HorSat2: A Saturation-Based Higher-Order Model Checker

Naoki Kobayashi

The University of Tokyo

Abstract. We present HorSat2, a saturation-based higher-order model checker. Higher-order model checking is a generalization of conventional model checking such as finite state or pushdown model checking, and it aims to check whether the tree generated by a given tree grammar, called a higher-order recursion scheme, satisfies a given property. Thanks to the expressiveness of higher-order recursion schemes, higher-order model checking has recently been applied to automated verification of higher-order functional programs. Like a previous higher-order model checker HorSat, HorSat2 uses Broadbent and Kobayashi's saturation-based algorithm for higher-order model checking, but we have applied a number of new optimizations. According to experiments, HorSat2 is significantly faster than other state-of-the-art higher-order model checkers, including TRecS, Preface, HorSat and HorSatZDD.

1 Introduction

Higher-order model checking [11, 6] is concerned about whether the tree generated by a given higher-order recursion scheme (HORS) satisfies a given property. HORS is a kind of tree grammar in which non-terminal symbols may take trees or higher-order functions on trees as arguments. It may be considered a generalization of conventional model checking such as finite-state/pushdown model checking, and has recently been applied to automated verification of higher-order functional programs [6, 9, 13] following the successful applications of conventional model checking to automated verification of imperative programs [3, 2].

Although the complexity of higher-order model checking is hyper-exponential (more precisely, k-EXPTIME complete for order-k HORS [11, 8]), practical higher-order model checking algorithms [6, 4, 13] have been recently developed, which run fast for typical inputs. Several higher-order model checkers, such as TRecS [6], HorSat [4], and Preface [13] have been implemented and used as backends of various automated analysis/verification tools for higher-order programs [9, 16, 12, 13, 17].

Those higher-order model checkers have been working reasonably well until recently, but some of the verification tools [10, 17] started to suffer from the bottleneck of the backend higher-order model checker. To overcome the problem, we have built a new higher-order model checker HorSat2. Like HorSat, HorSat2 is based on a saturation-based algorithm with an optimization based

on flow analysis [4], but we have employed a new flow-based optimization (inspired by Preface [13]), and applied a number of other optimizations (some of which were inspired by HorSatZDD [15]), taking the scalability into account. According to the experiments, HorSat2 is significantly faster than the previous higher-order model checkers. For example, for the $G_{k,m}$ benchmark [6, 5], it has been reported [13] that Preface could handle $G_{2,10000}$ in less than one minute, while HorSat2 can handle $G_{2,200000}$ (which consists of 200,006 rules), and $G_{7,10000}$ in less than one minute.

The rest of this paper is structured as follows. Section 2 introduces higher-order model checking and explains the functionality provided by HorSat2. Section 3 gives an overview of the algorithm and optimizations applied in HorSat2. Section 4 reports experimental comparison between HorSat2 and the previous higher-order model checkers.

2 Higher-Order Model Checking

A *higher-order recursion scheme* (HORS) is a higher-order tree grammar consisting of rewriting rules of the form

$$F\ x_1\ \cdots\ x_k \to t$$

where F is a non-terminal symbol, $x_1, \ldots, x_k$ are variables, and t is a term consisting of tree constructors, non-terminals, and variables. The part enclosed by `%BEGING` and `%ENDG` on the lefthand side of Figure 1 shows an example of HORS, written in the input format of HorSat2. Here, `br`, `open`, `read`, `close`, and `end` are tree constructors of arity 2, 1, 1, 1, and 0 respectively. The uppercase letters S and F denote non-terminals, and S is the start symbol.

There must be exactly one rule for each non-terminal. Furthermore, each non-terminal is associated with a type, and each rule must respect types. For the example above, S and F have type `o` (which is the type of trees) and $\mathtt{o} \to \mathtt{o}$.

A HORS describes a (possibly infinite) tree, obtained from S by repeatedly applying the rewriting rules. For the example in Figure 1, S can be rewritten as follows.

$$S \longrightarrow \mathtt{open}(F(\mathtt{close}\,\mathtt{end})) \longrightarrow \mathtt{open}(\mathtt{br}\,(\mathtt{read}(\mathtt{close}\,\mathtt{end}))\,(F(F(\mathtt{close}\ \mathtt{end})))) \longrightarrow$$
$$\mathtt{open}(\mathtt{br}(\mathtt{read}(\mathtt{close}\,\mathtt{end}))\,(\mathtt{br}(\mathtt{read}(F(\mathtt{close}\,\mathtt{end})))(F(F(F(\mathtt{close}\ \mathtt{end})))))) \longrightarrow \cdots$$

Thus, we obtain the infinite tree shown in the middle of Figure 1 as the limit of infinite rewriting.

Higher-order model checking [11] is the following problem:

Input: HORS $\mathcal{G}$ and a tree automaton $\mathcal{A}$
Output: Whether the tree generated by $\mathcal{G}$ is accepted by $\mathcal{A}$.

A tree automaton $\mathcal{A}$ may be an alternating parity tree automaton in general [11], but like many other higher-order model checkers, HorSat2 supports only trivial tree automata [1] (topdown tree automata with a trivial acceptance

```
%BEGING  /* HORS */
S -> open (F (close end)).
F k -> br (read k) (F (F k)).
%ENDG

%BEGINA  /* Automaton */
q0 open -> q1.
q1 br -> q1 q1.
q1 read -> q1.
q1 close -> q2.
q2 end -> .
%ENDA
```

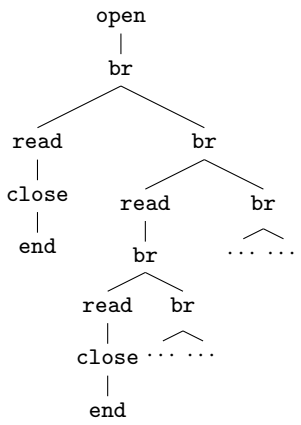

Fig. 1. A sample input for HorSat2 (left) and the tree generated by HORS (right)

condition). The part enclosed by `%BEGINA` and `%ENDA` on the lefthand side of Figure 1 shows an example of a trivial tree automaton in the input format of HorSat2. Each line defines a rewriting rule of the form: $q\ a \to q_1 \cdots q_k$, which should be read "upon reading a node labeled by a at state q, read its i-th child with q_i. A (possibly infinite) tree is accepted by an automaton $\mathcal{A}$ just if the run of $\mathcal{A}$ does not get stuck. The automaton given in Figure 1 accepts a tree if every finite path is labeled by an element of the regular expression $\texttt{open} \cdot (\texttt{read}|\texttt{br})^* \cdot \texttt{close} \cdot \texttt{end}$ (with `br` ignored) and every infinite path is labeled by a prefix of $\texttt{open} \cdot (\texttt{read}|\texttt{br})^\omega$. For the pair of HORS and tree automaton given in Figure 1, the answer to the higher-order model checking problem is "Yes". When the answer is "No", HorSat2 also outputs a counterexample, which is either a path (when the automaton is deterministic) or a finite subtree (when the automaton is alternating) that violates the property described by $\mathcal{A}$. For example, if an automaton $\mathcal{A}'$ describes the property "At most one `read` can occur above `close`", then the tree in Figure 1 is not accepted by $\mathcal{A}'$, and HorSat2 outputs "`(open,1)(br,2)(br,1)(read,1)(br,1)(read,1)(close,0)`" as a counterexample. It represents the path obtained from the tree on the righthand side of Figure 1 by taking the second branch at the first `br` node, and taking the first branch at all the other nodes, in which `read` occurs twice above `close`. More details on the input/output format of HORS are described in the distribution of HorSat2.

Higher-order model checking is k-EXPTIME complete for order-k HORS, even when restricted to trivial tree automata [8]. However, when certain parameters are fixed, it is polynomial (linear, when restricted to trivial tree automata) time in the size of HORS [6, 7]. Higher-order model checking may be considered a generalization of conventional model checking, where the order-0 case corresponds to finite state model checking, and the order-1 case corresponds to pushdown model checking,

Many verification problems for functional programs can be naturally reduced to higher-order model checking. For example, consider the following program [6]:

```
f x = if * then read x else (f x; f x)
main() = let y = open "foo" in (f y; close y)
```

The first line defines a recursive function f, which takes a file pointer x as an argument, and non-deterministically reads x, or recursively calls f twice. The second line defines a main function, which opens file "foo", calls f, and then closes the file. To check whether the file "foo" is used correctly as a read-only file, it suffices to transform the program above to a HORS that generates a tree expressing how the file is accessed by the program, and then use higher-order model checking to check that the tree represents valid usage of the file. Actually, the example of HORS given in Figure 1 is such a HORS. The tree constructors `br`, `open`, `close`, `read`, and `end` represent a non-deterministic branch, an open operation, a close operation, a read operation, and termination respectively. The rules for F and S correspond to the definitions of functions f and *main* respectively, obtained essentially by the CPS (continuation-passing-style) transformation [6]. The argument k of F corresponds to a continuation parameter, which describes how the file is accessed after a call to f returns. To check that the file "foo" is used as a read-only file, it suffices to check that every (finite) path of the tree generated by the HORS above is labeled by $\texttt{open} \cdot \texttt{read}^* \cdot \texttt{close} \cdot \texttt{end}$ (with `br` ignored). The automaton in Figure 1 describes that property.

The main advantages of using higher-order model checking (instead of conventional model checking) are: (i) HORS is much more expressive than finite state or pushdown systems; we can model complex control structures, such as higher-order recursion and exceptions accurately, and (ii) HORS is a high-level description close to a source program; as demonstrated by the example above, program verification can be easily reduced to higher-order model checking by using conventional program transformation techniques for functional programs, such as CPS transformation and λ-lifting.

3 Algorithm and Implementation

HorSat2 uses a saturation-based algorithm [4]. We briefly review the idea of the algorithm, and describe the new optimizations applied in HorSat2. The following description is admittedly cryptic; those who wish to understand the technical background may wish to consult [4]. The higher-order model checking problem can actually be reduced to the reachability problem of checking whether the start symbol S may be reduced to an element of the set **Error** of *error terms* (i.e., terms that represent invalid trees). Thus, it suffices to check whether $S \in Pre^*(\mathbf{Error})$ holds, where $Pre(U) = \{t \mid t \longrightarrow t' \in U\}$ and $Pre^*(U) = \cup_n Pre^n(U)$. The set $Pre^*(\mathbf{Error})$ is infinite in general, but can be *finitely* described by using types; one can design a type system such that $\mathbf{Terms}_\Gamma = \{t \mid \Gamma \vdash t : q_0\}$ coincides with $Pre^*(\mathbf{Error})$ for some type environment Γ. Furthermore, such Γ can be effectively computed as the least

fixedpoint (i.e, $\bigcup_n \mathcal{F}^n(\emptyset)$) of a function $\mathcal{F}$ on type environments, which satisfies: $Pre(\mathbf{Terms}_\Gamma) \subseteq \mathbf{Terms}_{\mathcal{F}(\Gamma)} \subseteq Pre^*(\mathbf{Error})$) [4].

Although the set $\bigcup_n \mathcal{F}^n(\emptyset)$ (which describes $Pre^*(\mathbf{Error})$) is finite, it is often too huge to compute. Thus, HorSat [4] applies an optimization based on flow analysis. Let **Reach** be any overapproximation of the set of reachable terms $\{t \mid S \longrightarrow^* t\}$. Then, as $S \in Pre^*(\mathbf{Error})$ if and only if $S \in Pre^*(\mathbf{Error}) \cap \mathbf{Reach}$, one can relax the condition on $\mathcal{F}$ to: $Pre(\mathbf{Terms}_\Gamma) \cap \mathbf{Reach} \subseteq \mathbf{Terms}_{\mathcal{F}(\Gamma)} \subseteq Pre^*(\mathbf{Error})$. The least fixedpoint Γ_ω of $\mathcal{F}$ then satisfies $Pre^*(\mathbf{Error}) \cap \mathbf{Reach} \subseteq \mathbf{Terms}_{\Gamma_\omega} \subseteq Pre^*(\mathbf{Error})$; hence $S \in Pre^*(\mathbf{Error})$ if and only if $S \in \mathbf{Terms}_{\Gamma_\omega}$. The design of such a function $\mathcal{F}$ is not unique. In HorSat [4], the following function is used.

$$\begin{aligned}\mathcal{F}(\Gamma) = \Gamma \cup \{F : \sigma_1 \to \cdots \to \sigma_k \to q \mid\\ F\,x_1 \cdots x_k \to t \in \mathcal{G}, t_i \in \mathbf{Flow}(x_i), \sigma_i \in \mathit{Types}_\Gamma(t_i),\\ \Gamma, x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash t : q\}\end{aligned}$$

where $\mathbf{Flow}(x_i) = \{s_i \mid C[F\,s_1 \cdots s_k] \in \mathbf{Reach}$ for some context $C\}$ (i.e., it is an overapproximation of the set of terms to which x_i may be bound), and $\mathit{Types}_\Gamma(t_i)$ is an overapproximation of the set of types of t_i under Γ. The set $\mathbf{Flow}(x_i)$ has been computed by using 0CFA [14].

The major changes in HorSat2 are as follows.

1. 0CFA has been replaced by a more accurate flow analysis, which is the same as the flow analysis used in Preface [13], except that we do not use type-based abstraction at all.
2. HorSat2 takes into account co-relations between the arguments of each non-terminal. In HorSat, if both $F\,t_{1,1}\,t_{1,2}$ and $F\,t_{1,1}\,t_{1,2}$ belong to **Reach** and $\sigma_{i,j} \in \mathit{Types}_\Gamma(t_{i,j})$, then all the types of the form $\sigma_{1,i} \to \sigma_{1,j} \to q$ (for $i, j \in \{1, 2\}$) were considered as candidates of the types of F (in other words, $F\,t_{1,1}\,t_{2,2}$ and $F\,t_{1,2}\,t_{2,1}$ are also considered to belong to **Reach**), resulting in a blow-up of the number of types. In HorSat2, only types of the form $\sigma_{1,i} \to \sigma_{1,i} \to q$ for $i \in \{1, 2\}$ are considered. To achieve this effect, the conditions $t_i \in \mathbf{Flow}(x_i) \wedge \sigma_i \in \mathit{Types}_\Gamma(t_i)$ above have been replaced by $F\,t_1 \cdots t_k \in \mathbf{Reach} \wedge (\sigma_1, \ldots, \sigma_k) \in \mathit{Types}_\Gamma(t_1, \ldots, t_k)$ (and **Reach** is computed by a more precise analysis, as mentioned above).
3. For each $F\,t_1 \cdots t_k \in \mathbf{Reach}$, the set $\mathit{Types}_\Gamma(t_1, \ldots, t_k)$ of types of $(t_1, \ldots, t_k)$ above is incrementally updated, as Γ increases.
4. Instead of adding $F : \sigma_1 \to \cdots \to \sigma_k \to q$ for every $(\sigma_1, \ldots, \sigma_k)$ such that $\Gamma, x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash t : q$, we pick, for each $F\,t_1 \cdots t_k \in \mathbf{Reach}$, just *one* tuple $(\sigma_1, \ldots, \sigma_k) \in \mathit{Types}_\Gamma(t_1, \ldots, t_k)$ such that $\Gamma, x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash t : q$, and add $F : \sigma_1 \to \cdots \to \sigma_k \to q$ to $\mathcal{F}(\Gamma)$. To ensure the soundness of this optimization, HorSat2 computes the *exact* set of types of $(t_1, \ldots, t_k)$ for $\mathit{Types}_\Gamma(t_1, \ldots, t_k)$; in HorSat, $\mathit{Types}_\Gamma(t)$ was an *overapproximation* of the set of types in t.

The second change above may blow up the size of $\mathit{Types}_\Gamma(t_1, \ldots, t_k)$ when the arity k is large; thus HorSat2 also provides an option to ignore the co-relations between arguments.

We have also applied a number of optimizations such as the replacement of list data structures with arrays and hashes in various places. The overall effect of those changes and optimizations was significant, as reported in the next section.

4 Experiments

We have compared HorSat2 with other recent model checkers, using the benchmarks of HorSatZDD [15]. The experiments were carried out on a machine with CPU Intel Xeon E5620 @ 2.40GHz and 4GB memory. For the space restriction, we show only abbreviated results. The following table shows the result for the $G_{k,m}$ and t_n benchmarks [13].[1]

	HorSat2	HorSat	Preface	HorSatZDD
$G_{2,m}$	200,000	250	10,000	20,000
$G_{k,10000}$	7	-	2	4
t_n	7,000	2,700	130	4,000

The table shows the maximum parameter that can be handled by each model checker within one minute. (For example, $200,000$ in the row $G_{2,m}$ means that $G_{2,200000}$ could be processed in less than one minute.)

The table below shows the total time for checking all the inputs (30 in total) in the second benchmark set (the upper-half of Table 1 in [15]).

HorSat2	HorSat	Preface	HorSatZDD
4.7 sec.	101.4 sec.	40.2 sec.	1318.6 sec.

Finally, we show the result for a new benchmark. The first four inputs have been borrowed from the experiments on fair termination verification of higher-order programs [10], and the latter two from verification of higher-order multi-threaded programs [17]. Times are in seconds, and the time-out was set to 300 seconds. The column "HorSat2 -m" shows the result for the optional mode to disable the optimization to keep track of the co-relations between arguments. For the first four inputs, the optional mode of HorSat2 is most effective; this is because the arities of non-terminal symbols are large in these inputs. For the latter two inputs, only HorSat2 could terminate; this is because the size of the automaton is huge, and the optimizations in HorSat2 effectively suppressed the blow-up of the size of the type environment Γ.

[1] $G_{k,m}$ is an order-k HORS of size $O(m)$, which generates a finite tree of size $\mathbf{exp}_k(m)$, where $\mathbf{exp}_0(x) = x$ and $\mathbf{exp}_{k+1}(x) = 2^{\mathbf{exp}_k(x)}$. Thus, when viewed as a state transition system, the number of states is $\mathbf{exp}_k(m)$.

	HorSat2	HorSat2 -m	HorSat	Preface	HorSatZDD
intro	3.9	0.1	1.1	time-out	8.5
intro-e	1.7	0.3	6.3	261.7	10.6
file	15.5	1.1	10.1	time-out	19.7
file-e	19.7	1.0	22.4	time-out	32.7
preach	21.9	time-out	time-out	time-out	time-out
preach-e	17.0	time-out	time-out	time-out	time-out

References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3(3) (2007)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: Integrated Formal Methods 2004. LNCS, vol. 2999, pp. 1–20. Springer (2004)
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. International Journal on Software Tools for Technology Transfer 9(5-6), 505–525 (2007)
4. Broadbent, C.H., Kobayashi, N.: Saturation-based model checking of higher-order recursion schemes. In: Proceedings of CSL 2013. LIPIcs, vol. 23, pp. 129–148 (2013)
5. Kobayashi, N.: A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In: Proceedings of FoSSaCS 2011. LNCS, vol. 6604, pp. 260–274. Springer (2011)
6. Kobayashi, N.: Model checking higher-order programs. Journal of the ACM 60(3) (2013)
7. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009. pp. 179–188. IEEE Computer Society Press (2009)
8. Kobayashi, N., Ong, C.H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. Logical Methods in Computer Science 7(4) (2011)
9. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proc. of PLDI. pp. 222–233. ACM Press (2011)
10. Murase, A., Terauchi, T., Kobayashi, N., Sato, R., Unno, H.: Temporal verification of higher-order functional programs. In: Proceedings of POPL 2016 (2016), to appear
11. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS 2006. pp. 81–90. IEEE Computer Society Press (2006)
12. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proc. of POPL. pp. 587–598. ACM Press (2011)
13. Ramsay, S., Neatherway, R., Ong, C.H.L.: An abstraction refinement approach to higher-order model checking. In: Proceedings of POPL 2014 (2014)
14. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie-Mellon University (May 1991)
15. Terao, T., Kobayashi, N.: A zdd-based efficient higher-order model checking algorithm. In: Proceedings of APLAS 2014. Lecture Notes in Computer Science, vol. 8858, pp. 354–371. Springer (2014)

16. Tobita, Y., Tsukada, T., Kobayashi, N.: Exact flow analysis by higher-order model checking. In: Proceedings of FLOPS 2012. LNCS, vol. 7294, pp. 275–289. Springer (2012)
17. Yasukata, K., Kobayashi, N., Matsuda, K.: Pairwise reachability analysis for higher order concurrent programs by higher-order model checking. In: Proceedings of CONCUR 2014. Lecture Notes in Computer Science, vol. 8704, pp. 312–326. Springer (2014)