Modular Verification of Procedure Equivalence in the Presence of Memory Allocation

Tim Wood[1], Shuvendu K. Lahiri[2], Sophia Drossopolou[1], and Susan Eisenbach[1]

[1] Imperial College
London, UK
[2] Microsoft Research
Redmond, USA

Abstract. For most high level languages, two procedures are equivalent if they transform a pair of isomorphic stores to isomorphic stores. However, tools for modular checking of such equivalence impose a stronger check where isomorphism is strengthened to equality of stores. This results in the inability to prove many interesting program pairs with recursion and dynamic memory allocation.
In this work, we present RIE, a methodology to modularly establish equivalence of procedures in the presence of memory allocation, cyclic data structures and recursion. Our technique addresses the need for finding witnesses to isomorphism with angelic allocation, supports reasoning about equivalent procedures calls when the stores are only locally isomorphic, and reasoning about changes in the order of procedure calls. We have implemented RIE by encoding it in the Boogie program verifier. We describe the encoding and prove its soundness.

Keywords: Program Equivalence, Program Verification, Version-aware Verification

1 Introduction

Program maintenance dominates the program lifecycle. A study of application bugs that took more than one attempt to fix [**Park2012**] found that 22%-33% of fixes required a supplementary fix, and found a diverse range of errors including incomplete refactorings. A study of refactorings [**Bavota2012**] found that across 12,922 refactorings from three software projects, 15% of refactorings induced a bug. *Automatic program equivalence verification* [**Godlin09**, **Lahiri2012**, **Partush2014**, **Felsing2014**] offers the potential to reduce problems by allowing a programmer to automatically (without programmer annotations) verify that the new version is behaviourally equivalent to the old.

The goal of these verification tools is to make the benefits of program equivalence verification available to programmers who are not verification experts. An automatic equivalence verification tool *takes a pair of programs as input and then outputs whether the programs are equivalent or not* (or perhaps times-out). Program equivalence is undecidable in the general case, however, some success has been achieved on programs with substantially similar structure. Since software is frequently modified in small incremental steps, versions tend to be structurally similar.

We know of two tools designed for fully-automatic program equivalence verification of programs with heaps: Symdiff [**Lahiri2012**] and RVT [**Godlin09**]. Symdiff [**Kawaguchi2010**, **Lahiri2012**, **Lahiri2013**, **Hawblitzel2013**] is built on top of the Boogie [**Barnett2005**] intermediate verification language which, in turn, uses the Z3 [**DeMoura2008**] satisfiability modulo theories (SMT) solver to discharge proof obligations. RVT uses a designed-for-purpose verification algorithm, which passes program fragments to the CMBC [**Clarke2003**] bounded model checker.

Symdiff relates heaps using equality (of arrays modelling the heaps), which we will call *e-equivalence*, so programs that differ in the order or amount of dynamic memory allocation or garbage cannot be verified as equivalent by Symdiff. RVT does support differences in allocation, but assumes that all heap data structures are tree-like.[3]

E-equivalence is too restrictive for programmers, who expect to be able to replace one procedure with another if the two have identical observable behaviour. A more intuitive notion of procedure equivalence for programs with dynamic memory allocation can be constructed using isomorphism between memory locations. Definitions of program equivalence based on a notion of isomorphism have been used in several formal systems [**Benton2007**, **Pitts2002**]. Our definition of equivalence is:

> *Two procedures are* equivalent*, if they terminate for the same set of initial stores, and if both procedures run to completion from isomorphic initial stores, they result in isomorphic final stores.*

Our definition of equivalence matches intuition: it allows for differences in the order or amount of memory allocation and garbage and is not restricted to tree-like structures. Achieving automatic verification presents several challenges:

Challenge 1 What kind of input do we need to give to an SMT solver so that it can even do the verification? We need to establish an isomorphism between unbounded heaps of arbitrary shape when this is computationally infeasible in general. Furthermore, a direct axiomatisation of isomorphism involves existentially quantifying the mapping between memory locations that characterises the isomorphism. SMT based verification systems are not very good at producing witnesses to such existentials and so a direct axiomatisation of isomorphism is ineffective.

Sometimes calls to equivalent procedures occur from stores that are not fully isomorphic, rather the stores are isomorphic in the footprint of the called procedures. This leads to the next two challenges:

[3] For details, see *Definition 2* (and the paragraph following) on page 5 of the 2009 paper by **Godlin09** [**Godlin09**].

Challenge 2 How can our tool determine when stores correspond in the footprint of a called procedure?

Challenge 3 What should we do about equivalent calls from non-isomorphic stores since they do not necessarily result in isomorphic stores?

Challenge 4 How do we decide which calls may be equivalent when there may be many possible candidates? Equivalent calls may not occur in the same order in each procedure, and moreover the procedure calls which correspond may differ from execution to execution depending on the initial state (i.e. program inputs).

1.1 Example

Consider the pair of equivalent procedures in Fig. 1 that differ in the order of memory allocation.

```
 1 lcopy(t,r) modifies {r} {          14 rcopy(t,r) modifies {r} {
 2   if(t != null  ∧  r != null) {     15   if(t != null  ∧  r != null) {
 3       rl := new;                     16       rl := new;
 4       rr := new;                     17       rr := new;
 5       n := new;                      18
 6                                      19       rcopy(t.r, rr);
 7       lcopy(t.l, rl);                20       rcopy(t.l, rl);
 8       lcopy(t.r, rr);                21
 9                                      22       n := new;
10       n.l := rl.v;                   23       n.l := rl.v;
11       n.r := rr.v;                   24       n.r := rr.v;
12       r.v := n;                      25       r.v := n;
13 } }                                  26 } }
```

Fig. 1. Both procedures copy a binary tree.

Both procedures are intended to copy the passed structure `t`. The procedures are equivalent on *any* input, whether the input is tree shaped or not. Our methodology RIE (**R**eplace **I**somorphism with **E**quality) and tool APE (**A**utomatic **P**rogram **E**quivalence tool) can discover that. The procedures differ in two ways: Firstly, the allocation of the copied node has been moved from before the recursive calls on line 5 to after the recursive calls on line 22. Secondly, the order of the recursive calls on lines 7 and 8 has been reversed on lines 19 and 20. The procedures are written in a simple language we call $\mathcal{L}$, formalised in Section 3.

The procedures are equivalent. An intuition as to why is: when `t` or `r` are `null` both procedures leave the heap unchanged. Otherwise, both procedures recursively copy all the nodes to the left, and all the nodes to the right, and return a newly allocated root node via the parameter `r`. The only pre-existing object modified by the procedures is the one pointed to by `r`. It is possible that `r` aliases a node reachable from `t`, but even so only the `v` field of `r` is written to, and only after the nodes have been copied. The objects allocated to `rl` and `rr` do not alias anything, so the recursive calls cannot modify the tree, and hence swapping the order of the recursive calls does not affect the result. The postcondition `modifies {r}` asserts that no existing object, other than `r`, is modified. For this example, our tool APE requires this framing assertion. Our approach can take advantage of any contracts that are available.

The procedures are not e-equivalent (so would fail to match in Symdiff) when the stores are related with equality. With non-deterministic allocation a procedure is not even equivalent to itself! It is straightforward to resolve with a deterministic allocator, e.g. one that starts at 0 and allocates the next address. Under this deterministic approach a procedure is e-equivalent with itself, but `lcopy` and `rcopy` are still not e-equivalent as the allocations on Line 5 and Line 22 are allocated different addresses.

The example illustrates the challenges in the following ways. **Challenge 1**: equivalence requires that the final stores are isomorphic, but the recursive calls are unbounded so a tool has to check isomorphism of a graph of arbitrary shape and unbounded size. **Challenge 2**: the stores are not equivalent prior to the recursive calls. For instance, in `lcopy` three allocations (Lines 3 to 5) have occurred before the recursive calls, but in `rcopy` only two allocations have occurred (Lines 16 and 17). **Challenge 3**: the stores after the related calls on Line 7 and Line 20 are not generally isomorphic, since the store after Line 20 contains the effects of two recursive calls but the store after Line 7 contains the effect of only one recursive call. **Challenge 4**: we do not know in advance which recursive call in `lcopy` (Lines 7 and 8) corresponds to which recursive call in `rcopy` (Lines 19 and 20).

1.2 Contributions

We propose a sound methodology, RIE, for establishing isomorphic procedure equivalence which is effective in an SMT solver. RIE enables our tool APE to tackle **challenge 1** by automatically establishing isomorphism using under-approximation and heap equality! RIE works by proving equivalence under the *angelic allocation* assumption; the memory locations are, as far as possible, assumed to be allocated in such a way as to make isomorphic heaps also equal.

We describe a simple language $\mathcal{L}$ for which isomorphism implies equivalence of observable behaviours, and give a formal definition of what it means for $\mathcal{L}$ to be closed under isomorphism.

RIE also simplifies challenges two, three and four. RIE allows us to use equality in place of isomorphism, so **challenge 2** is addressed by extending the notion of heap equality to support partial heap equality, which we then apply to an over-approximation of the footprint of the procedures. Furthermore RIE rescues us from the need to produce a witness (from **challenge 1**) to the correspondence between procedure behaviour, instead we address **challenge 3** by equating the write effects of equivalent procedures (soundly ignoring unobservable behavioural differences). We combine our technique with mutual summaries to address **challenge 4**.

In Section 2 we describe how RIE can be implemented to verify program equivalence. In Section 3 we formalise equivalence and isomorphism and outline RIE's soundness proof. In Section 4 we discuss the effectiveness and limitations of RIE. Finally in Section 5 we discuss some related work and conclude.

2 Encoding in a Verifier

Our tool APE takes as input an L program and produces as output 'success', 'failure', or times out. It does this by translating the input program into Boogie code, which is fed into the Z3 SMT solver. The source code is available at `https://github.com/lexicalscope/ape`.

In this section we illustrate RIE by showing how APE encodes the example from Section 1.1 and how that encoding helps overcome the challenges detailed in the introduction. In particular, we show how verification under the assumption of "angelic allocation" proceeds. Previous work typically takes the approach of abstracting or overapproximating programs with dynamic allocation [**Tennent2000**, **Koutavas2006**, **Benton2007**, **Tzevelekos2012**], *storeless semantics* [**Bozga2003**] go as far as abstracting away the observable store entirely. We make the surprising observation that under-approximating memory allocation is also a useful approach, and our formal system proves it sound. RIE establishes procedure equivalence by checking equivalence for only one pair of execution traces for each initial store. Specifically:

> *All* pairs of executions from isomorphic initial stores result in *isomorphic* final stores if at least *one* pair of executions from each initial store results in *equal* final stores.

In particular, it is not necessary to consider all pairs of isomorphic initial stores. We prove this in Section 3.

RIE combines our ideas about establishing isomorphism using SMT technology with the prior work on product programs and mutual summaries to produce an automatic program equivalence tool that can verify our example. Standard single program verification tools can be applied to the problem of procedure equivalence using *product programs* [**Barthe2004**, **Terauchi2005**, **Godlin09**, **Lahiri2012**, **Barthe2016**], which encode the bodies of a pair of procedures into a single procedure such that verifying a safety property of the product procedure is equivalent to verifying a relational property of the procedure pair [**Barthe2016**]. Furthermore, a technique called *mutual summaries* [**Hawblitzel2013**] can be applied to induce an SMT solver to search for interesting relations between procedure calls.

2.1 Angelic Allocation

APE checks equivalence for one pair of executions for each initial store. It does so by searching amongst the possible pairs of executions for a pair that result in equal stores (modulo garbage). Of interest, then, are pairs of executions where

```
27 procedure lcopy_rcopy(..., h:Heap, t,r:Ref) {
28 // make twelve copies of the initial heap, h
29 h_a0,h_a1,h_b0,h_b1,h_c0,h_c1,h_d0,h_d1,h_e0,h_e1,h_f0,h_f1
30     := h,h,h,h,h,h,h,h,h,h,h,h;
31
32 // inline each procedure once with variables renamed with the given suffix
33 inline lcopy with variable suffix _a0 and heap h_a0
34 inline rcopy with variable suffix _a1 and heap h_a1
35 assume rl_a0 = rl_a1 ∧ rr_a0 = rr_a1 ∧ n_a0 = n_a1;
36
37 // and again for correspondence b
38 inline lcopy with var suffix _b0 and heap h_b0
39 inline rcopy with var suffix _b1 and heap h_b1
40 assume rl_b0 = rl_b1 ∧ rr_b0 = n_b1 ∧ n_b0 = rr_b1;
41
42 // and so on for c,d,e,f
43
44 assert equal#heaps(...,h_a0,h_a1)
45     ∨ equal#heaps(...,h_b0,h_b1)
46     ∨ ... /* and so on for c,d,e,f */ }
```

Fig. 2. A product procedure encoding of equivalence verification under angelic memory allocation for procedures `lcopy` and `rcopy`.

particular allocation sites (`new`) in each procedure are allocated the same addresses. In Fig. 1 there are three allocation sites in each procedure. This gives six possible correspondences between allocation sites (the variables on the left of the equality are from `lcopy`, and the variables on the right are from `rcopy`):

$$\begin{array}{llll} a) & \texttt{rl} = \texttt{rl}, \texttt{rr} = \texttt{rr}, \texttt{n} = \texttt{n} & b) & \texttt{rl} = \texttt{rl}, \texttt{rr} = \texttt{n}, \texttt{n} = \texttt{rr} \\ c) & \texttt{rl} = \texttt{rr}, \texttt{rr} = \texttt{rl}, \texttt{n} = \texttt{n} & d) & \texttt{rl} = \texttt{rr}, \texttt{rr} = \texttt{n}, \texttt{n} = \texttt{rl} \\ e) & \texttt{rl} = \texttt{n}, \texttt{rr} = \texttt{rl}, \texttt{n} = \texttt{rr} & f) & \texttt{rl} = \texttt{n}, \texttt{rr} = \texttt{rr}, \texttt{n} = \texttt{rl} \end{array}$$

We do not know in advance which correspondence will be useful for verification[4]. In our example, it happens that correspondence (a) is the useful one. Pairs of (terminating) executions from the same initial store that have allocations in this correspondence will result in equal final stores. Hence, no *direct* checks for isomorphism are required. We will detail how procedure calls are handled shortly.

We induce the solver to search for the useful correspondence by constructing a pair of executions for each correspondence and using a disjunction to assert that at least one of them results in equivalent final stores. The Boogie-like pseudo code in Fig. 2 shows how APE encodes `lcopy` and `rcopy` into a single (Symdiff-style) product procedure. The `inline` commands (e.g. line 33) are not actual Boogie syntax but should be taken to mean that the statements from the body of the relevant procedure are copied into the product procedure at that point.

[4] It is interesting to note that in this example the variable names suggest which correspondence is important, perhaps indicating that there may be useful heuristics that could improve performance — such as trying correspondence (a) first.

```
function $Heap#ReachableEqual($h_1, $h_2:Heap, $roots:Roots) : bool {
 $Heap#ReachableEqual'($h_1, $h_2, $roots) ∧
 $Heap#ReachableEqual'($h_2, $h_1, $roots) }

function $Heap#ReachableEqual'($h_1, $h_2:Heap, $roots:Roots) : bool {
 (∀$a:Ref :: {...} (∀<alpha> $f:Field alpha ::
  $Reachable($h_1, $roots, $a) ⟹ $Read($h_1, $a, $f) = $Read($h_2, $a, $f)))}

function $Reachable($h:Heap, $roots:Roots, $a:Ref) : bool {
 (∃$r:Ref :: {...} $Root($roots, $r) ∧ $Reach($h, $r, $a)) }
```

Fig. 3. Extensional equality of the reachable heap region. Written in Boogie.

When the procedures are inlined, they are rewritten to work on their own private copy of the heap with fresh variable names. After each inlined pair a different correspondence between allocation sites is assumed (lines 35 and 40). Finally a disjunction is asserted to challenge the verifier to prove that the final heaps are equal for at least one of the inlined pairs (line 44).

Thus, RIE allows us to establish isomorphism using only heap equality!

2.2 Heap Equality

Here we described how APE establishes heap equality, and discuss why our approach is powerful. Tools can relate programs in an intensional or extensional way [**Benton2004**]. Intensionally equal heaps are defined in the same way, whereas extensionally equal heaps have the same observable properties. For example, the heaps[5] $h_1 = h_0[(5,\texttt{f}) \mapsto 7][(5,\texttt{g}) \mapsto 8]$ and $h_2 = h_0[(5,\texttt{g}) \mapsto 8][(5,\texttt{f}) \mapsto 7]$ are not intensionally equal, but they are extensionally equal. Extensional relationships provide a powerful means to reason about reordering of store updates.

APE uses the extensional axiomatisation of heap equality shown in Fig. 3. The axiomatisation allows procedures to create different garbage by only requiring equality of the reachable heap. The parameter `$roots` is a set of references, and it overapproximates the references that are on the stack. The predicate `$Reachable` is an axiomatisation of heap reachability and is discussed in Section 4.3.

We define equality between heaps using a pair of implications that say that if an object is reachable in either heap, its fields must be equal in both heaps. An alternative, and perhaps more obvious, definition would be that the reachable sets are equal, and that each object in the reachable set has equal fields in both the heaps. The definitions are equivalent — using a pair of implications implicitly relies on the fact that equal heaps have the same reachability relation. But our definition avoids causing the solver to directly try to prove that the reachable sets are equal. The solver is unable to directly prove that the reachability sets are the same on several examples.

[5] $h_0[(5,\texttt{f}) \mapsto 7]$ is the heap made by copying h_0 and setting field `f` of object 5 to 7.

2.3 Procedure Call

Challenges two, three and four in the introduction relate to the need to reason modularly about the behaviour of nested procedure calls, such as the recursive calls in the example Fig. 1. In this section we describe how our encoding in Fig. 4 leverages RIE to address these challenges.

```
57 function $abs_lcopy($strat:int, $h_pre:Heap, t:Ref,r:Ref, $h_post:Heap):bool;
58 function $abs_rcopy($strat:int, $h_pre:Heap, t:Ref,r:Ref, $h_post:Heap):bool;
59
60 procedure lcopy($strat:int, $h_pre:Heap, t:Ref,r:Ref) returns ($h_post:Heap)
61  free ensures $abs_lcopy($strat, $h_pre, t, r, $h_post);
62  ...
63  free ensures (∀<alpha> $a:Ref,$f:Field alpha ::
64   $a≠$Null ∧ $Allocated($h_pre,$a) ∧ $Read($h_pre,$a,$f)!=$Read($h_post,$a,$f)
65   ⟹$ReachableFromParams($h_pre, t, r, $a));
66 procedure rcopy($strat:int, $h_pre:Heap, t:Ref,r:Ref) returns ($h_post:Heap)
67  free ensures $abs_rcopy($strat, $h_pre, t, r, $h_post);
68  ...
69  free ensures (∀<alpha> $a:Ref,$f:Field alpha ::
70   $a≠$Null ∧ $Allocated($h_pre,$a) ∧ $Read($h_pre,$a,$f)!=$Read($h_post,$a,$f)
71   ⟹$ReachableFromParams($h_pre, t, r, $a));
72
73 axiom (∀$strat:int,$h_pre_0,$h_post_0,$h_pre_1,$h_post_1:Heap,
74         t_0,r_0,t_1,r_1:Ref ::
75  {$abs_lcopy($strat,$h_pre_0, t_0, r_0, $h_post_0),
76   $abs_rcopy($strat,$h_pre_1, t_1, r_1, $h_post_1)}
77  $abs_lcopy($h_pre_0, t_0, r_0, $h_post_0)∧
78  $abs_rcopy($h_pre_1, t_1, r_1, $h_post_1)
79  ∧$Heap#EqualFromParams($h_pre_0, t_0, r_0, $h_pre_1, t_1, r_1)
80   ⟹$SameDiff($h_pre_0, $h_post_0, $h_pre_1, $h_post_1));
81 function $SameDiff($h_pre_0, $h_post_0, $h_pre_1, $h_post_1:Heap) : bool {
82  (∀<alpha> $a:Ref,$f:Field alpha ::
83   ($Read($h_pre_0, $a, $f)≠$Read($h_post_0, $a, $f)
84   ⟹$Read($h_post_0, $a, $f)==$Read($h_post_1, $a, $f))∧
85   ($Read($h_pre_1, $a, $f)≠$Read($h_post_1, $a, $f)
86   ⟹$Read($h_post_0, $a, $f)==$Read($h_post_0, $a, $f))) }
```

Fig. 4. Mutual summary of the `lcopy` and `rcopy` procedures. Written in Boogie.

Equivalent procedure calls do not always occur from isomorphic stores (**challenge 2**). This is overcome by considering only the region of the heap reachable from the procedure parameters when trying to establish equivalence of procedure calls. This corresponds to the predicate `$Heap#EqualFromParams` on line 79, detailed in Section 2.2.

Equivalent procedure calls do not necessarily result in isomorphic stores (**challenge 3**). This is overcome through our choice of procedure summary (line 80) and a frame axiom. The write effects of a pair of equivalent procedure calls are related by the predicate `$SameDiff` (line 81). The frame axiom appears as a free postcondition[6] of every procedure (lines 63 and 69). Both are described below.

Equivalent procedure calls are summarised by the predicate `$SameDiff` that relates the pre and post stores of both calls. `$SameDiff` approximates the actual behaviour of the procedures a surprising way, although equivalent procedures can vary in the amount and shape of garbage, `$SameDiff` states that equivalent procedure calls will always have equal effects, we justify this in Section 3.

The framing axioms (lines 63 and 69) restrict the write effects of the procedures to the part of the heap that was reachable from the procedure parameters. The axiom follows from the semantics of $\mathcal{L}$. It is not known in advance which procedure calls might be equivalent (**challenge 4**). This is overcome by how the summary of the behaviour of equivalent procedures is encoded. Specifically, the encoding of mutual summaries presented by **Hawblitzel2013** [**Hawblitzel2013**] is used to induce the SMT solver to search for related pairs of procedure calls. We detailed this encoding below.

Fig. 4 shows how APE encodes the mutual summary for the procedures `lcopy` and `rcopy`. The encoding consists of several parts. Encountered calls to the procedures are abstracted by a pair of uninterpreted predicates (lines 57 and 58) which we call *abstraction predicates*. The predicates are uninterpreted so we precisely control their instantiation. They are given as free postconditions of the procedures `lcopy` (line 61) and `rcopy` (line 67). Encountering calls to these procedures causes the abstraction predicates to be instantiated in the solver's E-graph. We set *triggers* (lines 75 and 76) so that the solver will instantiate the mutual summary axiom's quantifiers (line 74) for each pair of instantiations of the abstraction predicates. Non-vacuous instantiations of the axiom occur when the solver is able to establish that the heaps reachable from the call parameters are equal, and hence the antecedent is satisfied.

During a Simplify [**Detlefs2005**] style SMT solver proof search the quantifiers that appear in an axiom are instantiated with ground terms from the solver's E-graph that match *triggers* associated with the quantifier. In-turn, the axiom is applied to those instantiations to introduce new terms into the E-graph and so on. Thus, APE controls the proof search by a combination of the logical meaning of the axioms and the quantifier triggers.

The synthetic parameter `$strat` of `lcopy` (line 60) and `rcopy` (line 66) is introduced to prevent the proof search from trying to establish equivalence between procedure calls occurring under different allocation correspondences. For example, in Fig. 2, the inlining of `lcopy` on line 33 and line 39 will contain recursive calls to `lcopy` and `rcopy` respectively — but it is not useful to find relations between these calls as they pertain to different allocations correspondences. Specifically, the disjunction on line 44 asserts nothing about the relationship between those heaps. The parameter `$strat` represents the allocation correspondence in effect for that call, and the mutual summary trigger (lines 75 and 76 of Fig. 4) restricts instantiation of the quantifiers to calls which occurred under the same allocation correspondence.

[6] A free postcondition may be assumed after a call, but is not checked.

We have completed our illustration of how the RIE methodology is implemented in a modular program equivalence tool. Discussion about the effectiveness and limitations of our approach is in Section 4.

3 Soundness of RIE

We now give a model of RIE and summarise a proof of its soundness. The semantics of $\mathcal{L}$ come in two flavours: the $\mathcal{V}$ semantics is the ordinary semantics of $\mathcal{L}$. The $\mathcal{A}$ semantics models our Boogie encoding, which includes the various **a**pproximations detailed in Section 2. We establish soundness of RIE by showing that equivalence under $\mathcal{A}$ implies equivalence under $\mathcal{V}$.

We expect a mapping between procedures names, $\mathcal{E}$, which pairs procedures that are suspected to be equivalent. RIE takes the program and $\mathcal{E}$ as input and tries to prove that indeed all pairs in $\mathcal{E}$ are equivalent.

We start with an overview of the semantics of $\mathcal{L}$, then define isomorphism and procedure equivalence. Then we detail how the various approximations are modelled in the $\mathcal{A}$ semantics. Finally we give RIE's soundness theorem[7].

3.1 Semantics of $\mathcal{L}$

$\mathcal{L}$ is a simple imperative language. Fig. 5 describes the standard aspects of $\mathcal{L}$, while Fig. 6 describes the non-standard aspects. The following points are interesting about our semantics:

- We distinguish between execution under $\mathcal{V}$ and $\mathcal{A}$ with a subscript, writing $\rightsquigarrow_{\mathcal{L}}$ where $\mathcal{L} \in \{\mathcal{V}, \mathcal{A}\}$.
- We split procedure call into two: a "call" rule and a "body" rule, similar to **Godlin09** [**Godlin09**], to treat procedure call concretely in $\mathcal{V}$ but abstractly in $\mathcal{A}$. The latter is a first ingredient in reflecting mutual summaries.
- Execution of pairs of procedures is included in the operational semantics, modelled by the rules COMV and COMA:

  $$\sigma_1, s_1 \parallel \sigma_2, s_2 \overset{tr_1, tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \sigma_3 \parallel \sigma_4$$

  meaning statement s_1 executed on store σ_1 results in store σ_3 producing trace tr_1, and similarly for statement s_2. These rules reflect product programs [**Lahiri2012**, **Barthe2016**, **Banerjee2016**].
- We require the program adhere to a specification given by Con[8].
- The semantics are instrumented to produce a trace of the states reached during execution which we use to distinguish particular executions.
- although our semantics is a big step semantics, we keep the whole calling context as part of the runtime configuration to allow us to give useful meaning to isomorphism of stores.

[7] Full details of the proof can be found in the first author's PhD thesis [**Wood2016**].

[8] We expect single program contracts to have been verified; programs with no contracts are also acceptable.

$$\begin{array}{lll}
s_a \in \mathit{AtomicStmt} & := & x := e \mid x := b \mid x := \texttt{new}() \mid \texttt{assume}\ b \mid \texttt{assert}\ b \mid \texttt{call}\ \texttt{p}(x, \ldots, x) \\
 & & \mid e.f := e \\
s \in \mathit{Stmt} & := & s_a \mid \texttt{if}(b)\{s_a\} \mid s; s \\
e \in \mathit{ScalarExpr} & := & \texttt{null} \mid x \mid e.f \mid \mathrm{b} \\
b \in \mathit{BoolExpr} & := & x = y \mid !b \mid b \wedge b \mid \texttt{true} \mid \texttt{false}
\end{array}$$

Sets $a \in \mathit{Addr}$, $x \in \mathit{Lid}$, $f \in \mathit{Fid}$, $p \in \mathit{Pid}$, $v \in \mathit{Val} \overset{\text{def}}{=} \mathit{Addr} \cup \{\texttt{true}, \texttt{false}\}$.
Set Addr has one reserved element $\texttt{null}$.

$$\begin{array}{ll}
h \in \mathit{Heap} & \overset{\text{def}}{=} (\mathit{Addr} \times \mathit{Fid}) \rightharpoonup \mathit{Val} \text{ where } \forall h, f : h(\texttt{null}, f) = \texttt{null}. \\
\phi \in \mathit{StackF} & \overset{\text{def}}{=} \mathit{Lid} \rightharpoonup \mathit{Val} \\
\tilde{\phi} \in \mathit{Stack} & \overset{\text{def}}{=} \mathit{StackF}^* \\
\sigma \in \mathit{Store} & \overset{\text{def}}{=} \mathit{Stack} \times \mathit{Heap}, \text{ where } \sigma(x) \text{ means } \phi(x) \text{ when } \sigma = (\tilde{\phi} \cdot \phi, h) \\
tr \in \mathit{Trace} & \overset{\text{def}}{=} (\mathit{Stmt} \times \mathit{Store} \times \mathit{Store})^* \\
\mathit{mkframe}(\sigma, x_1 \ldots x_n) & \overset{\text{def}}{=} (\sigma^{\texttt{stack}} \cdot [x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)], \sigma^{\texttt{heap}}) \\
\mathcal{B} : \mathit{Pid} \to \mathit{Stmt} & \text{looks up procedure bodies from procedure names.}
\end{array}$$

$\sigma^{\texttt{heap}} \overset{\text{def}}{=} h$ and $\sigma^{\texttt{stack}} \overset{\text{def}}{=} \tilde{\phi} \cdot \phi$ and $\sigma^{\texttt{pop}} \overset{\text{def}}{=} (\tilde{\phi}, h)$ and $\sigma^{\texttt{top}} \overset{\text{def}}{=} \phi$ when $\sigma = (\tilde{\phi} \cdot \phi, h)$

$$\frac{}{\sigma, e_1.f := e_2 \hookrightarrow (\sigma^{\texttt{stack}}, \sigma^{\texttt{heap}}[(\llbracket e_1 \rrbracket_\sigma, f) \mapsto \llbracket e_2 \rrbracket_\sigma])}\ \text{STORE}$$

$$\frac{a \notin \mathit{dom}(h) \quad \{f_1, \ldots, f_n\} = \mathit{Fid} \quad a \neq \texttt{null}}{(\tilde{\phi} \cdot \phi, h), x := \texttt{new}() \hookrightarrow (\tilde{\phi} \cdot \phi[x \mapsto a], h[(a, f_1) \mapsto \texttt{null}, \ldots, (a, f_n) \mapsto \texttt{null}]))}\ \text{NEW}$$

$$\frac{\sigma \vDash b}{\sigma, \texttt{assert}\ b \hookrightarrow \sigma}\ \text{ASSERTT} \qquad \frac{\sigma_1, s \hookrightarrow \sigma_2}{\sigma_1, s \rightsquigarrow_{\mathcal{L}}^{s, \sigma_1, \sigma_2} \sigma_2}\ \text{ATOM}$$

$$\frac{\sigma \vDash b}{\sigma, \texttt{assume}\ b \hookrightarrow \sigma}\ \text{ASSUME} \qquad \frac{\neg(\sigma \vDash b)}{\sigma, \texttt{assert}\ b \hookrightarrow \texttt{error}}\ \text{ASSERTF}$$

$$\frac{}{\sigma, x := e \hookrightarrow (\sigma^{\texttt{stack}}[x \mapsto \llbracket e \rrbracket_\sigma], \sigma^{\texttt{heap}})}\ \text{ASSIGN} \qquad \frac{\sigma_1 \vDash b \quad \sigma_1, s_a \rightsquigarrow_{\mathcal{L}}^{tr} \sigma_2}{\sigma_1, \texttt{if}(b)\{s_a\} \rightsquigarrow_{\mathcal{L}}^{tr} \sigma_2}\ \text{CONDT}$$

$$\frac{\neg(\sigma \vDash b)}{\sigma, \texttt{if}(b)\{s_a\} \rightsquigarrow_{\mathcal{L}}^{\texttt{if}(b)\{s_a\}, \sigma, \sigma} \sigma}\ \text{CONDF} \qquad \frac{\sigma_1, s_1 \rightsquigarrow_{\mathcal{L}}^{tr_1} \sigma_2 \quad \sigma_2, s_2 \rightsquigarrow_{\mathcal{L}}^{tr_2} \sigma_3}{\sigma_1, s_1; s_2 \rightsquigarrow_{\mathcal{L}}^{tr_1 \cdot tr_2} \sigma_3}\ \text{TRANS}$$

$$\frac{}{\llbracket \texttt{null} \rrbracket_\sigma = \texttt{null}} \qquad \frac{}{\llbracket x \rrbracket_\sigma = \sigma(x)} \qquad \frac{\neg(\sigma \vDash b)}{\sigma \vDash !b} \qquad \frac{\sigma \vDash b}{\llbracket b \rrbracket_\sigma = \texttt{true}} \qquad \frac{\neg(\sigma \vDash b)}{\llbracket b \rrbracket_\sigma = \texttt{false}}$$

$$\frac{}{\llbracket e.f \rrbracket_\sigma = \sigma^{\texttt{heap}}(\llbracket e \rrbracket_\sigma, f)} \qquad \frac{\sigma(x) = \sigma(y)}{\sigma \vDash x = y} \qquad \frac{\sigma(x) = \texttt{true}}{\sigma \vDash x} \qquad \frac{\sigma \vDash b_1 \quad \sigma \vDash b_2}{\sigma \vDash b_1 \,\&\&\, b_2}$$

Fig. 5. Grammar and Operations of $\mathcal{L}$

$$\frac{(\sigma_1, \sigma_3) \in \mathit{Con}(\mathtt{p}) \quad \sigma_1, \mathcal{B}(\mathtt{p}) \rightsquigarrow_{\mathcal{L}}^{tr} \sigma_3}{\sigma_1, \mathtt{body\ p} \rightsquigarrow_{\mathcal{L}}^{tr \cdot (\mathtt{ret}, \sigma_3, {\sigma_3}^{\mathtt{pop}})} {\sigma_3}^{\mathtt{pop}}}\ \text{BOD}$$

$$\frac{\mathit{mkframe}(\sigma_1, x_1 \ldots x_n), \mathtt{body\ p} \rightsquigarrow_{\mathcal{V}}^{tr} \sigma_3}{(\sigma_1, \mathtt{call\ p}(x_1 \ldots x_n) \rightsquigarrow_{\mathcal{V}}^{(\mathtt{call\ p}(x_1 \ldots x_n), \sigma_1, \sigma_3)} \sigma_3}\ \text{CALLV}$$

$$\frac{\begin{array}{c} \mathit{dom}(h_2) \supseteq \mathit{dom}(h_1) \\ ((\mathit{mkframe}(\sigma_1, x_1 \ldots x_n), h_1)(\tilde{\phi}_1 \cdot \phi, h_2)) \in \mathit{Con}(\mathtt{p}) \end{array}}{(\tilde{\phi}_1, h_1), \mathtt{call\ p}(x_1 \ldots x_n) \rightsquigarrow_{\mathcal{A}}^{(\mathtt{call\ p}(x_1 \ldots x_n), (\tilde{\phi}_1, h_1), (\tilde{\phi}_1, h_2))} (\tilde{\phi}_1, h_2)}\ \text{CALLA}$$

$$\frac{\sigma_1, s_1 \rightsquigarrow_{\mathcal{V}}^{tr_1} \sigma_2 \quad \sigma_2, s_2 \rightsquigarrow_{\mathcal{V}}^{tr_2} \sigma_4}{(\sigma_1, s_1) \parallel (\sigma_2, s_2) \rightsquigarrow_{\mathcal{V}}^{tr_1, tr_2} (\sigma_3, \sigma_4)}\ \text{COMV}$$

$$\frac{\mathcal{I}(tr_1, tr_2) \quad \mathcal{M}(tr_1, tr_2) \quad \sigma_1, s_1 \rightsquigarrow_{\mathcal{A}}^{tr_1} \sigma_2 \quad \sigma_2, s_2 \rightsquigarrow_{\mathcal{A}}^{tr_2} \sigma_4}{(\sigma_1, s_1) \parallel (\sigma_2, s_2) \rightsquigarrow_{\mathcal{A}}^{tr_1, tr_2} (\sigma_3, \sigma_4)}\ \text{COMA}$$

Fig. 6. Procedure call and composition rules of $\mathcal{L}$

– We assume loops will been encoded as recursive procedures.

We only discuss some of the rules of the operational semantics. NEW allocates a new object.[9] The address of the object must be fresh, and the object has all fields set to `null`. ASSIGN updates a stack variable x with the result of evaluating an expression e in the store σ.

BOD executes the body of a procedure `p` by looking up and executing `p`'s statements. Our assumption that procedure contracts have already been verified, is modelled by the requirement $(\sigma_1, \sigma_3) \in \mathit{Con}(\mathtt{p})$. Note that BOD pops the top of the final stack, while CALLA and CALLV push new frames (using the function *mkframe*).

CALLA models abstraction of procedure calls. The behaviour of the abstracted call is restricted by the procedure's contract $\mathit{Con}(\mathtt{p})$, and the call may not free any allocated address (All that RIE actually requires is that the language not have concrete addresses so any language with garbage collection which does not support pointer arithmetic can be handled.)

Angelic allocation in the rule COMA is modelled by the predicate over trace pairs $\mathcal{I}$. The behaviour of called procedures in the $\mathcal{A}$ semantics is modelled by the predicate over trace pairs $\mathcal{M}$. We define $\mathcal{I}$ and $\mathcal{M}$, as the paper progresses.

[9] In a concrete implementation we do not require that there is no garbage collection, just that the programmer cannot manipulate addresses.

3.2 Isomorphism

Stores are isomorphic if they differ only in the actual values of heap addresses or in garbage. An isomorphism is characterised by a bijection between values, π. Definition 1 says that σ_1 is isomorphic to σ_2 with relation π iff the stacks of σ_1 and σ_2 are the same height; for each corresponding stack frame the same variables are defined; and π is an injection. Where π is the uniquely defined relation that maps the stack variables of σ_1 to σ_2, commutes with field dereference, and preserves the meaning of `null`, `true`, and `false`.

Definition 1 (Isomorphism).
$\sigma_1 \approx_\pi \sigma_2 \overset{def}{\iff}$

- $|\sigma_1| = |\sigma_2| \ \wedge\ \forall i \leq |\sigma_1| : dom(\sigma_1[i]) = dom(\sigma_2[i])$
- π *is an injection*[10], *written* $in(\pi)$

Where π *is the smallest relation that satisfies:*

$$\begin{aligned}\pi = &\pi_v \cup \{\sigma_1[i](x) \mapsto \sigma_2[i](x) \mid x \in dom(\sigma_1[i]) \ \wedge\ i \leq |\sigma_1|\} \cup \\ &\{\sigma_1(a, f) \mapsto \sigma_2(\pi(a), f) \mid a \in dom(\pi)\}\end{aligned}$$

And $\pi_v \overset{def}{=} [\texttt{null} \mapsto \texttt{null}, \texttt{true} \mapsto \texttt{true}, \texttt{false} \mapsto \texttt{false}]$

In our notation: The number of stack frames in store σ is $|\sigma|$. The value of variable x in the i^{th} stack frame is $\sigma[i](x)$. The domain of the mapping π is $dom(\pi)$, and $dom(\sigma[i])$ is the set of variables defined in the i^{th} stack frame.

We also require that any contracts in the program are not sensitive to address values or garbage. We write $\sigma_{i\ldots j}$ to mean $\sigma_i, \ldots, \sigma_j$.

Definition 2 (Contracts in $\mathcal{L}$). *The contracts* $Con : Pid \to \mathcal{P}(Store \times Store)$, *are sets of pairs of Store representing the set of acceptable pre and post stores of each procedure. We require that:*

$$\begin{aligned}&\forall \boldsymbol{p}, \sigma_{1\ldots4}, \pi_{1,2} : \sigma_1 \approx_{\pi_1} \sigma_2 \ \wedge\ \sigma_3 \approx_{\pi_2} \sigma_4 \ \wedge\ (\sigma_1, \sigma_3) \in Con(\boldsymbol{p}) \ \wedge\ in(\pi_1 \cup \pi_2) \\ &\quad \implies (\sigma_2, \sigma_4) \in Con(\boldsymbol{p})\end{aligned}$$

Lemma 1 (Isomorphism is an equivalence relation). *The relation* $\approx$ *is an equivalence relation (reflexive, symmetric, transitive)*

The crucial property of $\approx$ is that it is closed under execution. Namely, executing a statement from isomorphic stores results in isomorphic executions. Executions are isomorphic iff the elements of their traces are pairwise isomorphic (written $tr_1 \approx tr_2$), and have isomorphic write effects.

Lemma 2 ($\mathcal{L}$ closed under isomorphism).
For every execution $\sigma_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \sigma_3$, *store* σ_2, *and injection* π_1 *if* $\sigma_1 \approx_{\pi_1} \sigma_2$ *then*

[10] $in(\pi) \overset{\text{def}}{\iff} \forall (a, b), (c, d) \in \pi : (a = c \iff b = d)$

– *there exists an execution* $\sigma_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \sigma_4$
– *if* $\mathcal{L} = \mathcal{V}$*, every execution* $\sigma_2, s \rightsquigarrow_{\mathcal{V}}^{tr_2} \sigma_4$ *is isomorphic to* $\sigma_1, s \rightsquigarrow_{\mathcal{V}}^{tr_1} \sigma_3$

Executions $\sigma_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \sigma_3$ *and* $\sigma_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \sigma_4$ *are* isomorphic *iff* $\exists \pi_{1,2}:$

$$\sigma_1 \approx_{\pi_1} \sigma_2 \ \wedge \ tr_1 \approx tr_2 \ \wedge \ in(\pi_1 \cup \pi_2) \ \wedge \ effect(\sigma_1, \sigma_3) \approx_{\pi_2} effect(\sigma_2, \sigma_4)$$

Where trace isomorphism is:

$$\begin{aligned} tr_1 \approx tr_2 &\overset{def}{\iff} tr_1 \approx_{\emptyset} tr_2 \\ tr_1 \approx_{\pi} tr_2 &\overset{def}{\iff} \exists n : |tr_1| = |tr_2| = n \ \wedge \ \exists \pi_1 \dots \pi_{2n} : \\ &\quad (\forall i \leq n : tr_1[i]{\downarrow_2} \approx_{\pi_{2i-1}} tr_2[i]{\downarrow_2}) \ \wedge \\ &\quad (\forall i \leq n : tr_1[i]{\downarrow_3} \approx_{\pi_{2i}} tr_2[i]{\downarrow_3}) \ \wedge \\ &\quad (\forall i, j \leq 2n : in(\pi \cup \pi_i \cup \pi_j)) \end{aligned}$$

And where $effect(\sigma_1, \sigma_3) \overset{def}{=} \sigma_3{}^{\texttt{heap}} \setminus \sigma_1{}^{\texttt{heap}}$

Proof. By induction on the derivation of $\sigma_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \sigma_3$. Most cases are straightforward since no instruction is sensitive to the actual value of addresses. Note that every instruction makes the set of reachable addresses smaller, apart from `new` which expands it by exactly one fresh address — this corresponds to the fact that addresses are never synthesised and garbage is never resurrected. □

A way to think of closure under isomorphism is that $\mathcal{L}$ is not sensitive to the actual values of addresses nor garbage. Many industrial languages (such as C, Java, Python, C♯, etc) contain features that are sensitive to the actual values of heap addresses or order of allocation. Such sensitivity is typically not central to the language and it is often not necessary to use such features.

3.3 Regional Isomorphism

As discussed in Section 2, APE works by establishing isomorphisms between the heap regions reachable from procedure parameters (Line 79 of Fig. 4), so we introduce a notion of isomorphism between heap regions. The heap regions reachable from two sequences of parameters W and X are isomorphic iff the sequences have the same length, and the relation (π) constructed by following all paths from the parameters is an injection:

Definition 3 (Regional Isomorphism).

$$\begin{aligned} &\sigma_1 \approx_{\pi}^{W,X} \sigma_2 \overset{def}{\iff} \\ &\quad |W| = |X| \ \wedge \ W \subseteq dom(\sigma_1{}^{\texttt{top}}) \ \wedge \ X \subseteq dom(\sigma_2{}^{\texttt{top}}) \ \wedge \ in(\pi) \end{aligned}$$

Where π *is the smallest relation which satisfies:*

$$\pi = \pi_v \cup \{\sigma_1(W[i]) \mapsto \sigma_2(X[i]) \,|\, i \leq |X|\} \cup \{\sigma_1(a, f) \mapsto \sigma_2(\pi(a), f) \,|\, a \in dom(\pi)\}$$

3.4 Procedure Equivalence

Procedures are equivalent if executing their bodies from isomorphic stores results in isomorphic stores. Executing `body p` means looking up and executing the statements that form the body of procedure `p`. Note that rule BOD pops the top stack frame before completing, so stores σ_3, σ_4 in Definition 4 are as observed by a caller. This means that equivalence relates to the observable behaviour of the procedure body, and that differences in local variables, etc, are ignored. The same definition of procedure equivalence applies to both the $\mathcal{A}$ and the $\mathcal{V}$ semantics.

Definition 4 (Procedure equivalence).

$$p_1 \overset{\mathcal{L}}{\approx} p_2 \overset{def}{\iff}$$

$$\forall \sigma_{1\ldots4} : \sigma_1 \approx \sigma_2 \ \wedge\ \sigma_1, \texttt{body}\ p_1 \parallel \sigma_2, \texttt{body}\ p_2 \rightsquigarrow_{\mathcal{I}} \sigma_3 \parallel \sigma_4 \implies \sigma_3 \approx \sigma_4$$

3.5 Angelic Allocation

We describe how angelic allocation is modelled by the predicate $\mathcal{I}$ in the $\mathcal{A}$ semantics rule COMA. The predicate selects the pairs of execution traces that exhibit desirable allocation patterns.

Predicate $\mathcal{I}$ (Definition 6) retains only traces with heap regions that are equal at particular isomorphic points (Definition 5) in the traces. In APE, these points correspond to procedure entry, equivalent procedure calls and allocation sites. APE only verifies procedures from equal (rather than isomorphic) initial stores, discards execution pairs which don't have interesting correspondences between allocations, and assumes procedures have equal effects. Because we are only trying to prove soundness of RIE, it is not necessary to fully specify how APE chooses which stores to equate. Rather, we prove that *any* assumption of store equality that the tool makes is sound, subject to the caveats in Definition 5.

Definition 5 (Isomorphic Points).
Any tool using RIE must define a function

$$pts : (Trace \times Trace) \rightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N} \times Lid^* \times Lid^*)$$

with the following properties:

1. *The same set of points is produced for isomorphic traces:*

$$\forall tr_{3,4} : tr_1 \approx tr_3 \ \wedge\ tr_2 \approx tr_4 \implies pts(tr_1, tr_2) = pts(tr_3, tr_4)$$

2. *The traces are isomorphic at each of the points:*

$$\forall (i, j, W, X) \in pts(tr_1, tr_2) : \exists \pi_1 : tr_1[i] \approxeq^{W,X}_{\pi_1} tr_2[j]$$

3. *If the initial stores of tr_1, tr_2 are isomorphic, then the isomorphism is injective with all the other isomorphisms. Otherwise the isomorphisms are empty.*

- $in(\pi \cup \Pi(tr_1, tr_2))$
- $\nexists \pi : fst(tr_1) \approx_\pi fst(tr_2) \implies \Pi(tr_1, tr_2) = \emptyset$

Where

$$\Pi(tr_1, tr_2) = \bigcup \{\pi \mid \exists (i, j, X, Y) \in pts(tr_1, tr_2) \wedge tr_1[i] \approxeq_\pi^{X,Y} tr_2[j]\}$$

Definition 6 (Angelic Allocation).

$$\mathcal{I}(tr_1, tr_2) \overset{def}{\iff} \Pi(tr_1, tr_2) \subseteq id$$

Definition 5 requires that the points are selected symmetrically for isomorphic traces. This symmetry is critical for the soundness of RIE, which must verify at least one pair of execution traces for each initial state. Furthermore, the definition requires that the union of the isomorphisms between the selected heap regions is an injection. This corresponds to the fact that RIE is implemented by equating allocation sites, and will be discussed further in later sections, particularly Section 4.

3.6 Mutual Summaries of Equivalent Procedures

APE uses mutual summaries, lines 73 and 80 in Fig. 4, to allow the verifier to use facts about the behaviour of equivalent procedure calls in its proofs. It is needed in order for procedure equivalence to be a transitive relation.

APE's use of mutual summaries is modelled by the rule CALLA, which over-approximates the behaviour of concrete procedure call. And the predicate $\mathcal{M}$ in the rule COMA, which restricts the traces to those where the procedure pairs in $\mathcal{E}$ behave equivalently.

The antecedent $\sigma_1 \approxeq_{\pi_1}^{\{x_1 \dots x_n\},\{y_1 \dots y_n\}} \sigma_2$ expresses that the regions reachable from the parameters are isomorphic (as needed for challenge 2), while the conclusion $\mathit{effect}(\sigma_1, \sigma_3) \approx_{\pi_2} \mathit{effect}(\sigma_2, \sigma_4)$ expresses that the procedures have isomorphic write effects (as needed for challenge 3). In our example, the encoding of the antecedent is `$Heap#EqualFromParams` on line 79 of Fig. 4, while the encoding of the conclusion is `$SameDiff` on line 80 of Fig. 4.

Definition 7 (Mutual Summaries of Equivalent Procedures).

$$\begin{aligned}\mathcal{M}(tr_1, tr_2) \overset{def}{\iff} & \forall \pi_1, \sigma_{1\dots4}, (p_1, p_2) \in \mathcal{E} : \\ & (\texttt{call } p_1(x_1 \dots x_n), \sigma_1, \sigma_3) \in tr_1 \wedge \\ & (\texttt{call } p_2(y_1 \dots y_n), \sigma_2, \sigma_4) \in tr_2 \wedge \\ & \sigma_1 \approxeq_{\pi_1}^{\{x_1 \dots x_n\},\{y_1 \dots y_n\}} \sigma_2 \\ & \implies \exists \pi_2 : in(\pi_1 \cup \pi_2) \wedge \mathit{effect}(\sigma_1, \sigma_3) \approx_{\pi_2} \mathit{effect}(\sigma_2, \sigma_4)\end{aligned}$$

3.7 Soundness of RIE

We now give the theorem which guarantees soundness of RIE, and describe some keys points in its proof. Theorem 1 states that if all pairs in $\mathcal{E}$ are mutually terminating and equivalent under the $\mathcal{A}$ semantics, then they are also equivalent under the $\mathcal{V}$ semantics. Mutual termination (mt) means that both procedures terminate for the same set of initial stores[11].

Theorem 1 (RIE is sound).

$$\begin{aligned}&\textit{If } \forall(p_3, p_4) \in \mathcal{E} : mt_{\mathcal{V}}(p_3, p_4) \;\wedge\; p_3 \overset{\mathcal{A}}{\approx} p_4\\ &\textit{Then } \forall(p_1, p_2) \in \mathcal{E} : p_1 \underset{\mathcal{V}}{\approx} p_2\end{aligned}$$

Where $mt_{\mathcal{L}}(p_3, p_4) \overset{def}{\iff} \forall\sigma_{1\ldots3} : \sigma_1 \approx \sigma_2 \implies$

$$\Big(\sigma_1, \texttt{body } p_3 \rightsquigarrow_{\mathcal{L}} \sigma_3 \implies \exists\sigma_4 : \sigma_2, \texttt{body } p_4 \rightsquigarrow_{\mathcal{L}} \sigma_4\Big) \;\wedge$$
$$\Big(\sigma_1, \texttt{body } p_4 \rightsquigarrow_{\mathcal{L}} \sigma_3 \implies \exists\sigma_4 : \sigma_2, \texttt{body } p_3 \rightsquigarrow_{\mathcal{L}} \sigma_4\Big)$$

Proof. The proof proceeds by showing that for any pair of executions (tr_1, tr_2) from isomorphic initial stores in the $\mathcal{V}$ semantics there exist an isomorphic execution (tr_5 with $tr_1 \approx tr_5$) such that that $\mathcal{I}$ and $\mathcal{M}$ hold for (tr_1, tr_5) and thus the tr_5 and tr_2 executions compose by $\|$ in the $\mathcal{A}$ semantics. Then by the assumptions and transitivity of $\approx$ we know that tr_1, tr_2 end in isomorphic stores. The proof goes by an inner induction nested within an outer induction. We now write the proof in some more detail:

Assume $\forall(\texttt{p}_3, \texttt{p}_4) \in \mathcal{E} : mt_{\mathcal{V}}(\texttt{p}_3, \texttt{p}_4) \;\wedge\; \texttt{p}_3 \overset{\mathcal{A}}{\approx} \texttt{p}_4$.
To show:

$$\forall(\texttt{p}_1, \texttt{p}_2) \in \mathcal{E}, \sigma_{1\ldots4}, tr_{1,2} :$$

$$\sigma_1 \approx \sigma_2 \;\wedge\; \sigma_1, \texttt{body p}_1 \parallel \sigma_2, \texttt{body p}_2 \overset{tr_1, tr_2}{\rightsquigarrow_{\mathcal{V}}} \sigma_3 \parallel \sigma_4 \implies \sigma_3 \approx \sigma_4$$

First part: From Definition 5 we see that there is a $\pi_2 = \Pi(tr_1, tr_2)$ (1). By Lemma 3 (below) there exists a third execution $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_3} \sigma_5$ such that tr_1 is isomorphic to tr_3 with π_2, i.e. $tr_1 \approx_{\pi_2} tr_3$, which means by Lemma 1 (symmetry) we get $tr_3 \approx_{\pi_2^{-1}} tr_1$ (2). Take any point $(i, j, X, Y) \in pts(tr_1, tr_2)$. To show: $tr_3[i] \approx_{id}^{X,Y} tr_2[j]$. By (1) exists π_4 such that $tr_1[i] \approx_{\pi_4}^{X,Y} tr_2[j]$ and $\pi_4 \subseteq \pi_2$ (4). By (2) there exists π_3 such that $tr_3[i] \approx_{\pi_3} tr_1[i]$ and $\pi_3 \subseteq \pi_2^{-1}$ (3). Hence, by Lemma 1 (transitivity), we have that $tr_3[i] \approx_{\pi_3 \circ \pi_4}^{X,Y} tr_2[j]$, we know π_3

[11] We could produce a total definition of procedure equivalence by including a notion of mutual termination [**Hawblitzel2013**, **Elenbogen2015**] in Definition 4. However, APE does not yet reason about the termination behaviour of the procedures. A total notion of procedure equivalence is important, particularly where a transitive procedure equivalence relation is needed. Since our tool takes the same basic approach as Symdiff, it should be straightforward to incorporate existing mutual termination checking techniques [**Godlin08**, **Hawblitzel2013**, **Elenbogen2015**].

composed with π_4 is large enough because X is a subset of the stack variables defined in $tr_1[i]$, and $dom(\pi_3)$ includes the values of all stack variables. By (3), (4), we know that π_3 composed with π_4 is a subset of the identity relation. So $tr_3[i] \approxeq_{id}^{X,Y} tr_2[j]$. Because $tr_1 \approx tr_3$ we have by definition $pts(tr_1, tr_2) = pts(tr_3, tr_2)$. Then we get $\forall(i,j,X,Y) \in pts(tr_3, tr_2) : tr_3[i] \approxeq_{id}^{X,Y} tr_2[j]$. And thus we have $\Pi(tr_3, tr_2) \subseteq id$, which in turn gives us $\mathcal{I}(tr_3, tr_2)$ (5).
Second part: We now proceed by induction on the size of the derivation of $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_3} \sigma_5$.

It remains to show that, tr_2 is also a trace under $\mathcal{A}$ and that there is an $\mathcal{A}$ trace tr_5 isomorphic to tr_3 such that $\mathcal{M}(tr_5, tr_2)$ holds. Note that it is trivial to prove that apart from $\|$ all the rules of $\mathcal{A}$ semantics overapproximate the $\mathcal{V}$ semantics (l1). By (l1) and induction on the derivation of $\sigma_2, \texttt{body p}_2 \rightsquigarrow_{\mathcal{V}}^{tr_2} \sigma_4$ we get $\sigma_2, \texttt{body p}_2 \rightsquigarrow_{\mathcal{A}}^{tr_2} \sigma_4$.
Base case: there are no procedure calls in $\sigma_1, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_1} \sigma_3$. By Lemma 2 then $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_3} \sigma_5$ also has no procedure calls. By (l1) and induction on the derivation of $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_3} \sigma_5$ we get $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{A}}^{tr_3} \sigma_5$. Since there are no procedure calls, trivially $\mathcal{M}(tr_3, tr_2)$ and $\sigma_2, \texttt{body p}_1 \parallel \sigma_2, \texttt{body p}_2 \rightsquigarrow_{\mathcal{A}}^{tr_3, tr_2} \sigma_5 \parallel \sigma_4$. From the antecedent we know that $lst(tr_3) \approx lst(tr_2)$, and since $tr_1 \approx tr_3$ then by transitivity of $\approx$ we have $lst(tr_1) \approx lst(tr_2)$. And since $lst(tr_1) = \sigma_3$ and $lst(tr_2) = \sigma_4$ base case is done.
Inductive step: there are procedure calls in $\sigma_1, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_1} \sigma_3$.

To show: there exists an execution $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{A}}^{tr_5} \sigma_7$ such that $\mathcal{M}(tr_5, tr_2)$ and $tr_5 \approx_{id} tr_3$ (6). Proceed by an inner induction over the derivation of $\sigma_2, \texttt{body p}_1 \rightsquigarrow_{\mathcal{V}}^{tr_3} \sigma_5$. Most cases are trivial. The interesting cases are CALLV where the called procedure is in $\mathcal{E}$, and the inductive case TRANS.
Inner case CALLV where the called procedure is in $\mathcal{E}$. By case there is $\sigma_2, \texttt{call p}_3(x_1 \ldots x_n) \rightsquigarrow_{\mathcal{V}} \sigma_5$ and $(\texttt{p}_3, \texttt{p}_4) \in \mathcal{E}$. Rule CALLV can only be applied if a shallower tree is derivable for the body of the called procedure. Therefore we apply the outer induction hypothesis, the antecedent $mt_{\mathcal{V}}(\texttt{p}_3, \texttt{p}_4)$, and Lemma 3, to deduce that there exists σ_7 such that $\sigma_2, \texttt{call p}_4(x_1 \ldots x_n) \rightsquigarrow_{\mathcal{V}}^{tr_5} \sigma_7$ and $\sigma_5 \approx_{id} \sigma_7$. From CALLA we see that $\sigma_2, \texttt{call p}_3(x_1 \ldots x_n) \rightsquigarrow_{\mathcal{A}}^{(\texttt{call p}_3(x_1 \ldots x_n), \sigma_2, \sigma_7)} \sigma_7$ (intuition: we swap the behaviour of $\texttt{p}_3$ for the behaviour of $\texttt{p}_4$). Take $tr_5 = (\texttt{call p}_3(x_1 \ldots x_n), \sigma_2, \sigma_7)$ and we have $tr_3 \approx_{id} tr_5$. To show: $\mathcal{M}(tr_5, tr_2)$. Take arbitrary $(\texttt{call p}_4(y_1 \ldots y_n), \sigma_8, \sigma_{10}) \in tr_2$ such that $\sigma_2 \approxeq_{\pi_5}^{\{x_1 \ldots x_n\}, \{y_1 \ldots y_n\}} \sigma_8$. To show: $\exists \pi_6 : in(\pi_5 \cup \pi_6) \wedge \mathit{effect}(\sigma_2, \sigma_5) \approx_{\pi_6} \mathit{effect}(\sigma_8, \sigma_{10})$. This follows straightforwardly from Lemma 2 and the fact that the same procedure $\texttt{p}_4$ was executed to obtain σ_5 as to obtain σ_{10}. Inner case **done**.
Inner case TRANS By case there is $\sigma_2, s_1; s_2 \rightsquigarrow_{\mathcal{V}}^{tr_3} \sigma_5$ and hence exists $\sigma_9, tr_{7,9}$ such that $\sigma_2, s_1 \rightsquigarrow_{\mathcal{V}}^{tr_7} \sigma_9, s_2 \rightsquigarrow_{\mathcal{V}}^{tr_9} \sigma_5$. The proof goes as expected, by applying the inner induction hypothesis twice with one slight complexity. The first application constructs another execution of s_1 with the desired properties; but that execution's final store is not σ_9! Rather it is some other store (say σ_{11}) that is isomorphic to σ_9. Before we apply the inner induction

hypothesis a second time, we use Lemma 3 to construct an execution isomorphic to $\sigma_9, s_2 \leadsto_{\mathcal{V}}^{tr_9} \sigma_5$ but with initial store σ_{11}. $\mathcal{M}$ holds for the resultant traces by the same argument as the CALLV case. Inner case **done**.
Hence $\mathcal{M}(tr_5, tr_2)$. From (5) and (6) we also have $\mathcal{I}(tr_5, tr_2)$. So we get that $\sigma_2, \texttt{body p}_1 \parallel \sigma_2, \texttt{body p}_2 \leadsto_{\mathcal{A}}^{tr_5, tr_2} \sigma_7 \parallel \sigma_4$ is an execution under the $\mathcal{A}$ semantics. Finally, from the antecedent we know that $lst(tr_3) \approx lst(tr_4)$, and since $tr_1 \approx tr_3 \approx tr_5$ and $tr_2 \approx tr_4$ then by transitivity of $\approx$ we have $lst(tr_1) \approx lst(tr_2)$. And since $lst(tr_1) = \sigma_3$ and $lst(tr_2) = \sigma_4$ we are done.
□

The soundness proof of Theorem 1 relies on constructing alternative executions that are isomorphic using the identity bijection, Lemma 3 states that all such alternative executions are derivable in $\mathcal{L}$.

Lemma 3 (Sufficent non-determinism). *Given statement s, stores $\sigma_{1\ldots3}$, mapping π_1, and alternative allocation strategy π_2, such that:*

- *σ_1 and σ_2 are isomorphic with mapping π_1: $\sigma_1 \approx_{\pi_1} \sigma_2$*
- *s can execute to completion from σ_1: $\sigma_1, s \leadsto_{\mathcal{L}}^{tr_1} \sigma_3$*
- *π_1 and π_2 map common addresses in the same way $in(\pi_1 \cup \pi_2)$*
- *$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \sigma_1{}^{\texttt{heap}} \iff a_2 \in \sigma_2{}^{\texttt{heap}})$*

Then there exists an isomorphic execution $\sigma_2, s \leadsto_{\mathcal{L}}^{tr_2} \sigma_4$ such that:

$$tr_1 \approx_{\pi_2} tr_2 \wedge \forall (a_1, a_2) \in \pi_2 : (a_1 \in \sigma_3{}^{\texttt{heap}} \iff a_2 \in \sigma_4{}^{\texttt{heap}})$$

Proof. By induction on the derivation of $\sigma_1, s \leadsto_{\mathcal{L}}^{tr_1} \sigma_3$ the alternative execution is constructed. In particular note that because π_2 does not map between allocated and unallocated addresses, the appropriate alternative address is always unallocated when an allocation statement is reached. And that, since $in(\pi_1 \cup \pi_2)$ then all addresses that were already allocated at the start of the execution do not need to be allocated alternatives. The proof goes through for both $\mathcal{V}$ and $\mathcal{A}$. □

4 Discussion

The number of correspondences between allocations (Section 2) is factorial in the maximum number of allocation sites in either procedure. Hence RIE is only practical for relatively small numbers of allocation sites. However, this is not as restrictive as it may seem because our approach is modular. In practice, when loops are encoded as procedure calls, then many interesting procedures contain only small numbers of allocations. In some cases it is also possible to split a procedure into chunks or abstract common parts. It is also likely that the applicability of this technique can be significantly extended by using additional static analysis to eliminate some of the permutations in advance. For example, the types of the objects being allocated could be used to eliminate permutations that aligned objects of different types.

Framing of procedure calls is important in verifying equivalence for many examples. APE has a fairly naive approach to framing and disjointness of heap regions, which restricts the class of examples it can currently deal with. However, our techniques, and choice of Dafny [**Leino10**] style heap encoding, should be amenable to a more powerful framing methodology. Improving APE's framing support is likely to significantly improve its completeness.

We considered many alternative approaches to establishing isomorphism. The natural approach using existentials does not work very well. We investigated several approaches using universal quantification. We tried defining heaps to be isomorphic if all pairs of paths that lead to related addresses in one heap also lead to related addresses in the other. We tried several approaches for limiting which, and what depth of paths should be considered by the solver. But the underlying doubly exponential complexity of comparing all pairs of paths impedes the applicability of that approach. The requirement that disjoint heap effects of procedure calls commute was an important design force: many alternative approaches required extensive additional axioms to handle the various cases, whereas our current approach of enumerating allocators seems to handle many cases naturally.

4.1 Examples

There are a collection of programs available from `https://github.com/lexicalscope/ape#automatic-procedure-equivalence-tool` that show the capabilities of APE. Several of them have been listed in Fig. 7 with timings performed on an Intel Core i5-3210M@2.5GHz processor with 8GB memory.

Description	**Timing**
Empty program	1.5s
Change order of allocation	1.5s
Change amount of garbage allocated	1.5s
Making the same procedures calls	1.5s
Isomorphism of heap reachable from call parameters	1.8s
Allocations moved past calls	1.5s
Recursive calls reordered	8s
Copying a cyclic data structure	4s
Inserting a row into a table	31s
Copying a list	7s
Copying a tree	130s

Fig. 7. Examples with timings

RIE's approach of using equality to establish isomorphism does prevent APE from establishing isomorphism in some cases where it would be helpful to do so. The example in Fig. 8 is from a refactoring of some code which manipulates a doubly linked list. Both procedures add an element to a list, but first remove it if it is already present. The left procedure has a redundant check that the item is in the list, in the right procedure this redundancy is removed.

The isomorphism between lines 88 and 105 relates the addresses in `rf0` and `rf`. The isomorphism between lines 91 and 105 relates the addresses in `rf1` and `rf`. If we were to assume equality for both of these isomorphisms then we would have `rf` $=$ `rf0` $=$ `rf1`. However, `rf0` is allocated by the `new` statement on line 87 whereas `rf1` is allocated by the subsequent `new` statement on line 90. The semantics of `new` require that each allocation gives an address which was not previously allocated — i.e. that `rf0` $\neq$ `rf1`.

RIE, therefore, restricts the selected points in Definition 6 to prevent contradictory isomorphisms being selected. Due to this restriction a verifier using RIE alone may fail to produce a proof for some procedures that are in fact equivalent according to our definitions. Any tool using RIE in practice may choose to equate one pair of calls to `find`, but it must find some other way to deal with the other pair of calls (such as manually adding an additional specification of `find`).

```
 87 new rf0;                                  104 new rf;
 88 find(l,x,rf0); _ _ _ _ _ _ _ _ _ _ _ _ _ _105 find(l,x,rf);
 89 if(!rf0.v.sentinel) {                 _ - 106
 90   new rf1;                      _ - -     107
 91   find(l,x,rf1); - - - - - - -            108
 92   node := rf1.v;                          109 node := rf.v;
 93   if (!node.sentinel) {                   110 if(!node.sentinel){
 94     node.prev.next := node.next;          111   node.prev.next := node.next;
 95     node.next.prev := node.prev;          112   node.next.prev := node.prev;
 96   }                                       113 }
 97   add(l,x);                               114 add(l,x);
 98 } else {                                  115
 99   add(l,x);                               116
100 }                                         117
101                                           118
102 find(l,x,r) modifies {r} ...              119 find(l,x,r) modifies {r} ...
103 add(l,x) ...                              120 add(l,x) ...
```

Fig. 8. A difficult example where two stores in one execution are isomorphic with the same store in the other execution.

4.2 Definitions of Isomorphism and Procedure Equivalence

Our definition of procedure equivalence is useful because it is a contextual equivalence [**Milner1977**] for $\mathcal{L}$. This means that given equivalent procedures $\mathtt{p}_1, \mathtt{p}_2$ and a program that calls $\mathtt{p}_1$, one can always change the program to call $\mathtt{p}_2$ instead without affecting the observable behaviour of the program. Of particular interest to programmers is Corollary 1: the relation $\approx$ preserves the meaning of all assertions.

Corollary 1 (Isomorphism is assertion preserving).

$$\forall \sigma_1, \sigma_2, b : \sigma_1 \approx \sigma_2 \implies (\sigma_1 \vDash b \iff \sigma_2 \vDash b)$$

Proof. Follows from Lemma 2 □

Interestingly, it is possible to define isomorphism almost equivalently as the least-fixed-point interpretation of the relation:

$$\sigma_1 \approx' \sigma_2 \overset{\text{def}}{\iff}$$
$$\sigma_1 = \sigma_2 = \sigma_\emptyset \vee \left(\exists s_a, \sigma_{3,4} : \sigma_3 \approx' \sigma_4 \wedge \sigma_3, s_a \rightsquigarrow_{\mathcal{C}} \sigma_1 \wedge \sigma_4, s_a \rightsquigarrow_{\mathcal{C}} \sigma_2\right)$$

where $\sigma_\emptyset$ is the empty store. That is, it could be defined as a smallest relation closed under the atomic operations of the semantics. However, even though the semantics is naturally closed under $\approx'$, the definition is not as helpful when trying to decide if a particular pair of stores are isomorphic. Regardless, a definition in this least-fixed-point style would allow us to construct a notion of isomorphism even for a semantics where we did not know an appropriate direct definition. Perhaps it is interesting to consider what assertion language would be preserved for any particular semantics given such a definition.

4.3 Reachability

Establishing reachability enables APE to prove interesting examples, but is ancillary to the focus of this paper RIE and angelic allocation. Still, our definition of equivalence allows differences in garbage (which is unreachable memory), and APE use reachability to reason about read and write effect framing as described in Section 2.3 — so an useful axiomatisation of reachability is needed.

Fig. 9 shows our Boogie encoding of reachability. We give several axioms for the predicate `$Reach`, which the tool instantiates in different circumstances controlled by various triggers (controlled programatically, users cannot write them). Rather than precisely deciding the reachability set, often it is necessary to prove disjointness of certain heap regions. For example, garbage objects are disjoint from the reachable region, and a property of a region is preserved over a procedure call if the region is disjoint from the call effects. Our choice of axioms enable the tool to establish a lack of reachability by showing either that there are no outgoing (line 130) or no incoming (line 127) edges to a particular heap region. Although the axioms line 127 and line 130 are logically equivalent, we use triggers to unroll them in different situations.

5 Related Work and Conclusions

The study of program equivalence arguably pre-dates the study of functional correctness. In his 1969 paper [**Hoare1969**], **Hoare1969** identified that "Many [previous] axiomatic treatments of computer programming [**Yanov1958**,

```
// Reachability is uninterpreted but axiomatised
function $Reach($h:Heap, $a:Ref, $b:Ref) : bool;

axiom (∀$h:Heap, $a:Ref :: $Reach($h, $a, $a));
axiom (∀$h:Heap, $a,$b:Ref :: {...}
 $Reach($h, $a, $b) ∧ $NoInboundEdges($h,$b) ⟹ $a = $b);
axiom (∀$h:Heap, $a,$b:Ref :: {...} $Reach($h,$a,$b) ⟹
 $a = $b ∨
 (∃$c:Ref,$f:Field Ref :: $Reach($h, $a, $c) ∧ $Edge($h, $c, $f, $b)));
axiom (∀$h:Heap, $a,$b:Ref :: {...} $Reach($h,$a,$b) ⟹
 $a = $b ∨
 (∃$c:Ref,$f:Field Ref :: $Edge($h, $a, $f, $c) ∧ $Reach($h, $c, $b)));

function $NoInboundEdges($h:Heap, $a:Ref) : bool
 { (∀$b:Ref, $f:Field Ref :: !$Edge($h,$b,$f,$a)) }
```

Fig. 9. The partial axiomatisation of reachability used by APE, written in Boogie. The triggers are elided `{...}`. The function `$Read(h,a,f)` is the value of field `f` of object `a` in heap `h`, the predicate `$Allocated($h,$a)` holds if object `$a` is allocated in heap `$h`. The predicate `$Edge($h, $a, $f, $c)` holds if the field `$f` of object `$a` has the value `$c` in heap `$h`.

Igarishi1964, **DeBarker1968**] tackle the problem of proving the equivalence, rather than the correctness, of algorithms". To date, practical approaches to program equivalence rely on structural similarity of the programs. Many works focus on methods to account for some structural differences. The importance of program structure in proving program equivalence was observed by Dijkstra in 1972 [**Dahl1972**], where he also observes that programmers are often called upon to modify existing programs. Key developments in program equivalence have come from research into *non-interference* in *secure information flow* and compiler *translation validation*. Non-interference is the property that the values of secret inputs do not influence public outputs. Translation validation provides assurances that the program output by a compiler is correct with respect to the input program. Translation validation concerns itself with correctness of particular compiler runs, and does prove the compiler implementation correct. Non-interference can be formalised in terms of program equivalence [**Joshi2000**], or more generally as a safety property over pairs of program traces [**Barthe2004**, **Terauchi2005**]. Methods for reducing safety properties over trace pairs to safety properties over single traces have been explored [**Leino2008a**, **Benton2006**] and generalised, particularly via product programs [**Barthe2011**, **Barthe2013**] and similar [**Zaks2008**, **Stepp11**, **Tristan11**]. *Product programs* combine a pair of programs into one, such that useful invariants can be formulated at interesting points in the product, and can generalise to relations between programs [**Barthe2016**]. Compiler translation validation [**Kundu09**, **Stepp11**, **Tristan11**, **Le2014**, **Zaks2008**] is inherently a program equivalence question. Many techniques have been applied, several variations of product programs [**Zaks2008**], constructing bisimulations between control flow graphs [**Kundu09**], iteratively applying equality axioms [**Stepp11**], or normalising [**Tristan11**] graph representations of the programs. *Relational Hoare Logic* [**Benton2004**, **Benton2007**, **Yang2007**, **Beringer2011**, **Barthe2016**] (RHL) was proposed by **Benton2004** in 2004 [**Benton2004**], in the course of proving the correctness of various compiler optimisations. The Hoare triple $\{P\}S\{Q\}$ is extended to a Hoare quadruple by inclusion of two statements, rather than one, $\{P\}C_1 \sim C_2\{Q\}$. The pre and post conditions are lifted to relations over stores. RHL has been extended by various rules to account for differences in structure between the programs [**Benton2004**, **Barthe2016**]. **Barthe2016** [**Barthe2016**] pointed out that RHL is closely related to the idea of product programs. Several formal works tackle the problem of proving program equivalence in the presence of dynamic memory allocation. **Pitts2002** uses a simulation between memory locations when defining a semantic approach to program equivalence [**Pitts2002**], the memory model is flat not a heap. **Benton2007** uses isomorphism between heap regions when proposing an RHL that supports dynamic allocation [**Benton2007**]. **Yang2007** constructs a relational separation logic with support for dynamic allocation [**Yang2007**]. **Sumner2010** propose a different approach, canonical memory addresses are constructed based on program control flow and syntactic elements. **Banerjee2016** [**Banerjee2016**] propose a logic for weaving programs with structural differences so that relational properties of programs can be expressed. They extend this with a region logic to support reasoning about encapsulation in dynamically allocating programs; catering for equivalence between programs which vary the representation of objects.

5.1 Fully Automatic Equivalence Verification Tools

To our knowledge there are four other tools with the objective of fully automated verification of procedure equivalence for imperative programs: Symdiff [**Lahiri2012**], RVT [**Godlin09**], SCORE [**Partush2014**], and Rêve [**Felsing2014**]. Symdiff [**Lahiri2012**] uses program verification to prove or provide counter examples of equivalence. It uses mutual summaries, and can infer intermediate summaries to establish equivalence. Conditional equivalence [**Hawblitzel2013**] can show partial equivalence over a subset of procedure inputs and construct summaries of interprocedural behavioural differences. Symdiff is built on Boogie [**Barnett2005**]. Symdiff has no built-in support for procedures that differ in memory allocation. RVT [**Godlin09**] proves equivalence of some C programs. RVT generates loop and recursion free program fragments, which are verified by the CMBC [**Clarke2003**] bounded model checker. Loops are encoded as recursive functions. Recursive calls are replaced by uninterpreted functions. Recently support for unbalanced recursive functions has been added [**Strichman2016**]. RVT is extended to dynamic data structures involving pointers by generating (symbolic) bounded tree-like data structures as inputs for procedures. These initial tree-like structures are isomorphic up to some bound. RVT then verifies (up to the same bound) that those data structures remain isomorphic, at procedure calls and procedure return. The bound is determined by a syntactic overapproximation of the maximum depth of modification. No rigorous proof is presented for this extension to pointers. Rêve [**Felsing2014**] and SCORE [**Partush2014**] support numerical programs without heaps. Rêve uses Horn constraints to verify equivalence of deterministic imperative programs with unbounded integer variables. Rêve infers inductive coupling predicates and as such can deal with loops and recursion where, for example, the number and meaning of procedure parameters has changed. The authors of Rêve propose in the future to extend their tool with an RVT-like approach to the heap. SCORE uses abstract interpretation over an interleaving of the programs. A good interleaving is found by searching. SCORE deals with numerical programs. For non-equivalent programs SCORE can compute an overapproximation of the semantic difference. The precision of this overapproximation is related to the size of the syntactic difference.

5.2 Conclusion

We defined procedure equivalence, and a sound methodology RIE, for automatically proving equivalence of programs which vary in dynamic memory allocation. We described our RIE encoding APE for equivalence verification (available at `https://github.com/lexicalscope/ape`). Our approach is fully automatic, and applicable to programs which manipulate heap data structures of any shape.