightarrow \varphi(i) \wedge ((i = N) \vee (\forall x \in \{i+1, \ldots, N\}.\varphi(x))). \end{aligned}$$

It should be now clear how to describe $\exists x \in A.\varphi(x)$ in the functional programming language.

The above discussion assumes that $A = \{0, \ldots, N\}$ for some natural number N. However, on closer examination, one notices that only the following assumptions are actually used.

- The following data are programmable:
  - The minimum element $\texttt{zero} := 0$.
  - The comparison $\texttt{ismax}(x) := (x = N)$ to the maximum element $N \in A$.
  - The successor function $\texttt{succ}(x) := x + 1$.
- The maximum element N is reachable from 0 by iterative application of the successor, i.e.,

  $$\texttt{ismax}(\texttt{succ}^k(\texttt{zero})) \quad = \quad \texttt{true}$$

  for some k. Furthermore

  $$A \quad = \quad \{\texttt{zero}, \texttt{succ}(\texttt{zero}), \ldots, \texttt{succ}^k(\texttt{zero})\}.$$

Let us call this structure an *enumeration structure*. This structure allows us to program universal and existential quantifiers for a function type by the above-described method:

$$P_{\forall x^A.\varphi} := \texttt{let rec}\, f = t\, \texttt{in}\, f(\texttt{zero})$$

where t is

$$\lambda x^A.P_\varphi(x)\ \texttt{and if ismax}(x)\ \texttt{then true else}\ f\ (\texttt{succ}(x))$$

The set of boolean functions (as well as the set of boolean values) is not a subset of the natural numbers, but it has properties similar to subsets of the natural numbers in the sense that there are constants and operations as above. Details shall be discussed in the next subsection.

Assuming appropriate implementations of $\texttt{zero}_A$, $\texttt{ismax}_A$, and $\texttt{succ}_A$ for all types A, our transformation is defined by induction on the length of the logical formula, as follows:

$$\begin{aligned} P_{\varphi\wedge\psi} &:= P_\varphi\ \texttt{and}\ P_\psi \\ P_{\varphi\vee\psi} &:= P_\varphi\ \texttt{or}\ P_\psi \\ P_{f(x_1,\ldots,x_k)} &:= f(x_1,\ldots,x_k) \\ P_{\neg\varphi} &:= \texttt{not}\ P_\varphi \end{aligned}$$

and the translations of quantifiers are given by

$$P_{\forall x^A.\varphi} := \left[\begin{array}{l} \texttt{let rec f x} = P_\varphi\ \texttt{and} \\ \quad \texttt{if ismax}_A(x)\ \texttt{then} \\ \quad\quad \texttt{true} \\ \quad \texttt{else} \\ \quad\quad \texttt{f}\,(\texttt{succ}_A(x)) \\ \texttt{in} \\ \quad \texttt{f}(\texttt{zero}_A) \end{array}\right]$$

and

$$P_{\exists x^A.\varphi} := \left[\begin{array}{l} \texttt{let rec f x} = P_\varphi\ \texttt{or} \\ \quad \texttt{if ismax}_A(x)\ \texttt{then} \\ \quad\quad \texttt{false} \\ \quad \texttt{else} \\ \quad\quad \texttt{f}\,(\texttt{succ}_A(x)) \\ \texttt{in} \\ \quad \texttt{f}(\texttt{zero}_A) \end{array}\right]$$

where $\texttt{let rec}\ f\,x = t\,\texttt{in}\,s$ is an abbreviation for $\texttt{let rec}\ f = (\lambda x.t)\,\texttt{in}\,s$.

As we shall see in the next subsection, since the size of $\texttt{zero}_A$, $\texttt{ismax}_A$, and $\texttt{succ}_A$ is linear with respect to the size of A, we obtain the following result.

Theorem 2. *There exists a linear-time reduction $\varphi \mapsto P_\varphi$ from the HOQBF satisfiability problem to higher-order model checking. Furthermore, the reduction maps an order-n formula to order-$(n+1)$ model checking.*

3.3 Enumeration Structure of Each Type

We describe an implementation of an enumeration structure for each type A. The definition is by induction on A.

Enumeration Structure of Booleans The set of values $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$ of type bool has an obvious enumeration structure.

$$\begin{aligned} \texttt{zero}_{\texttt{bool}} &:= \texttt{false} \\ \texttt{ismax}_{\texttt{bool}}(x) &:= x \\ \texttt{succ}_{\texttt{bool}}(x) &:= \texttt{true}. \end{aligned}$$

The definition of $\texttt{succ}_{\texttt{bool}}$ has arbitrariness. The value of $\texttt{succ}_{\texttt{bool}}$ for the "maximum" value can be anything (since it is essentially undefined), so we can set, for example, $\texttt{succ}_{\texttt{bool}}(\texttt{true}) = \texttt{false}$ instead of the above definition (i.e. $\texttt{succ}_{\texttt{bool}}(\texttt{true}) = \texttt{true}$). So $\texttt{succ}_{\texttt{bool}}(x) := \texttt{not}(x)$ is another possible implementation.

Enumeration Structure of Functions Taking Booleans Before discussing arbitrary function types, let us first focus on function types whose argument type is bool. Let $A = \texttt{bool} \to B$ and assume an enumeration structure for B. We define $\texttt{zero}_A \colon A$, $\texttt{succ}_A \colon A \to A$, and $\texttt{ismax}_A \colon A \to \texttt{bool}$ with the expected properties.

By the induction hypothesis, the set $[\![B]\!]$ can be seen as an initial segment $\{0, 1, \ldots, N\}$ of natural numbers for some N. The idea is to identify a function $f \in [\![\texttt{bool} \to B]\!]$ with a pair $(f(\texttt{true}), f(\texttt{false})) \in [\![B]\!] \times [\![B]\!]$. So there is a bijective correspondence between functions in $[\![A]\!] = [\![\texttt{bool} \to B]\!]$ and two-digit numbers in the base-$(N+1)$ system. The idea is to define the zero and the successor for the two-digit numbers in the base-$(N+1)$ system.

The zero is expressed as $(0, 0)$. So it is the constant function to 0 in $\{0, \ldots, N\} \cong [\![B]\!]$:

$$\texttt{zero}_{\texttt{bool}\to B} := \lambda x^{\texttt{bool}}.\texttt{zero}_B.$$

The successor of (n, m) is defined as follows:

- If $m < N$, then the successor is $(n, m+1)$.
- If $m = N$ and $n < N$, then the successor is $(n+1, 0)$.
- If $m = n = N$, the successor is undefined.

This observation gives the following definition of the successor.

$$\begin{aligned} &\texttt{succ}_{\texttt{bool}\to B}(x) := \texttt{if ismax}_B(x(\texttt{false}))\ \texttt{then} \\ &\quad \lambda b^{\texttt{bool}}.\texttt{if}\, b\, \texttt{then succ}_B(x(b))\ \texttt{else zero}_B \\ &\texttt{else} \\ &\quad \lambda b^{\texttt{bool}}.\texttt{if}\, b\, \texttt{then}\, x(b)\, \texttt{else succ}_B(x(b)). \end{aligned}$$

The (n, m) is the maximum number if and only if $n = m = N$:

$$\begin{aligned} \texttt{ismax}_{\texttt{bool}\to B}(x) \quad &:= \texttt{ismax}_B(x(\texttt{true}))\ \texttt{and} \\ &\quad\ \texttt{ismax}_B(x(\texttt{false})). \end{aligned}$$

Enumeration Structure of General Function Types The above given enumeration structure of first-order function types can be easily extended to arbitrary function types. In the above discussion, we identify elements in $[\![\texttt{bool} \to B]\!]$ as two-digit numbers in the base-$(N+1)$ system, where $[\![B]\!] \cong \{0, 1, \ldots, N\}$. Here, the number of digits, 2, comes from the number of elements in $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$. For a general type C with $[\![C]\!] \cong \{0, 1, \ldots, M\}$, there is

a bijective correspondence between elements in $[\![C \to B]\!]$ and $(M+1)$-digit numbers in the base-$(N+1)$ system. So it suffices to implement the zero, the comparison with the maximum and the successor function for $(M+1)$-digit numbers in the base-$(N+1)$ system. The implementation will be slightly more cumbersome due to the increased number of digits, but there are no inherent difficulties.

4 Synthesizing Skolem Functions

The previous section provided a translation from higher-order QBFs to higher-order functional programs. The truth value of an input formula coincides with the evaluation result of the generated program, so efficient program verification tools are applicable to solve the higher-order QBF solving.

Some applications concern not only the truth or falsity of a logical formula, but also the assignment to existentially quantified variables that would make the logical formula true. Such an assignment can be expressed as *Skolem functions*.

Example 1. *Consider the formula*

$$\forall x^{\texttt{bool}}.\exists y^{\texttt{bool}}.\forall z^{\texttt{bool}}.(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z),$$

which is true. An appropriate choice for y may depend on the value of x, so it can be expressed as a function $Y\colon \texttt{bool} \to \texttt{bool}$ (taking the value of x as its argument). We seek a function Y that makes the formula

$$\forall x^{\texttt{bool}}.\forall z^{\texttt{bool}}.(x \vee Y(x) \vee z) \wedge (\neg x \vee \neg Y(x) \vee \neg z),$$

true, and

$$Y(x) \quad := \quad \neg x$$

is an example of such a function.

Example 2. *Consider the formula*

$$\forall f^{\texttt{bool}\to\texttt{bool}}.\exists g^{\texttt{bool}\to\texttt{bool}}.\exists z^{\texttt{bool}}.(f(g(z)) \leftrightarrow z),$$

which is true. An appropriate choice for g and z depends on f, so we would like to find functions $G\colon (\texttt{bool} \to \texttt{bool}) \to \texttt{bool} \to \texttt{bool}$ and $Z\colon (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}$ that make

$$\forall f^{\texttt{bool}\to\texttt{bool}}.\big(f(G(f, Z(f))) \leftrightarrow Z(f)\big)$$

true. The requirement is satisfied by

$$G(f,x) \;:=\; \top \quad \textit{and} \quad Z(f) \;:=\; f(\top).$$

Our higher-order QBF solving method based on the reduction to the higher-order model-checking program can be used to synthesize such Skolem functions. When a higher-order model checker answers "yes," it generates a witness to the correctness of the answer, from which Skolem functions can be extracted. This witness is known as a *derivation tree for a refinement intersection type system*, but we limit our discussion here to what is necessary for this paper; see, e.g., (Kobayashi 2013) for a general theory.

For simplicity, we will focus on the second-order QBF.

A witness consists of specifications that each part of the input program satisfies. The specifications for a part of `bool` type are $\{\top, \bot\}$ meaning the result of the evaluation. Specifications for function types have two kinds: *prime* ones and *collective* ones.[3] A prime specification expresses an output to a specified input: for example, a prime specification for $\texttt{bool} \to \texttt{bool} \to \texttt{bool}$ is that "the function returns $\top$ if it takes $\top$ as the first argument and $\bot$ as the second argument," which we write as $\top \to \bot \to \top$. We allow the "don't care" argument for specification: for example, $\top \to * \to \top$ means that the function returns true if its first argument is $\top$, whatever the second argument is. This specification is met by the disjunction function but not by the conjunction function since $\mathit{conj}(\top, \bot) = \bot$. A collective specification is simply a finite set of prime specifications, meaning that all prime specifications in the set is satisfied. For example, $\{\top \to * \to \bot,\ \ \bot \to * \to \top\}$ is a collective specification, and there exists a unique function that satisfies this specification (namely, the function $f(x,y) := \neg x$). A higher-order model checker returns a specification for (the parts of) the input program which suffices to prove the correctness of the answer of the model checker.

Example 3. *Recall Example 1. Let $\varphi(x,y,z) := (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$. The translation in the previous section yields a program that contains $\texttt{P}_\varphi$, which is a program with three free variables `x`, `y` and `z`. Regarding $\texttt{P}_\varphi$ as a function of type $\texttt{bool} \to \texttt{bool} \to \texttt{bool} \to \texttt{bool}$, a possible witness returned by a model checker says that $\texttt{P}_\varphi$ satisfies the following (collective) specification:*

$$\big\{\top \to \bot \to * \to \top,\quad \bot \to \top \to * \to \top\big\}.$$

Intuitively, this means that $\varphi(\top, \bot, *) = \varphi(\bot, \top, *) = \top$. The witness suggests that we should set $y = \bot$ when $x = \top$ and $y = \top$ when $x = \bot$. This is the Skolem function given in Example 1.

Our translation of a second-order QBF generates a program in which a subprogram taking a function as an argument appears. For such higher-order functions, a witness is similarly given. A slight difference is that a specification of function arguments is given as a collective specification (rather than boolean values). For example, $\Phi \to \bot \to \top$ where $\Phi = \{\top \to \top\}$ means that the function returns $\top$ if its first argument satisfies Φ and the second argument is $\bot$. An example of a function that meets this specification is $H(f,z) = f(\neg z)$.

Example 4. *Recall Example 2. Let $\psi(f,g,z) := (f(g(z)) \leftrightarrow z)$. The translation in the previous section yields a program that contains P_ψ, which is a program with three free variables `f`, `g` and `z`. Regarding $\texttt{P}_\psi$ as a function of type $(\texttt{bool} \to \texttt{bool}) \to (\texttt{bool} \to \texttt{bool}) \to \texttt{bool} \to \texttt{bool}$, a possible witness returned by a model checker says that $\texttt{P}_\psi$ satisfies the following (collective) specification:*

$$\left\{ \begin{array}{l} \{\top \to \top\} \to \{* \to \top\} \to \top \to \top, \\ \{\top \to \bot\} \to \{* \to \top\} \to \bot \to \top \end{array} \right\}.$$

This witness suggests that it is better to divide the first argument f into two cases, namely, when it satisfies $\{\top \to \top\}$ and when it satisfies $\{\top \to \bot\}$ (i.e., the case where $f(\top) =$

[3] Collective specifications are usually called *intersection types*, whereas prime specifications as *prime types*.

problem	O	V	A	result	time (s)
example1	1	3	2	SAT	0.032
example2	2	3	1	SAT	0.024
aaai23_ex1	2	5	4	SAT	0.437
aaai23_ex2	2	7	4	UNSAT	12.274
sym-asym-b	2	5	0	UNSAT	0.190
sym-asym-bb	2	9	0	T/O	N/A
sym-asym-id	2	7	1	SAT	0.070
SB-theorem	2	16	3	SAT	0.152
left-total	2	3	1	SAT	0.035
right-total	2	3	1	UNSAT	0.035
left-uniq	2	4	0	UNSAT	0.035
right-uniq	2	4	0	SAT	0.032
refl-cl-exist	2	11	2	SAT	3.415
refl-cl-uniq	2	23	2	SAT	124.717
sym-cl-exist	2	13	2	SAT	7.717
sym-cl-uniq	2	27	2	M/O	N/A
tran-cl-exist	2	15	2	SAT	30.286
tran-cl-uniq	2	31	2	M/O	N/A
f_h-neq-g_h	3	3	2	SAT	0.111
cps-arity1	3	5	4	SAT	11.785
cps-arity2	4	6	4	M/O	N/A

Table 1: Experiment results on the HOQBF benchmark set

$\top$ and the case where $f(\top) = \bot$). The witness also suggests that, in the former case, g and z can be chosen so that $g(*) = \top$ and $z = \top$, and in the latter case, $g(*) = \top$ and $z = \bot$. This is the Skolem function given in Example 2.

As the above examples demonstrate, in general, appropriate Skolem functions for a formula

$$\mathcal{Q}_1 x_1 \ldots \mathcal{Q}_n x_n . \varphi(x_1, \ldots, x_n)$$

(where $\mathcal{Q}_i \in \{\forall, \exists\}$, x_i is an ordinary or functional variable, and φ is quantifier-free) can be synthesized from the witness given to the subprogram $\mathtt{P}_\varphi$ by a higher-order model checker. The witness is a (collective) specification for a function with arguments $x_1, \ldots, x_n$ and suggests, for each universally quantified variable, an appropriate case distinction for the values for the variable and, for each existentially quantified variable, the value that the variable should be set.

5 Implementation and Evaluation

We implemented the reduction from HOQBF satisfiability (HOSAT) to higher-order model checking proposed in Section 3 and developed a HOSAT solver we call HOMCSAT[4] by using HORSAT2 (Broadbent and Kobayashi 2013; Kobayashi 2016) as the backend higher-order model checker. HOMCSAT can read Boolean, QBF, and DQBF satisfiability problems in prenex CNF, written in the DIMACS, QDIMACS, and DQDIMACS formats, respectively, and solve them by converting them to HOSAT. It also supports reading a custom format for specifying non-prenex, non-CNF HOSAT problem instances.

[4]Available from https://github.com/hiroshi-unno/coar.

We prepared a HOSAT benchmark set consisting of 21 problems to conduct preliminary experiments for evaluating HOMCSAT. The experiments were conducted on a machine with a 12th Gen Intel(R) Core(TM) i7-1270P 2.20 GHz processor and 32 GB of memory. The experimental results are summarized in Table 1. In the "problem" column, `example1` and `example2` are the HOQBFs from Examples 1 and 2, respectively, while `aaai23_ex1` and `aaai23_ex2` are from Examples 1 and 2 in (Jiang 2023), respectively. Cases `sym-asym-b` and `sym-asym-bb` assert the existence of symmetric and anti-symmetric binary relations on $\mathtt{bool}$ and $\mathtt{bool} \times \mathtt{bool}$, respectively; the former is unsatisfiable, while the latter is satisfiable. These benchmarks were designed to assess solver scalability as the depth of quantifier nesting increases, by changing the bit width of the bitvector. Case `sym-asym-id` asserts that symmetric and anti-symmetric binary relations are characterized as subsets of the identity relation, while `SB-theorem` is a HOQBF representing the Schröder-Bernstein theorem, which states that for sets A and B, if there exist injective functions from A to B and from B to A, then there exists a bijection from A to B. Cases `left-total`, `right-total`, `left-uniq`, and `right-uniq` assert that all $\mathtt{bool} \to \mathtt{bool}$ functions are, respectively, left-total, right-total, left-unique, and right-unique. However, only cases `left-total` and `right-uniq` are satisfiable. Cases `refl-cl-uniq` and `refl-cl-exists` represent the uniqueness and existence of the reflexive closure, respectively, while `sym-cl-*` and `trans-cl-*` make similar assertions regarding the symmetric and transitive closures, respectively. Case `f_h-neq-g_h` is an order-3 problem instance representing

$$\forall f^{(\mathtt{bool}\to\mathtt{bool})\to\mathtt{bool}} . \exists g^{(\mathtt{bool}\to\mathtt{bool})\to\mathtt{bool}} . \forall h^{\mathtt{bool}\to\mathtt{bool}} . \neg(f(h) \leftrightarrow g(h)).$$

Cases `cps-arity1` and `cps-arity2` assert the existence of CPS-transformed equivalent functions of type $\mathtt{bool} \to (\mathtt{bool} \to \mathtt{bool}) \to \mathtt{bool}$ and $\mathtt{bool} \to ((\mathtt{bool} \to \mathtt{bool}) \to \mathtt{bool}) \to \mathtt{bool}$, respectively, for all functions of type $\mathtt{bool} \to \mathtt{bool}$ and $\mathtt{bool} \to \mathtt{bool} \to \mathtt{bool}$. These benchmarks were intended to test solver scalability as the arity of functions increases, and consequently, the order of the CPS-transformed functions increases.

In the table, the column labeled "O" indicates the order of each problem, "V" represents the number of variables, "A" indicates the maximum number of quantifier alternations along a path from the root to a leaf in the syntax tree of the HOQBF, and in the "result" column, "SAT" and "UNSAT" show that HOMCSAT correctly judged whether the problem is satisfiable or unsatisfiable, respectively. "T/O" indicates a timeout (300 seconds), and "M/O" represents out of memory. The column labeled "time (s)" shows the time in seconds taken to conclude "SAT" or "UNSAT."

As the first HOSAT solver, HOMCSAT was able to solve various HOQBFs that express typical properties of boolean functions and binary relations, which involve higher-order quantifiers and quantifier alternations, yielding promising results. At the same time, the experimental results highlight both the limitations of HORSAT2, the backend higher-order

model checker used, and possible directions for improvement. Since HORSAT2 has primarily been applied to program verification, it struggles to scale when dealing with cases like those in our benchmark set that involve extensive branching due to numerous quantifiers, a situation uncommon in programs written by humans. This extensive branching leads to the "*intersection refinement types*," which HORSAT2 searches for as witnesses of satisfiability, becoming too large and complex to manage effectively. This is reflected in problems such as `sym-asym-b`, which involves proving the existence of a binary relation on `bool` and was solved in under 0.2 seconds, but `sym-asym-bb`, which expands the search space to binary relations on `bool` $\times$ `bool`, timed out. Similarly, out-of-memory errors occur with `sym-cl-uniq`, `trans-cl-uniq`, and `cps-arity2`. The results do not clearly reveal the relationship between parameters such as the number of quantified variables, order, and quantifier alternation depth, and the time required for solving. We believe this is because HORSAT2 tends to be peaky in performance, making it challenging to measure their scalability based on these parameters. Possible remedies include improving HORSAT2 by compactly representing the "*intersection refinement types*" using binary decision diagrams (BDDs) (Lee 1959; Bryant 1986) or zero-suppressed binary decision diagrams (ZDDs) (Minato 1993), as symbolic model checkers do, and incorporating abstraction, pruning, and propagation techniques similar to those used in SAT, QBF, and DQBF solvers. With these extensions, as the higher-order model checker scales to HOSAT instances with more quantifiers, future work could involve applying HOMCSAT to second-order solving for relaxed memory models, which, like in our current experiments, would involve reasoning about combinations of typical properties concerning n-ary relations, as explored in (Cooksey et al. 2020).

6 Related Work

HOQBF is an elevation from SOQBF by permitting quantification over variables of arbitrary order, whereas SOQBF subsumes DQBF by allowing an arbitrary number of alternations between universal and existential second-order quantification.

The satisfiability checking of SOQBF has complexity corresponding to the exponential-time hierarchy (EXPH) (Dawar, Gottlob, and Hella 1998), as was established in (Lohrey 2012; Lück 2016). The connection between SOQBF and EXPH is similar to that between QBF and the polynomial-time hierarchy (PH) (Stockmeyer 1976). Prior work (Jiang 2023) provides a sound and complete proof system for SOQBF and an algorithm for countermodel extraction from a refutation proof.

Chistikov et al. formulates HOQBF and shows its computational complexity. Essentially, HOSAT is TOWER-complete for the unbounded case and is complete for the weak k-EXP hierarchy when the formula is of order at most k and has d quantifier alternations for d being odd (Chistikov et al. 2022). In contrast, in this work, we focus on the algorithmic solving for HOQBF.

In this paper, instead of extending existing QBF and DQBF solving techniques for checking the satisfiability of HOQBFs, we applied the techniques of higher-order model checking, which have been developed independently from the SAT community, bridging the gap between the two research communities. Since Ong (2006) established the decidability of higher-order model checking, practical model checkers based on intersection refinement types or collapsible pushdown systems such as TRECS (Kobayashi 2009a), GTRECS (Kobayashi 2011), TRAVMC (Neatherway, Ramsay, and Ong 2012), C-SHORE (Broadbent et al. 2013), HORSAT (Broadbent and Kobayashi 2013), and PREFACE (Ramsay, Neatherway, and Ong 2014) have been actively researched and developed, despite the inherently high computational complexity.

Although higher-order model checking has primarily been applied to the verification of higher-order functional programs, addressing a wide range of specifications (Kuwahara et al. 2014, 2015; Murase et al. 2016) and language features (Kobayashi 2013; Kobayashi, Tabuchi, and Unno 2010; Ong and Ramsay 2011; Unno, Tabuchi, and Kobayashi 2010; Kobayashi, Sato, and Unno 2011; Sato, Unno, and Kobayashi 2013; Matsumoto, Kobayashi, and Unno 2015; Kobayashi, Lago, and Grellois 2020; Dal Lago and Ghyselen 2024; Sekiyama and Unno 2024) as program verification problems, to the best of our knowledge, this paper is the first to apply it to Boolean satisfiability problems, particularly those involving reasoning with higher-order quantifiers. As discussed in Section 5, we not only obtained promising results for this novel application but also identified the limitations of existing higher-order model checkers and proposed directions for future improvements.

7 Conclusion and Future Work

We have developed the first HOQBF solver based on the reduction to higher-order model checking. We implemented the proposed approach and evaluated it using a HOSAT benchmark set, with the higher-order model checker HORSAT2 as the backend. Despite the high computational complexity, our solver HOMCSAT successfully solved various higher-order examples, yielding promising results. At the same time, we also identified problem instances that could not be solved due to HORSAT2 being a bottleneck and discussed possible directions for improving HORSAT2. The application of higher-order model checking in this work differs significantly from previous applications, as it involves extensive branching caused by higher-order quantifiers. This makes it an important future work to introduce techniques developed for solving SAT, QBF, and DQBF, such as abstraction, pruning, and propagation, into higher-order model checking. Once these enhancements are made, we plan to explore full-scale applications such as memory consistency verification (Cooksey et al. 2020), planning for multi-agent systems, secure system synthesis (Jiang 2023), and solving quantified bit-vector arithmetic problems (Jonáš and Strejček 2018).

Acknowledgments

We thank the anonymous reviewers for their comments, which improved the paper. This research was partially supported by JSPS KAKENHI Grant Numbers JP20H04162, JP20H05703, JP24H00699, JP23K24826, and JP23K24820, and NSTC 111-2923-E-002-013-MY3 of Taiwan.

References

Broadbent, C.; Carayol, A.; Hague, M.; and Serre, O. 2013. C-SHORe: A Collapsible Approach to Higher-order Verification. In *ICFP '13*, 13–24. ACM.

Broadbent, C.; and Kobayashi, N. 2013. Saturation-Based Model Checking of Higher-Order Recursion Schemes. In *CSL '13*, volume 23 of *LIPIcs*, 129–148. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8): 677–691.

Chistikov, D.; Haase, C.; Hadizadeh, Z.; and Mansutti, A. 2022. Higher-Order Quantified Boolean Satisfiability. In *MFCS '22*, volume 241 of *LIPIcs*, 33:1–33:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Cooksey, S.; Harris, S.; Batty, M.; Grigore, R.; and Janota, M. 2020. PrideMM: Second Order Model Checking for Memory Consistency Models. In *Formal Methods. FM 2019 International Workshops*, 507–525. Springer.

Dal Lago, U.; and Ghyselen, A. 2024. On Model-Checking Higher-Order Effectful Programs. *Proceedings of the ACM on Programming Languages*, 8(POPL).

Dawar, A.; Gottlob, G.; and Hella, L. 1998. Capturing Relativized Complexity Classes without Order. *Mathematical Logic Quarterly*, 44(1): 109–122.

Jiang, J.-H. R. 2023. Second-Order Quantified Boolean Logic. In *AAAI '23*, volume 37, 4007–4015.

Jonáš, M.; and Strejček, J. 2018. On the complexity of the quantified bit-vector arithmetic with binary encoding. *Information Processing Letters*, 135: 57–61.

Kobayashi, N. 2009a. Model-checking higher-order functions. In *PPDP '09*, 25–36. ACM.

Kobayashi, N. 2009b. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL '09*, 416–428. ACM.

Kobayashi, N. 2011. A Practical Linear Time Algorithm for Trivial Automata Model Checking of Higher-Order Recursion Schemes. In *FoSSaCS '11*, volume 6604 of *LNCS*, 260–274. Springer.

Kobayashi, N. 2013. Model Checking Higher-Order Programs. *Journal of the ACM*, 60(3): 20:1–20:62.

Kobayashi, N. 2016. HorSat2. https://www-kb.is.s.u-tokyo.ac.jp/~koba/horsat2/.

Kobayashi, N.; Lago, U. D.; and Grellois, C. 2020. On the Termination Problem for Probabilistic Higher-Order Recursive Programs. *Logical Methods in Computer Science*, Volume 16, Issue 4.

Kobayashi, N.; and Ong, C.-H. L. 2009. Complexity of Model Checking Recursion Schemes for Fragments of the Modal Mu-Calculus. In *ICALP '09*, volume 5556 of *LNCS*, 223–234. Springer.

Kobayashi, N.; Sato, R.; and Unno, H. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI '11*, 222–233. ACM.

Kobayashi, N.; Tabuchi, N.; and Unno, H. 2010. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL '10*, 495–508. ACM.

Kuwahara, T.; Sato, R.; Unno, H.; and Kobayashi, N. 2015. Predicate Abstraction and CEGAR for Disproving Termination of Higher-order Functional Programs. In *CAV '15*, LNCS. Springer.

Kuwahara, T.; Terauchi, T.; Unno, H.; and Kobayashi, N. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *ESOP '14*, volume 8410 of *LNCS*, 392–411. Springer.

Lee, C. Y. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4): 985–999.

Lohrey, M. 2012. Model-checking hierarchical structures. *Journal of Computer and System Sciences*, 78(2): 461–490.

Lück, M. 2016. Complete Problems of Propositional Logic for the Exponential Hierarchy. *CoRR*, abs/1602.03050.

Matsumoto, Y.; Kobayashi, N.; and Unno, H. 2015. Automata-Based Abstraction for Automated Verification of Higher-Order Tree-Processing Programs. In *APLAS '15*, 295–312. Springer.

Minato, S.-i. 1993. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC '93*, DAC '93, 272–277. ACM.

Murase, A.; Terauchi, T.; Kobayashi, N.; Sato, R.; and Unno, H. 2016. Temporal Verification of Higher-order Functional Programs. In *POPL '16*, 57–68. ACM.

Neatherway, R. P.; Ramsay, S. J.; and Ong, C.-H. L. 2012. A Traversal-based Algorithm for Higher-order Model Checking. In *ICFP '12*, 353–364. ACM.

Ong, C.-H. L. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *LICS '06*, 81–90. IEEE.

Ong, C.-H. L.; and Ramsay, S. J. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL '11*, 587–598. ACM.

Ramsay, S. J.; Neatherway, R. P.; and Ong, C.-H. L. 2014. A Type-directed Abstraction Refinement Approach to Higher-order Model Checking. In *POPL '14*, 61–72. ACM.

Sato, R.; Unno, H.; and Kobayashi, N. 2013. Towards a scalable software model checker for higher-order programs. In *PEPM '13*, 53–62. ACM.

Sekiyama, T.; and Unno, H. 2024. Higher-Order Model Checking of Effect-Handling Programs with Answer-Type Modification. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2).

Stockmeyer, L. J. 1976. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1): 1–22.

Unno, H.; Tabuchi, N.; and Kobayashi, N. 2010. Verification of Tree-Processing Programs via Higher-Order Model Checking. In *APLAS '10*, volume 6461 of *LNCS*, 312–327. Springer.