Verifying Liveness Properties of ML Programs

ACM SIGPLAN Workshop on ML, 18th September 2011, Tokyo

M M Lester R P Neatherway C-H L Ong S J Ramsay

Department of Computer Science, University of Oxford

Abstract

Higher-order recursion schemes are a higher-order analogue of Boolean Programs; they form a natural class of abstractions for functional programs. We present a new, efficient algorithm for checking CTL properties of the trees generated by higher-order recursion schemes, which is an extension of Kobayashi's intersection type-based model checking technique. We show that an implementation of this algorithm, THORS, performs well on a number of small examples and we demonstrate how it can be used to verify liveness properties of OCaml programs. Example properties include statements such as "all opened sockets are eventually closed" and "the lock is held until the file is closed".

1. Introduction

Higher-Order Recursion Schemes (HORS) are a kind of higher-order tree grammar for generating a (potentially infinite) tree. They are in essence closed, ground-type terms of the simply-typed lambda calculus with recursion and uninterpreted first-order constants. Because to the close relationship between the lambda-calculus and functional programming languages, HORS are a natural model of computation for functional programs. They provide, in particular, an extremely accurate account of higher-order functions. Moreover, HORS are well-suited to the purpose of verification since they have a decidable mu-calculus model checking problem. That is, given a mu-calculus property ϕ and a HORS $\mathcal{G}$, the problem of whether the tree generated by $\mathcal{G}$ satisfies ϕ can be solved effectively, albeit with a rather challenging worst-case time complexity: n-EXPTIME where n is the largest order of any function in $\mathcal{G}$ [7].

Following Kobayashi [3], we aim to verify properties of a given functional program by first constructing a HORS $\mathcal{G}$ which generates the (possibly infinite) computation tree of the program—i.e. a tree whose paths represent runs of the program that are labelled by observations of interest—and then model checking $\mathcal{G}$. Kobayashi restricted his attention to checking only safety properties, but even in this more constrained setting the model-checking problem is complete for $(n-1)$-EXPTIME. However, in an attempt to perform well outside of the worst-case, a follow-up paper [4] presented an algorithm based on partial evaluation and heuristic search which was shown to work remarkably well in practice.

We have extended this approach to the verification of properties expressible in the Alternation Free Mu-Calculus (AFMC), thus allowing for the specification of *both* safety *and* liveness properties. In particular, this allows for the verification of every property expressible in the Computation Tree Logic (CTL). Our algorithm employs techniques similar to those introduced by Kobayashi [4] and, in addition, comprises a weak Büchi game solver which has been heavily optimised for our particular domain.

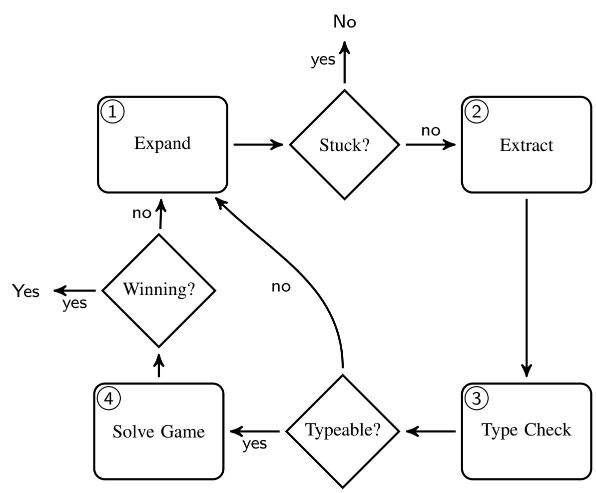

Figure 1. Algorithm in outline

We have built an implementation of our algorithm, THORS (Types for Higher Order Recursion Schemes), written in OCaml. We have used THORS to verify safety and liveness properties of a number of interesting OCaml programs; the performance in our initial experiments has been promising. THORS can be tested through a web interface at `https://mjolnir.comlab.ox.ac.uk/thors/`. A complete account is available in a technical report [6].

2. Algorithm

The core algorithm of our tool takes as input a HORS $\mathcal{G}$ and an Alternating Weak Tree automaton $\mathcal{A}$. Typically, $\mathcal{G}$ will be an abstraction of the functional program under consideration, generated from its function definitions. Meanwhile, $\mathcal{A}$ encodes the temporal logic property ϕ to be checked. The algorithm decides whether $\mathcal{A}$ has an accepting run over $\mathcal{G}$.

Our algorithm is based around an intersection type system, which is similar to that of Kobayashi and Ong [5]. Atomic types in the system are states of $\mathcal{A}$. $\mathcal{G}$ is typeable in the system if and only if $\mathcal{A}$ accepts the tree that $\mathcal{G}$ generates. Typability depends not only on every function in $\mathcal{G}$ having a valid typing, but also on there being a winning strategy in a certain parity game constructed from the types and type environments used in this typing. The size of the full parity game usually prohibits its explicit construction. However, by forbidding weakening in our type system, we have discovered that it is possible to consider only a small, relevant fragment of the game, making its construction tractable in many cases.

The structure of the algorithm is shown in Figure 1. In Stage 1, $\mathcal{G}$ is partially evaluated to obtain information about the behaviour of the program. If it is immediately apparent that a trace of the program violates the property, then the algorithm terminates and reports that the problem is a No instance. Otherwise, the algorithm proceeds to Stage 2, where heuristics are used to select candidate types for functions of $\mathcal{G}$ on the basis of the partial evaluation.

Stage 3 type-checks the functions of $\mathcal{G}$ using these types. If the candidate types are not self-consistent, types are discarded until either a consistent set is found or all types have been discarded. In the latter case, the algorithm returns to Stage 1 to evaluate the program further and obtain more candiate types. In the former case, the valid typing witnesses the existence of a run tree of $\mathcal{A}$ over $\mathcal{G}$, but does not determine whether this is accepting; intuitively, it checks safety, but not liveness.

Thus the algorithm proceeds to Stage 4, which uses the types and the corresponding type environments to construct a weak Büchi game (a kind of parity game) for which the existence of a winning strategy indicates that the run tree is accepting. If a winning strategy exists, then the algorithm terminates and the problem is a Yes instance. Otherwise, the algorithm returns to Stage 1 to evaluate the program further in an attempt to find other run trees.

If $\mathcal{A}$ is deterministic, any valid run tree is unique, so instead of looping back to Stage 1 from Stage 4, the algorithm can terminate and return No. If $\mathcal{A}$ is non-deterministic and the problem is a Yes instance, then termination is guaranteed. But on a No instance, the algorithm may loop forever. However, we can solve this problem by running a second copy of the algorithm in parallel on the complement automaton $\overline{\mathcal{A}}$ and negating its result if it terminates first.

3. Examples

We discuss two examples constructed from ML programs. The full technique translates a Resource Usage Language [2] program directly to a HORS, using a bisimulation to prove correctness. Techniques for abstraction from ML are not covered in this paper. The translation to HORS uses a CPS transform to (i) preserve ML call-by-value semantics in call-by-name HORS, and (ii) generate a computation tree of resource accesses.

3.1 Intercept

For this example, we take a network-oriented OCaml program. This program reads an arbitrary amount of data from a network socket into a queue and then forwards the data to another socket. The full program can be found online [1]; an abstracted form in ML-like syntax follows:

```
let rec g y n = for i in 1 to n do write(y); done; close(y)
let rec f x y n = if b then read(x); f(x,y,n+1)
                       else close(x); g(y,n)
let t = open_out "socket2"
let s = open_in "socket1" in f(s,t,0)
```

For this program it would be useful to confirm that if the "in" socket stops transmitting data then the "out" socket is eventually closed ($AG\, close_{in} \Rightarrow AF\, close_{out}$). In order to distingush between these two resources, the alphabet in the image of the translation includes duplicate access primitives.

$$
\begin{array}{rcl}
S & \rightarrow & \mathit{Newr}\ \mathit{C1} \\
\mathit{C1}\ x & \rightarrow & \mathit{Neww}\ (\mathit{C2}\ x) \\
\mathit{C2}\ x\ y & \rightarrow & F\ x\ y\ \mathit{Zero}\ \mathit{end} \\
F\ x\ y\ n\ k & \rightarrow & \mathit{br}\ (\mathit{Read}\ x\ (F\ x\ y\ (\mathit{Succ}\ n)\ k)) \\
 & & \quad (\mathit{Closer}\ x\ (G\ y\ n\ k) \\
G\ y\ n\ k & \rightarrow & n\ (\mathit{Write}\ y)\ (\mathit{Closew}\ y\ k) \\
I\ x\ y & \rightarrow & x\ y \\
K\ x\ y & \rightarrow & y \\
\mathit{Newr}\ k & \rightarrow & \mathit{newr}\ (k\ I) \\
\mathit{Neww}\ k & \rightarrow & \mathit{neww}\ (k\ I) \\
\mathit{Closer}\ x\ k & \rightarrow & x\ \mathit{closer}\ k \\
\mathit{Closew}\ x\ k & \rightarrow & x\ \mathit{closew}\ k \\
\mathit{Read}\ x\ k & \rightarrow & x\ \mathit{read}\ k \\
\mathit{Write}\ x\ k & \rightarrow & x\ \mathit{write}\ k \\
\mathit{Zero}\ f\ x & \rightarrow & x \\
\mathit{Succ}\ n\ f\ x & \rightarrow & f\ (n\ f\ x)
\end{array}
$$

THORS verifies that this HORS satisfies the property in 35ms. The scheme is order 4, while the property automaton has 2 states and the parity game has 31 nodes.

3.2 Unbounded file access

Our second example analyses a file with an unbounded number of file accesses. The program reads from a file for an unspecified length of time, before closing it and opening another.

```
let rec g x = if b then close(x); g(open_in n)
                   else read(x); g(x) in
let s = open_in "foo" in g(s)
```

For this program, we wish to ensure that every opening of a file is followed by a finite number of reads and a close. Note that the program itself need not terminate. In CTL, we represent this with the property $AG\,(\mathit{newr} \Rightarrow AXA(\mathit{read}\ U\,\mathit{close}))$. A translated version of the program is:

$$
\begin{array}{rcl}
S & \rightarrow & \mathit{Newr}\ (G\ \mathit{end}) \\
G\ k\ x & \rightarrow & \mathit{br}\ (\mathit{Close}\ x\ (\mathit{Newr}\ (G\ \mathit{end})))\ (\mathit{Read}\ x\ (G\ k\ x)) \\
I\ x\ y & \rightarrow & x\ y \\
K\ x\ y & \rightarrow & y \\
\mathit{Newr}\ k & \rightarrow & \mathit{brnew}\ (\mathit{newr}\ (k\ I))\ (k\ K) \\
\mathit{Close}\ x\ k & \rightarrow & x\ \mathit{close}\ k \\
\mathit{Read}\ x\ k & \rightarrow & x\ \mathit{read}\ k
\end{array}
$$

For the program to meet its specification we must, as is common when verifying liveness properties, impose a fairness constraint. Here we exclude any path containing an infinite sequence of reads, modelling an environment for our program that does not include files of infinite length.

THORS verifies that this HORS satisfies the property in 1ms. The scheme is order 4, while the property automaton has 3 states and the parity game has 17 nodes.

4. Future Work

Although THORS performs well on many examples, developing better heuristics would increase the range of programs and properties we can practically verify. Furthermore, the abstraction from OCaml programs to HORS is currently manual; it would need to be automated in a practical verification tool.

References

[1] B. Durak. `http://abaababa.ouvaton.org/caml/`, Oct. 2010.

[2] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.*, 27(2):264–313, 2005.

[3] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 416–428. ACM, 2009. ISBN 978-1-60558-379-2.

[4] N. Kobayashi. Model-checking higher-order functions. In A. Porto and F. J. López-Fraguas, editors, *PPDP*, pages 25–36. ACM, 2009. ISBN 978-1-60558-568-0.

[5] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188. IEEE Computer Society, 2009. ISBN 978-0-7695-3746-7.

[6] M. M. Lester, R. P. Neatherway, C.-H. L. Ong, and S. J. Ramsay. Model checking liveness properties of higher-order functional programs, 2011. URL `https://mjolnir.comlab.ox.ac.uk/papers/thors.pdf`.

[7] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90. IEEE Computer Society, 2006.