HECTOR: An Equivalence Checker for a Higher-Order Fragment of ML

David Hopkins[1] Andrzej S. Murawski[2] C.-H. Luke Ong[1]

[1] Department of Computer Science, University of Oxford, UK
[2] Department of Computer Science, University of Leicester, UK

Abstract. We present HECTOR, an observational equivalence checker for a higher-order fragment of ML. The input language is RML, the canonical restriction of standard ML to ground-type references. HECTOR accepts programs from a decidable fragment of RML identified by us at ICALP'11, which comprises programs of short-type (order at most 2 and arity at most 1) that may contain free variables whose arguments are also of short-type. This is an expressive fragment that contains complex higher-order types, and includes many examples from the literature which have proven challenging to verify using other methods. To our knowledge, HECTOR is the first fully-automated equivalence checker for higher-order, call-by-value programs. Both sound and complete, the tool relies on the fully abstract game semantics of RML to construct, on-the-fly, visibly pushdown automata which precisely capture program behaviour. These automata are then checked for language equivalence, and if they are inequivalent a counterexample (in the form of a separating context) is constructed.

1 Introduction

ML-like languages combine the power of higher-order functions with imperative constructs and mutable state. We consider the call-by-value language RML, which is essentially the canonical restriction of Standard ML to ground-type references. We are interested in a notion of program equivalence called observational equivalence. Two terms $\Gamma \vdash M_1, M_2$ are *observationally equivalent* just if for every program context $C[-]$ such that $\Gamma \vdash C[M_i] : \mathsf{unit}$, we have that $C[M_1]$ converges if and only if $C[M_2]$ converges. This definition says that two programs are equivalent if one can replace one by the other in any context without affecting the outcome of the computation. Observational equivalence is extremely useful when refactoring or updating code; if the updated version of a function is observationally equivalent to the older version then the changes cannot break any existing code which calls it. This makes observational equivalence an intuitively natural and practically relevant notion of equivalence. Unfortunately, it is notoriously difficult to reason about. Take the programs below.

$$F_1 \equiv \mathbf{let}\, a = \mathbf{ref}\, 0\, \mathbf{in}\, \mathbf{let}\, r = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f.(r := !r + 1; a := f(!r); r := !r - 1; !a)$$
$$F_2 \equiv \lambda f.f(1)$$

It may appear that these two terms should be equivalent, as F_1 uses local variables to return $f(1)$. However, they are separated by the term $G \equiv \lambda F.F(\lambda x.F(\lambda y.y))$. This forces a nested call of F_i. In $G\,F_1$ this call will be performed before r has been

decremented. Hence, $G\,F_1$ evaluates to 2 whereas $G\,F_2$ returns 1. However, the terms $\mathbf{let}\,c\;=\;\mathbf{ref}\,0\,\mathbf{in}\,\lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c:=0;f();c:=1;f();!c)$ and $\lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f();f();1)$ are equivalent. While the context can use nested calls in the same manner to reset the value of c to 0, any such state changes must be made in a well-bracketed manner and so the terms cannot be separated.

2 Theory and Implementation

We will make use of the fully abstract game semantics of RML [7]. This model views program execution as the playing of a game between the program and its environment. The type sequent $\Gamma \vdash \theta$ determines the rules of a two player game $[\![\Gamma \vdash \theta]\!]$ to be played between P (the program) and O (the environment). Play proceeds by the players taking it in turn to play a move (which can be either a question or an answer), equipped with a *justification pointer* to an earlier move. These pointers model the variable-to-binder and call-to-return relation within the play. We say a play is *complete* if every question has been answered. The denotation of a program $\Gamma \vdash M : \theta$ is a strategy $[\![\Gamma \vdash M]\!]$ for playing the game $[\![\Gamma \vdash \theta]\!]$. Strategies are described using a set of plays which form a playbook telling P how to play. The game model is *fully abstract* in the sense that two programs are observationally equivalent if and only if the sets of complete plays of their denotations are equal [1].

In [7] we identified the *O-strict* fragment of RML. This is the fragment for which the justification pointers from O-moves are always uniquely reconstructible from the underlying move sequence (although those from P-moves can still be ambiguous). This consists of terms-in-context of the shape $x_1 : \Theta_3, \cdots, x_n : \Theta_3 \vdash M : \Theta_2$ where Θ_2, Θ_3 are defined as follows.

$$\Theta_0 \;::=\; \mathsf{unit} \mid \mathsf{int} \qquad \Theta_2 \;::=\; \Theta_0 \mid \Theta_1 \to \Theta_0 \mid \mathsf{int\ ref}$$
$$\Theta_1 \;::=\; \Theta_0 \mid \Theta_0 \to \Theta_1 \mid \mathsf{int\ ref} \qquad \Theta_3 \;::=\; \Theta_0 \mid \Theta_2 \to \Theta_3 \mid \mathsf{int\ ref}$$

If we let a *short* type be a type of order at most two and arity at most one, then the O-strict fragment consists of programs of short types which may contain free identifiers all of whose argument types are short.

We went on to show that the strategies corresponding to terms of the O-strict fragment of RML (with finite data types) can be precisely captured using visibly pushdown automata (VPA). VPA are a subclass of pushdown automata in which the stack action (push, pop, or neither) is determined by the input letter [3]. This gives them highly desirable closure properties; in particular, language equivalence is decidable in polynomial time. Our translation from strategies to VPA allowed us to show that observational equivalence for O-strict terms is EXPTIME-complete [7].

We have now implemented our algorithm into a tool called HECTOR (Higher-order Equivalence Checker for Terms of O-strict RML). Our VPA are constructed inductively over the normal forms of the language, following [7]. Given two such VPA, using a product construction [3], it is easy to construct another to accept their symmetric difference. Then our two programs are equivalent if, and only if, the language accepted by the resulting automaton is empty. We choose to follow an on-the-fly model checking approach as this has proved successful for the game semantics based model checker

MAGE [4]. That is, when constructing our automata, we just return a function from states to the list of transitions out of that state. This function will build up the transition relation only as it is called during our exploration of the automaton. This can allow us to avoid constructing the entire automaton as we can halt the search as soon as a counter-example is found. On-the-fly reachability for pushdown systems using summary edges was described by Alur et al. [2] and we follow their approach. This essentially proceeds as a depth-first search, recording push- and pop-sites so that additional summary edges can be added when two matching transitions are found.

A web interface for HECTOR can be found at `http://mjolnir.cs.ox.ac.uk/~davh/cgi-bin/rml/input/`. Our tool allows programs to be compared, can generate separating contexts where appropriate, and can display the VPA translation of a given term, which represents its game semantics.

3 Examples and Experiments

In this section we consider a number of examples that HECTOR can handle. Where applicable we also compare its performance against HOMER, a game semantics based equivalence checker [8] for the 3rd-order fragment of Idealized Algol (IA). The main difference between RML and IA is that IA uses call-by-name evaluation (and block-allocated variables), which lead to game models that differ significantly [1]. A direct comparison between the two tools is therefore tricky, but we can attempt to use examples which have similar behaviour under both call-by-name and call-by-value evaluation. A further difference is that HOMER does not take advantage of on-the-fly construction but always builds up the entire model.

"Tricky" Examples Several examples in the literature are known to be challenging to verify. In addition to the first inequivalence in Section 1 due to Stark [12], the following have been analysed respectively by Pitts and Stark [11], and by Dreyer et al. [6].

(i) $\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c := 1; f(); !c) \;\cong\; \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f(); 1)$
(ii) $\mathbf{let}\, c = \mathbf{ref}\, 0\, \mathbf{in}\, \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(c := 0; f(); c := 1; f(); !c) \;\cong\; \lambda f^{\mathsf{unit}\to\mathsf{unit}}.(f(); f(); 1)$

They are known to be extremely tricky to prove using methods based on logical relations. All three of these examples are in the O-strict fragment and HECTOR can easily handle them as seen in the table below.

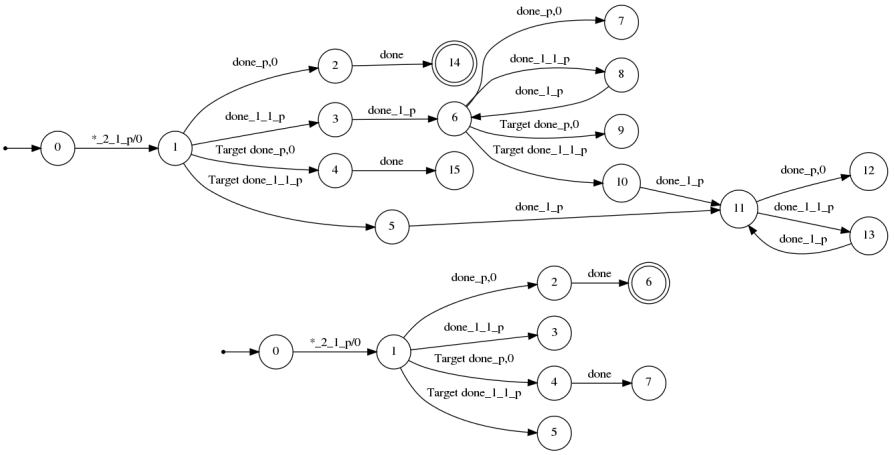
Example	Time to Compare	Time to Generate Counter-Example	State Space
(i) [11]	180ms	N.A.	67
(ii) [6]	130ms	N.A.	231
Sec. 1 [12]	150ms	50ms	57

No-Snapback Another non-obvious example is below.

$$p : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \vdash$$
$$\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, p(\lambda y.x := 1); \mathbf{if}\, !x = 1\, \mathbf{then}\, \Omega\, \mathbf{else}\, () \cong p(\lambda y.\Omega)$$

Here Ω is the term which immediately diverges. In the first term, if p ever applies its functional argument to anything then x will be assigned the value 1. This ensures

that when p terminates, the computation will diverge. Conversely, if p does not use its argument then x will have the value 0 so when p finishes the computation terminates. The effect is the same as passing p an argument which will diverge if used. The fact that they are equivalent shows that there is no term which can undo the side-effects caused by running its argument. The VPA translations of these programs as produced by HECTOR are shown below. The reachable states are somewhat different in each case as the divergence occurs at different points. However, in both cases a final state can only be reached if p's argument is never called.



Scope Extrusion Consider the following terms.

$$
\begin{aligned}
M_1 &\equiv F : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \vdash \\
&\qquad \mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, F(\lambda y.\mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1; !x) \\
M_2 &\equiv F : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \vdash \\
&\qquad F(\lambda y.\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}\, \mathbf{if}\, !x = 0\, \mathbf{then}\, x := y\, \mathbf{else}\, x := y - 1; !x) \\
M_3 &\equiv F : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \vdash F(\lambda y.y)
\end{aligned}
$$

The only difference between the first two terms is the location of the $\mathbf{let}\, x = \mathbf{ref}\, 0\, \mathbf{in}$ - binding. However, this makes a big difference to their behaviour. In the first term, the value of x persists between calls so when F calls its argument a second time the value in x will be the value of y from the first call. On the other hand, in M_2 a new reference of value 0 is allocated each time the argument is called. Hence, the guard will always be true and so we have $M_1 \ncong M_2 \cong M_3$. For the inequivalence, HECTOR generates a separating context as a counter-example. A readable version of the context produced is shown on the right. It can be seen that this binds F to a function which applies its argument twice, the first time passing it 1 and the second time 0. It then checks whether the return value from the second call is 0. When M_2 is placed in the context the check will pass as F's argument is

```
(fun f .
let _ = f (fun g .
      let _ = (g 1)
      in let z = (g 0)
      in assert((z = 0));
      3)
in ())
(fun F .[])
```

the identity function. However, when M_1 is used the check fails and so the terms are separated.

All the examples in this section can be checked by HECTOR in less than a second.

Sorting Sorting algorithms are a challenging example for any model checker due to the complex interplay between control-flow and state. We can use HECTOR to compare different sorting algorithms for equivalence. The table below compares the length of time required to check the equivalence of bubble sort and insert sort on lists of length n containing 3-valued elements. For comparison we include the time taken by HOMER, as well as the state space of the final automaton and the biggest intermediate automaton HOMER produces. As can be seen, HECTOR is outperformed by HOMER. We suspect that this is due to the added complexities of the call-by-value semantics over the call-by-name. However, we can also check the sorting algorithms when they are parameterised by a comparison function $compare : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$. In this case a malicious context could pass in a comparison function which does not act as a total order and can use this function to gain more information about the internals of the algorithm. Hence, the two programs are no longer equivalent. Due to the added size of the model when parameterised in this manner, HOMER runs out of memory for lists of length 10. On the other hand, due to the on-the-fly approach HECTOR finds the counter-example almost immediately and so does not have to construct the entire model.

n	HECTOR to Compare	Counter-example	States	HOMER	Final States	Max States
5	3s	N/A	716	1.5s	496	496
7	1min	N/A	5,000	10s	2,800	33,000
10	95min	N/A	120,000	7.5min	60,000	900,000
With A Comparison Function						
5	220ms	120ms	96	2.25min	75,000	75,000
7	225ms	225ms	132	Time Out	Time Out	Time Out
10	300ms	500ms	186	Time Out	Time Out	Time Out
15	400ms	2s	276	Time Out	Time Out	Time Out

Kierstead Terms An interesting family of higher-order terms are the Kierstead terms.

$$K_{n,i} \equiv f : ((\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}) \to \mathsf{unit} \vdash f(\lambda x_1.f(\lambda x_2. \ldots f(\lambda x_n.x_i()) \ldots))$$

For $i \neq j$, $K_{n,i} \ncong K_{n,j}$. In differentiating these terms the location of justification pointers from P-moves is critical (HECTOR uses tags on the moves to encode the location of these pointers.) We can compare the performance of HECTOR against that of HOMER on the equivalent call-by-name family of Kierstead terms. Again since this is an inequivalence, HECTOR outperforms HOMER as we do not have to construct the entire model. The timing data is shown in the table below.

n	HECTOR to Compare	Counter-example	States	HOMER	Final States	Max States
10	120ms	80ms	150	1s	74	1,400
25	140ms	200ms	366	6s	194	4,000
50	180ms	800ms	576	22s	356	7,000
100	530ms	4.5s	1,600	2min	800	18,000
200	2min	9s	37,000	7min	1,300	42,000

4 Related Work, Conclusions and Further Directions

We have presented HECTOR, an equivalence checker for a higher-order fragment of ML. Our algorithm utilises the fully abstract game semantics of RML. We believe this is the only known procedure for deciding observational equivalence of higher-order ML programs. As HECTOR is the first implementation of this algorithm, a fair comparison with existing tools is difficult. Compared with the call-by-name equivalence checker HOMER, our tool performs much better on inequivalences, thanks to the on-the-fly approach, but not as well on equivalences (which is not surprising as call-by-value game models are more complex constructions [7]). The only other game semantics based verification tool that uses on-the-fly model generation is MAGE [4], which is restricted to 2nd-order, (call-by-name) Idealized Algol programs. MAGE can only check reachability. Other tools, notably TRECS [10] and HMC [9], can verify safety properties of ML programs, but not equivalence.

In future work we hope to expand the language accepted by HECTOR. We know that observational equivalence is undecidable for most types outside the O-strict fragment but there are still a few remaining types whose decidability is unknown. It is also possible to introduce a limited form of recursion into the language, although VPA are no longer sufficiently expressive and we would require the power of DPDA. Additionally, we would like to improve the performance of HECTOR, possibly using predicate abstraction in the style of [5].

References

1. S. Abramsky and G. McCusker. Call-by-value games. In *CSL*, 1997.
2. R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS*, 2005.
3. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.
4. A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In *TACAS*, 2008.
5. A. Bakewell and D. R. Ghica. Compositional predicate abstraction from game semantics. In *TACAS*, 2009.
6. Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP*, pages 143–156, 2010.
7. D. Hopkins, A. S. Murawski, and C.-H L. Ong. A fragment of ML decidable by visibly pushdown automata. In *ICALP*, 2011.
8. D. Hopkins and C.-H L. Ong. HOMER: A higher-order observational equivalence model checker. In *CAV*, 2009.
9. Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, pages 470–485, 2011.
10. Naoki Kobayashi. Model-checking higher-order functions. In *PPDP*, pages 25–36, 2009.
11. A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, 1998.
12. I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, Univ. of Cambridge, 1995.