A Bounded Model Checking Technique for Higher-Order Programs

Yu-Yang Lin and Nikos Tzevelekos

Queen Mary University of London

Abstract. We present a Bounded Model Checking technique for higher-order programs based on defunctionalization and points-to analysis. The vehicle of our study is a higher-order calculus with general references. Our technique is a symbolic state syntactical translation based on SMT solvers, adapted to a setting where the values passed and stored during computation can be functions of arbitrary order. We prove that our algorithm is sound and provide a prototype implementation with experimental results showcasing its performance. The first novelty of our technique is a presentation of defunctionalization using nominal techniques, which provides a theoretical background to proving soundness of our technique, coupled with SSA adapted to higher-order values. The second novelty is our use of defunctionalization and points-to analysis to directly encode general higher-order functional programs.

1 Introduction

Bounded Model Checking [3] (BMC) is a model checking technique that allows for highly automated and scalable SAT/SMT-based verification and has been widely used to find errors in C-like languages [5,15,8,1]. BMC amounts to bounding the executions of programs by unfolding loops only up to a given bound, and model checking the resulting execution graph. Since the advent of CBMC [5], the mainstream approach additionally proceeds by symbolically executing program paths and gathering the resulting path conditions in propositional formulas which can then be passed on to SAT/SMT solvers. Thus, BMC performs a syntactic translation of program source code into a propositional formula, and uses the power of SAT/SMT solvers to check the bounded behaviour of programs. Being a Model Checking technique, BMC has the ability to produce *counterexamples*, which are execution traces that lead to the violation of desired properties. A specific advantage of BMC over unbounded techniques is that it avoids the full effect of state-space explosion at the expense of full verification. On the other hand, since BMC is inconclusive if the formula is unsatisfiable, it is generally regarded as a bug-finding or underapproximation technique.

The above approach has been predominantly applied to imperative, first-order languages and, while tools like CBMC can handle C++ (and, more recently, JAVA bytecode), the treatment of higher-order programs is currently limited. In particular, there is no direct analogue of the syntactic translations available for imperative languages in the higher-order case. This is what we address herein. We

propose a symbolic BMC procedure for higher-order functional and imperative programs that may contain free variables of ground type based on *defunctionalization* [20] and *points-to analysis* [2]. Our contributions include: (a) a novel syntactical translation to apply BMC to languages with higher-order methods and state; (b) a proof that the approach is sound; (c) an optimisation based on points-to analysis to improve scalability; (d) and a prototype implementation of the procedure with experimental results showcasing its performance.

As with most approaches to software BMC, we translate a given program into a propositional formula for an SMT solver to check for satisfiability, where formulas are satisfiable only if a violation is reachable within a given bound. Where in first-order programs BMC places a bound on loop unfolding, in the higher-order setting we place the bound on nested method applications. The main challenge for the translation then is the symbolic execution of paths which involve the flow of higher-order terms, by either variable binding or use of the store. We first solve the problem of higher-order store by adapting the standard technique of Static Single Assignment (SSA) to a setting where variables and references can be of higher-order. To handle higher-order terms in particular, we use an approach from operational semantics, whereby each method is uniquely identified by a name. Here, defunctionalization occurs at the semantics level, with methods being passed and stored as unique values, called *names*, during execution. This acts as a form of defunctionalization by applying each method through a repository. The following section will describe this in more detail. We capture program behaviour by also uniquely identifying every sub-term in the program tree with a return variable; analogous to how CBMC [5] captures the behaviour of sequencing commands in ANSI-C programs.

To give a simple example of our approach, consider the following code, where r is a reference of type $\mathsf{int} \to \mathsf{int}$, f is a variable bound to a method of type $\mathsf{int} \to (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$, g, h are variables of type $\mathsf{int} \to \mathsf{int}$, and n, x are variables of type int.

```
1  let f = λ x,g,h. if (x <= 0) then g else h
2  in
3  r := f n (λ x. x−1) (λ x. x+1);
4  assert(!r n >= n)
```

In the code above, a function is assigned to reference r. In a symbolic setting, it is not immediately obvious which function to call when dereferencing r in line 4. Luckily, we know that when calling f in line 3, its value can only be the one bound to it in line 1. Thus, a first transformation of the code could be:

```
3  r := if (n <= 0) then (λ x. x−1) else (λ x. x+1);
4  assert(!r n >= n)
```

The assignment in line 3 can be facilitated by using a return variable *ret* and method names for $(\lambda x.x - 1)$ and $(\lambda x.x + 1)$:

```
1  let m1 = λ x. x−1 in let m2 = λ x. x+1 in
2  let ret = if (n <= 0) then m1 else m2 in
3  r := ret;
```

```
4  assert(!r n >= n)
```

Here lies the challenge as we now need to decide how to symbolically dereference r. The simplest solution is try to match $!r$ with all existing functions of matching type, in this case $m1$ and $m2$:

```
1  let m1 = λ x. x−1 in let m2 = λ x. x+1 in
2  let ret = if (n <= 0) then m1 else m2 in
3  r := ret;
4  let ret' = match !r with
5             | m1 −> m1 n
6             | m2 −> m2 n in
7  assert(ret' >= n)
```

Performing the substitutions of $m1, m2$, we can read off the following formula for checking falsity of the assertion:

$$(ret' < n) \wedge (r = m1 \Rightarrow ret' = n - 1) \wedge (r = m2 \Rightarrow ret' = n + 1) \wedge (r = ret) \\ \wedge (n <= 0 \Rightarrow ret = m1) \wedge (n > 0 \Rightarrow ret = m2)$$

The above is true e.g. for $n = 0$, and hence the code violates the assertion.

These ideas underpin the core of our BMC translation, which is presented in Section 3 and proven sound later on. The language we examine, HOREF, is a higher-order language with general references and integer arithmetic. While correct, one can quickly see that the translation is inefficient when trying to resolve the flow of functions to references and variables. In effect, it explores all possible methods of the appropriate type that have been created so far, and relies on the solver to pick the right one. This is why a data-flow analysis is required, which we present in Section 4. We optimise the translation by restricting such choices according to a simple points-to analysis. In Section 5 we present an implementation of our technique in a BMC tool for a higher-order OCAML-like syntax extending HOREF, and test it on several example programs adapted from the MOCHI benchmark [13]. Finally, we compare our tool with MOCHI and an implementation of our bounded operational semantics in ROSETTE [24].

Related Work

While common in symbolic evaluation, defunctionalization with points-to analysis, to our knowledge, has not been used to translate entire general higher-order programs into SAT/SMT-based BMC encodings. As such, we believe we present a different and sound approach to model checking higher-order terms. BMC being a common technique, there exist several similar encodings. For example, [10,7,6] are bounded approaches based on relational logic that verify JAVA programs using SAT/SMT solvers. Being applied to JAVA, and especially prior to JDK 8, these approaches do not cope with terms and store of arbitrary order. In every case, methods are inlined statically, which is not always possible with function abstractions. In [6] we observe a case of exhaustive method application that restricts concrete method invocations by their type. This is similar in concept to our approach, but is only applied to resolve dynamic dispatch.

More common are verification tools for general higher-order programs that are not based on a direct syntactical BMC encodings. Two main techniques followed are higher-order recursion schemes modelling [12,18], and symbolic execution [21,9,11]. In the first category, MoCHi [13] performs full verification of OCaml programs by translating them into higher-order recursion schemes checked with specialised tools. In the second category, Rosette [24], Kaplan [14], and Rubicon [16] perform symbolic evaluation for Racket and Ruby respectively by using solver-aided languages for functional and imperative higher-order programs. On the other hand, the tool implemented in [17] performs a contracts-based symbolic execution that allows evaluating symbolically *arbitrarily open* higher-order programs. From these approaches, we choose MoCHi and Rosette as representatives for a comparison in Section 5.

Finally, tools based on Cbmc are inherently similar to our BMC encoding and procedure as we take inspiration from the Cbmc translation, and add symbolic defunctionalization to cope with higher-order syntax. Overall, tools based on symbolic execution are able to produce the most extensionally similar implementations, while intentionally, our approach is closer in idea to Cbmc with defunctionalization.

2 The Language: HORef

Here we present a higher-order language with higher-order global state. The syntax consists of a call-by-value λ-calculus with global references. Its types are given by the grammar: $\theta ::= \mathtt{unit} \mid \mathtt{int} \mid \theta \times \theta \mid \theta \to \theta$.

We use countably infinite sets $\mathtt{Meths}$, $\mathtt{Refs}$ and $\mathtt{Vars}$ for methods, global references and variables, ranged over by m, r and x respectively, and variants thereof; while i is for ranging over the integers. Each of these sets is typed, that is, it can be expressed as a disjoint union as follows: $\mathtt{Meths} = \biguplus_{\theta,\theta'} \mathtt{Meths}_{\theta,\theta'}$, $\mathtt{Refs} = \biguplus_{\theta} \mathtt{Refs}_{\theta}$, $\mathtt{Vars} = \biguplus_{\theta} \mathtt{Vars}_{\theta}$.

The syntax and typing rules are given in Figure 1. We assume a set of arithmetic operators $\oplus$, which we leave unspecified as they do not affect the analysis. Assertions are used for the specification of safety properties to be checked. We extend the syntax with usual constructs: $r{+}{+}$ is $r := !r + 1$, and $T; T'$ stands for $\mathtt{let}\ \_ = T\ \mathtt{in}\ T'$. Booleans are represented by 0 and $i \neq 0$.

As usual, a variable occurrence is *free* if it is not in the scope of a matching (λ/$\mathtt{let}$/$\mathtt{letrec}$)-binder. Terms are considered modulo α-equivalence and, in particular, we may assume that no variable occurs both as free and bound in the same term. We call a term *closed* if it contains no free variables.

Remark 1. By typing variable, reference and method names, we do not need to provide a context in typing judgements, this choice made for simplicity.

The references we use are **global**: a term can use references from the set $\mathtt{Refs}$ but not create them locally or pass them as arguments, and in particular there is no $\mathtt{ref}$ type. Adding ML-like local references is orthogonal to our analysis and it does not seem to present inherent difficulties (we would be treating the dynamic creation of references similarly to how we deal with method names).

$$
\begin{array}{rcl}
\texttt{Terms} \ni T & ::= & \texttt{assert}(T) \mid x \mid m \mid i \mid () \mid \langle T, T\rangle \mid T \oplus T \mid r := T \mid !r \mid \pi_1\, T \mid \pi_2\, T \mid \lambda x.T \\
& & \mid T\, T \mid \texttt{if}\ T\ \texttt{then}\ T\ \texttt{else}\ T \mid \texttt{let}\ x = T\ \texttt{in}\ T \mid \texttt{letrec}\ x = \lambda x.T\ \texttt{in}\ T \\
\texttt{Vals} \ni v & ::= & x \mid m \mid i \mid () \mid \langle v, v\rangle \\
\texttt{ECxts} \ni E & ::= & \bullet \mid \texttt{assert}(E) \mid r := E \mid E \oplus T \mid v \oplus E \mid \langle E, T\rangle \mid \langle v, E\rangle \mid \pi_j E \mid E\, T \mid v\, E \\
& & \mid \texttt{let}\ x = E\ \texttt{in}\ T \mid \texttt{if}\ E\ \texttt{then}\ T\ \texttt{else}\ T \mid (\!|E|\!) \\
\texttt{CForms} \ni M & ::= & \texttt{assert}(v) \mid v \mid r := v \mid !r \mid v \oplus v \mid \pi_1\, v \mid \pi_2\, v \mid x\ v \mid m\ v \mid \lambda x.M \\
& & \mid \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ M \mid \texttt{let}\ x = M\ \texttt{in}\ M \mid \texttt{letrec}\ x = \lambda x.M\ \texttt{in}\ M
\end{array}
$$

$$
\frac{T : \texttt{int}}{\texttt{assert}(T) : \texttt{unit}} \quad \frac{}{() : \texttt{unit}} \quad \frac{}{i : \texttt{int}} \quad \frac{x \in \texttt{Vars}_\theta}{x : \theta} \quad \frac{m \in \texttt{Meths}_{\theta,\theta'}}{m : \theta \to \theta'} \quad \frac{T : \texttt{int} \qquad T_0, T_1 : \theta}{\texttt{if}\ T\ \texttt{then}\ T_1\ \texttt{else}\ T_0 : \theta}
$$

$$
\frac{T_1, T_2 : \texttt{int}}{T_1 \oplus T_2 : \texttt{int}} \quad \frac{T_1 : \theta_1 \qquad T_2 : \theta_2}{\langle T_1, T_2\rangle : \theta_1 \times \theta_2} \quad \frac{\langle T_1, T_2\rangle : \theta_1 \times \theta_2}{\pi_i \langle T_1, T_2\rangle : \theta_i} \quad \frac{r \in \texttt{Refs}_\theta}{!r : \theta} \quad \frac{r \in \texttt{Refs}_\theta \qquad T : \theta}{r := T : \texttt{unit}}
$$

$$
\frac{T' : \theta \to \theta' \qquad T : \theta}{T'\, T : \theta'} \quad \frac{T : \theta' \qquad x : \theta}{\lambda x.T : \theta \to \theta'} \quad \frac{x, T : \theta \qquad T' : \theta'}{\texttt{let}\ x = T\ \texttt{in}\ T' : \theta'} \quad \frac{x, \lambda y.T : \theta \to \theta'' \qquad T' : \theta'}{\texttt{letrec}\ x = \lambda y.T\ \texttt{in}\ T' : \theta'}
$$

Fig. 1. Grammar and typing rules for HOREF.

On the other hand, methods are dynamically created during execution, and for that reason we will be frequently referring to them simply as names. The terminology comes from nominal techniques [19]. On a related note, λ-abstractions are not values in our language. This is due to the fact that in the semantics these get evaluated to method names.

Bounded Operational Semantics We next present a bounded operational semantics for HOREF, which we capture with our bounded BMC routine. The semantics is parameterised by a bound k which, similarly to loop unwinding in procedural languages, limits the depth of method (i.e. function) calls within an execution. A bound $k = 0$ in particular means that, unless no method calls are made, execution will terminate with no return value. Consequently, in this bounded operational semantics, all programs must halt. Note at this point that the unbounded semantics of HOREF, allowing arbitrary recursion, can be obtained e.g. by allowing bound values $k = \infty$.

The bounded operational semantics is given in Figure 2. It is defined by means of a small-step transition relation with transitions of the form:

$$(T, R, S, k) \to (T', R', S', k')$$

It uses *values* and *evaluation contexts*, which are in turn defined in Figure 1. By abuse of notation, we extended the term syntax to be able to mark nested method calls explicitly, by use of *evaluation boxes* of the form $(\!|...|\!)$. We use this to correctly bound nested function calls.

A ***configuration*** is a quadruple (T, R, S, k) where T is a typed term and:

$$(\texttt{assert}(j), R, S, k) \to ((), R, S, k) \qquad (!r, R, S, k) \to (S(r), R, S, k)$$
$$(r := v, R, S, k) \to ((), R, S[r \mapsto v], k) \qquad (\pi_i\langle v_1, v_2\rangle, R, S, k) \to (v_i, R, S, k)$$
$$(i_1 \oplus i_2, R, S, k) \to (i, R, S, k) \quad (i = i_1 \oplus i_2) \qquad (\lambda x.T, R, S, k) \to (m, R[m \mapsto \lambda x.T], S, k)$$
$$(\texttt{if } j \texttt{ then } T_1 \texttt{ else } T_0, R, S, k) \to (T_1, R, S, k) \qquad (\texttt{if } 0 \texttt{ then } T_1 \texttt{ else } T_0, R, S, k) \to (T_0, R, S, k)$$
$$(\texttt{let } x = v \texttt{ in } T, R, S, k) \to (T\{v/x\}, R, S, k) \qquad (\langle\!|v|\!\rangle, R, S, k) \to (v, R, S, k+1)$$
$$(mv, R, S, k) \to (\langle\!|T\{v/x\}|\!\rangle, R, S, k-1) \quad (R(m) = \lambda x.T)$$
$$(\texttt{letrec } f = \lambda x.T \texttt{ in } T', R, S, k) \to (T'\{m/f\}, R[m \mapsto \lambda x.T], S, k)$$
$$(E[T], R, S, k) \to (E[T'], R', S', k') \quad \text{where } (T, R, S, k) \to (T', R', S', k')$$

Fig. 2. Bounded operational semantics rules. In all cases, $k \neq \texttt{nil}$ and $j \neq 0$.

- $R : \texttt{Meths} \rightharpoonup \texttt{Terms}$ is a finite map, called a ***method repository***, such that for all $m \in \mathrm{dom}(R)$, if $m \in \texttt{Meths}_{\theta\to\theta'}$ then $R(m) = \lambda x.T : \theta \to \theta'$.
- $S : \texttt{Refs} \rightharpoonup \texttt{Vals}$ is a finite map, called a ***store***, such that for all $r \in \mathrm{dom}(S)$, if $r \in \texttt{Refs}_\theta$ then $S(r) : \theta$.
- $k \in \{\texttt{nil}\} \cup \mathbb{N}$ is the nested calling bound, where decrementing k beyond zero results in $\texttt{nil}$.

A closed configuration is one whose components are all closed. We call a configuration (T, R, S, k) ***valid*** if all methods and references appearing in T, R, S are included in $\mathrm{dom}(R)$ and $\mathrm{dom}(S)$ respectively. We also call a transition sequence $(T, R, S, k) \twoheadrightarrow (T', R', S', k')$ *valid*, where $\twoheadrightarrow$ is the reflexive and transitive closure of $\to$, if T' is a value. Note that failing an assertion results to a stuck configuration. Thus, no assertions can be violated in a valid transition sequence. Moreover, we can see that all terms must eventually evaluate to a value, or fail an assertion, or consume the bound and reach $\texttt{nil}$.

3 A Bounded Translation for HORef

We next present an algorithm which, given an initial configuration, produces a tuple containing propositional formulas and context components that capture its bounded semantics. Without loss of generality, we define the translation on terms in ***canonical form***, ranged over by M and variants, which are presented in Figure 1. This provision is innocuous as transforming a term in canonical form can be achieved in linear time with standard methods.

The algorithm receives a valid configuration (M, R, S, k) as input, where M is in canonical form and may only contain free variables of ground type, and proceeds to perform the following sequence of transformations:

$$(M, R, S, k) \xmapsto{init} (M, R, C_S, C_S, \phi_S, \top, \top, k) \xmapsto{[\![\cdot]\!]} (ret, \phi, R', C, D, \alpha, pc)$$

The first step is an initialisation step that transforms the tuple in the form appropriate for the main translation $[\![\cdot]\!]$, which is the essence of the entire bounded translation. We proceed with $[\![\cdot]\!]$ and will be returning to *init* later on.

$[\![\cdot]\!]$ operates on symbolic configurations of the form $(M, R, C, D, \phi, \alpha, pc, k)$, where M and R are a term and a repository respectively, k is the bound, and:

- $C, D : \texttt{Refs} \rightharpoonup \texttt{SSAVars}$ are single static assignment (SSA) maps where $\texttt{SSAVars}$ is the set of variables of the form r_i (for each $r \in \texttt{Refs}$), such that i is the number of times that r has been assigned to so far. The map C is counting all the assignments that have taken place so far, whereas D only counts those in the current path. E.g. $C(r) = r_5$ if r has been assigned to five times so far in every path looked at. We write $C[r]$ to mean *update* C *with reference* r: if $C(r) = r_i$, then $C[r] = C[r \mapsto r_{i+1}]$, where r_{i+1} is fresh.
- ϕ is a propositional formula containing the (total) behaviour so far.
- α is a propositional formula consisting of a conjunction of statements representing assertions that have been visited by $[\![\cdot]\!]$ so far.
- pc is the path condition that must be satisfied to reach this configuration.

The translation returns tuples of the form $(ret, \phi, R, C, D, \alpha, pc)$, where:

- $\phi, R, C, D, \alpha, pc$ have the same reading as above, albeit for *after reaching the end of all paths from the term* M.
- ret is a propositional variable representing the return value of the initial configuration.

Finally, returning to *init*, we have that:

- the initial SSA maps C_S simply map each r in the domain of S to the SSA variable r_0;
- ϕ_S stipulates that each r_0 be equal to its corresponding value $S(r)$;
- since there is no computation preceding M, its α and pc are simply $\top$ (true).

The BMC translation is given in Figure 3. In all cases in the figure, ret is a fresh variable and $k \neq \texttt{nil}$. We also assume a common domain $\Pi = dom(C) = dom(D) \subseteq \texttt{Refs}$, which contains all references that appear in M and R.

The translation stops when either the bound is $\texttt{nil}$, or when every path of the given term has been explored completely. The base cases add clauses mapping return variables to actual values of evaluating M. Inductive cases build the symbolic trace of M by recording in ϕ all changes to the store, and adding clauses for return variables at each sub-term of the program, thus building a control flow graph by relating said return variables and chaining them together in the formula. Wherever branching occurs, the chaining is guarded.

In the translation, defunctionalization occurs because every method call is replaced with a call to the repository using its respective name as an argument. Because this is a symbolic setting, however, it is possible to lose track of the specific name desired. Particularly, when applying variables as methods (xv, with $x : \theta$), we encode in the behaviour formula an n-ary decision tree where n is the number of methods to consider. In such cases, we assume that x could be any method in the repository R. We call this case ***exhaustive method application***. This case seems to be fundamental for applying BMC to higher-order terms with

state, and is necessary for defunctionalization. To explore plausible paths only, we restrict R to type θ (denoted $R \restriction \theta$). In Section 4) we will be applying a points-to analysis to restrict this further.

To illustrate the algorithm, we look at a few characteristic cases:

[nil case] When the translation consumes its bound, we end up translating some $[\![M, R, C, D, \phi, \texttt{nil}]\!]$. In this case, we simply return a fresh variable ret representing the final value, and stipulate in the program behaviour that ret is equal to some default value (the latter is needed to ensure a unique model for ret). The breaching of the bound is recorded as an assertion violation, and a reserved propositional variable `inil` us used for that purpose. The returned path condition is set to false.

[let case] In $[\![\text{let } x = M \text{ in } M', R, C, D, \phi, k]\!]$, we first compute the translation of M. Using the results of said translation, we can substitute in M' the fresh variable ret_1 for x, and compute its translation. In the latter step, we also feed the updated repository R_1, SSA maps C_1 and D_1, program behaviour ϕ_1, assertions α_1 (actually, a conjunction of assertions), and the accumulated path condition $pc \wedge pc_1$. To finish, we return ret_2 and the newly updated repository R_2, SSA maps, C_2 and D_2, assertions α_2. The path condition returned is $pc_1 \wedge pc_2$, reflecting the new path conditions gathered.

[xv case] In $[\![xv, R, C, D, \phi, k]\!]$ we see exhaustive method application in action. We first restrict the repository R to type θ to obtain the set of names identifying all methods of matching type for x. If no such methods exist, this means that the binding of x had not succeeded due to breaching the bound earlier on, so `defval` is returned. Otherwise, for each method m_i in this set, we obtain the translation of applying m_i to the argument v. This is done by substituting v for y_i in the body of m_i. After translating all method applications, all paths are joined in ψ, by constructing an n-ary decision tree that includes the state of the store in each path. We do this by incrementing all references in C_n, and adding the clauses $C'_n = D_i(r)$ for each path. These paths are then guarded by the clauses $(x = m_i)$. Finally, we return a formula that includes ψ and the accumulation of all intermediate ϕ_i's, the accumulation of repositories, the final SSA map, accumulation of assertions and new path conditions. Note that we return C'_n as both the C and D resulting from translating this term. This is because all branches have been joined, and any term sequenced after this one should have all updates available to it.

Remark 2. The difference between reading (D) and writing (C) is noticeable when branching. Branching can occur in two ways: through a conditional statement, and by performing symbolic method application where we have lost track of the concrete method; more precisely, when M is of the form xv. In the former case, we branch according to the return value of the condition (denoted by ret_b), and each branch translates M_0 and M_1 respectively. In this case, both branches read from the same map D_b, but may contain different assignments, which we accumulate in C. The formula $\psi_0 \wedge \psi_1$ then encodes a binary decision node in the control flow graph through guarded clauses that represent the path conditions. Similar care is taken with branching caused by symbolic method application.

Base Cases:

- $[\![\texttt{assert}(v), R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = ()) \wedge \phi, R, C, D, (pc \implies (v \neq 0)) \wedge \alpha, \top)$
- $[\![M, R, C, D, \phi, \alpha, pc, \texttt{nil}]\!] = (ret, (ret = \texttt{defval}) \wedge \phi, R, C, D, \alpha \wedge (pc \implies \texttt{inil}), \bot)$
- $[\![v, R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = v) \wedge \phi, R, C, D, \alpha, \top)$
- $[\![!r, R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = D(r)) \wedge \phi, R, C, D, \alpha, \top)$
- $[\![\lambda x.M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = m) \wedge \phi, R', C, D, \alpha, \top)$
  where $R' = R[m \mapsto \lambda x.M]$ and m fresh
- $[\![\pi_i\, v, R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = \pi_i\, v) \wedge \phi, R, C, D, \alpha, \top)$
- $[\![v_1 \oplus v_2, R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = v_1 \oplus v_2) \wedge \phi, R, C, D, \alpha, \top)$
- $[\![r := v, R, C, D, \phi, \alpha, pc, k]\!] =$ let $C' = C[r]$ in let $D' = D[r \mapsto C'(r)]$ in
  $(ret, ((ret = ()) \wedge (D'(r) = v)) \wedge \phi, R, C', D', \alpha, \top)$

Inductive Cases:

- $[\![\texttt{let}\ x = M\ \texttt{in}\ M', R, C, D, \phi, \alpha, pc, k]\!] =$
  let $(ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = [\![M, R, C, D, \phi, \alpha, pc, k]\!]$ in
  let $(ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_2) = [\![M'\{ret_1/x\}, R_1, C_1, D_1, \phi_1, \alpha_1, pc \wedge pc_1, k]\!]$ in
  $(ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_1 \wedge pc_2)$
- $[\![\texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M', R, C, D, \phi, \alpha, pc, k]\!] =$
  let m, f' be fresh in
  let $R' = R[m \mapsto \lambda x.M\{f'/f\}]$ in $[\![M'\{f'/f\}, R', C, D, \phi \wedge (f' = m), \alpha, pc, k]\!]$
- $[\![m\, v, R, C, D, \phi, \alpha, pc, k]\!] =$
  let $R(m)$ be $\lambda x.N$ in $[\![N\{v/x\}, R, C, D, \phi, \alpha, pc, k-1]\!]$
- $[\![\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, C, D, \phi, \alpha, pc, k]\!] =$
  let $(ret_0, \phi_0, R_0, C_0, D_0, \alpha_0, pc_0) = [\![M_0, R, C, D, \phi, \alpha, pc \wedge (v = 0), k]\!]$ in
  let $(ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = [\![M_1, R_0, C_0, D_b, \phi_0, \alpha_0, pc \wedge (v \neq 0), k]\!]$ in
  let $C' = C_1[r_1] \cdots [r_n]$ $(\Pi = \{r_1, \ldots, r_n\})$ in
  let $\psi_0 = (v = 0) \implies ((ret = ret_0) \wedge \bigwedge_{r \in \Pi} (C'(r) = D_0(r)))$ in
  let $\psi_1 = (v \neq 0) \implies ((ret = ret_1) \wedge \bigwedge_{r \in \Pi} (C'(r) = D_1(r)))$ in
  $(ret, \psi_0 \wedge \psi_1 \wedge \phi_1, R, C', C', \alpha_1, ((pc_0 \wedge (v = 0)) \vee (pc_1 \wedge (v \neq 0))))$
- $[\![x^{\theta}\, v, R, C, D, \phi, \alpha, pc, k]\!] =$
  if $R \restriction \theta = \emptyset$ then $(ret, (ret = \texttt{defval}) \wedge \phi, R, C, D, \alpha, \bot)$ else
  let $R \restriction \theta$ be $\{m_1, ..., m_n\}$ and (R, C, ϕ, α) be $(R_0, C_0, \phi_0, \alpha_0)$ in
  for each $i \in \{1, ..., n\}$:
  let $R(m_i)$ be $\lambda y_i.N$ in let $(ret_i, \phi_i, R_i, C_i, D_i, \alpha_i, pc_i) =$
  $[\![N_i\{v/y_i\}, R_{i-1}, C_{i-1}, D, \phi_{i-1}, \alpha_{i-1}, pc \wedge (x = m_i), k-1]\!]$ in
  let $C'_n = C_n[r_1] \cdots [r_j]$ $(\Pi = \{r_1, \ldots, r_j\})$ in
  let $\psi = \bigwedge_{i=1}^{n} \left((x = m_i) \implies ((ret = ret_i) \wedge \bigwedge_{r \in \Pi} (C'_n(r) = D_i(r))) \right)$ in
  let $pc'_n = \bigvee_{i=1}^{n} (pc_i \wedge (x = m_i))$ in $(ret, \psi \wedge \phi_n, R_n, C'_n, C'_n, \alpha_n, pc'_n)$

Fig. 3. The BMC translation.

```
r := r0;
letrec f =
  λ x. if x then (r++; f (x − 1))
  else (λ y. assert (y = !r + x))
in let g = f n in g n
```

$$
\begin{aligned}
\phi = {} & (r_1 = r_0) \wedge (ret_4 = m_2) \wedge \\
& (ret_3 = (n = 0)?ret_4 : ret_5) \wedge \\
& (r_2 = r_1 + 1) \wedge (ret_6 = m_3) \wedge \\
& (ret_5 = (n - 1 = 0)?ret_6 : ret_7) \wedge \\
& (r_3 = r_2 + 1) \wedge (ret_7 = \texttt{defval}) \wedge \\
& (ret_3 = \texttt{defval}) \implies (ret = ()) \wedge \\
& (ret_3 = m_3) \implies (ret = ()) \wedge \\
& (ret_3 = m_2) \implies (ret = ())
\end{aligned}
$$

$$
\begin{aligned}
\alpha = {} & ((ret_3 = m_3) \implies (n = r_2 + n - 1)) \wedge \\
& ((ret_3 = m_2) \implies (n = r_1 + n - 1)) \wedge \\
& ((ret_3 = \texttt{defval}) \implies \texttt{inil})
\end{aligned}
$$

```
let r1 = r0 in
new m1 =
  λ x. if x then (r++; m1 (x−1))
  else (λ y. assert(y = !r + x)) in
let ret3 =
  if n then (let r2 = r1 + 1 in
  if n−1 then (let r3 = r2 + 1 in defval)
  else (new m3 =
        λ y. assert(y = r2+n−1) in m3))
  else (new m2 =
        λ y. assert(y = r1 + n) in m2)
  in match ret3 with
  | defval → defval
  | m3  → assert(n = r2 + n−1)
  | m2  → assert(n = r1 + n)
```

Fig. 4. Translation example (clockwise from top left): original program, unwound program, constraints produced.

Example 3. To illustrate the intuition of our translation, consider the example in Figure 4 for $k = 2$. We build a model ϕ, where, for economy, we directly return variables instead of renaming them, and properties α. To construct the formula, we traverse the term in order, and add clauses in order of traversal. Note that the **else** branch is always explored first. The program is first unwound and transformed to SSA form with exhaustive method application (at **match** ret3 **with**). Note that all assignments have been replaced with let-bindings. This is because we convert references to SSA variables. In addition, we use keyword **new** to add fresh names to the repository.

Bounded Model Checking with the Translation The steps to do a k-bounded model checking of some initial configuration using the bounded translation algorithm described previously are as follows. First, given a valid configuration (M, R, S, k), let us set:

$$C_S = \{r \mapsto r_0 \mid r \in dom(S)\} \qquad \phi_S = \bigwedge\nolimits_{r \in dom(S)} (r_0 = S(r))$$

Thus, starting from (M, R, S, k):

1. Build $init(M, R, S, k) = (M, R, C_S, C_S, \phi_S, \top, \top, k)$.
2. Compute the translation: $[\![M, R, C_S, C_S, \phi_S, \top, \top, k]\!] = (ret, \phi, R', C, D, \alpha, pc)$.
3. Check $\phi \wedge \texttt{inil} \wedge \neg\alpha$ for satisfiability:
   (a) If satisfiable, we have a model for $\phi \wedge \texttt{inil} \wedge \neg\alpha$ that provides values for all open variables $\vec{x}$ and, therefore, a reachable assertion violation.
   (b) Otherwise, check $\phi \wedge \neg\texttt{inil} \wedge \neg\alpha$ for satisfiability:[1]

[1] Thus, the role of `inil` is to switch off/on the recording of reaching the bound as an assertion violation.

i. If satisfiable, the bound was reached, so we increment k by one, and repeat from step (2).
ii. Otherwise, the bound was not reached, so the program has been fully verified.

Soundness In this section we prove that our BMC algorithm is sound for input terms that are closed or contain open variables of ground type.

We start off with some definitions. An ***assignment*** $\sigma : \texttt{Vars} \rightharpoonup \texttt{CVals}$ is a finite map from variables to closed values. Given a term M, we write $M\{\sigma\}$ for the term obtained by applying σ to M. On the other hand, applying σ to a method repository R, we obtain the repository $R\{\sigma\} = \{m \mapsto R(m)\{\sigma\} \mid m \in \mathrm{dom}(R)\}$ – and similarly for stores S. Then, given a valid configuration (M, R, S, k), we have $(M, R, S, k)\{\sigma\} = (M\{\sigma\}, R\{\sigma\}, S\{\sigma\}, k)$.

Given a formula ψ and an assignment σ, we say σ ***represents*** ψ, and write $\sigma \simeq \psi$, if σ satisfies ψ (written $\sigma \vDash \psi$) and ψ implies σ ($\forall x \in dom(\sigma).\ \psi \implies x = \sigma(x)$). Given an assignment σ, we define a formula σ° representing it by: $\sigma^\circ = \bigwedge_{x \in \mathrm{dom}(\sigma)} (x = \sigma(x))$.

We define $[\![M, R, S, k]\!] = [\![init(M, R, S, k)]\!]$.

Theorem 4 (Soundness). *Given a valid configuration (M, R, S, k) whose open variables are of ground type, suppose $[\![M, R, S, k]\!] = (ret, \phi, R'', C, D, \alpha, pc)$. Then, for all assignments σ_0 closing (M, R, S, k), 1 and 2 are equivalent:*

1. $\exists E, R', S'.(M, R, S, k)\{\sigma_0\} \twoheadrightarrow (E[\texttt{assert}(0)], R', S', k')$
2. $\exists \sigma' \supseteq \sigma_0.\ \sigma' \models \phi \wedge \texttt{inil} \wedge \neg\alpha.$

Moreover, if $\phi \wedge \texttt{inil} \wedge \neg\alpha$ is not satisfiable then 3 and 4 are equivalent:

3. $\exists M', R', S'.(M, R, S, k)\{\sigma_0\} \twoheadrightarrow (M', R', S', \texttt{nil})$
4. $\exists \sigma' \supseteq \sigma_0.\ \sigma' \models \phi \wedge \neg\texttt{inil} \wedge \neg\alpha.$

Proof. (1) $\implies$ (2) and (3) $\implies$ (4) follow directly from Lemma 5 below. For the reverse directions, we rely on the fact that the semantics is bounded so, in every case, $(M, R, S, k)\{\sigma_0\}$ should either reach a value, or a failed assertion, or hit the bound. Moreover, the semantics is deterministic, in the sense that the configurations that can be eventually reached may only differ in the choice of fresh names used in the transition sequence (see Appendix A). These two facts allow us to employ Lemma 5 also for the reverse directions. □

Lemma 5 (Correctness). *Given $M, R, C, D, \phi, \alpha, pc, k, \sigma$ such that $\sigma \simeq \phi$, $(M, R, D, k)\{\sigma\}$ is valid, and $[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$ there exists $\sigma' \supseteq \sigma$ such that $\sigma' \simeq \phi'$ and:*

- *if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (v, \hat{R}, \hat{S}, \hat{k})$ then $\sigma' \vDash (pc \implies (pc' \wedge (ret = v))) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha')$, $R'\{\sigma'\} \supseteq \hat{R}$, and $D'\{\sigma'\} = \hat{S}$.*
- *if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (E[\texttt{assert}(0)], \hat{R}, \hat{S}, \hat{k})$ then $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha')$*
- *if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (\hat{M}, \hat{R}, \hat{S}, \texttt{nil})$ then $\sigma' \vDash (pc \implies \neg pc') \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha') \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha')$.*

Base Cases:

$PT(M, R, pt, nil) = (ret, \varnothing, R, pt)$ $\quad PT(\langle v_1, v_2\rangle, R, pt, k) = (ret, \langle pt(v_1), pt(v_2)\rangle, R, pt)$
$PT(m, R, pt, k) = (ret, \{m\}, R, pt)$ $\quad PT(\lambda x.M, R, pt, k) = (ret, \{m\}, R[m \mapsto \lambda.M], pt)$
$PT(x, R, pt, k) = (ret, pt(x), R, pt)$ $\quad PT(r := v, R, pt, k) = (ret, \varnothing, R, pt[r \mapsto pt(v)])$
$PT(!r, R, pt, k) = (ret, pt(r), R, pt)$ $\quad PT(\pi_i\, v, R, pt, k) = (ret, \pi_i\,(pt(v)), R, pt)$
$PT(v_1 \oplus v_2, R, pt, k) = (ret, \varnothing, R, pt)$ $\quad PT(v, R, pt, k) = (ret, \varnothing, R, pt)$ where $v = i, ()$
$PT(\texttt{assert}(v), R, pt, k) = (ret, \varnothing, R, pt)$

Inductive Cases:

$PT(\texttt{let}\ x = M\ \texttt{in}\ M', R, pt, k) =$
  let $(ret_1, A_1, R_1, pt_1) = PT(M, R, pt, k)$ in $PT(M'\{ret_1/x\}, R_1, pt_1[ret_1 \mapsto A_1], k)$
$PT(\texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M', R, pt, k) =$
  let m, f' be fresh in $PT(M'\{f'/f\}, R[m \mapsto \lambda x.M\{f'/f\}], pt[f' \mapsto \{m\}], k)$
$PT(m\, v, R, pt, k) =$ let $R(m)$ be $\lambda x.N$ in $PT(N\{v/x\}, R, pt, k)$
$PT(\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, pt, k) =$
  let $(ret_0, A_0, R_0, pt_0) = PT(M_0, R, pt, k)$ in
  let $(ret_1, A_1, R_1, pt_1) = PT(M_1, R_0, pt_b, k)$ in $(ret, A_0 \cup A_1, R_1, merge(pt_0, pt_1))$
$PT(x^{\theta}\, v, R, pt, k) =$
  let R be R_0 and $pt(x)$ be $\{m_1, ..., m_n\}$ in
  if $n = 0$ then $(ret, \emptyset, R_0, pt)$ else: for each $i \in \{1, ..., n\}$:
    let $R(m_i)$ be $\lambda y_i.N$ inlet $(ret_i, A_i, R_i, pt_i) = PT(N_i\{v/y_i\}, R_{i-1}, pt, k)$ in
  $(ret, A_1 \cup ... \cup A_n, R_n, merge(pt_1, \ldots, pt_n))$

Fig. 5. The points-to analysis algorithm.

4 A Points-to Analysis for Names

The presence of exhaustive method application in our BMC translation is a primary source of state space explosion. As such, a more precise filtering of R is necessary for scalability. In this section we describe a simple analysis to restrict the number of methods considered. We follow ideas from points-to analysis, which typically computes an overapproximation of the *points-to set* of each variable inside a program, that is, the set of locations that it may point to.

Our analysis computes the set of methods that may be bound to each variable while unfolding and. We do this via a finite map $pt : (\texttt{Refs} \cup \texttt{Vars}) \rightharpoonup \texttt{Pts}$ where $\texttt{Pts}$ contains all *points-to sets* and is given by: $\texttt{Pts} \ni A ::= X \mid \langle A, A\rangle$ where $X \subseteq_{fin} \texttt{Meths}$. Thus, a points-to set is either a finite set of names or a pair of points-to sets. These need to be updated whenever a method name is created, and are assigned to references or variables according to the following cases:

$r := M$	add in pt: $r \mapsto pt(M)$
$\texttt{let}\ x = M\ \texttt{in}\ M'$	add in pt: $x \mapsto pt(M)$
xM	add in pt: $ret(M) \mapsto pt(M)$

where $ret(M)$ is the variable assigned to the result of M. The $\texttt{letrec}$ binder follows a similar logic. The need to have sets of names, instead of single names, in

the range of pt is that the analysis, being symbolic, branches on conditionals and applications, so the method pointed to by a reference cannot be decided during the analysis. Thus, when joining after branching, we merge the pt maps obtained from all branches.

The points-to algorithm is presented in Figure 5. Given a valid configuration (M, R, S, k), the algorithm returns $PT(M, R, S, k) = (ret, A, R, pt)$, where A is the points-to set of ret, and pt is the overall points-to map computed.

The merge of points-to maps is given by:

$$merge(pt_1, \ldots, pt_n) = \{x \mapsto \bigcup_i \hat{pt}_i \mid x \in \bigcup_i \mathrm{dom}(pt_i)\}$$

where $\hat{pt}_i(x) = pt_i(x)$ if $x \in dom(pt_i)$, $\emptyset$ otherwise, and $A \cup B$ is defined by $\langle A_1 \cup B_1, A_2 \cup B_2 \rangle$ if $A, B = \langle A_1, A_2 \rangle, \langle B_1, B_2 \rangle$, and just $A \cup B$ otherwise.

The optimised BMC translation We can now incorporate the points-to analysis in the BMC translation to get an optimised translation which operates on symbolic configurations augmented with a points-to map, and returns:

$$[\![M, R, C, D, \phi, \alpha, pc, pt, k]\!]_{PT} = (ret, \phi', R', C', D', \alpha, pc, A, pt')$$

The optimised BMC translation is defined by lock-stepping the two algorithms presented above (i.e. $[\![\_]\!]$ and $PT(\_)$) and letting $[\![\_]\!]$ be informed from $PT(\_)$ in the xM case, which now restricts the choices of names for x to the set $pt(x)$. Its soundness is proven along the same lines as the basic algorithm.

To illustrate the significance of reducing the set of names, consider the program on the right which recursively generates names to compute triangular numbers. Without points-to analysis, since f creates a new method per call, and the translation considers all methods of matching type per recursive call, the number of names to apply at depth $m \leq n$ when translating $f(n)$ is approximately $m!$. This means that the number of paths explored grows by the factorial of n, with the total number of methods created being the left factorial sum $!n$, and total number of names considered being the derangement of n. In contrast, $f'(n)$ only considers n names with a linear growth in number of paths. With points-to analysis, the number of names considered and created in f is reduced to that of f'.

```
letrec f = λ x.
  if x ≤ 0 then 0
  else let g = (λ y.x + y) in
  g (f (x−1))
in
letrec f' = λ x.if x ≤ 0 then 0
                 else x + (f' (x−1))
in assert(f n = f' n)
```

5 Implementation and Experiments

We implemented the translation algorithm in a prototype tool to model check higher-order programs called BMC-2. The implementation and benchmarks can be found at:

https://github.com/LaifsV1/BMC-2

The tool takes program source code written in an ML-like language, and produces a propositional formula in SMT-LIB 2 format. This is then be fed to an SMT solver such as Z3. Syntax of the input language is based on the subset of OCaml that corresponds to HORef. Differences between OCaml and our concrete syntax are for ease of parsing and lack of type checking. For instance, all input programs must be either written in "Barendregt Convention", meaning all bound variables must be fresh, or such that variables have the same type globally. Additionally, all bound variables are annotated with types. Internally, BMC-2 implements an abstract syntax that extends HORef with vector arguments and integer lists. This means that functions can take multiple arguments at once. Lists are handled for testing, but not discussed here as they are not relevant to the theory. BMC-2 itself was written in OCaml.

To illustrate our input language, on the right is sample program `mc91-e` from [13] translated from OCaml to our syntax. The keyword `Methods` is used to define all methods in the repository. The keyword `Main` is used to define the main method. For this sample program, our tool builds a translation with $k = 1$ for which Z3 correctly reports that the assertion fails if $n = 102$. Details about experiments will be provided later.

```
Methods:
mc91 (x:Int) :(Int) =
  if x >= 101 then x + −10
  else mc91 (mc91 (x + 11));
Main (n:Int) :(Unit):
  if n <= 102
  then assert((mc91 n)==91)
  else skip
```

Benchmarks We tested our implementation on a set of 40 programs that include a selection from the MoCHi benchmark [13]; a set of higher-order programs written in OCaml, originally used to test the higher-order model checking tool MoCHi [13], and subsequently used for benchmarking [23,4,22]. We added custom samples with references (`ref-1`, `ref-1-e`, `ref-2`, `ref-2-e`, `ref-3`), as well as programs of varying lengths–100, 200 and 400 lines of code–constructed by combining the other samples. To combine programs, we refactored and concatenated methods and main methods from different files into a single file, and switch between the methods based on user input, thus forcing BMC-2 to consider all mains. In this set we have of safe and unsafe programs denoted by the `-e` termination in their filename. Unsafe programs were constructed by slight modifications to the assertions of the original safe programs. For our experiment, the programs were manually translated to our input language and checked using our tool and Z3. Care was taken to keep all sample programs as close to the original source code as our concrete syntax allows. All experiments ran on an Ubuntu machine equipped with an Intel Core i7-6700 CPU clocked at 3.40GHz and 16GB RAM. All tests were set to time-out after 3 minutes, and up to a maximum bound $k = 15$. These limits were chosen due to the combinatorial nature of model checking and the sample programs used. BMC-2 ran twice per program per bound, and the average was recorded.

Figure 6 plots the average time taken for BMC-2 to check all the benchmark programs. We can observe that performance of BMC-2 heavily depends on the program it is checking, making the possibility of full verification entirely dependent on the nature of the program. For example, `ack`, which is an implementation of

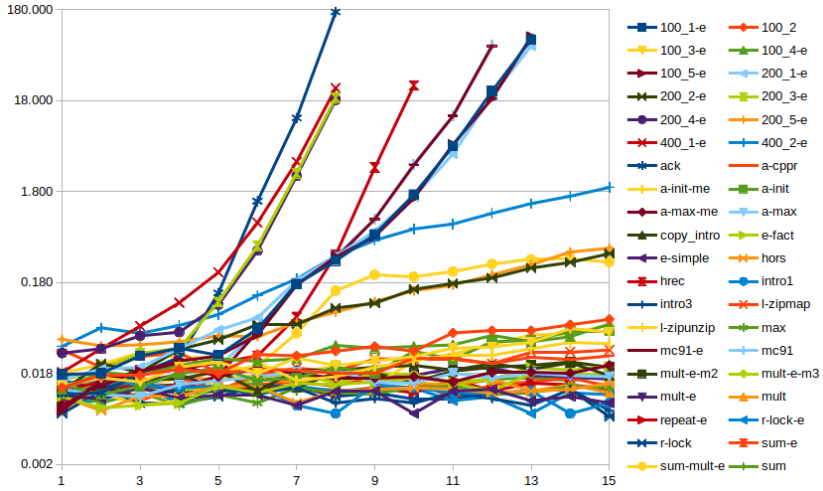

Fig. 6. Average execution time (s) for BMC-2 on bounds $k = 1..15$.

the Ackermann function, is a deeply recursive program that diverges rapidly, and thus cannot be translated by our algorithm any better than its normal growth. This agrees with the intuition that BMC is not appropriate to find bugs in deep recursion. As mentioned before, however, BMC has been shown to be effective on shallow bugs in industry. This can be seen with our examples for 100 to 400 lines of code, which were correctly shown to have bugs, with little difficulty despite the increase in program size.

In addition to testing BMC-2, we also ran comparison experiments on prior tools MoCHi [13] and Rosette [24]. All experiments ran on the same machine used to test BMC-2. These tools will be described in more detail in the following section. For Rosette, we used an implementation of our bounded semantics in Rosette provided to us by an anonymous reviewer. With the semantics implemented, we compare Rosette's symbolic execution of HORef, to BMC-2 with Z3's translation and solving of the same terms.

	4	7	10	13	15	MoCHi
100_1-e	0.034	0.173	1.661	84.130	-	c
100_2	0.020	0.028	0.032	0.053	0.071	c
100_3-e	0.021	0.027	0.028	0.040	0.051	10.734
200_1-e	0.034	0.188	1.572	71.296	-	m
200_2-e	0.033	0.063	0.151	0.259	0.372	-
200_3-e	0.034	2.849	-	-	m	1.742
400_1-e	0.108	3.805	-	-	m	m
400_2-e	0.061	0.196	0.696	1.321	1.991	-
ack	0.027	11.519	-	-	-	0.525
a-cppr	0.031	0.020	0.026	0.027	0.028	28.584
a-init	0.018	0.016	0.032	0.042	0.053	c
e-fact	0.009	0.014	0.016	0.021	0.022	0.629
e-simple	0.010	0.008	0.007	0.009	0.009	0.098
hrec	0.020	0.075	26.175	-	-	0.867
r-lock-e	0.009	0.013	0.013	0.007	0.011	0.216
ref-2	0.013	0.008	0.011	0.012	0.010	u
ref-2-e	0.011	0.013	0.010	0.013	0.011	u
ref-3	0.019	0.018	0.047	0.211	0.211	u

Table 1. Time taken (s) for BMC-2 ($k = 1..15$) and MoCHi, where $-$, c, m and u respectively indicate *timeout*, *crash*, *out-of-memory* and *unsupported*

Comparison with MoCHi Though the goals of each tool are different, we attempted to compare our approach to MoCHi. Being unable to build from source, we decided to used the Dockerfile on the Ubuntu machine from before. In Table 1, we have the time taken for BMC-2 and MoCHi for a smaller set of programs– the full range of results can be found in Appendix C. We noticed that MoCHi is very sensitive to the operators and functions used in the assertions, while

BMC-2 appears to be less dependent on these. For instance, checking `mult-e` with `assert(mult m m <= mult n n)` was three orders of magnitude slower than the original, while, at $k = 1$, BMC-2 takes 0.012 seconds; an increase of 20% from the original 0.010s to find a bug. We also noticed that MoCHi is less consistent with larger programs. For 100 to 400 lines of code, MoCHi correctly found bugs in 4 out of 12 samples, but halted unexpectedly on the remaining 8. BMC-2 found all 11 bugs of the 12 programs, and found no bugs in the safe program. Finally, we included 5 examples with references, which BMC-2 correctly checked. Attempting to check them with MoCHi suggests references are unsupported.

Comparison with Rosette for Racket We found that BMC-2 and Rosette are very similar in their ability to check higher-order programs. Since Racket is a stateful higher-order language like HORef, and Rosette employs a symbolic virtual machine with symbolic execution techniques for Racket, we can expect this similarity. Fundamentally, Rosette and BMC-2 provide different approaches to verification as the former is related to symbolic evaluation, while the latter is a monolithic BMC translation. We were particularly interested in Rosette's ability to implement bounded verification for higher-order programs. We defined our bounding mechanism in Rosette, and compared its symbolic evaluation to BMC-2 on the Ubuntu machine. Figure 7 showcases this comparison. We found

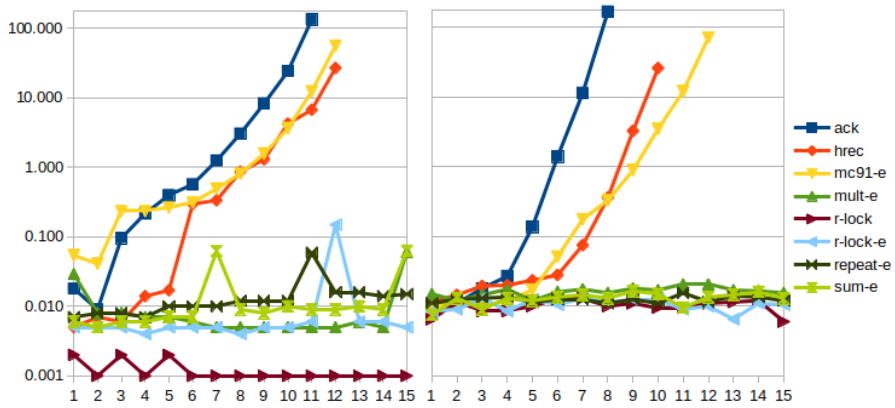

Fig. 7. Execution time (s) of $k = 1 \ldots 15$ for Rosette (left) and BMC-2 (right).

that Rosette and BMC-2 are comparable in scalability, with BMC-2 being less optimised for some diverging programs such as `ack`. This could be due to the way Rosette performs *type-driven state merging*, which provides opportunities for concretization. In contrast, we perform a suboptimal SSA transformation which could benefit from *dominance frontiers* for optimal merging of control flow. BMC, however, has the theoretical advantage of faster compilation time over symbolic execution [24]. Since testing in Rosette involves manually (re)translating our examples into Racket, our comparison is based on a sample of 8 programs from the MoCHi benchmarks.

References

1. N. Amla, R. P. Kurshan, K. L. McMillan, and R. Medel. Experimental analysis of different techniques for bounded model checking. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003.
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
3. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
4. T. C. Burn, C. L. Ong, and S. J. Ramsay. Higher-order constrained horn clauses for verification. *PACMPL*, 2(POPL):11:1–11:28, 2018.
5. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
6. G. Dennis, F. S. Chang, and D. Jackson. Modular verification of code with SAT. In L. L. Pollock and M. Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 109–120. ACM, 2006.
7. J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 195–204. ACM, 2007.
8. V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
9. W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *Software Engineering, IEEE Transactions on*, SE-3:266– 278, 08 1977.
10. J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.
11. J. King. A new approach to program testing. 10:228–233, 06 1975.
12. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 416–428, New York, NY, USA, 2009. ACM.
13. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 222–233. ACM, 2011.
14. A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 151–164. ACM, 2012.

15. J. Morse, M. Ramalho, L. C. Cordeiro, D. Nicole, and B. Fischer. ESBMC 1.22 - (competition contribution). In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 405–407. Springer, 2014.
16. J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In W. Tracz, M. P. Robillard, and T. Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 60. ACM, 2012.
17. P. C. Nguyễn and D. Van Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 446–456, New York, NY, USA, 2015. ACM.
18. C. . L. Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90, Aug 2006.
19. A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013.
20. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
21. R. S. Boyer, B. Elspas, and K. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10:234–245, 06 1975.
22. R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In E. Albert and S. Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013*, pages 53–62. ACM, 2013.
23. T. Terao and N. Kobayashi. A zdd-based efficient higher-order model checking algorithm. In J. Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2014.
24. E. Torlak and R. Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 530–541. ACM, 2014.

Appendix A Nominal determinacy

While the operational semantics is bounded in depth, the reduction tree of a given term can still be infinite because of the non-determinacy involved in evaluating λ-abstractions: the rule non-deterministically creates a fresh name m and extends the repository with m mapped to the given λ-abstraction. This kind of non-determinism, which can be seen as *determinism up to fresh name creation*, is formalised below.

Let us consider permutations $\pi : \texttt{Meths} \to \texttt{Meths}$ such that, for all m, if $m \in \texttt{Meths}_{\theta\to\theta'}$ then $\pi(m) \in \texttt{Meths}_{\theta\to\theta'}$. We call such a permutation π *finite* if the set $\{a \mid \pi(a) \neq a\}$ is finite. Given a syntactic object X (e.g. a term, repository, or store) and a finite permutation π, we write $\pi \cdot X$ for the object we obtain from X if we swap each name a appearing in it with $\pi(a)$. Put otherwise, the operation $\cdot$ is an action from finite permutations of $\texttt{Meths}$ to the set of objects X. Given a set $\Delta \subseteq \texttt{Meths}$ and objects X, X', we write $X \sim_\Delta X'$ whenever there exists a finite permutation π such that:

$$\pi \cdot X = X' \wedge \forall a \in \Delta.\ \pi(a) = a$$

and say that X and X' are *nominally equivalent* up to Δ.

In the following lemma we write $\twoheadrightarrow_n$ for the n-step composition of $\to$.

Lemma 6 (Nominal determinacy). *Let (T, R, S, k) be a valid configuration, $(T, R, S, k) \twoheadrightarrow_n (T', R', S', k')$, and let $\Delta = \mathrm{dom}(R) \cup \mathrm{dom}(S)$. Then, for all (T'', R'', S'', k'') we have $(T, R, S, k) \twoheadrightarrow_n (T'', R'', S'', k'')$ iff $(T', R', S', k') \sim_\Delta (T'', R'', S'', k'')$.*

Appendix B Proof of Correctness Lemma

Lemma 5 [Correctness] Given $M, R, C, D, \phi, \alpha, pc, k, \sigma$ such that $\sigma \simeq \phi$, $(M, R, D, k)\{\sigma\}$ is valid, and:

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

there exists $\sigma' \supseteq \sigma$ such that $\sigma' \simeq \phi'$ and:

- if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (v, \hat{R}, \hat{S}, \hat{k})$ then $\sigma' \vDash (pc \implies (pc' \wedge (ret = v))) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha')$, $R'\{\sigma'\} \supseteq \hat{R}$, and $D'\{\sigma'\} = \hat{S}$.
- if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (E[\texttt{assert}(0)], \hat{R}, \hat{S}, \hat{k})$ then $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha')$
- if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (\hat{M}, \hat{R}, \hat{S}, \texttt{nil})$ then $\sigma' \vDash (pc \implies \neg pc') \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha') \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha')$

where $\twoheadrightarrow$ is the reflexive transitive closure of $\to$.

Proof. Consider $\sigma \simeq \phi$, a valid configuration $(M, R, D, k)\{\sigma\}$, and its corresponding translation $[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$.

Now, by induction on the length of the transition sequence produced by the operational semantics, and, lexicographically, by induction on the size of the term, we have the following cases for configurations in canonical form.

Terminal configurations:

- $k = \texttt{nil}$.
  From the operational semantics we have $(M, R, D, \texttt{nil})\{\sigma\}$ as a struck configuration. From the translation we have

$$\begin{aligned}&[\![M, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = \texttt{defval}) \wedge \phi, R, C, D, \alpha \wedge (pc \implies \texttt{inil}), \bot)\end{aligned}$$

  We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = \texttt{defval}) \wedge \phi))$ and

$$\begin{aligned}\sigma' \vDash &(pc \implies \neg\bot) \\ &\wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies (\alpha \wedge (pc \implies \texttt{inil}))) \\ &\wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies \texttt{inil})))\end{aligned}$$

  Let us choose $\sigma' = \sigma[ret \mapsto \texttt{defval}]$. Since $\sigma \simeq \phi$, and since ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies \top) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies (\alpha \wedge (pc \implies \texttt{inil}))) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies \texttt{inil})))$ holds.
- $M = \texttt{assert}(0)$.
  From the operational semantics we have $(\texttt{assert}(0), R, D, k)\{\sigma\}$ as a struck configuration. From the translation we have

$$\begin{aligned}&[\![\texttt{assert}(0), R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = ()) \wedge \phi, R, C, D, \alpha \wedge (pc \implies (0 \neq 0)), \top)\end{aligned}$$

  We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = ()) \wedge \phi))$ and

$$\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies (0 \neq 0))))$$

  Let us choose $\sigma' = \sigma[ret \mapsto ()]$. Since $\sigma \simeq \phi$, and ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies \bot)))$ holds.
- $M = v$.
  From the operational semantics we have $(v, R, D, k)\{\sigma\}$ as a struck configuration. From the translation we have

$$\begin{aligned}&[\![v, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = v) \wedge \phi, R, C, D, \alpha, \top)\end{aligned}$$

  We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = v) \wedge \phi))$ and

$$\begin{aligned}&\sigma' \vDash (pc \implies (\top \wedge (ret = v))) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

  Let us choose $\sigma' = \sigma[ret \mapsto v]$. Since $\sigma \simeq \phi$, and ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (\top \wedge (ret = v))) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.
- $M = (\!|v|\!)$. Similar to the case above.

Non-terminal configurations:

- $M = \texttt{assert}(i)$ where $i \neq 0$.
  From the operational semantics we have

$$(\texttt{assert}(i), R, D, k)\{\sigma\} \rightarrow ((), R, D, k)\{\sigma\}$$

  From the translation we have

$$\begin{aligned}&[\![\texttt{assert}(i), R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = ()) \wedge \phi, R, C, D, \alpha \wedge (pc \implies (i \neq 0)), \top)\end{aligned}$$

  We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = ()) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = ())) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies (\alpha \wedge (pc \implies (i \neq 0)))) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

  Let us choose $\sigma' = \sigma[ret \mapsto ()]$. Since $\sigma \simeq \phi$, and ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = ()))$ holds because σ' maps ret to $()$, and $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.
- $M = r := v$.
  From the operational semantics we have

$$(r := v, R, D, k)\{\sigma\} \rightarrow ((), R, D[r \mapsto v], k)\{\sigma\}$$

  From the translation we have

$$\begin{aligned}&[\![r := v, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad \text{let } C' = C[r] \text{ in let } D[r \mapsto C'(r)] \text{ in} \\ &\quad (ret, (ret = ()) \wedge (D'(r) = v) \wedge \phi, R, C', D', \alpha, \top)\end{aligned}$$

  We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = ()) \wedge (D'(r) = v) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = ())) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies (\alpha \wedge (pc \implies (i \neq 0)))) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D[r \mapsto C'(r)]\{\sigma'\} = D[r \mapsto v]\{\sigma'\}\end{aligned}$$

  Let us choose $\sigma' = \sigma[ret \mapsto (), C'(r) \mapsto v]$. Since $\sigma \simeq \phi$, ret is the only new variable in ϕ', and D' maps r to $C'(r)$, which σ' maps to v, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = ()))$ holds because σ' maps ret to $()$, and $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ holds trivially, and $D[r \mapsto C'(r)]\{\sigma'\} = D[r \mapsto v]\{\sigma'\}$ holds because σ' maps $C'(r)$ to v.

– $M = !r$.
From the operational semantics we have

$$(!r, R, D, k)\{\sigma\} \to (v, R\{\sigma\}, D\{\sigma\}, k) \quad \text{where } v = D\{\sigma\}(r) = \sigma(D(r))$$

From the translation we have

$$\begin{aligned}&[\![!r, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = D(r)) \wedge \phi, R, C, D, \alpha, \top)\end{aligned}$$

We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = D(r)) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = v)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto D(r)]$. Since $\sigma \simeq \phi$, and ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = v))$ holds because $\sigma' \supset \sigma$, $\sigma(D(r)) = v$ and $\sigma'(ret) = D(r)$, and we know $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

– $M = v_1 \oplus v_2$.
From the operational semantics we have

$$(v_1 \oplus v_2, R, D, k)\{\sigma\} \to (v, R\{\sigma\}, D\{\sigma\}, k) \quad \text{where } v = \sigma(v_1) \oplus \sigma(v_2)$$

From the translation we have

$$\begin{aligned}&[\![v_1 \oplus v_2, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = (v_1 \oplus v_2)) \wedge \phi, R, C, D, \alpha, \top)\end{aligned}$$

We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = (v_1 \oplus v_2)) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = v)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto v]$. Since $\sigma \simeq \phi$, and ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = v))$ holds because $\sigma' \supseteq \sigma$, $\sigma(v_1) \oplus \sigma(v_2) = v$ and σ' maps ret to v. Lastly, we know $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

– $M = \pi_i\, v$.
From the operational semantics we have

$$(\pi_i\, v, R, D, k)\{\sigma\} \to (v_i, R\{\sigma\}, D\{\sigma\}, k) \quad \text{where } \sigma(v) = \langle v_1, v_2 \rangle$$

From the translation we have

$$[\![\pi_i\, v, R, C, D, \phi, \alpha, pc, k]\!] =$$

$$(ret, (ret = (\pi_i\, v)) \wedge \phi, R, C, D, \alpha, \top)$$

We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = (\pi_i\, v)) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\sigma' \vDash (pc \implies (ret = v_i)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$$
$$R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}$$

Let us choose $\sigma' = \sigma[ret \mapsto v_i]$. Since $\sigma \simeq \phi$, and ret is the only new variable in ϕ', we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = v))$ holds because $\sigma' \supseteq \sigma$, $\sigma(\pi_i\, v) = v_i$ and σ' maps ret to v_i. Lastly, we know $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

– $M = \lambda x.N$.
From the operational semantics we have

$$(\lambda x.N, R, D, k)\{\sigma\} \to (\hat{m}, R[\hat{m} \mapsto \lambda x.N]\{\sigma\}, D\{\sigma\}, k)$$

From the translation we have

$$[\![\lambda x.N, R, C, D, \phi, \alpha, pc, k]\!] = (ret, (ret = m) \wedge \phi, R[m \mapsto \lambda x.N], C, D, \alpha, \top)$$

We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = m) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\sigma' \vDash (pc \implies (ret = \hat{m})) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$$
$$R[m \mapsto \lambda x.N]\{\sigma'\} \supseteq R[\hat{m} \mapsto \lambda x.N]\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}$$

Let us choose $\sigma' = \sigma[ret \mapsto m]$. Since $\sigma \simeq \phi$ and ret is the only new variable in ϕ', and choosing $\hat{m}$ such that $\hat{m} = m$ by Lemma 6 (nominal determinism of the operational semantics), we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = \hat{m}))$ holds because $\sigma' \supset \sigma$, $m = \hat{m}$ by Lemma 6, and σ' maps ret to m. Lastly, we know $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha)$ holds. Additionally, $D\{\sigma'\} = D\{\sigma'\}$ holds trivially, and $R[m \mapsto \lambda x.N]\{\sigma'\} \supseteq R[\hat{m} \mapsto \lambda x.N]\{\sigma'\}$.

– $M = mv$.
From the operational semantics we have

$$(mv, R, S, k)\{\sigma\} \to (N\{v/x\}, R, S, k-1)\{\sigma\}$$

where $R(m) = \lambda x.N$.
From the translation we have

$$[\![mv, R, C, D, \phi, \alpha, pc, k]\!] = [\![N\{v/x\}, R, C, D, \phi, \alpha, pc, k-1]\!]$$

As such, this case holds directly from the inductive hypothesis.

– $M = \texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M'$.
From the operational semantics we have

$$(\texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M', R, S, k)\{\sigma\} \to (M'\{m/f\}, R[m \mapsto \lambda x.N], S, k)\{\sigma\}$$

where $R(m) = \lambda x.N$.
From the translation we have

$$[\![\texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M', R, C, D, \phi, \alpha, pc, k]\!] = \\ [\![M\{f'/f\}, R[m \mapsto \lambda x.N\{f'/f\}], C, D, \phi \wedge (f' = m), \alpha, pc, k]\!]$$

Let $\sigma' = \sigma[f' \mapsto m]$ such that $\sigma' \simeq (\phi \wedge (f' = m))$. As such, this case holds directly from the inductive hypothesis.
– $M = (\texttt{let}\ x = v\ \texttt{in}\ M')$. Similar to the case above.
– $M = \texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0$.
From the operational semantics we have

$$(\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, S, k)\{\sigma\} \to (M_i, R, S, k-1)\{\sigma\}$$

where $i = 0$ if $\sigma(v) = 0$, and $i = 1$ otherwise.
From the translation we have

$$[\![\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, C, D, \phi, \alpha, pc, k]\!] = \\ \text{let } (ret_0, \phi_0, R_0, C_0, D_0, \alpha_0, pc_0) = [\![M_0, R, C, D, \phi, \alpha, pc \wedge (v = 0), k]\!] \text{ in} \\ \text{let } (ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = [\![M_1, R_0, C_0, D_b, \phi_0, \alpha_0, pc \wedge (v \neq 0), k]\!] \text{ in} \\ \text{let } C' = C_1[r_1] \cdots [r_n]\ (\Pi = \{r_1, \ldots, r_n\}) \text{ in} \\ \text{let } \psi_0 = (v = 0) \implies ((ret = ret_0) \wedge \bigwedge_{r \in \Pi}(C'(r) = D_0(r))) \text{ in} \\ \text{let } \psi_1 = (v \neq 0) \implies ((ret = ret_1) \wedge \bigwedge_{r \in \Pi}(C'(r) = D_1(r))) \text{ in} \\ (ret, \psi_0 \wedge \psi_1 \wedge \phi_1, R, C', C', \alpha_1, ((pc_0 \wedge (v = 0)) \vee (pc_1 \wedge (v \neq 0))))$$

We now have two cases for $\sigma(v)$:
1. $\sigma(v) = 0$.
   (1) By the inductive hypothesis we have $\sigma_0 \supseteq \sigma$, where $\sigma_0 \simeq \phi_0$, and all necessary conditions P_0 are satisfied.
   (2) By Lemma 7, we know $\exists \sigma_1 \supseteq \sigma_0.(\sigma_1 \simeq \phi_1)$.
   Let $\sigma' = \sigma_1[ret \mapsto ret_0, C'(r) \mapsto D_0(r)]$. Since $\sigma' \supset \sigma_1 \supseteq \sigma_0$, we know that P_0 is also satisfied. Thus, this case holds by (1) and (2).
2. $\sigma(v) \neq 0$.
   (1) By Lemma 7, we know $\exists \sigma_0 \supseteq \sigma.(\sigma_0 \simeq \phi_0)$.
   By Lemma 8, we know R must be preserved in R_0, so $R_0\{\sigma_0\} \supseteq R$. Thus, by the inductive hypothesis:
   (2) we have $\sigma_1 \supseteq \sigma_0 \supseteq \sigma$, where $\sigma_1 \simeq \phi_1$, and all necessary conditions P_1 are satisfied.
   Let $\sigma' = \sigma_1[ret \mapsto ret_1, C'(r) \mapsto D_1(r)]$. Since $\sigma' \supset \sigma_1$, we know that P_1 is also satisfied. Thus, this case holds by (1) and (2).

– $M = xv$.
From the operational semantics we have

$$(xv, R, S, k)\{\sigma\} \rightarrow (N_i\{v/y_i\}, R, S, k-1)\{\sigma\}$$

where $\sigma(x) = m_i$ and $R(m_i = \lambda y_i.N_i)$.
Let $\sigma_0 = \sigma$.
From the translation we have

$$\begin{array}{l}
[\![x^\theta\ v, R, C, D, \phi, \alpha, pc, k]\!] = \\
\quad \text{if } R \upharpoonright \theta = \emptyset \text{ then } (ret, (ret = \texttt{nil}) \wedge \phi, R, C, D, \alpha, pc) \text{ else} \\
\quad \text{let } R \upharpoonright \theta \text{ be } \{m_1, ..., m_n\} \text{ and } (R, C, \phi, \alpha) \text{ be } (R_0, C_0, \phi_0, \alpha_0) \text{ in} \\
\quad \text{for each } i \in \{1, ..., n\} : \\
\qquad \text{let } R(m_i) \text{ be } \lambda y_i.N \text{ in} \\
\qquad \text{let } (ret_i, \phi_i, R_i, C_i, D_i, \alpha_i, pc_i) = \\
\qquad\quad [\![N_i\{v/y_i\}, R_{i-1}, C_{i-1}, D, \phi_{i-1}, \alpha_{i-1}, pc \wedge (x = m_i), k-1]\!] \text{ in} \\
\quad \text{let } C'_n = C_n[r_1] \cdots [r_j]\ (\Pi = \{r_1, \ldots, r_j\}) \text{ in} \\
\quad \text{let } \psi = \bigwedge_{i=1}^{n} \left(\begin{array}{l} (x = m_i) \implies \\ ((ret = ret_i) \wedge \\ \quad \bigwedge_{r \in \Pi} (C'_n(r) = D_i(r))) \end{array} \right) \text{ in} \\
\quad \text{let } pc'_n = \bigvee_{i=1}^{n} (pc_i \wedge (x = m_i)) \text{ in} \\
\quad (ret, \psi \wedge \phi_n, R_n, C'_n, C'_n, \alpha_n, pc'_n)
\end{array}$$

Since $R \upharpoonright \theta = \{m_1, \ldots, m_n\}$, it must be the case that $i \in \{1 \ldots n\}$.
It must be the case then that either (1) $i = 1$ or (2) $1 < i \leq n$.

1. (1) By the inductive hypothesis, we have $\exists \sigma_1 \supseteq \sigma_0.(\sigma_1 \simeq \phi_1)$, and all necessary properties P_0 hold.
   (2) By Lemma 7 applied repeatedly, we have $\exists \sigma_n \supseteq \cdots \supseteq \sigma_1.(\sigma_n \simeq \phi_n)$.
   Since $\sigma_n \supseteq \sigma_1$, properties P_0 hold, so this case holds by (1) and (2).
2. (1) By Lemma 7 applied repeatedly, we know $\exists \sigma_{i-1} \supseteq \cdots \supseteq \sigma_0.(\sigma_{i-1} \simeq \phi_{i-1}) \wedge \cdots \wedge (\sigma_0 \simeq \phi_0)$.
   (2) By Lemma 8 applied repeatedly, we also know $R_{i-1} \supseteq \cdots \supseteq R_0$.
   (3) By the inductive hypothesis, we know $\exists \sigma_i \supseteq \sigma_{i-1}.(\sigma_i \simeq \phi_i)$ and the necessary properties P_i hold.
   (4) By Lemma 7 again applied repeatedly, we know $\exists \sigma_n \supseteq \cdots \supseteq \sigma_i.(\sigma_n \simeq \phi_n) \wedge \cdots \wedge (\sigma_i \simeq \phi_i)$.
   Since $\sigma_n \supseteq \sigma_i$, properties P_i hold, so this holds by (1), (2), (3) and (4).

– $M = (\texttt{let}\ x = M'\ \texttt{in}\ M'')$, with M' not a value.
Let us write M as $E[M']$. From the operational semantics we have

$$(E[M'], R, S, k)\{\sigma\} \rightarrow (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow \ldots$$

where $(M', R, S, k)\{\sigma\} \to (\hat{M}, \hat{R}, \hat{S}, \hat{k})\{\sigma\}$.
We now have the following translation.

$$\begin{aligned}&[\![E[M'], R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad \text{let } (ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = [\![M', R, C, D, \phi, \alpha, pc, k]\!] \text{ in} \\ &\quad \text{let } (ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_2) = [\![E[ret_1], R_1, C_1, D_1, \phi_1, \alpha_1, pc \wedge pc_1, k]\!] \text{ in} \\ &\quad (ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_1 \wedge pc_2)\end{aligned}$$

Since $E[M']$ can lead to either (1) some value, (2) an assertion $\texttt{assert}(0)$, or (3) a stuck configuration where the bound is $\texttt{nil}$, we have three cases to consider.

1. $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.
   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies (pc_1 \wedge (ret_1 = \hat{v_1}))) \wedge ((pc \wedge \alpha \wedge \texttt{inil}) \implies \alpha_1)$ and $R_1\{\sigma_1\} \supseteq R, D_1\{\sigma_1\} = \hat{S}$.

   $(E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1}) \twoheadrightarrow (\hat{v}', \hat{R}', \hat{S}', \hat{k})'$.
   (2) By the inductive hypothesis: $\sigma_2 \vDash (pc_1 \implies (pc_2 \wedge (ret_2 = \hat{v}'))) \wedge ((pc_1 \wedge \alpha_1 \wedge \texttt{inil}) \implies \alpha_2)$ and $R_2\{\sigma_2\} \supseteq \hat{R}_1, D\{\sigma_1\} = \hat{S}$.

   Since $\sigma_2 \supseteq \sigma$, we know $\sigma_2 \simeq \phi_2$.
   From (1) and (2), we know $\sigma_2 \vDash (pc \implies (pc_1 \wedge pc_2 \wedge (ret = \hat{v}\ ))) \wedge ((pc \wedge \alpha \wedge \texttt{inil}) \implies \alpha_2)$. Case holds.
2. (a) $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\texttt{assert}(0)], \hat{R_1}, \hat{S_1}, \hat{k_1})$.
   (1) By the inductive hypothesis: $\sigma_1 \vDash ((pc \wedge \alpha \wedge \texttt{inil}) \implies \neg\alpha_1)$ such that $\sigma_1 \simeq \phi_1$.
   (2) By Lemma 7, we know $\exists \sigma_2 \supseteq \sigma_1 . \sigma_2 \simeq \phi_2$

   By Lemma 9, and (1) and (2), we know $\alpha_2 \implies \alpha_1$, thus $\neg\alpha_1 \implies \neg\alpha_2$. We therefore have $\sigma_1 \vDash ((pc \wedge \alpha \wedge \texttt{inil}) \implies \neg\alpha_2)$. Case holds.
   (b) $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.
   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies (pc_1 \wedge (ret_1 = \hat{v_1}))) \wedge ((pc \wedge \alpha \wedge \texttt{inil}) \implies \alpha_1)$ and $R_1\{\sigma_1\} \supseteq R, D_1\{\sigma_1\} = \hat{S}$.

   $(E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1}) \twoheadrightarrow (E'[\texttt{assert}(0)], \hat{R}', \hat{S}', \hat{k})'$
   (2) By the inductive hypothesis: $\sigma_2 \vDash ((pc \wedge pc_1 \wedge \alpha_1 \wedge \texttt{inil}) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$.

   From (1) and (2) we have $\sigma_2 \vDash (pc \implies \neg(pc_1 \wedge pc_2)) \wedge (\alpha \wedge pc \wedge \texttt{inil}) \implies \neg\alpha_2$. Case holds.
3. (a) $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{M_1}], \hat{R_1}, \hat{S_1}, \texttt{nil})$.
   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies \neg pc_1) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_1) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_1)$ such that $\sigma_1 \simeq \phi_1$.
   (2) By Lemma 7, we know $\exists \sigma_2 \supseteq \sigma_1 . \sigma_2 \simeq \phi_2$.
   (3) By Lemma 9, we know $\alpha_2 \implies \alpha_1$, so $\neg\alpha_1 \implies \neg\alpha_2$.

   From (1) we have that $\sigma_1 \vDash (pc \implies \neg(pc_1 \wedge pc_2))$.

From (1) and (3) we have that $\sigma_1 \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_2)$.
From (2) we have that $\sigma_2 \vDash (pc \implies \neg(pc_1 \wedge pc_2)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$. Case holds.

(b) $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.
(1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies (pc_1 \wedge (ret_1 = \hat{v_1}))) \wedge ((pc \wedge \alpha \wedge \texttt{inil}) \implies \alpha_1)$ and $R_1\{\sigma_1\} \supseteq R, D_1\{\sigma_1\} = \hat{S}$.

$(E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1}) \twoheadrightarrow (E'[\hat{M'}], \hat{R'}, \hat{S'}, \texttt{nil})'$
(2) By the inductive hypothesis: $\sigma_2 \vDash ((pc \wedge pc_1) \implies \neg pc_2) \wedge ((\texttt{inil} \wedge \alpha_1 \wedge pc \wedge pc_1) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha_1 \wedge pc \wedge pc_1) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$.

From (1) and (2) we have $\sigma_2 \vDash (pc \implies \neg(pc_1 \wedge pc_2)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$. Case holds.

Lemma 7 (Uniqueness of the translation). *Given an assignment σ and formula ϕ such that $\sigma \cong \phi$, and a translation*

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

we know there exists some $\sigma' \supseteq \sigma$ such that $\sigma' \cong \phi'$.

Proof. Assuming $\sigma \cong \phi$, by induction on k and then by structural induction on M, we have the base cases:

1. $k = \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto \texttt{defval}]$.
2. $M = \texttt{assert}(()v)$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto ()]$.
3. $M = v$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto v]$.
4. $M = !r$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto D(r)]$.
5. $M = \lambda x.N$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto m]$.
6. $M = \pi_i\, v$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto \pi_i\, v]$.
7. $M = v_1 \oplus v_2$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto v_1 \oplus v_2]$.
8. $M = r := v$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto (), D'(r) = v]$.

With base cases done, we have the following inductive cases:

1. $M = \texttt{let}\ x = M'\ \texttt{in}\ M''$:
   (1) By the inductive hypothesis on $[\![M, R, C, D, \phi, \alpha, pc, k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   (2) By the inductive hypothesis on $[\![M'\{ret_1/x\}, R_1, C_1, D_1, \phi_1, \alpha_1, pc \wedge pc_1, k]\!]$, we have $\sigma_2 \simeq \phi_2$.
   This case holds by (1) and (2).
2. $M = \texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M'$:
   Let $\sigma' = \sigma[f' \mapsto m]$ such that $\sigma' \simeq (\phi \wedge (f' = m))$.
   (1) By the inductive hypothesis on $[\![M'\{f'/f\}, R', C, D, \phi \wedge (f' = m), \alpha, pc, k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   This case holds by (1).

3. $M = mv$:
   (1) By the inductive hypothesis on $[\![N\{v/x\}, R', C, D, \phi, \alpha, pc, k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   This case holds by (1).
4. $M = \texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0$:
   (1) By the inductive hypothesis on $[\![M_0, R, C, D, \phi, \alpha, pc \wedge (v = 0), k]\!]$, we have $\sigma_0 \simeq \phi_0$.
   (2) By the inductive hypothesis on $[\![M_1, R_0, C_0, D_0, \phi_0, \alpha_0, pc \wedge (v \neq 0), k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   We now have two cases on $\sigma_1(v)$:
   (a) $\sigma_1(v) = 0$. Choose $\sigma' = \sigma_1[ret \mapsto ret_0, C'(r) \mapsto D_0(r)]$ for all $r \in \Pi$.
   (b) $\sigma_1(v) = i \neq 0$. Choose $\sigma' = \sigma_1[ret \mapsto ret_1, C'(r) \mapsto D_1(r)]$ for all $r \in \Pi$.
5. $M = xv$:
   (1) By the inductive hypothesis on $[\![N_1, R_0, C_0, \ldots]\!]$ to $[\![N_n, R_{n-1}, C_{n-1}, \ldots]\!]$, we have $\exists \sigma_n \supseteq \cdots \supseteq \sigma_1 \supseteq \sigma_0.(\sigma_n \simeq \phi_n) \wedge \cdots \wedge (\sigma_1 \simeq \phi_1) \wedge (\sigma_0 \simeq \phi_0)$.
   Since $\sigma(x) \in R$, let $\sigma(x) = m_i$ for $i \in \{1..n\}$.
   Let $\sigma' = \sigma_n[ret \mapsto ret_i, C'_n(r) \mapsto D_i(r)]$ for all $r \in \Pi$.
   This case holds by (1).

Lemma 8 (Preservation of the repository). *Given a translation*

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

we know the input repository must be preserved; i.e. $R' \supseteq R$.

Proof. By inspection of the translation rules.

Lemma 9 (Propagation of preconditions). *Given a translation*

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

we know that preconditions ϕ *and* α *must be propagated and included in* ϕ' *and* α'*; i.e.* $\phi' = \psi \wedge \phi$ *and* $\alpha' = \beta \wedge \alpha$ *where* $[\![M, R, C, D, \top, \top, \top, k]\!] = (ret, \psi, R', C', D', \beta, pc'')$

Proof. By inspection of the translation rules.

Appendix C Time Taken (s) for BMC-2 and MoCHi

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	MoCHi
100_1-e	0.018	0.018	0.028	0.034	0.029	0.056	0.173	0.312	0.605	1.661	5.640	22.778	84.130	-	-	c
100_2	0.013	0.017	0.017	0.020	0.018	0.029	0.028	0.032	0.036	0.032	0.050	0.053	0.053	0.062	0.071	c
100_3-e	0.013	0.016	0.015	0.021	0.024	0.018	0.027	0.022	0.025	0.028	0.034	0.035	0.040	0.056	0.051	10.734
100_4-e	0.012	0.019	0.017	0.022	0.026	0.025	0.026	0.037	0.034	0.036	0.037	0.047	0.040	0.046	0.063	13.547
100_5-e	0.007	0.015	0.019	0.025	0.028	0.048	0.168	0.344	0.576	1.520	5.911	18.821	90.544	-	-	2.345
200_1-e	0.016	0.020	0.021	0.034	0.054	0.074	0.188	0.348	0.660	1.572	4.635	20.209	71.296	-	-	m
200_2-e	0.012	0.023	0.019	0.033	0.042	0.063	0.063	0.094	0.106	0.151	0.175	0.201	0.259	0.298	0.372	-
200_3-e	0.015	0.020	0.030	0.034	0.110	0.453	2.849	19.257	-	-	-	-	-	-	m	1.742
200_4-e	0.030	0.034	0.047	0.051	0.102	0.401	2.661	18.285	-	-	-	-	-	-	m	47.022
200_5-e	0.043	0.036	0.037	0.040	0.047	0.046	0.065	0.085	0.111	0.146	0.168	0.218	0.282	0.388	0.428	m
400_1-e	0.019	0.034	0.060	0.108	0.232	0.814	3.805	24.594	-	-	-	-	-	-	m	m
400_2-e	0.036	0.057	0.050	0.061	0.080	0.129	0.196	0.350	0.523	0.696	0.789	1.033	1.321	1.594	1.991	-
ack	0.011	0.011	0.018	0.027	0.138	1.403	11.519	168.670	-	-	-	-	-	-	-	0.525
a-cppr	0.033	0.021	0.028	0.031	0.018	0.020	0.020	0.018	0.018	0.026	0.026	0.023	0.027	0.026	0.028	28.584
a-init-me	0.018	0.023	0.031	0.027	0.027	0.024	0.015	0.017	0.021	0.025	0.029	0.029	0.034	0.040	0.038	c
a-init	0.014	0.011	0.013	0.018	0.021	0.015	0.016	0.021	0.018	0.032	0.026	0.040	0.042	0.053	0.053	c
a-max-me	0.008	0.013	0.020	0.023	0.017	0.021	0.018	0.016	0.016	0.017	0.015	0.019	0.019	0.018	0.022	5.348
a-max	0.012	0.011	0.010	0.014	0.015	0.017	0.019	0.020	0.015	0.014	0.018	0.016	0.018	0.017	0.017	0.636
copy_intro	0.016	0.019	0.016	0.016	0.019	0.012	0.020	0.020	0.021	0.022	0.020	0.023	0.020	0.024	0.017	c
e-fact	0.010	0.008	0.008	0.009	0.013	0.011	0.014	0.014	0.015	0.016	0.015	0.016	0.021	0.020	0.022	0.629
e-simple	0.010	0.010	0.013	0.010	0.011	0.011	0.008	0.012	0.012	0.007	0.012	0.012	0.009	0.010	0.009	0.098
hors	0.010	0.007	0.010	0.010	0.013	0.013	0.009	0.012	0.012	0.013	0.012	0.011	0.012	0.012	0.014	0.321
hrec	0.012	0.015	0.020	0.020	0.024	0.028	0.075	0.362	3.284	26.175	-	-	-	-	-	0.867
intro1	0.014	0.015	0.014	0.013	0.013	0.013	0.008	0.007	0.014	0.013	0.011	0.011	0.012	0.007	0.009	0.123
intro3	0.011	0.013	0.015	0.011	0.012	0.011	0.013	0.009	0.010	0.009	0.011	0.010	0.008	0.012	0.007	0.110
l-zipmap	0.009	0.012	0.015	0.016	0.012	0.018	0.021	0.019	0.026	0.026	0.026	0.023	0.031	0.031	0.033	c
l-zipunzip	0.013	0.011	0.008	0.009	0.016	0.017	0.019	0.022	0.014	0.025	0.030	0.037	0.048	0.049	0.062	c
max	0.012	0.009	0.011	0.008	0.011	0.009	0.013	0.011	0.011	0.013	0.013	0.011	0.011	0.013	0.013	0.172
mc91-e	0.010	0.013	0.010	0.012	0.017	0.052	0.177	0.330	0.895	3.513	12.128	71.434	-	-	-	5.119
mc91	0.012	0.011	0.013	0.018	0.021	0.054	0.179	0.348	0.890	3.565	12.137	73.717	-	-	-	0.173
mult-e-m2	0.009	0.012	0.012	0.011	0.012	0.012	0.013	0.019	0.018	0.015	0.019	0.021	0.022	0.024	0.021	0.426
mult-e-m3	0.012	0.014	0.014	0.012	0.012	0.016	0.018	0.014	0.018	0.018	0.014	0.012	0.018	0.018	0.017	0.152
mult-e	0.015	0.012	0.015	0.018	0.012	0.016	0.018	0.016	0.018	0.017	0.021	0.021	0.017	0.017	0.016	c
mult	0.013	0.015	0.016	0.016	0.016	0.016	0.015	0.017	0.016	0.014	0.014	0.016	0.012	0.014	0.011	0.153
repeat-e	0.011	0.013	0.013	0.014	0.012	0.013	0.013	0.011	0.013	0.011	0.016	0.012	0.014	0.014	0.012	0.189
r-lock-e	0.009	0.009	0.013	0.009	0.013	0.011	0.013	0.012	0.013	0.013	0.009	0.010	0.007	0.011	0.011	0.216
r-lock	0.007	0.012	0.009	0.009	0.010	0.013	0.013	0.010	0.011	0.010	0.010	0.011	0.012	0.012	0.006	0.135
sum-e	0.008	0.013	0.009	0.013	0.012	0.014	0.015	0.013	0.017	0.015	0.010	0.014	0.015	0.016	0.013	0.145
sum-mult-e	0.007	0.011	0.013	0.016	0.019	0.022	0.050	0.146	0.218	0.207	0.237	0.286	0.323	0.328	0.301	0.238
sum	0.011	0.010	0.013	0.013	0.013	0.014	0.012	0.012	0.011	0.013	0.013	0.016	0.016	0.015	0.017	0.330
ref-1	0.007	0.009	0.014	0.013	0.011	0.007	0.007	0.011	0.012	0.013	0.010	0.011	0.012	0.011	0.011	u
ref-1-e	0.023	0.007	0.019	0.016	0.020	0.020	0.008	0.010	0.013	0.014	0.013	0.014	0.014	0.012	0.009	u
ref-2	0.011	0.007	0.011	0.013	0.009	0.010	0.008	0.011	0.012	0.011	0.010	0.010	0.012	0.009	0.010	u
ref-2-e	0.011	0.008	0.009	0.011	0.009	0.012	0.013	0.013	0.013	0.010	0.008	0.011	0.013	0.011	0.011	u
ref-3	0.012	0.013	0.015	0.019	0.016	0.015	0.018	0.022	0.028	0.047	0.099	0.209	0.211	0.211	0.211	u