Modular Verification of Equivalence for Memory Allocating Procedures

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing

Timothy Wood
Department of Computing
Imperial College London

February 27, 2017

Abstract

Verifying the equivalence of programs has been applied in many situations: for example, proving the correctness of bug-fixes, refactorings, compilation, and optimisation, proving program continuity, proving non-interference in secure information flow, proving abstraction and refinement relationships between programs, and proving that programs conform to differential privacy policies. Verifying the equivalence of heap manipulating procedures where the order and amount of memory allocations differ is challenging for state-of-the-art equivalence verifiers. We describe a fully automatic program equivalence tool, and propose a verification methodology, for such dynamically allocating programs.

Recent years have seen significant progress toward fully automatic program equivalence verification, with the release of several tools taking a variety of approaches. Two main approaches are to use a weakest-precondition based program verifier or a bounded model checker. One such tool has built in support for programs that differ in the order of memory allocation, it uses a bounded model checker to discharge some proof obligations and restricts the allowable shapes of heap data structures to trees.

We describe a fully automatic program equivalence verification tool for a simple object oriented language. It has a notion of procedure equivalence that is powerful enough to allow procedures with different orders and amounts of memory allocation or garbage creation to be considered equivalent, with no restrictions on heap shapes. Our tool establishes equivalence by verifying that procedures result in isomorphic heaps. The tool is built on top of an off-the-shelf weakest-precondition based verifier which itself uses an SMT solver to discharge proof obligations.

A naïve encoding of procedure equivalence would require the verification tool to produce a witness to the heap isomorphism before and after procedure calls, which SMT based tools are not very good at. Instead we propose a modular verification methodology, called *RIE*, that allows us to soundly establish heap isomorphism by checking that an approximation preserves heap equality. RIE then allows us to assume that: whenever we can establish an isomorphism between parts of stores that these stores are in fact equal, and that whenever equivalent procedures are called in an isomorphic manner their effects are equal. RIE also allows our tool to handle some cases where there is not a simulation between the recursive procedure calls of the programs being compared. We prove, and provide intuitions, that RIE is sound for a simple programming language that includes non-deterministic allocation, unbounded recursion, and unbounded heap updates.

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

This work is my own, ideas from the work of others are appropriately acknowledged and referenced.

Acknowledgements

I have had interesting, informative and engaging conversations with many people in the course of developing this work. In particular I am grateful to Reuben Rowe, Alexander Summers, Sylvan Clebsch, Juliana Franco, Rabih Mohsen, Robert Chatley, Susan Eisenbach and several others I have accidentally omitted. Alex and Reuben both offered me generous discussion about various aspects of my work, and made me aware of some areas of related work. Robert also inspired the final example in chapter 1. Susan has influenced me throughout my career so far, has given me valuable advice and feedback on many occasions, and bears some responsibility for my deciding to undertake this degree. The mistakes are all mine.

I was fortunate to have feedback and conversation on my early efforts from Cristian Cadar and several members of his research group including Tomasz Kuchta, Daniel Liew, Petr Hosek and Hristina Palikareva. Alastair Donaldson's course and feedback were also valuable and enjoyable, and I was fortunate to have discussions with Jeroen Ketema from his research group.

Rustan Leino was kind enough to discuss my ideas with me on several occasions, his knowledge of and enthusiasm for program verification were both inspiring. I consider myself very fortunate to have had the opportunity to attend his lectures on the Dafny programming language.

I am grateful to Shuvendu Lahiri for originally suggesting that we should investigate isomorphism and equality in the context of a Symdiff style tool, for several discussions throughout the progress of this work and for sharing his extensive knowledge of the field.

Sophia Drossopoulou has provided informative, inspiring and challenging supervision and conversation throughout. Her undergraduate teaching bears significant responsible for my interest in programming languages and program correctness.

Contents

List of Definitions

List of Theorems

List of Figures

Symbol	Comment	Ref
ϕ	$Store \stackrel{\text{def}}{=} Stack \times Heap$ Store is a sequence of stack frames and a heap	2.1.1
h	$Heap \stackrel{\text{def}}{=} (Addr \times Fid) \rightharpoonup Val$ Heap maps object fields to the values	2.1.1
$\tilde{\sigma}$	$Stack \stackrel{\text{def}}{=} StackF^*$ Stack is a sequence of Stackframes	2.1.1
σ	$StackF \stackrel{\text{def}}{=} Lid \rightharpoonup Val$ Stackframe maps local variables to values	2.1.1
$\lvert\tilde{\sigma}\rvert$	$StackF^* \to \mathbb{N}$ Number of frames in a sequence	2.1.1
$\tilde{\sigma}[i]$	$StackF^* \times \mathbb{N} \rightharpoonup StackF$ The i^{th} element in a sequence	2.1.1
$dom(\sigma)$	$(Lid \rightharpoonup Val) \to \mathcal{P}(Lid)$ The domain of a relation	2.1.1
ϕ^h	$Heap$ The heap of ϕ	2.1.1
$\phi^{\tilde{\sigma}}$	$Stack$ The stack of ϕ	2.1.1
ϕ^{ctx}	$Store$ The calling context of ϕ. Suppose $\phi = (\tilde{\sigma} \cdot \sigma, h)$ then $\phi^{ctx} = (\tilde{\sigma}, h)$	2.1.1
tr^{ϕ}	$Trace$ A trace with initial state ϕ	2.1.1
$\mathcal{B}(\texttt{f})$	$Pid \to Stmt$ The body of procedure `f`	2.1.1
$\mathcal{V}(\texttt{f}, n)$	$Pid \times \mathbb{N} \rightharpoonup Lid$ Name of the n^{th} parameter of procedure `f`	2.1.1
π	$ValRel \stackrel{\text{def}}{=} \mathcal{P}(Val \times Val)$ Relation between values	2.2.1
$\phi_1 \approx_\pi \phi_2$	$\approx \subseteq Store \times ValRel \times Store$ Reachable parts of ϕ_1, ϕ_2 are isomorphic, with characteristic bijection π	2.2.1
$tr_1 \approx tr_2$	$\approx \subseteq Trace \times Trace$ The traces differ only in the actual values of allocated addresses	4.4.1
$effect(h_1, h_2)$	$Heap \times Heap \to Heap$ The writes between h_1 and h_2	
$(\phi_1, \phi_2) \in Con(\texttt{f})$	$Pid \to \mathcal{P}(Store \times Store)$ Store ϕ_1 statisfies precondition of `f`, pair (ϕ_1, ϕ_2) statisfies postcondition	2.1.1
$\phi_1, s \rightsquigarrow^{tr}_{\mathcal{L}} \phi_2$	$\rightsquigarrow_c \subseteq Store \times Stmt \times \times \mathcal{L} Trace \times Store$ Statement s executes with semantics $\mathcal{L}$ from store ϕ_1 to ϕ_2 with trace tr	2.1.1
$com(\pi_1, \ldots, \pi_n)$	$com \subseteq \mathcal{P}(ValRel)$ If relations π_j, π_k map the same address, they map it to the same address	2.3.1
$mpts(tr_1, tr_2)$	$(Trace \times Trace) \to \mathcal{P}(\mathbb{N} \times \mathbb{N} \times Lid^* \times Lid^*)$ A subset of the ways of selecting isomorphic points from two traces representing the verification strategy of a tool	3.4.1

Chapter 1

Introduction

1.1 Overview

Program maintenance dominates the program lifecycle; fixing bugs and adding features is recurring, expensive, and error-prone. A study of operating system bugs [60] found that at least 14.8%-24.4% of patches were incorrect. A study of application bugs that took more than one attempt to fix [45] found that 22%-33% of fixes required a supplementary fix, and found a diverse range of errors including incomplete refactorings. A study of refactorings [9] found that, across 12,922 refactorings from three software projects, 15% of refactorings induced a bug, with some particular kinds of refactorings having much higher rates of problems. When modifying a program it is clearly a challenge to avoid accidental unwanted changes in program behaviour. *Version aware program verification* [24, 25, 31, 34, 35, 39, 47] offers the potential for a programmer to automatically verify that the new version is, fully or partially, behaviourally equivalent to the old.

We know of four other tools designed with the goal of fully-automatic program equivalence verification Symdiff [34], RVT [24], SCORE [46], and Rêve [21]. SCORE and Rêve support numerical programs. Symdiff is built on a general purpose program verifier Boogie [3]. Only RVT has built-in support for dynamic memory allocation. RVT uses a designed-for-purpose verification algorithm, which passes certain program fragments to the CBMC [16] bounded model checker, and assumes tree-like heap data-structures.

In Symdiff procedures are considered behaviourally equivalent if, given equal pre states, they produce equal post states [34]. We call this existing notion of equivalence *equi-equivalence* to distinguish from notions that we will introduce. Equi-equivalence is useful for programmers because replacing a procedure with an equi-equivalent one does not affect the overall meaning of a program. Equi-equivalence verification is particularly successful in compiler translation validation [33, 43, 51, 56], where it is common to model the heap as arrays and require that these arrays are equal across procedure calls [33, 34, 43].

Equi-equivalence is too restrictive for programs that allocate memory. Many programs that a programmer reasonably considers equivalent are not in fact equi-equivalent. Procedures with a different number or order of allocations, or that produce different garbage, are usually not equi-equivalent. Procedures that make different rearrangements of existing heap objects are also usually not equi-equivalent. Surprisingly, by definition a single procedure that allocates memory is not equi-equivalent with itself, due to the non-deterministic nature of memory allocation.

Typical strategies that researchers have employed to mitigate this problem are:

- Considering programs without dynamic allocation, such as in compiler translation validation. How-

ever many programs include dynamic allocation.

- Modelling allocation as deterministic and employing equi-equivalence verification [34]. Such a tool would assume a memory allocator that simply always allocates the next free address. If such an allocator is assumed, then programs that allocate exactly the same heap objects in the same order may be considered equivalent. But using a deterministic allocator alongside equi-equivalence verification does not allow programs that re-order or add or remove allocations to be verified as equivalent.

- Using store[1] isomorphism, rather than equality, by maintaining explicit bounded heap graphs during symbolic execution [50]. This does not allow for maintaining heaps symbolically as formulae in a verification condition, preventing the use of standard program verifiers.

- Assume tree shaped heap data structures and verify up to a bound on loop and recursion-free program fragments [24].

Equi-equivalence is too restrictive for programs that allocate memory, but a more powerful 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 [12, 15, 48, 54]. Later we give an isomorphism based definition of procedure equivalence, suitable for modular verification of program equivalence. Our definition of equivalence allows for differences in the order or amount of memory allocation and garbage and is not restricted to tree-like structures.

We have built a **procedure equivalence verification tool**[2] for a simple object oriented language that can automatically verify the equivalence of some programs that differ in the amount and order of dynamic memory allocations. Our tool follows the Symdiff approach, so is based on a general purpose program verifier, and does not make any assumptions about the shapes of data-structures. It can work for heap-manipulating procedures that include unbounded recursion and unbounded heap updates with no inherent restrictions on aliasing and cycles within heap data-structures. Achieving this presented several challenges:

Challenge 1. Directly and automatically establishing isomorphism between unbounded heaps of arbitrary shape 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, like the one underlying Symdiff, are not very good at producing witnesses to such existentials, and so a direct axiomatisation of isomorphism is ineffective at automatically proving equivalence.

Challenge 2. Many interesting examples contain nested or recursive procedure calls. Such equivalent calls to equivalent procedures do not necessarily occur from isomorphic stores. Rather the stores correspond in the footprint of the called procedures.

Challenge 3. Since equivalent calls do not necessarily occur from isomorphic stores, they do not necessarily result in isomorphic stores — rather the calls have corresponding behaviour.

Challenge 4. In general it is not known in advance which calls may be equivalent, and there may be many candidates. In particular, 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] Store means the stack and the heap.
[2] Sourcecode available at `https://github.com/lexicalscope/ape`

Challenge 5. Because the question of verifying program equivalence is undecidable in general, and has high computational complexity in many specific cases, we have to design our Boogie encoding carefully so that predicates are unfolded enough but not too much.

We propose a verification methodology, called RIE, that avoids the requirement to find an isomorphism witness and simplifies the other challenges. We have proved that RIE is sound and implemented it in our tool. We imagine an *angelic memory allocator* that, where possible, allocates memory in such a way that equivalent statements in fact result in equal effects. We then ask the verifier to determine if such an allocation is in fact possible, because whenever it is possible the stores are isomorphic and the procedures are equivalent. RIE stands for *Replace Isomorphism with Equality*.

Fully featured program verifiers are large and complex to build, so we use an off-the-shelf modular single program verifier[3] as the basis for our tool. The concept of encoding relational [5, 11, 55], properties of pairs of programs into a single procedure so that standard single program analysis tools can be applied is widely known [7, 25, 31, 38]. SMT based program verification tools have been shown to be effective for automatic procedure equivalence verification [34]. SMT based tools offer the potential for a high degree of automation, in particular we will combine our technique with *mutual summaries* [25] to induce the solver to search for related procedure calls.

1.2 Procedure Equivalence

> *Two procedures are equivalent if no permissible calling context can distinguish which of the procedures was executed.*

Our notion of *procedure equivalence* differs from equi-equivalence in the following ways: it allows procedures to rearrange existing objects and still be considered equivalent; it considers procedure equivalence only from the point of view of the caller, thus allowing us to take full account of the procedure precondition; it allows equivalent procedures to differ in the amount of garbage they allocate; it naturally accounts for procedures that allocate objects at different memory addresses; it is useful when reasoning about irreconcilable calling contexts — we can soundly use facts about procedure equivalence even when the global state of the executions (e.g. the state of the memory allocator, the number or shape of reachable objects) can no longer be equated.

1.3 Organisation

In this dissertation we will: give several illustrative examples of establishing procedure equivalence using RIE; describe RIE in detail; prove that RIE is sound; describe our experimental RIE based tool and its application to several examples; discuss the applicability of our technique and some remaining challenges.

1.4 Introduction to examples

The rest of this chapter illustrates procedure equivalence and RIE with examples. The examples are written in $\mathcal{BL}$. Section 2.1 contains $\mathcal{BL}$'s operational semantics. $\mathcal{BL}$ is a simple, but quite standard, object oriented language. We call the stack and heap together the *store*. To simplify exposition, $\mathcal{BL}$ procedures do not return values, instead values must be passed back to the calling context via the heap.

[3] Such a verifier usually has no inbuilt support for verifying procedure equivalence.

$\mathcal{BL}$ has no statements which can distinguish the actual memory location of an object (lemma 4.1.3). Locations can be compared for equality, but no other information about the memory location can be ascertained. Loops are encoded as recursive procedures.

1.5 Example - different allocation order

In this section we show two equivalent procedures which allocate memory in different orders, and discuss how RIE classifies them as equivalent. Our tool can automatically verify this example. Recursive procedures `lcopy` and `rcopy` (Listing 1.1) both copy the tree passed as parameter `t`. The copy is returned via the heap cell passed as parameter `r`. Procedure `lcopy` copies the left children (`t.l`) followed by the right children (`t.r`), whereas `rcopy` first copies the right children then the left. The postcondition `modifies {r}` asserts that only the fields of object `r` may be modified, all other existing objects must be unchanged. Note that we do not assume or require that parameter `t` does in fact point to a tree shaped heap data structure.

Procedures `lcopy` and `rcopy` are equivalent because when called from equivalent calling contexts, they result in equivalent calling contexts, i.e. immediately after the procedures return, the reachable objects differ only in the actual memory locations where objects are allocated. Precisely, two stores are equivalent iff they are *isomorphic*: i.e. iff there exists a bijection, between their reachable memory locations, that preserves the shape of the store (definition 2.2.1).

Aside from differences in allocation order, this example also has other interesting features. Calls to equivalent procedures occur in stores that are only partially isomorphic because: the object `n` is allocated before the calls in `lcopy` and afterward in `rcopy`; the two sides of the tree are copied in different orders. Note that because the recursive calls and allocations happen in a different order, then there is no useful simulation relationship between the statements in `lcopy` and `rcopy`. RIE instead considers the overall effect of each procedures when determining equivalence.

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

```
19 rcopy(t,r)
20   modifies {r}
21 {
22   if(t != null) {
23     if(r!=null) {
24       rl := new;
25       rr := new;
26
27       rcopy(t.r, rr);
28       rcopy(t.l, rl);
29
30       n := new;
31       n.l := rl.v;
32       n.r := rr.v;
33       r.v := n;
34     }
35   }
36 }
```

Figure 1.1: Two procedures that copy a binary tree. Procedure `lcopy` copies the left nodes first and then the right nodes. The parent node is allocated before copying the children. Procedure `rcopy` copies the right nodes and then the left nodes. The parent node is allocated after copying the children. The postcondition `modifies {r}` asserts that no existing object, other than `r`, is modified.

First consider the question: is a procedure equi-equivalent with itself? Imagine that the procedure

`lcopy` (fig. 1.1) is executed twice from the same initial state. Will both executions always result in the same final state? The answer is, perhaps surprisingly, no. The allocation on line 8 may allocate non-deterministically different addresses in each execution, so the final states may differ.

It may be instructive to consider why the procedures `lcopy` and `rcopy` are not equi-equivalent even if a simple deterministic allocator is assumed. For example consider an allocator that always allocates the next free address. To show a counter example to equi-equivalence, we consider executions where the then branches are taken: the recursive call to `rcopy` on line 27 occurs after one allocation, but the corresponding call to `lcopy` on line 11 occurs after the number of allocations that occur as a result of the call on line 10 plus three. Thus, the nodes of the tree produced by `lcopy` will be allocated at different memory addresses than the corresponding nodes in the tree produced by `rcopy`. The final states will not be equal under such an allocator.

Although procedures `lcopy` and `rcopy` are not equi-equivalent, they are equivalent according to our notion of equivalence: the same shaped tree will be produced regardless of the order of copying the children.

One way to reason about the equivalence of `lcopy` and `rcopy` is to use inductive reasoning of the following form. Hypothesise that recursive isomorphic calls to `lcopy` and `rcopy` are equivalent. Assume that the procedures are executed from isomorphic initial stores. Base case: if `t==null` then the procedures are equivalent, since they both have no effect.

Inductive case: By the modifies clause and otherwise: the stores reachable from the call parameters on lines 10 and 28 are isomorphic; the stores reachable from the call parameters on lines 11 and 27 are isomorphic. Note that the stores are not completely isomorphic, but the parts reachable from the call parameters are.

By above and the induction hypothesis twice: the calls on lines 10 and 28 have isomorphic effects. The calls on lines 11 and 27 have isomorphic effects. We can use the induction hypothesis even though the pre-stores of the calls are only partially isomorphic, because we know from the semantics of $\mathcal{BL}$ that the footprint of a procedure is confined to the part of the store reachable from the call parameters.

By above, the `modifies` clause of the recursive calls, the fact that `rr` and `rl` are not aliases, and standard reasoning about the effects of the other statements: the stores at 18 and 36 are isomorphic in all calling contexts.

1.6 Our Methodology

The main challenges involved in automatically constructing a proof like the one in section 1.5 were presented in section 1.1. Challenge one is establishing isomorphism between a pair of unboundedly large graphs prior to the recursive procedure calls. Even if that challenge is overcome it is still necessary to provide a sufficiently complete and efficient set of equivalence axioms to allow the solver to determine whether the statements subsequent to the procedure calls (line 10 and line 27) do indeed establish isomorphism by procedure exit, despite potentially reaching intermediate stores which are not isomorphic.

The crucial idea in RIE is that whenever we can establish an isomorphism, and it is useful and consistent to do so, then we can soundly assume that the isomorphic store parts are equal. As an aid to intuition, this can equivalently be thought of as: rewriting the address of one of the stores according to the isomorphism; or as proceeding with verification from equal store parts whenever isomorphism is proved; or as verifying under a semantics of angelic memory allocation, where objects are allocated canonical addresses such that isomorphic store parts are equal.

Using this crucial idea immediately solves the problem of inventing equivalence axioms, because once we have taken the store parts to be equal we can use the standard axioms for dealing with statements operating on equal values. This is particularly effective when using an SMT based verification tool chain, since SMT deals naturally with equality. Happily this idea also significantly simplifies challenge one, since we are not faced with constructing a witness to the existence of an isomorphism, rather we can take the identity relation on addresses as a candidate witness and only have to show that the store updates are consistent with such an isomorphism.

Assuming that isomorphic store parts are equal corresponds to verifying procedure equivalence under an approximate semantics. Using such an approximate semantics allows us to make further related assumptions, which we introduce below. In chapter 3 we prove that this approximation is sound — that whenever the verifier can prove procedure equivalence under the approximate semantics, the procedures are in fact equivalent under the precise semantics.

Our approximate semantics only considers executions where:

(A1) the initial stores are equal, rather than isomorphic

(A2) selected isomorphic stores are in fact equal

(A3) calls to equivalent procedures are replaced by calls to the same procedure

(A4) isomorphic calls have *identical*, rather than isomorphic, effects[4].

Furthermore, RIE:

- can be combined with standard (single procedure) verification contracts in a useful way, allowing programmers to manually add additional specification if necessary
- is modular [22, 25, 28] and supports programs with unbounded recursion and heap updates

```
37 if(t != null) {
38   if(r!=null) {
39     n := new;
40     rl := new;
41     rr := new;
42
43     lcopy(t.l, rl);
44     lcopy(t.r, rr);
45
46     n.l := rl.v;
47     n.r := rr.v;
48     r.v := n;
49   }
50 }
```

```
51 if(t != null) {
52   if(r!=null) {
53     rl := new;
54     rr := new;
55
56     lcopy(t.r, rr);
57     lcopy(t.l, rl);
58
59     n := new;
60     n.r := rr.v;
61     n.l := rl.v;
62     r.v := n;
63   }
64 }
```

Figure 1.2: The body of procedure `lcopy` on the left. And on the right, the body of procedure `rcopy` with all calls to `rcopy` replaced by calls to `lcopy`

We now describe how RIE is applied to prove equivalence of `rcopy` and `lcopy`. The verifier must discharge an assertion that whenever `rcopy` and `lcopy` are executed from equal initial states, (A1), they produce isomorphic final states. If the verifier is able to do so then we know that the procedures `rcopy`

[4]You may notice that this item is a consequence of item (A2) and item (A3)

and `lcopy` are in fact equivalent under the precise semantics of $\mathcal{BL}$. Executing the procedure under the approximate semantics differs from precise execution in that:

1. By item (A3), all calls to `rcopy` are replaced by calls to `lcopy`, as illustrated in fig. 1.2. We know that if `rcopy` and `lcopy` are indeed equivalent then the meaning of the procedure bodies will be preserved
2. Since the stores reachable from the call parameters on lines 11 and 27 are isomorphic, by item (A2), only executions where those store parts are equal are admitted. Also for the calls on lines 10 and 28.
3. Calls to a procedure from equal stores have equal effects, by item (A4).

Observations

This example illustrates the following features of our approach:

- since we take the isomorphic store parts to be equal, the verifier does not need to produce a witness to the isomorphism after the recursive calls on lines 10 and 28, and on lines 11 and 27.
- The order of the equivalent calls copying the left and right children is reversed between `lcopy` and `rcopy`, equivalent calls do not have to appear in the same order in each procedure.
- The calls in `lcopy` and `rcopy` can be considered equivalent even though the differing number of allocations before the recursive calls means that the calling contexts are not fully equivalent.
- procedures may be recursive and allocate unbounded amounts of memory[5].
- the equivalence proof takes advantage of the procedure specification, in this case the `modifies {r}` postcondition.

Other features of our approach not illustrated by this particular example:

- procedures may make different rearrangements of existing objects and still be equivalent.
- equivalence is from the viewpoint of the calling context.

1.7 Example — rearrangements of existing objects

Procedures which make different rearrangements of existing objects may be equivalent in specific calling contexts. For example, consider the procedures `swap` and `skip` in Listing 1.3. Executing the body of `swap` from any store where `this.f` aliases `this.g` will leave the calling context isomorphic to the one produced by executing `skip` from that store. However, executing the body of `swap` from a store where the object at `this.f` has a different number of fields to the object at `this.g` result in a calling context that is not isomorphic to the one produced by executing `swap`.

Procedures `skip` and `swap` are not equivalent per se, rather their equivalence depends on which initial stores are permissible. It is standard to specify the permissible initial stores of a procedure in the procedure's precondition. RIE can be combined with preconditions to widen which procedures are considered equivalent. Although RIE allows for `skip` and `swap` to be verified, some implementation questions remain and our tool cannot verify this example.

[5]So an exhaustive or bounded model checking approach would not generally be suitable

```
1 swap(this,x) {
2         t := this.f;
3         this.f := this.g;
4         this.g := t;
5 }
```

```
6 skip(this,x) {
7         assert true;
8 }
9
10
```

Figure 1.3: Two procedures, skip which does nothing, and swap which swaps the values of two fields.

1.8 Completeness

RIE works alongside existing procedure equivalence verification techniques to expand the class of procedures that can be proved equivalent automatically. However, there is an inherent limit to the technique of assigning canonical addresses to objects that means proving the equivalence of some procedure pairs requires additional manual annotation of the program[6]. The limit is that for any given execution of a procedure each object is allocated exactly one address, and each address is associated with at most one object. This limit stems directly from the nature of a heap which is a mapping from addresses to objects.

This limit occurs in procedure pairs where there are several isomorphic store parts that could usefully be assumed to be equal, but doing so would require the same address to be given to more than one object. The rest of this section describes such an example, and shows that although RIE still helps with proving procedure equivalence, it is not complete by itself.

This example is a refactoring of some code that uses a linked list. The example is difficult because, as we will see, two calls in the first procedure are isomorphic with one call in the second procedure, but the isomorphisms map one address in the second procedure with two different addresses in the first procedure. This inhibits a verifier using RIE from assuming isomorphism to be equality in *all* the places where it would be useful, although it can still do so in *some* of the places where it is useful.

Figure 1.4 shows equivalent list manipulating procedures `addLru_a` and `addLru_b`. They first remove any existing occurrence of an object from the list, then add the object to the end of the list, thus maintaining the least recently used item at the start of the list. The procedure `addLru_a` has more code than is necessary because the `remove` procedure itself first checks that the object is in the list before removing it. On noticing this redundancy, a programmer may remove the conditional, and call to `contains`, from `addLru_a` to produce the equivalent procedure `addLru_b`.

Due to the modular nature of our approach, it cannot directly verify the equivalence of procedures `addLru_a` and `addLru_b`, because the called procedures (`add`, `remove`, `contains`), are insufficiently specified. In this situation it is sometimes productive to inline each procedure call once and try to verify the resulting procedures instead. Figure 1.6 shows the body of `addLru_a` on the left with all calls inlined once, and on the right the body of `addLru_b` with all calls inlined once. We call the inlined versions `addLru_a'` and `addLru_b'`

The relevant list procedures are in fig. 1.5. The list is doubly linked, with sentinel nodes representing the start and end. Each node in the list has `next` and `prev` pointers, and a flag `sentinel` which indicates the sentinel nodes.

The procedures in fig. 1.6 are equivalent. The call on line 55 is isomorphic with the call on line 89. Furthermore, due to the postcondition of `find`, the call on line 61 is also isomorphic with the call on line 89. This allows us to reason that the conditions on lines 58, 63 and 92 are all true or all false, i.e. `rf.v.sentinel` = `rf0.v.sentinel` = `rf1.v.sentinel`. The rest of the proof is standard.

[6] or perhaps by some other clever technique we are not yet familiar with

```
addLru_a(l,x) {
  new rc;
  contains(l, x, rc);
  if(rc.v) {
    remove(l, x);
    add(l,x);
  } else {
    add(l,x);
  }
}
```

```
addLru_b(l,x) {
  remove(l,x);
  add(l,x);
}
```

Figure 1.4: A difficult example, where the equivalence of these procedures relies on the equivalence of two calls to `contains` in `addLru_a` (one call is inside `remove`) with one call to `contains` in `addLru_b` (this call is also inside `remove`).

```
remove(l,x) {
   new rf;
   find(l,x,rf);
   node := rf.v;
   if !node.sentinel {
     node.prev.next := node.next;
     node.next.prev := node.prev;
   }
}

add(l,x) {
   new node;
   node.v := x;
   node.sentinel := false;
   node.next := l.last;
   node.prev := l.last.prev;
   l.last.prev.next := node;
   l.last.prev := node;
}

contains(l,x,r) {
   new rf;
   find(l,x,rf);
   node := rf.v;
   r.v := !node.sentinel
}

find(l,x,r) modifies {r} {
   ...
}
```

Figure 1.5: The list manipulation procedures that are relevant to our example

When we try to automatically verify the equivalence of `addLru_a'` and `addLru_b'`, RIE allows the verifier to take isomorphic calls to be equal (see section 1.5). However, if it does so for more than one pair of stores, this could sometimes lead to a contradiction with the semantics of `new`. Even an angelic allocator may have no way to allocate the addresses such that *both* pairs of isomorphic store parts are equal. To avoid this, we insist that the union of all isomorphisms that the verifier chooses to assume are equal must be an injection. We write that the *isomorphisms are compatible* iff their union is an injection.

The isomorphism between lines 55 and 89 relates the addresses in `rf0` and `rf`. The isomorphism between lines 61 and 89 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 54 whereas `rf1` is allocated by the subsequent `new` statement on line 60. The semantics of `new` require that each allocation gives an address which was not previously allocated — i.e. that `rf0` $\neq$ `rf1`.

Due to this restriction to compatible isomorphisms, 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. For instance, it is possible to progress with this example by manually providing an additional mutual summary [25] relating the behaviour of calls to find more precisely.

```
51 // addLru_a'
52 // contains(l, x, rc);
53 new rc;
54 new rf0;
55 find(l,x,rf0);
56 node := rf0.v;
57 rc.v := !node.sentinel
58 if(rc.v) {
59   // remove(l,x);
60   new rf1;
61   find(l,x,rf1);
62   node := rf1.v;
63   if(!node.sentinel){
64     node.prev.next := node.next;
65     node.next.prev := node.prev;
66   }
67   // add(l,x);
68   new node;
69   node.v := x;
70   node.sentinel := false;
71   node.next := l.last;
72   node.prev := l.last.prev;
73   l.last.prev.next := node;
74   l.last.prev := node;
75 } else {
76   // add(l,x);
77   new node;
78   node.v := x;
79   node.sentinel := false;
80   node.next := l.last;
81   node.prev := l.last.prev;
82   l.last.prev.next := node;
83   l.last.prev := node;
84 }
```

```
85 // addLru_b'
86
87 // remove(l,x);
88 new rf;
89 find(l,x,rf);
90 node := rf.v;
91
92 if(!node.sentinel){
93
94
95
96
97
98   node.prev.next := node.next;
99   node.next.prev := node.prev;
100 }
101 // add(l,x);
102 new node;
103 node.v := x;
104 node.sentinel := false;
105 node.next := l.last;
106 node.prev := l.last.prev;
107 l.last.prev.next := node;
108 l.last.prev := node;
109
110
111
112
113
114
115
116
117
118
```

Figure 1.6: A difficult example with procedure calls inlined once

1.9 Summary

We make the following contributions. A verification methodology, RIE, for automatically proving contextual equivalence of procedures which differ in the order and amount of memory allocation. RIE is suitable for implementation using a standard SMT based program verifier. We give a proof that RIE is sound. And describe a program verification tool, which implements RIE, for a simple object-orientated programming language. The tool demonstrates how an angelic memory allocator can be modelled in the Boogie [3] intermediate verification language. Our tool adapts the mutual summaries [25] approach and has a SYMDIFF-like [34] design. The tool can automatically verify some complex examples involving recursive procedures that allocate unbounded amounts of memory in different orders.

Chapter 2

Preliminaries

This chapter includes the definitions of our language $\mathcal{BL}$ and some basic formal results. The style of this chapter is formal and precise. The reader would normally skim this chapter then refer back when a definition or basic result is needed.

2.1 Operational Semantics of $\mathcal{BL}$

This section details the operational semantics of $\mathcal{BL}$, as well as the operational semantics that will be used to model the behaviour of our tool and in the proof of theorem 3.2.1 (page 36).

The operational semantics of $\mathcal{BL}$ produce a trace of the states reached during execution. Since $\mathcal{BL}$ is non-deterministic we use these execution traces to distinguish particular executions. The semantics of $\mathcal{BL}$ are parameterised by $\mathcal{L} \in \{\mathcal{C}, \mathcal{V}, \mathcal{R}, \mathcal{K}, \mathcal{A}\}$, representing the precise semantics of $\mathcal{BL}$ and several approximations. The approximate $\mathcal{A}$ semantics models RIE, as we discuss in chapter 5.

$\mathcal{BL}$ includes recursive procedure calls, but not loops. Objects have fields, and values are heap addresses or booleans. Procedure calls do not return a value, but returning values can be encoded by passing them via the heap. Conditional execution of atomic statements can be used to encode if-then-else blocks.

Our rule for procedure call evaluates the procedure body after extending the stack. Though our semantics is a big step semantics, we keep the whole calling context to allow us to give useful meaning to isomorphism of stores.

Definition 2.1.1 (Semantics of $\mathcal{BL}$) $\mathcal{BL}$

Our language $\mathcal{BL}$ consists of

- *Infinite enumerable set Addr of addresses, ranged over by a, with one reserved element* `null`*. Extended to* $Val \stackrel{def}{=} Addr \cup \{\texttt{true}, \texttt{false}\}$*, ranged over by v.*
- *Enumerable sets Lid of variable names, ranged over by x; Fid of field names, ranged over by f; Pid of procedure names, ranged over by f.*
- *Heap modelled as a partial map,* $Heap \stackrel{def}{=} (Addr \times Fid) \rightharpoonup Val$*. Ranged over by h, and* $\forall h, f : h(\texttt{null}, f) = \texttt{null}$*.*
- *Stack frames* $StackF \stackrel{def}{=} Lid \rightharpoonup Val$*, ranged over by* σ*.*

- *Stacks* $Stack \stackrel{def}{=} StackF^*$ *is a sequence of stack frames*[a]*, ranged over by* $\tilde{\sigma}$*. The* i^{th} *element is written* $\tilde{\sigma}[i]$*, and* $\tilde{\sigma} \cdot \sigma$ *means the stack obtained by appending* $\tilde{\sigma}$ *with* σ*.*
- *Stores* $Store \stackrel{def}{=} Stack \times Heap$*, ranged over by* ϕ*. The heap of* ϕ *is* ϕ^h*, and the stack* $\phi^{\tilde{\sigma}}$*;* $\phi(x)$ *is the value of the variable* x *in the topmost frame of* $\phi^{\tilde{\sigma}}$*; and* $\phi(x, \overline{f})$ *is the value obtained by following the, possibly empty, sequence of fields* $\overline{f}$ *from the object at address* $\phi(x)$*.*
- *Statements* $Stmt$*, defined by the grammar in figure 2.1.*
- *Function* $\mathcal{B} : Pid \to Stmt$ *looks up procedure bodies from procedure names. And partial function* $\mathcal{V} : Pid \times \mathbb{N} \rightharpoonup Lid$ *looks up parameter names from procedure names.*
- *Execution traces* $Trace \stackrel{def}{=} (Stmt \times Store \times Store)^*$
- *A set* $Sem \stackrel{def}{=} \{\mathcal{C}, \mathcal{V}, \mathcal{R}, \mathcal{K}, \mathcal{A}\}$ *ranged over by* $\mathcal{L}$ *used to indicate which semantics are required.*
- *A big step execution relation defined in figures 2.1-2.2.*

$$\rightsquigarrow_{\mathcal{L}} \subseteq Store \times (Stmt \cup \texttt{body}) \times Sem \to Trace \times Store$$

- *Function* $Con : Pid \to \mathcal{P}(Store \times Store)$ *representing the pre and post conditions of a procedure. The contract is a set of pairs of Store representing the set of acceptable pre and post stores of the procedure. No contract may discriminate between stores based on the actual values of memory addresses:*

$$\begin{array}{l} \forall \mathtt{f}, \phi_{1\ldots4}, \pi_{1,2} : \\ \quad \phi_1 \approx_{\pi_1} \phi_2 \ \wedge \ \phi_3 \approx_{\pi_2} \phi_4 \ \wedge (\phi_1, \phi_3) \in Con(\mathtt{f}) \ \wedge \ com(\pi_1, \pi_2) \\ \quad \Longrightarrow \\ \quad (\phi_2, \phi_4) \in Con(\mathtt{f}) \end{array}$$

- *The function* $\texttt{alloc} : Heap \times Addr \to Heap$

$$(a_1, f, v) \in \texttt{alloc}(h, a_2) \stackrel{def}{\iff} (a_1, f, v) \in h \ \vee (a_1 = a_2 \ \wedge \ v = \texttt{null})$$

- *The function* $mkframe : Store \times Pid \times Lid^* \to Store$

$$mkframe(\phi, \mathtt{f}, x_1 \ldots x_n) \stackrel{def}{=} (\phi^{\tilde{\sigma}} \cdot [\mathcal{V}(\mathtt{f}, 1) \mapsto \phi(x_1), \ldots, \mathcal{V}(\mathtt{f}, n) \mapsto \phi(x_n)], \phi^h)$$

- *The function* $pop(\tilde{\sigma} \cdot \sigma, h) \stackrel{def}{=} (\tilde{\sigma}, h)$

[a] Although the semantics are bigstep we carry the calling context to allow us to give a global meaning to isomorphism

RIE is designed to work with programs that have single program contracts that have already been verified. Programs with no contracts are also fine. Verifying a program involves proving that before every procedure call, the procedure precondition is established, and that given the precondition each procedure entails its postcondition. The verified semantics $\rightsquigarrow_{\mathcal{V}}$ model execution of such a verified program, and as such are defined only for programs where all procedure contracts are valid. We do not say how such contracts should be verified. Note that $\rightsquigarrow_{\mathcal{V}}$ is equivalent to $\rightsquigarrow_{\mathcal{C}}$ in the case that all procedures have the trivial contract $Store \times Store$.

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

$$\frac{}{\phi, e_1.f := e_2 \hookrightarrow (\phi^\sigma, \phi^h[([\![e_1]\!]_\phi, f) \mapsto [\![e_2]\!]_\phi])}\ \text{STORE}$$

$$\frac{a \notin \mathit{dom}(\phi^h) \qquad a \neq \texttt{null}}{\phi, x := \texttt{new}() \hookrightarrow (\phi^\sigma[x \mapsto a], \texttt{alloc}(\phi^h, a))}\ \text{NEW}$$

$$\frac{}{\phi, x := e \hookrightarrow (\phi^\sigma[x \mapsto [\![e]\!]_\phi], \phi^h)}\ \text{ASSIGN} \qquad \frac{\phi \vDash b}{\phi, \texttt{assert}\ b \hookrightarrow \phi}\ \text{ASSERTT}$$

$$\frac{\phi \vDash b}{\phi, \texttt{assume}\ b \hookrightarrow \phi}\ \text{ASSUME} \qquad \frac{[\![b]\!]_\phi = \texttt{true}}{\phi \vDash b} \qquad \frac{\neg(\phi \vDash b)}{\phi, \texttt{assert}\ b \hookrightarrow \texttt{error}}\ \text{ASSERTF}$$

$$\frac{}{[\![\texttt{null}]\!]_\phi = \texttt{null}} \qquad \frac{}{[\![x]\!]_\phi = \phi(x)} \qquad \frac{}{[\![e.f]\!]_\phi = \phi^h([\![e]\!]_\phi, f)}$$

$$\frac{\phi_1 \vDash b \qquad \phi_1, s_a \rightsquigarrow_{\mathcal{L}}^{tr} \phi_2}{\phi_1, \texttt{if}(b)\{s_a\} \rightsquigarrow_{\mathcal{L}}^{tr} \phi_2}\ \text{CONDT} \qquad \frac{\neg(\phi \vDash b)}{\phi, \texttt{if}(b)\{s_a\} \rightsquigarrow_{\mathcal{L}}^{\texttt{if}(b)\{s_a\}, \phi, \phi} \phi}\ \text{CONDF}$$

$$\frac{\phi_1, s \hookrightarrow \phi_2}{\phi_1, s \rightsquigarrow_{\mathcal{L}}^{s, \phi_1, \phi_2} \phi_2}\ \text{ATOM} \qquad \frac{\phi_1, s_1 \rightsquigarrow_{\mathcal{L}}^{tr_1} \phi_2 \qquad \phi_2, s_2 \rightsquigarrow_{\mathcal{L}}^{tr_2} \phi_3}{\phi_1, s_1; s_2 \rightsquigarrow_{\mathcal{L}}^{tr_1 \cdot tr_2} \phi_3}\ \text{TRANS}$$

Where

- $[\![x = y]\!]_\phi = \mathit{if}\ \phi(x) = \phi(y)\ \mathit{then}\ \texttt{true}\ \mathit{else}\ \texttt{false}$
- $[\![!b]\!]_\phi = \mathit{if}\ [\![b]\!]_\phi = \texttt{false}\ \mathit{then}\ \texttt{true}\ \mathit{else}\ \texttt{false}$
- $[\![b_1 \,\&\&\, b_2]\!]_\phi = \mathit{if}\ [\![b_1]\!]_\phi = \texttt{true} \wedge [\![b_2]\!]_\phi = \texttt{true}\ \mathit{then}\ \texttt{true}\ \mathit{else}\ \texttt{false}$

Figure 2.1: Grammar and Operations of $\mathcal{BL}$

$$\frac{mkframe(\phi_1, \texttt{f}, x_1 \dots x_n), \texttt{body f} \rightsquigarrow^{tr}_{\mathcal{L}} \phi_3 \qquad \mathcal{L} \neq \mathcal{A}}{\phi_1, \texttt{call f}(x_1 \dots x_n) \rightsquigarrow^{(\texttt{call f}(x_1 \dots x_n), \phi_1, \phi_3^{ctx})}_{\mathcal{L}} \phi_3^{ctx}} \text{CALLV}$$

$$\frac{\begin{array}{c} dom(h_2) \supseteq dom(\phi_1^h) \qquad \phi_2 = (\phi_1^{\tilde{\sigma}} \cdot \sigma, h_2) \\ (mkframe(\phi_1, \texttt{f}, x_1 \dots x_n), \phi_2) \in Con(\texttt{f}) \end{array}}{\phi_1, \texttt{call f}(x_1 \dots x_n) \rightsquigarrow^{(\texttt{call f}(x_1 \dots x_n), \phi_1, \phi_2^{ctx})}_{\mathcal{A}} \phi_2^{ctx}} \text{CALLA}$$

$$\frac{\phi_1, \mathcal{B}(\texttt{f}) \rightsquigarrow^{tr}_{\mathcal{C}} \phi_3}{\phi_1, \texttt{body f} \rightsquigarrow^{tr}_{\mathcal{C}} \phi_3} \text{BODC} \qquad \frac{(\phi_1, \_) \in Con(\texttt{f}) \qquad \phi_1, \mathcal{B}(\texttt{f}) \rightsquigarrow^{tr}_{\mathcal{V}} \phi_3}{\phi_1, \texttt{body f} \rightsquigarrow^{tr}_{\mathcal{V}} \phi_3} \text{BODV}$$

$$\frac{(\phi_1, \_) \in Con(\texttt{f}) \qquad \phi_1, ren_{\mathcal{E}}(\mathcal{B}(\texttt{f})) \rightsquigarrow^{tr}_{\mathcal{L}} \phi_3 \qquad \mathcal{L} \in \{\mathcal{R}, \mathcal{K}, \mathcal{A}\}}{\phi_1, \texttt{body f} \rightsquigarrow^{tr}_{\mathcal{L}} \phi_3} \text{BODA}$$

$$\frac{\phi_1, s_1 \rightsquigarrow^{tr_1}_{\mathcal{L}} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow^{tr_2}_{\mathcal{L}} \phi_4 \qquad \mathcal{L} \in \{\mathcal{C}, \mathcal{V}, \mathcal{R}\}}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow^{tr_1, tr_2}_{\mathcal{L}} \phi_3 \parallel \phi_4} \text{COMV}$$

$$\frac{misos(tr_1, tr_2) \subseteq id \qquad \phi_1, s_1 \rightsquigarrow^{tr_1}_{\mathcal{K}} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow^{tr_2}_{\mathcal{K}} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow^{tr_1, tr_2}_{\mathcal{K}} \phi_3 \parallel \phi_4} \text{COMK}$$

$$\frac{misos(tr_1, tr_2) \subseteq id \qquad \mathcal{U}(tr_1, tr_2) \qquad \phi_1, s_1 \rightsquigarrow^{tr_1}_{\mathcal{A}} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow^{tr_2}_{\mathcal{A}} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow^{tr_1, tr_2}_{\mathcal{A}} \phi_3 \parallel \phi_4} \text{COMA}$$

Figure 2.2: Procedure call and composition

2.2 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 2.2.1 defines isomorphism of stores. A store ϕ is a sequence of stack frames and a heap. The value in field f of an object a is written $\phi(a, f)$. The number of stack frames is denoted by $|\phi|$. The value of variable x in the i^{th} stack frame is $\phi[i](x)$. The domain of the mapping π is $dom(\pi)$, and $dom(\phi[i])$ is the set of variables defined in the i^{th} stack frame. Section 2.1 contains the full definition of *Store*.

In words, definition 2.2.1 says that $\phi_1 \approx_\pi \phi_2$ iff π is a relation where: the stacks of ϕ_1 and ϕ_2 are the same height; for each corresponding stack frame the same variables are defined and π maps between them; π commutes with field dereference for all reachable objects; and π is an injection that preserves the meaning of `null`, `true`, and `false`.

Definition 2.2.1 (Isomorphism)
Define relation $\approx \subseteq \mathit{Store} \times \mathcal{P}(\mathit{Val} \times \mathit{Val}) \times \mathit{Store}$, *such that:*

$$\phi_1 \approx_\pi \phi_2 \overset{def}{\iff}$$

- $|\phi_1| = |\phi_2| \ \wedge\ \forall i \leq |\phi_1| : \mathit{dom}(\phi_1[i]) = \mathit{dom}(\phi_2[i])$
- π *is an injection, written* $\mathit{in}(\pi)$

Characterising relation π *is inductively defined:*

$$\pi_v \cup \{\phi_1[i](x) \mapsto \phi_2[i](x) \mid x \in \mathit{dom}(\phi_1[i]) \ \wedge\ i \leq |\phi_1|\} \cup \{\phi_1(a, f) \mapsto \phi_2(\pi(a), f) \mid a \in \mathit{dom}(\pi)\}$$

And

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

The bijection (π) is the least-fixed-point interpretation of the definition. Exactly one bijection characterises any pair of isomorphic stores (lemma 2.2.2), which helpfully shortens some parts of the proof.

Lemma 2.2.2 (Isomorphism is unique)

$$\forall \phi_{1,2}, \pi_{1,2} : \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ \phi_1 \approx_{\pi_2} \phi_2 \implies \pi_1 = \pi_2$$

Full proof on page 83.

Proof Outline. Take two bijections $\pi_{1,2}$ between an arbitrary isomorphic pair of stores. Without loss of generality, by induction on the definition of $\approx$ every element of π_1 is in π_2. □

2.3 Compatibility of Isomorphisms

A surprising aspect of RIE is the requirement that equality can only be assumed for a set of compatible isomorphisms (see section 1.8)[1], consistent with the semantics of NEW. This is achieved using the intran-

[1] In fact we actually implemented an unsound version of the tool before we realised what the correct condition should be. We tried using subset and other similar transitive relations to restrict the selection of isomorphisms. We found that transitive relations are too restrictive to be useful when proving equivalence of programs that create different garbage. Such transitive relations also make it difficult to relate isomorphisms that refer to different parts of a store.

sitive relation *com*. A set of mappings satisfy *com* whenever their union is an injection. We call such sets of mappings *compatible* (definition 2.3.1).

Because an injection provides a unique alternative address for every address in its domain, we can consider a compatible set of mappings as a (partial) description of an *alternative allocation strategy*. The selected isomorphisms describe an alternate allocation strategy which can be used to construct an alternative execution — one that is related by the identity isomorphism at the selected places. The rest of this section is a discussion of the key lemmas about compatibility. The lemmas revolve around the idea that compatibility ensures that we can always pick the desired address when building the alternative execution.

Definition 2.3.1 (Compatibility of injections)

$$com(\Pi) \overset{def}{\iff} in\Big(\bigcup \Pi\Big)$$

where $in(\pi) \overset{def}{\iff} \forall (a,b),(c,d) \in \pi : (a = c \iff b = d)$

Several $\mathcal{BL}$ instructions can cause heap objects to become unreachable. Relation *com* has the property that if injections π_1 and π_2 are compatible then π_2 is compatible with any subset of π_1. This property makes *com* useful for garbage creating executions, since an instruction that creates garbage may preserve compatibility.

Lemma 2.3.2 (Compatible with a smaller injection)

$$\forall \pi_{1\ldots3} : com(\pi_1, \pi_2) \land \pi_3 \subseteq \pi_1 \implies com(\pi_3, \pi_2)$$

Full proof on page 84.

Proof Outline. Directly. All pairs from π_1 and π_2 are compatible, and all elements of π_3 are in π_1. □

The NEW instruction allocates an address. If the address is in the alternative allocation strategy then it must be possible to allocate the related address, in the alternative execution, while maintaining compatibility. The relation *com* has the property that for compatible injections π_1, π_2 any element of π_2 can be added to π_1 while still maintaining compatibility (lemma 2.3.3).

Lemma 2.3.3 (Can add compatible element to injection)

$$\forall \pi_{1,2}, a_{1,2} : com(\pi_1, \pi_2) \land (a_1, a_2) \in \pi_2 \implies com(\pi_1 \cup (a_1, a_2), \pi_2)$$

Full proof on page 85.

Proof Outline. Directly. Since all pairs from $\pi_1 \cup \pi_2$ are compatible, and (a_1, a_2) is in π_2. □

The alternative allocation strategy is partial, not every allocation may have a defined alternative address. For example, some temporary memory allocated within a procedure but garbage at procedure exit may not appear in any isomorphism. Still, it is necessary to allocate some alternative address in the alternative execution. The relation *com* has the property that for compatible injections π_1, π_2 any pair of addresses disjoint from the domain/range of $\pi_1 \cup \pi_2$ can be added to π_1 while still maintaining compatibility (lemma 2.3.4). Thus we can pick an arbitrary alternative address, provided it does not appear in the alternative allocation strategy.

Lemma 2.3.4 (Can add disjoint element to injection)

$$\begin{aligned}&\forall \pi_{1,2}, a_{1,2}:\\ &\quad com(\pi_1,\pi_2) \ \wedge\ a_1 \notin dom(\pi_1 \cup \pi_2) \ \wedge\ a_2 \notin rng(\pi_1 \cup \pi_2)\\ &\quad \Longrightarrow\\ &\quad com(\pi_1 \cup (a_1,a_2), \pi_2)\end{aligned}$$

Full proof on page 85.

Proof Outline. Directly, since a_1 doesn't appear in the domain and a_2 doesn't appear in the range no incompatibility can occur. □

In these ways *com* allows us to ensure that we can always allocate an appropriate address, and thus always construct an execution with our desired allocation strategy.

2.4 Reachability

Our desire to reason about garbage creating procedures lead us to a notion of isomorphism that is closely related to heap reachability. Some properties of reachability are important in our proof and are described in this chapter. Later, in lemma 2.4.11 we prove the relationship between heap reachability and isomorphism. In chapter 6 we will discuss how heap reachability features in the implementation of our tool.

Definition 2.4.1 (Reachable addresses)
We define the function reach : Store → $\mathcal{P}(Addr)$ as the least solution of

$$\begin{aligned}a \in reach(\phi) \ \overset{def}{\Longleftrightarrow}\ &(\exists i, x:\ \phi[i](x) = a) \ \vee\\ &\big(\exists a' \in reach(\phi):\ \phi^h(a', f) = a\big)\end{aligned}$$

Several properties of $\approx$ are derived from the fact that *reach* is a function, for example lemma 2.2.2. Lemma 2.4.2 states that *reach* is well defined.

Lemma 2.4.2 (*reach* is a function)

$$\forall \phi, A_{1,2} : (\phi, A_1) \in reach \ \wedge (\phi, A_2) \in reach \implies A_1 = A_2$$

Full proof on page 87.

Proof Outline. By induction on the definition of *reach*. □

In order to prove later that $\mathcal{BL}$ does not allow address synthesis – i.e. to show that only the NEW instruction can increase the set of reachable addresses – we will observe several facts about how reachability is affected by each instruction (lemma 4.1.5, lemma 4.1.8 lemma 4.1.10, etc). The proofs of these facts rely on the following arguments:

1. Expressions which evaluate to an address, always evaluate to a reachable address (lemma 2.4.3), because all paths from the stack through the heap lead to a reachable address (lemma 2.4.5)
2. Any path through a modified field or variable must be constructed from either a new address or by composing reachable path fragments. Due to item 1 and lemmas 2.4.4 and 2.4.5

All expressions that evaluate to an address evaluate to a reachable address. This is because $\mathcal{BL}$ does not allow addresses to be synthesised or otherwise manipulated (for example due to pointer arithmetic). This property of $\mathcal{BL}$ is crucial in allowing equivalent procedures to differ in the amount and shape of garbage that they create.

Lemma 2.4.3 (Expressions evaluate to reachable address)

$$\forall \phi, e, a : [\![e]\!]_\phi = a \implies a \in \mathit{reach}(\phi)$$

Full proof on page 88.

Two further lemmas are useful. Lemma 2.4.4 says that there is a path from a stack variable via a sequence of fields to every reachable address. Lemma 2.4.5 says that every path from a stack variable via a sequence of fields leads to a reachable address.

Lemma 2.4.4 (Path to reachable address)

$$\forall \phi, a \in \mathit{reach}(\phi) : \exists i, x, \overline{f} : \phi^h(\phi^{\tilde{\sigma}}[i](x), \overline{f}) = a \ \wedge \ \mathit{acyclic}(\phi, \phi^{\tilde{\sigma}}[i](x), \overline{f})$$

where

$$\mathit{acyclic}(\phi, a, \overline{f}) \overset{def}{\iff} \nexists \overline{g} \cdot \overline{g}' = \overline{f}, \overline{h} \cdot \overline{h}' = \overline{f} : \overline{g} \neq \overline{h} \ \wedge \ \phi^h(a, \overline{g}) = \phi^h(a, \overline{h})$$

Full proof on page 89.

Lemma 2.4.5 (Paths are reachable)

$$\forall \phi, i, x, \overline{f} : \phi^h(\phi^{\tilde{\sigma}}[i](x), \overline{f}) = a \implies a \in \mathit{reach}(\phi)$$

And

$$\forall \phi, a_1 \in \mathit{reach}(\phi), a_2, \overline{f} : \phi^h(a_1, \overline{f}) = a_2 \implies a_2 \in \mathit{reach}(\phi)$$

Full proof on page 89.

2.4.1 Isomorphism in Calling Context

Poping the top of the stack reduces the reachable set. We will use this to establish that any alternative execution we construct preserves any calling context isomorphism of the original trace.

Lemma 2.4.6 (Calling context reachability smaller)

$$\forall \phi : \mathit{reach}(\phi^{ctx}) \subseteq \mathit{reach}(\phi)$$

Full proof on page 89.

Proof Outline. Straightforwardly by induction on the definition of *reach*. □

Lemma 2.4.7 (Isomorphism implies calling context isomorphism)

$$\forall \phi_{1,2} : \phi_1 \approx_{\pi_1} \phi_2 \implies \phi_1^{ctx} \approx_{\pi_2} \phi_2^{ctx} \ \wedge \ \pi_2 = \pi_1{\downarrow_{\mathit{reach}(\phi_1^{ctx})}} \cup \pi_v$$

Where $\pi_1{\downarrow_{\mathit{reach}(\phi_1^{ctx})}}$ *means the relation produced by restricting the domain of* π_1 *to the reachable*

addresses of (ϕ_1^{ctx})

Full proof on page 90.

Proof Outline. By induction on the definition of $\approx$ and lemma 2.4.6. □

2.4.2 Reachable gets Smaller

Once an allocated address becomes unreachable, it stays unreachable. The reachable set increases only if new addresses are allocated. Specifically, if an address is reachable in a later trace element, it must be either reachable or unallocated in every earlier trace element. And if an address is unallocated in a trace element it is unallocated in every earlier trace element.

Lemma 2.4.8 (Reach gets smaller)

$$\begin{aligned}
&\forall \mathcal{L} \neq \mathcal{A}, s, \phi_{1,3}, i \leq j, a_1 \leq |tr|, k, l \in \{2,3\}: \\
&\quad \phi_1, s \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \ \wedge \ k \leq l \\
&\quad \implies \\
&\quad \left(a_1 \in \mathit{reach}(tr_1[j]{\downarrow}l) \implies a_1 \in \mathit{reach}(tr_1[i]{\downarrow}k) \ \vee \ a_1 \notin tr_1[i]{\downarrow}k^h\right) \wedge \\
&\quad \left(a_1 \notin tr_1[j]{\downarrow}l^h \implies a_1 \notin tr_1[i]{\downarrow}k^h\right)
\end{aligned}$$

Full proof on page 93.

Furthermore, all write effects involve reachable or newly allocated addresses. Nothing gets written to the garbage, and references to the garbage never get written anywhere.

Definition 2.4.9 (Write effects)

$$\mathit{effect}(\phi_1, \phi_3) \overset{def}{=} \phi_3^h \setminus \phi_1^h$$

Lemma 2.4.10 (Effects are reachable)

$$\begin{aligned}
&\forall \mathcal{L} \neq \mathcal{A}, s, \phi_{1,2}, a_{1,2}, f: \\
&\quad \phi_1, s \overset{tr}{\rightsquigarrow_{\mathcal{L}}} \phi_2 \ \wedge (a_1, f, a_2) \in \mathit{effect}(\phi_6, \phi_4) \\
&\quad \implies \\
&\quad (a_1 \in \mathit{reach}(\phi_1) \ \vee \ a_1 \notin \phi_1^h) \ \wedge (a_2 \in \mathit{reach}(\phi_1) \ \vee \ a_2 \notin \phi_1^h)
\end{aligned}$$

Full proof on page 96.

2.4.3 Isomorphism and Reachability

If a pair of states $\phi_{1,2}$ are isomorphic with bijection π (i.e. $\phi_1 \approx_\pi \phi_2$) then the domain of π is the reachable addresses of ϕ_1 union the values $\{\texttt{null}, \texttt{true}, \texttt{false}\}$, and the range of π is the reachable addresses of ϕ_2 union the values $\{\texttt{null}, \texttt{true}, \texttt{false}\}$.

Lemma 2.4.11 (Domain of isomorphism is reachable addresses)

$$\forall \phi_{1,2}, \pi: \ \phi_1 \approx_\pi \phi_2 \implies \mathit{reach}(\phi_1) \cup \mathit{dom}(\pi_v) = \mathit{dom}(\pi)$$

Full proof on page 96.

2.5 Composition of Isomorphism

The bijections which characterise isomorphisms can be composed. This is particularly useful in the proof of lemma 4.3.1 (Isomorphism is an equivalence relation).

Definition 2.5.1 (Isomorphism composition)

$$\pi_1 \circ \pi_2 \stackrel{def}{=} \{(a, c) \mid \exists b : (a, b) \in \pi_1 \ \wedge (b, c) \in \pi_2\}$$

Composition has several useful properties. The composition of two injections is itself an injection (lemma 2.5.2). And also, the composition of an injection with its inverse is a subset of the identity relation (lemma 2.5.3).

Lemma 2.5.2 (Composition of injections is an injection)

$$\forall \pi_{1,2} : \ in(\pi_1) \ \wedge \ in(\pi_2) \implies in(\pi_1 \circ \pi_2)$$

Full proof on page 98.

Lemma 2.5.3 (Injection composed with inverse is identity)

$$\forall \pi_{1\ldots3} : \ in(\pi_1) \ \wedge \ \pi_2 \subseteq \pi_1 \ \wedge \ \pi_3 \subseteq \pi_1^{-1} \implies \pi_2 \circ \pi_3 \subseteq id$$

And also

$$\forall \pi : \ in(\pi) \implies \pi \circ \pi^{-1} = id{\downarrow_{dom(\pi)}}$$

Full proof on page 99.

Chapter 3

RIE Methodology

In this chapter we define procedure equivalence and our RIE methodology for ascertaining it. We state RIE's soundness theorem, and outline the main lemmas in its proof for a simple language called $\mathcal{BL}$. The full proofs are in appendix B.

Our definition of procedure equivalence is contextual [40, 41], procedures are equivalent when no difference in behaviour is observable in any permitted calling context. This corresponds to the notion of equivalence typically used in refactoring [44]. Our definition is powerful enough to cope with: changes in allocation order; differences in garbage; and different rearrangements of existing objects.

Automatically verifying contextual equivalence of procedures that differ in allocation order is challenging for an SMT based verifier. We propose a verification methodology, RIE, under which it is easier for such tools to verify procedure equivalence. RIE assumes some isomorphic store parts are equal and ignores unobservable differences in called procedures. We describe why RIE helps verification tools in section 3.6.

We formalise RIE as an approximate semantics called $\mathcal{A}$. We prove soundness, i.e. that if procedures are equivalent when executed under the approximate $\mathcal{A}$ semantics, then they are also equivalent when executed under the ordinary semantics called $\mathcal{V}$ (theorem 3.2.1). We present important semantic rules as they arise, but the full semantics are in section 2.1.

3.1 Procedure Equivalence

Procedures are equivalent whenever no permissible calling context can distinguish which of the procedures was executed. In order to establish procedure equivalence a verification tool must be able to judge when a pair of stores satisfies this indistinguishability condition. For our language, $\mathcal{BL}$, isomorphic stores cannot be distinguished by any calling context. We now define procedure equivalence directly in terms of isomorphism between stores, but we defer defining isomorphism precisely until chapter 4. Later we will show how RIE makes it possible to establish such isomorphisms in an SMT based tool.

Procedures are equivalent whenever they always result in isomorphic calling contexts when executed to completion from isomorphic initial stores. We write $\mathtt{f}_1 \lessapprox \mathtt{f}_2$ to mean procedure $\mathtt{f}_1$ is equivalent to $\mathtt{f}_2$ under semantics $\rightsquigarrow_{\mathcal{L}}$. The symbol $\mathcal{L}$ ranges over the set of semantics. In this chapter we formalise our tool in terms of the real semantics of $\mathcal{BL}$, called $\mathcal{V}$, and an approximate semantics called $\mathcal{A}$. Semantics $\mathcal{V}$ and $\mathcal{A}$ vary only in how they treat procedure call. In the soundness proof some other semantics appear for technical reasons.

Since we are interested in the relative behaviour of pairs of procedures, our semantics include a rule

for executing a pair of procedures. The notation $\rightsquigarrow_{\mathcal{L}}$ represents execution with semantics $\mathcal{L}$, and

$$\phi_1, s_1 \parallel \phi_2, s_2 \overset{tr_1, tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \parallel \phi_4$$

means statement s_1 executes to completion from store ϕ_1, producing trace tr_1, resulting in store ϕ_3 and statement s_2 executes to completion from store ϕ_2, producing trace tr_2, resulting in store ϕ_4. Since $\mathcal{BL}$ is non-deterministic, our semantics are instrumented to produce a trace. Traces will be discussed in more detail shortly.

The rules for $\parallel$ in the $\mathcal{V}$ semantics and $\mathcal{A}$ semantics are given in fig. 3.1. Rule COMV is similar to the idea of product procedure from Symdiff [34]. In Symdiff, a product procedure executes the bodies of a pair of procedures on disjoint copies of the initial state. Symdiff uses mutual summaries to model the mutual behaviour of pairs of procedure *calls*. We use two predicates R_1 and R_2 in the rule for $\parallel$ in the $\mathcal{A}$ semantics to give an operational meaning to the mutual behaviour of pairs of procedure calls. Moreover, we also model the mutual behaviour of procedure *bodies*. We discuss this further, and define R_1 and R_2, as the chapter progresses.

$$\frac{\phi_1, s_1 \rightsquigarrow_{\mathcal{V}}^{tr_1} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow_{\mathcal{V}}^{tr_2} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow_{\mathcal{V}}^{tr_1, tr_2} \phi_3 \parallel \phi_4}\ \text{COMV}$$

$$\frac{R_1(tr_1, tr_2) \qquad R_2(tr_1, tr_2) \qquad \phi_1, s_1 \rightsquigarrow_{\mathcal{A}}^{tr_1} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow_{\mathcal{A}}^{tr_2} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow_{\mathcal{A}}^{tr_1, tr_2} \phi_3 \parallel \phi_4}\ \text{COMA}$$

Figure 3.1: Rule for executing procedure pairs in the $\mathcal{V}$ semantics and $\mathcal{A}$ semantics. The predicates R_1 and R_2 model the mutual behaviour of the executions by restricting what pairs of traces (tr_1, tr_2) are allowed, and are described in detail later on

Definition 3.1.1 defines procedure equivalence under semantics $\mathcal{L}$. Procedures are equivalent if executing their bodies, from isomorphic stores, results in isomorphic calling contexts. Executing `body f` means looking up and executing the statements that form the body of procedure `f`. The calling context of store ϕ is written ϕ^{ctx}, and is ϕ with the top stack frame popped[1]. Think of ϕ^{ctx} as the store that would result if procedure return was executed from store ϕ.

Definition 3.1.1 (Procedure equivalence)

$$\mathtt{f}_1 \lessapprox \mathtt{f}_2 \overset{def}{\iff} \forall \phi_{1\ldots4} : \phi_1 \approx \phi_2 \wedge \phi_1, \mathtt{body}\ \mathtt{f}_1 \parallel \phi_2, \mathtt{body}\ \mathtt{f}_2 \rightsquigarrow_{\mathcal{L}} \phi_3 \parallel \phi_4 \implies \phi_3^{ctx} \approx \phi_4^{ctx}$$

Defining procedure equivalence in terms of the calling context, ϕ^{ctx}, gives us contextual equivalence. This means that no code executed after either of the equivalent procedures returns can distinguish (by reading the heap or otherwise) which procedure executed.

To make the following exposition clearer, we split procedure call into two rules. One rule for `call` which pushes a new stack frame, and a second rule for `body` which looks up and executes the body of a procedure. We do this for reasons similar to why "body" is used by Godlin [22]. We will shortly detail the $\mathcal{V}$ semantics and $\mathcal{A}$ semantics rules for `call` and `body` which differ. In particular the $\mathcal{A}$ semantics abstracts procedure call and transforms the statements of the procedure body before executing them.

[1] Or just ϕ if there is no top frame.

3.2 Soundness of RIE

Theorem 3.2.1 states that proving $\mathtt{f}_1, \mathtt{f}_2$ equivalent under the approximate $\mathcal{A}$ semantics is sufficient to prove their equivalence under the precise $\mathcal{V}$ semantics, provided some other conditions are valid. The mapping $\mathcal{E}$ must be provided[2], and contains pairs of *allegedly* equivalent procedures. If all of the procedures in $\mathcal{E}$ are infact equivalent under the $\mathcal{A}$ semantics, and mutually terminate, then $\mathtt{f}_1$ and $\mathtt{f}_2$ are equivalent under the $\mathcal{V}$ semantics. Mutual termination means that both procedures terminate for the same set of initial stores[3], we discuss this and define $mt_{\mathcal{V}}(\mathtt{f}_3, \mathtt{f}_4)$ precisely in section 5.4.

Theorem 3.2.1 (RIE is sound)
Given a mapping between procedure names $\mathcal{E}$, and a pair of procedures $(\mathtt{f}_1, \mathtt{f}_2) \in \mathcal{E}$:
If

$$\forall (f_3, f_4) \in \mathcal{E} : \; mt_{\mathcal{R}}(f_3, f_4) \;\wedge\; f_3 \overset{\mathcal{A}}{\approx} f_4$$

Then

$$f_1 \overset{\mathcal{V}}{\approx} f_2$$

Full proof on page 102.

Proof Outline. By lemma 5.3.1 (Modular procedure equivalence), lemma 5.2.1 (Discard non equal isomorphisms), and lemma 5.1.1 (Replace equivalent calls). □

The $\mathcal{V}$ semantics and $\mathcal{A}$ semantics differ in three rules as summarised in table 3.1. These rules are discussed in detail in the rest of the chapter.

rule	$\mathcal{V}$	$\mathcal{A}$	
procedure call	precise	abstracted	fig. 3.5/fig. 3.5
procedure body	precise	rewritten according to $\mathcal{E}$	fig. 3.2/fig. 3.2
par composition	precise	only executions where isomorphic store parts are equal	fig. 3.1/fig. 3.6

Table 3.1: Differences between the $\mathcal{V}$ and $\mathcal{A}$ semantics.

3.3 Replacing procedures with equivalent procedures

Calls to equivalent procedures may have significantly different effects on the heap, even though those differences are never observable in the calling context[4]. They may allocate objects at different addresses (example in section 1.5), they may create different amounts and shapes of garbage, they may even make different arrangements of existing objects (example in section 1.7). Despite this we have proved that it is sound to ignore these differences in effect! Intuitively, *we may assume away unobservable differences in called procedure behaviour without affecting the equivalence of the calling procedures.* This amounts to replacing all calls to equivalent procedures by calls to the same procedure.

[2] $\mathcal{E}$ must map each procedure to at most one allegedly equivalent procedure.
[3] We could produce a total definition of procedure equivalence by including a notion of mutual termination [19, 25]. However, our tool 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. Infact, the replacement of equivalent procedures in the $\mathcal{A}$ semantics relies on such a notion of transitivity, see section 5.1.
[4] By definition equivalent procedures have equivalent observable effects.

When two procedures $(\mathtt{f}_1, \mathtt{f}_2)$ are equivalent we can replace a call to $\mathtt{f}_1$ with a call to $\mathtt{f}_2$ without affecting the observable meaning of the caller. For example, fig. 1.2 shows the bodies of the procedures in fig. 1.1 with all calls to `rcopy` replaced by calls to `lcopy`. The procedures in fig. 1.1 are equivalent if the bodies in fig. 1.2 are equivalent.

The $\mathcal{V}$ semantics rule for executing the body of a procedure is in fig. 3.2. Our use of a `body` rule is similar in spirit to a combination of how "body" is used in Godlin's `PROC-P-EQ` rule [22] and how "inline" is used in Hawblitzel's `MUTUALCHECK` procedure [25]. Separating the `body` and `call` rules allows us to independently vary our treatment of procedure call vs executing the body of a procedure, and also to avoid the possibility of circularity in the modular equivalence proofs.

In the $\mathcal{V}$ semantics, the body of function `f` is executed by looking up the statements of the body, using the global function $\mathcal{B}$, and executing them. In the $\mathcal{A}$ semantics the statements are transformed before execution, as described below. We assume that the procedure contracts have already been checked and are valid, so a procedure body may only be executed from a store that satisfies its precondition, written $(\phi_1, \_) \in \mathit{Con}(\mathtt{f})$ (see section 2.1).

$$\frac{(\phi_1, \_) \in \mathit{Con}(\mathtt{f}) \qquad \phi_1, \mathcal{B}(\mathtt{f}) \rightsquigarrow^{tr}_{\mathcal{V}} \phi_3}{\phi_1, \mathtt{body\ f} \rightsquigarrow^{tr}_{\mathcal{V}} \phi_3}\ \text{BODV}$$

$$\frac{(\phi_1, \_) \in \mathit{Con}(\mathtt{f}) \qquad \phi_1, \mathit{ren}_{\mathcal{E}}(\mathcal{B}(\mathtt{f})) \rightsquigarrow^{tr}_{\mathcal{A}} \phi_3}{\phi_1, \mathtt{body\ f} \rightsquigarrow^{tr}_{\mathcal{A}} \phi_3}\ \text{BODA}$$

Where $\mathit{ren}_{\mathcal{E}}(s) \stackrel{\text{def}}{=}$

$$\begin{cases} \mathtt{call\ f}_2(x_1, \ldots, x_n) & \text{if } s = \mathtt{call\ f}_1(x_1, \ldots, x_n) \wedge (\mathtt{f}_1, \mathtt{f}_2) \in \mathcal{E} \\ \mathit{ren}_{\mathcal{E}}(s_1); \mathit{ren}_{\mathcal{E}}(s_2) & \text{if } s = s_1; s_2 \\ \mathtt{if}(b)\{\mathit{ren}_{\mathcal{E}}(s_1)\} & \text{if } s = \mathtt{if}(b)\{s_1\} \\ s & \text{otherwise} \end{cases}$$

Figure 3.2: Execution of a procedure body in $\mathcal{V}$ semantics and $\mathcal{A}$ semantics

In the approximate $\mathcal{A}$ semantics the rule (fig. 3.2) for executing the body of a procedure first replaces calls to equivalent procedures by calls to the same procedure. The statements of the procedure body are looked up as usual, $\mathcal{B}(\mathtt{f})$, but before they are executed procedure calls are replaced syntactically according to the mapping $\mathcal{E}$. The syntax of $\mathcal{BL}$ is in fig. 2.1.

There are other ways that we could have formalised the concept of assuming away unobservable differences, however treating it as replacing equivalent procedure calls has some ancillary benefits. For example, if one were planning to apply an inference tool to the programs it may be helpful to replace equivalent procedure calls with calls to the same procedure first, and such like.

3.4 Assuming isomorphic $\implies$ equal

We prove that to establish procedure equivalence it is sufficient to verify only some specific pairs of executions. In particular a tool can verify only executions such that at some points where the executions are isomorphic, those isomorphisms are the identity isomorphism![5] Another way to think of this, is that

[5] As normal, the identity isomorphism is an isomorphism characterised by a bijection where each mapped address is mapped to itself.

when parts of two stores are isomorphic we can soundly assume those store parts are infact equal.

Many isomorphisms may be found between stores reached at different points in execution. But it is not always useful for the verifier to assume that a particular isomorphism is the identity isomorphism. So our methodology is parameterised by a strategy for selecting interesting isomorphisms.

As illustrated by the example in section 1.8 (page 20), it is important for the soundness of our methodology that equality is assumed in a manner which is consistent with the semantics of memory allocation. We require that all of the isomorphisms for which equality is assumed must be *compatible*. Recall that isomorphisms are compatible iff their union is an injection.

3.4.1 The approximation

In the approximate $\mathcal{A}$ semantics isomorphic store parts must be equal. This is achieved by means of a predicate in the antecedent of the procedure composition rule of the $\mathcal{A}$ semantics. This predicate corresponds to R_1 in fig. 3.1, which also shows the rule for procedure composition in the precise $\mathcal{V}$ semantics. The $\mathcal{A}$ semantics rule for procedure composition is shown again with R_1 defined in fig. 3.3. Rule COMA permits only executions with traces where function *misos* produces the identity isomorphism.

The semantics of $\mathcal{BL}$ produce a trace for each execution. A trace is an indexed sequence of the stores reached during an execution. Each element of the trace is a tuple containing the statement executed, the store before the statement was executed, and the store after the statement was executed. Consider fig. 3.4 (page 41) where two executions and their traces, tr_l and tr_r, are shown. The left trace is from an execution of the statements on the left of fig. 1.2, the right trace is from an execution of the statements on the right of fig. 1.2. Both traces have the same initial store. In fig. 3.4 (page 41), the third element of trace tr_l is (`lcopy(t.l, rl)`$, \phi_5, \phi_7$) and is written $tr_l[3]$. Procedure call executes in a single big step.

The restriction, $misos(tr_1, tr_2) \subseteq id$, that COMA imposes on the traces is described in definition 3.4.1 (Selected isomorphic points) and discussed in the remainder of this section.

$$\frac{misos(tr_1, tr_2) \subseteq id \qquad R_2(tr_1, tr_2) \qquad \phi_1, s_1 \rightsquigarrow^{tr_1}_{\mathcal{A}} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow^{tr_2}_{\mathcal{A}} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow^{tr_1, tr_2}_{\mathcal{A}} \phi_3 \parallel \phi_4)} \text{ COMA}$$

Figure 3.3: Procedure composition in the $\mathcal{A}$ semantics

3.4.2 Restricting executions to be related by the identity isomorphism

The restriction $misos(tr_1, tr_2) \subseteq id$, that COMA imposes on the traces acts as a filter, allowing only executions that are related by the identity isomorphism in the manner given by the function *misos*. This filter corresponds to assuming that isomorphic store parts are equal. There are zero or more isomorphisms between stores that arise in the traces tr_1, tr_2. The function *misos* returns the union of these arising isomorphisms. By restricting this union to be a subset of the identity bijection, COMA allows only executions with traces that are related by the identity isomorphism.

We relate pairs of traces at *points*. A point identifies an element in each trace, and also a part of the pre-store of each identified trace element. For example, in fig. 3.4, the point $(3, 4, [t.l, rl], [t.l, rl])$ identifies the third element of tr_l and the fourth element of tr_r. It also indicates the part of the store rooted at the paths $[t.l, rl]$ in both trace elements.

A pair of stores can be partially isomorphic. We write $\phi_1 \approx_{\pi}^{[x_1 \ldots x_n],[y_1 \ldots y_n]} \phi_2$ to mean that the part of the store ϕ_1 rooted at the variables $[x_1 \ldots x_n]$ is isomorphic with the part of store ϕ_2 rooted at the variables $[y_1 \ldots y_n]$. And that relation π characterises the isomorphism. We sometimes omit the characterising relation if we do not need to refer to it. In fig. 3.4, $\phi_5 \approx^{[t.l,rl],[t.l,rl]} \phi_8$.

The isomorphisms between the traces do not proceed monotonically. An earlier store in one trace may be related to a later store in the other trace. In this case $\phi_5 \approx^{[t.l,rl],[t.l,rl]} \phi_8$ and also $\phi_{11} \approx^{[t.r,rr],[t.r,rr]} \phi_4$. This presents some technical challenges for the proof of soundness of theorem 3.2.1, which are detailed in chapter 5.

In fig. 3.4, an interesting set of isomorphic points for our tool is:

$$\{(1, 1, [t, r], [t, r]), (6, 2, [t.r, rr], [t.r, rr]), (3, 4, [t.l, rl], [t.l, rl])\}$$

These points correspond to procedure entry and procedure call. The restriction $misos(tr_l, tr_r) \subseteq id$ will only allow executions that are related by the identity isomorphisms at these points. In this sense RIE underapproximates the full behaviour of $\mathcal{BL}$. Executions which allocate addresses in any way that does not produce the identity isomorphism at these points are discarded.

3.4.3 Selecting useful points

As illustrated by the example in section 1.8 (page 20), there are sometimes several incompatible ways in which particular procedure executions may be isomorphic. To account for this, the restriction *misos* is only partially defined. Thus, a tool is free to choose any function that complies with the requirements given in definition 3.4.1.

Definition 3.4.1 contains two main concepts. First, that the isomorphisms are compatible, i.e. that $\bigcup \Pi$ is an injection (definition 2.3.1). We write this as $com(\Pi)$. This ensures that the union produced by *misos* remains compatible with the semantics of `new`. In the soundness proof this arises as a requirement that the result of *misos* has an inverse, see chapter 5.

Second, that the function *misos* does not discriminate between isomorphic traces. Traces are isomorphic if their stores are pairwise isomorphic throughout the entire trace (see definition 4.4.1). The soundness of RIE depends on the rule COMA not discarding too many executions. A tool must examine at least one representative trace from each equivalence class of isomorphic traces.

Definition 3.4.1 (Selected isomorphic points) *Given* $tr_{1,2}$

- *The function misos is the union of the isomorphic points identified by a function mpts*

$$misos : (Trace \times Trace) \to \mathcal{P}(Val \times Val)$$
$$misos(tr_1, tr_2) \stackrel{def}{=} \bigcup \left\{ \pi \mid \exists (i, j, X, Y) \in mpts(tr_1, tr_2) \ \wedge \ tr_1[i] \approx_{\pi}^{X,Y} tr_2[j] \right\}$$

- *The function mpts gives a set of points given a pair of traces. Any tool using RIE must define a concrete mpts function with the following properties:*

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

The same set of points is produced for isomorphic traces

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

The traces are isomorphic at each of the points

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

And the union of all the isomorphisms is an injection

$$\mathit{com}(\mathit{misos}(tr_1, tr_2))$$

All the isomorphic points must be compatible with the initial isomorphism

$$\forall \pi : \mathit{fst}(tr_1) \approx_\pi \mathit{fst}(tr_2) \implies \mathit{com}(\pi, \mathit{misos}(tr_1, tr_2))$$

And if the initial stores are not isomorphic then the empty set of points is produced

$$\nexists \pi : \mathit{fst}(tr_1) \approx_\pi \mathit{fst}(tr_2) \implies \mathit{mpts}(tr_1, tr_2) = \emptyset$$

Where Lid^* is a sequence of local variable names, see section 2.1. The property $tr_1[i] \approx^{X,Y}_\pi tr_2[j]$ means that the part of the pre-store of trace element $tr_1[i]$ reachable from local variables X is isomorphic with the part of the pre-store of trace element $tr_2[j]$ reachable from local variables Y.

To illustrate this we consider again fig. 3.4. There are many points in traces tr_l, tr_r where an isomorphism exists between some part of the stores. For example, since the heap reachable from the stack variable t does not change throughout either execution, any pair of indices are isomorphic if we consider only the store reachable from t. There is an isomorphism at point $(4, 5, [t], [t])$ and also $(5, 4, [t], [t])$.

One could imagine a naïve choice of *mpts* that simply attempted to return all isomorphisms between the traces. However, this is not allowed by definition 3.4.1. For example, there is an isomorphism at point $(3, 2, [rl], [rr])$, the isomorphism maps the address allocated in `rl` by tr_l to the address allocated in `rr` by tr_r; call this mapping $\pi_1 = \{\phi_5(rl) \mapsto \phi_4(rr)\}$. There is also an isomorphism at point $(3, 4, [rl], [rl])$, the isomorphism maps the address allocated in `rl` by tr_l to the address allocated in `rl` by tr_r; call this mapping $\pi_2 = \{\phi_5(rl) \mapsto \phi_8(rl)\}$. Mappings π_1 and π_2 are *in*compatible. We know from the semantics of memory allocation in $\mathcal{BL}$ that $\phi_4(rr) \neq \phi_8(rl)$, and hence that $\pi_1 \cup \pi_2$ is *not* an injection. Rather, a tool must provide an *mpts* that choose a particular subset of the possible isomorphic points.

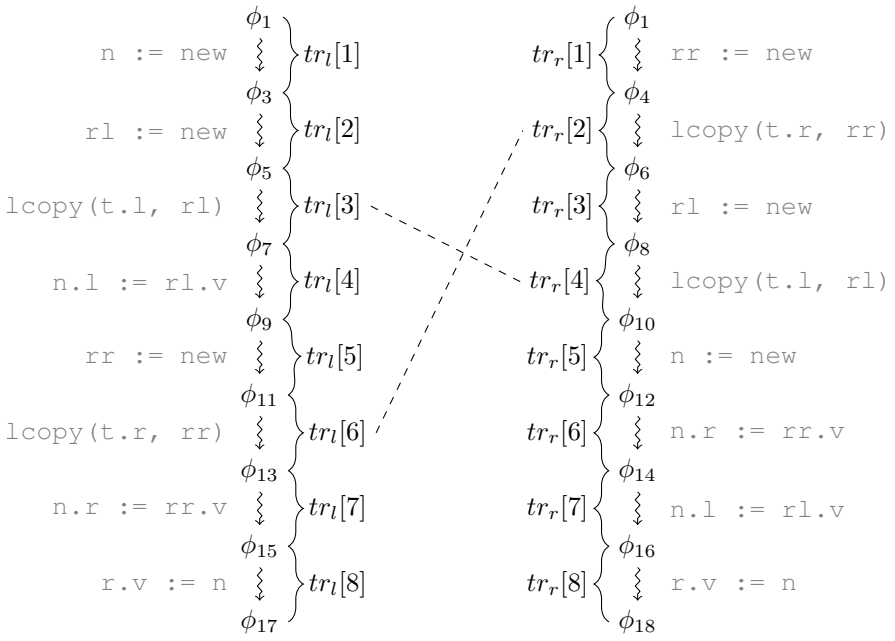

Figure 3.4: Traces from executing the example in Figure 1.2. The left hand trace is an execution of `lcopy`, the right hand trace is an execution of `rcopy`. There is an isomorphism between the parts of the state reachable from the call parameters at the points $(3, 4, [t.l, rl], [t.l, rl])$ and also at $(6, 2, [t.r, rr], [t.r, rr])$, as indicated by the dashed lines. Note that there are also many other part isomorphisms, such as $(3, 2, [rl], [rr])$ — which is that store parts reachable from the local variable rl of ϕ_5 and rr of ϕ_4. Not all of the isomorphisms are compatible.

3.5 Modular Proof of Procedure Equivalence

Program verification is often undertaken in a modular fashion. This is done to improve the scalability of verification tools. Procedures are verified independently, meaning that each procedure is verified exactly once. A non-modular analysis would need to verify procedures within particular calling contexts, leading to procedures being verified many times each.

Modular verification of procedure equivalence has some additional challenges over standard single-procedure modular verification. This was identified as early as 1972 [17]. Instead of considering equivalence of whole programs, we pair up the procedures and consider their equivalence modularly. Thus we need the programs to have a similar procedural decomposition in order to compare them [17]. We adapt the approaches of Godlin [22], which uses an inference rule for modular procedure equivalence that is similar to Hoare's rule for recursive procedure invocation [28], and Hawblitzel [25]. Benton [12] recognises the role of bijections between heap regions in reasoning about the equivalence of dynamically allocating programs.

In the approximate $\mathcal{A}$ semantics procedure call is treated abstractly, as is typical in modular verification. In single program verification, procedure calls are abstracted by their contracts. However, in program equivalence verification we would like to use facts about the mutual or relative behaviour of procedure calls across executions of the pair of procedures being verified. Furthermore, in RIE we need to introduce facts about memory allocation that concern the entire execution trace of the procedure pair.

We formalise this mutual behaviour of procedure bodies by defining the predicate R_2 from fig. 3.1. This predicate further restricts the allowed traces in the $\mathcal{A}$ semantics procedure composition rule COMA, which is shown in full in fig. 3.6. Predicate R_2 and the rules for procedure call are discussed below.

$$\frac{\mathit{mkframe}(\phi_1, \mathtt{f}, x_1 \dots x_n), \mathtt{body\ f} \rightsquigarrow_{\mathcal{V}}^{tr} \phi_3}{\phi_1, \mathtt{call\ f}(x_1 \dots x_n) \rightsquigarrow_{\mathcal{V}}^{(\mathtt{call\ f}(x_1 \dots x_n), \phi_1, \phi_3^{ctx})} \phi_3^{ctx}}\ \text{CALLV}$$

$$\frac{\begin{array}{c}\mathit{dom}(h_2) \supseteq \mathit{dom}(\phi_1^h) \qquad \phi_2 = (\phi_1^{\tilde{\sigma}} \cdot \sigma, h_2) \\ (\mathit{mkframe}(\phi_1, \mathtt{f}, x_1 \dots x_n), \phi_2) \in \mathit{Con}(\mathtt{f})\end{array}}{\phi_1, \mathtt{call\ f}(x_1 \dots x_n) \rightsquigarrow_{\mathcal{A}}^{(\mathtt{call\ f}(x_1 \dots x_n), \phi_1, \phi_2^{ctx})} \phi_2^{ctx}}\ \text{CALLA}$$

Figure 3.5: Procedure call in the $\mathcal{V}$ semantics and $\mathcal{A}$ semantics

In the $\mathcal{V}$ semantics, a procedure call is executed by pushing a new stack frame containing the call parameters and then executing the body of the procedure (fig. 3.5). The $\mathcal{A}$ semantics abstracts procedure behaviour by its specification (fig. 3.5). Calling a procedure `f` results in a non-deterministic new heap h, and stack frame σ, that satisfy the contract of `f`. The stack of store ϕ_1 is written $\phi_1^{\tilde{\sigma}}$.

One can think of the rule COMA as enforcing consistent mutual behaviour of these abstracted but equivalent procedure calls across both procedure body executions. The rule allows pairs of executions only if every pair of isomorphic calls to a procedure have isomorphic effects. Isomorphic calls are found from the trace, where $(\mathtt{call\ f}(x_1 \dots x_n), \phi_1, \phi_3) \in tr_1$ means that procedure `f` was called in trace tr_1 with parameters $(x_1 \dots x_n)$ from store ϕ_1 resulting in store ϕ_3.

In the example, fig. 3.4, since $\phi_5 \approx^{[t.l,rl],[t.l,rl]} \phi_8$, rule COMA enforces that all admitted traces have an isomorphism between $\mathit{effect}(\phi_5, \phi_7)$ and $\mathit{effect}(\phi_8, \phi_9)$.

$$\frac{misos(tr_1, tr_2) \subseteq id \qquad \mathcal{U}(tr_1, tr_2) \qquad \phi_1, s_1 \rightsquigarrow_{\mathcal{A}}^{tr_1} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow_{\mathcal{A}}^{tr_2} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow_{\mathcal{A}}^{tr_1, tr_2} \phi_3 \parallel \phi_4} \text{COMA}$$

Where $\mathcal{U}(tr_1, tr_2) \stackrel{\text{def}}{\iff}$

$$\begin{aligned}
&\forall \pi_1, \mathtt{f}, \phi_{1\ldots4} : \\
&\quad (\mathtt{call\ f}(x_1 \ldots x_n), \phi_1, \phi_3) \in tr_1 \ \wedge \\
&\quad (\mathtt{call\ f}(y_1 \ldots y_n), \phi_2, \phi_4) \in tr_2 \ \wedge \\
&\quad \phi_1 \approx_{\pi_1}^{\{x_1 \ldots x_n\}, \{y_1 \ldots y_n\}} \phi_2 \\
&\quad \Longrightarrow \\
&\quad \exists \pi_2 : \ com(\pi_1, \pi_2) \ \wedge \ effect(\phi_1, \phi_3) \approx_{\pi_2} effect(\phi_2, \phi_4)
\end{aligned}$$

Figure 3.6: Procedure composition in the $\mathcal{A}$ semantics

3.6 Summary

In our tool, we use an SMT solver, and Symdiff like heap encoding, to automatically modularly prove equivalence of recursive procedures, see chapter 6. A naïve encoding of procedure equivalence involves asking the SMT solver to prove an isomorphism by providing a witness to the bijection between stores. In general SMT solvers are good at checking that certain properties hold, but are less good at finding witnesses that have some property. Thus, the solver is almost never able to do this, resulting in solver timeouts. Instead, we avoid needing to find such a bijection since:

- we often do not need to produce the bijection. Rather it is enough to prove that, assuming the initial isomorphism, the statements preserve isomorphism
- several sound underapproximations of program behaviour allow us to use the identity bijection as the initial isomorphism and after procedure call

This approach corresponds to asking the verifier to verify an assertion about the final states of the procedures under a semantics restricted to carefully selected executions that are easier for the verifier. We have proved that these approximations are sound, in the sense that **whenever the verification succeeds the statements are indeed equivalent**.

In the next chapters we give the full details of $\mathcal{BL}$ and its properties, and describe the proof of theorem 3.2.1. Some readers may wish to skip forward to chapter 6, which describes the implementation of RIE in our tool.

Chapter 4

Properties of Isomorphism

Equivalence of procedures in our simple language $\mathcal{BL}$ is defined in terms of isomorphism $\approx$, see definition 3.1.1 (page 35). The choice of relation $\approx$ is tightly constrained by the necessity for $\approx$ to be both useful to programmers and still allow us to prove theorem 3.2.1 (RIE is sound). The relation $\approx$ must have the following properties:

- $\mathcal{BL}$ must be closed under the relation $\approx$ (section 4.1)
- The relation $\approx$ must preserve the meaning of all assertions (section 4.2).
- The relation $\approx$ must be an equivalence relation (section 4.3)

This chapter proves $\approx$ has the above properties.

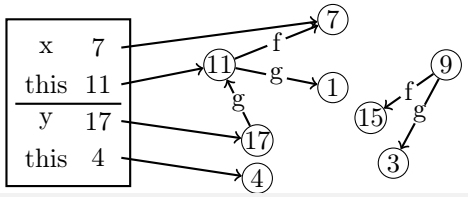

Figure 4.1: A store with two stack frames, five reachable objects and three unreachable objects

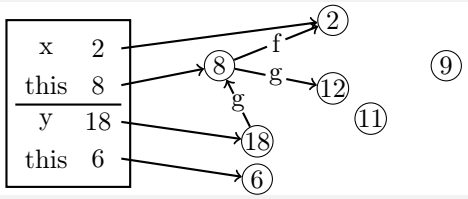

Figure 4.2: A store with two stack frames, five reachable objects and two unreachable objects. The store in this figure is isomorphic with the store in fig. 4.1

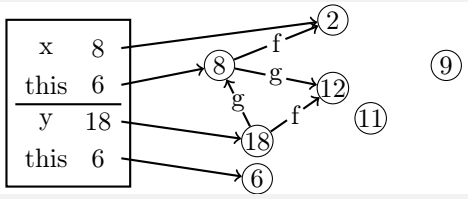

Figure 4.3: A store with two stack frames and five reachable objects. The store in this figure is not isomorphic with the store in fig. 4.1 nor the store in fig. 4.2

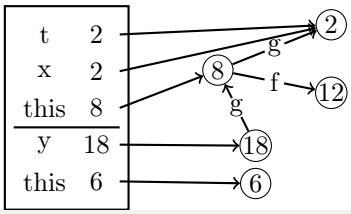

Figure 4.4: A store similar to fig. 4.2 but with the fields of object 8 swapped. Garbage has been omitted.

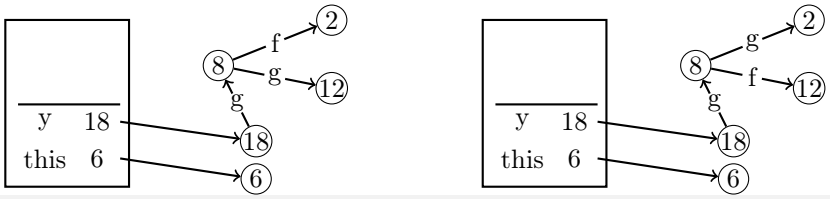

Figure 4.5: Calling context shape after executing `skip` on the left, and after executing `swap` on the right. The calling contexts are isomorphic, even though the fields of object 8 have been swapped.

4.1 Closure under isomorphism

Our definition of procedure equivalence is useful because $\mathcal{BL}$ is closed under $\approx$ (lemma 4.1.3). Executing a statement s to completion from stores related by $\approx$ always leads to stores related by $\approx$, via executions with isomorphic effects. Therefore no statement can distinguish between stores related by $\approx$.

Furthermore, given isomorphic initial stores related by π_1, we can construct executions of s such that the final stores are related by any sufficiently large injection π_3, provided that π_3 conforms to certain restrictions. The restrictions are that: $\pi_3 \cup \pi_1$ is an injection, written $com(\pi_1, \pi_3)$; and π_3 is consistent with any initially allocated addresses. This is because $\mathcal{BL}$ has the property that the `new` instruction can non-deterministically pick *any* unallocated address. We call such an injection π_3 an *alternative allocation strategy.*

Altogether lemma 4.1.3 establishes that:

- whenever alternative allocation strategies are identified, there exist executions corresponding to those strategies
- an execution from a particular store is representative of all executions from all isomorphic starting stores

The above properties are crucial for the proof of theorem 3.2.1 (RIE is sound). Soundness requires that the approximation include sufficient executions — particularly that those executions are representative of the full set of possible executions.

Definition 4.1.1 (Isomorphic Executions)
Executions $\phi_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \phi_3$ *and* $\phi_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \phi_4$ *are isomorphic*[a] *iff* $\exists \pi_{1,2}$:

$$\phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ tr_1 \approx tr_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ effect(\phi_1, \phi_3) \approx_{\pi_2} effect(\phi_2, \phi_4)$$

[a] see definition 4.4.1

Definition 4.1.2 (Statement closed under isomorphic)
A statement s is closed under isomorphism iff for every execution $\phi_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \phi_3$*, store* ϕ_2*, and injection* π_1 *if* $\phi_1 \approx_{\pi_1} \phi_2$ *then*

- *there exists an execution* $\phi_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \phi_4$
- *if* $\mathcal{L} \neq \mathcal{A}$*, every execution* $\phi_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \phi_4$ *is isomorphic to* $\phi_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \phi_3$

Lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism)

All statements are closed under isomorphism.

Moreover, given alternative allocation strategy π_2 *such that*

$$com(\pi_1, \pi_2) \ \wedge\ \forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h)$$

There exists an isomorphic execution $\phi_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \phi_4$ *such that*

$$tr_1 \approx_{\pi_2} tr_2 \ \wedge\ \forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h)$$

Full proof on page 103.

Proof Outline. The proof of lemma 4.1.3 proceeds by induction on the structure of the derivation of an execution. The base cases of the induction correspond to the atomic instructions of $\mathcal{BL}$. Sequential composition and procedure call are the inductive cases.

The base cases of the proof corresponding to instructions *STORE* (lemma 4.1.4), *ASSIGN* (lemma 4.1.7), *NEW* (lemma 4.1.9) and the inductive case corresponding to procedure call, are of particular interest because they potentially modify the store.

The inductive case TRANS goes by noting that executions only expand the set of reachable addresses by allocation (lemma 2.4.8 (Reach gets smaller)) — i.e. nothing that was garbage becomes reachable again. And also that effects are confined to the reachable part of the store (lemma 2.4.10 (Effects are reachable)). We can safely sequentially compose executions, because anything that the second execution works on is either new or must have been reachable in its initial store and therefore is in any isomorphism involving that store. □

The rest of this section details the main lemmas used in proving the interesting cases of the above proof.

4.1.1 STORE

The *STORE* instruction is closed under isomorphism (lemma 4.1.4).

Lemma 4.1.4 (STORE Closed)

$$\forall e_{1,2}, f : e_1.f := e_2 \text{ is closed under isomorphism}$$

Full proof on page 110.

A single heap update can affect an arbitrary number of paths through the heap, and create an arbitrary amount of garbage. So in order to prove that the *STORE* instruction is closed under isomorphism we observe that there is a close relationship between the notions of heap reachability and store isomorphism. The domain of an isomorphism from the store ϕ is exactly the set of addresses reachable (definition 2.4.1) from the stack variables of ϕ (lemma 2.4.11), and expressions always evaluate to a reachable address (lemma 2.4.3). Therefore, executing a store instruction never increases the set of addresses that are reachable. Only addresses that were already reachable can be written to the heap by STORE.

Lemma 4.1.5 (STORE reduces *reach*)

$$\forall \phi_{1,3}, e_{1,2}, f : \phi_1, e_1.f := e_2 \hookrightarrow \phi_3 \implies reach(\phi_3) \subseteq reach(\phi_1)$$

Full proof on page 114.

It remains to show that the executions of the store instruction preserve isomorphism. This is proved by induction on the definition of $\approx$. The critical step is to observe that, given $\phi_1 \approx_\pi \phi_2$, an expression that evaluates to a value v in store ϕ_1, will evaluate to the value $\pi(v)$ in store ϕ_2.

Lemma 4.1.6 (Isomorphism preserves expression)

$$\forall \phi_{1,2}, \pi, v, e : \phi_1 \approx_\pi \phi_2 \wedge v = [\![e]\!]_{\phi_1} \implies \pi(v) = [\![e]\!]_{\phi_2}$$

Full proof on page 115.

4.1.2 ASSIGN

Lemma 4.1.7 states that the *ASSIGN* instruction is closed under isomorphism. A single stack update can affect an arbitrary number of paths through the heap.

Lemma 4.1.7 (ASSIGN Closed)

$$\forall x, e : x := e \text{ is closed under isomorphism}$$

Full proof on page 118.

The proof of lemma 4.1.7 proceeds similarly to lemma 4.1.4 (STORE Closed). Again, the key observation is that isomorphism is closely related to heap reachability and ASSIGN can only reduce the set of reachable addresses.

Lemma 4.1.8 (ASSIGN reduces *reach*)

$$\forall \phi_{1,3}, e, x : \phi_1, x := e \hookrightarrow \phi_3 \implies \mathit{reach}(\phi_3) \subseteq \mathit{reach}(\phi_1)$$

Full proof on page 122.

4.1.3 NEW

Lemma 4.1.9 states that NEW is closed under isomorphism, and furthermore that we can pick the allocated address to maintain compatibility with some given bijection. The lemma means that *any* unallocated address can be used to construct the alternate execution (lemma 2.3.3 (Can add compatible element to injection) and lemma 2.3.4 (Can add disjoint element to injection)).

Lemma 4.1.9 (NEW Closed)

$$\forall x : x := \texttt{new}() \text{ is closed under isomorphism}$$

Full proof on page 123.

The proof is similar to the proof of lemma 4.1.7 (ASSIGN Closed), but we also have to show that it is possible to pick an appropriate new address with which to construct the alternate execution. It is possible to pick an appropriate address because lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism) requires that the alternate allocation strategy is consistent with the initially allocated addresses. Specifically, that the alternate allocation strategy must only map an allocated address to an allocated address and never an allocated address to an unallocated address (and vice vesa).

It is also important that the new instruction does not synthesise or reuse any existing address. If the new instruction did reuse an address that was previously garbage, then this would place further unhelpful restrictions on the choice of isomorphic points (see definition 3.4.1).

Lemma 4.1.10 (NEW doesn't synthesise existing addresses)

$$\begin{aligned}&\forall \phi_{1,3}, x, a : \\ &\quad \phi_1, x := \mathit{new} \hookrightarrow \phi_3 \ \wedge \\ &\qquad \implies \mathit{reach}(\phi_3) \subseteq \mathit{reach}(\phi_1) \cup \{a, \mathtt{null}\}\end{aligned}$$

Full proof on page 131.

Proof Outline. Note that $\phi_3 = (\phi^\sigma[x \mapsto a], \mathtt{alloc}(\phi^h, a))$, and that all fields of the new object have the value `null` □

4.2 Non-vacuity of isomorphism

There is at least one other relation under which executing a statement of $\mathcal{BL}$ from any pair in the relation leads to another pair in the relation — the trivial relation that relates every store to every store (*Store* × *Store*). In order for our notion of procedure equivalence to be useful for programmers we need to avoid any such trivial relations. Therefore we insist that the isomorphism $\approx$ must preserve the meaning of all assertions (lemma 4.2.1).

Lemma 4.2.1 (Isomorphism is assertion preserving)

$$\forall \phi_1, \phi_2, b : \ \phi_1 \approx \phi_2 \implies (\phi_1 \vDash b \iff \phi_2 \vDash b)$$

Full proof on page 132.

Proof Outline. By induction on the structure of the assertion. No expression is sensitive to the actual values of addresses, apart from `null` which is always mapped to itself. □

4.3 Isomorphism is an equivalence relation

The proof of theorem 3.2.1 (RIE is sound) depends on the fact that the relation $\approx$ is an equivalence relation. In particular, it follows from transitivity of $\approx$ that establishing isomorphism of two stores is sufficient to establish isomorphism between all of their equivalence class of isomorphic stores. We may think of each equivalence class as representing all the alternative, but indistinguishable, allocation strategies (modulo reachability). It is also important that equal stores are isomorphic (reflexivity), since we choose to use the pairs of equal stores to represent the pairs of isomorphic stores.

Lemma 4.3.1 (Isomorphism is an equivalence relation)

It is transitive

$$\forall \phi_{1\ldots3}, \pi_{1,2} : \ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge \ \phi_2 \approx_{\pi_2} \phi_3 \implies \phi_1 \approx_{\pi_1 \cdot \pi_2} \phi_3$$

It is reflexive

$$\forall \phi : \ \phi \approx_{id} \phi$$

It is symmetric

$$\forall \phi_{1,2}, \pi : \ \phi_1 \approx_{\pi} \phi_2 \implies \phi_2 \approx_{\pi^{-1}} \phi_1$$

Full proof on page 133.

4.4 Trace Isomorphism

In order to distinguish between non-deterministic executions the semantics are instrumented to produce a trace for each execution. Trace isomorphism is written $tr_1 \approx tr_2$ and defined in the natural way. Traces are isomorphic if they are the same length and their elements are pairwise isomorphic. Sometimes we want to restrict the trace isomorphism to be compatible with some mapping π: then we write $tr_1 \approx_\pi tr_2$. In section 2.1 a trace is defined as a sequence of $Stmt \times Store \times Store$. Given the ith element of trace tr is $tr[i] = (x := y, \phi_1, \phi_2)$, we write $tr[i]{\downarrow_2}$ to mean ϕ_1 and $tr[i]{\downarrow_3}$ to mean ϕ_2.

Definition 4.4.1 (Isomorphism of Traces)

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

4.5 Discussion

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

$$\phi_1 \approx \phi_2 \iff^{\text{def}} \phi_1 = \phi_2 = \phi_\emptyset \ \vee\ \left(\exists s_a, \phi_{3,4} : \phi_3 \approx \phi_4 \ \wedge\ \phi_3, s_a \rightsquigarrow_{\mathcal{C}} \phi_1 \ \wedge\ \phi_4, s_a \rightsquigarrow_{\mathcal{C}} \phi_2\right)$$

where $\phi_\emptyset$ is the empty store. That is, as a smallest relation closed under the atomic operations of the semantics. However, even though the semantics is naturally closed under such a definition, 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. It is interesting to consider what assertion language would be preserved for any particular semantics given such a definition.

Another way to think of closure under isomorphism is that $\mathcal{BL}$ is neither sensitive to the actual values of heap addresses nor garbage. Many industrial languages (such as Java, Python, C$^\sharp$, etc) do in fact contain features that are sensitive to the actual values of heap addresses or order of address allocation. Still, such address sensitivity is typically not central to the language and it is often not necessary to use such features.

Chapter 5

RIE Soundness

RIE is sound because whenever procedures are verified as equivalent under the approximate $\mathcal{A}$ semantics they are also equivalent under the precise semantics of $\mathcal{BL}$. This chapter outlines the proof of the soundness theorem 3.2.1.

For expository reasons, our proof of theorem 3.2.1 proceeds in stages via several intermediate semantics, showing preservation of procedure equivalence at each stage. The semantics we use are in the set $\{\mathcal{C}, \mathcal{V}, \mathcal{R}, \mathcal{K}, \mathcal{A}\}$, each element of which indicates a variant operational semantics of $\mathcal{BL}$. We depict the relationship between these semantics in fig. 5.1 and discuss in detail as this chapter progresses. Each semantics varies from the other in one or two rules.

semantics	purpose
$\mathcal{C}$	precise semantics of $\mathcal{BL}$
$\mathcal{V}$	as above but where all procedure contracts are known to be valid
$\mathcal{R}$	as above but calls to equivalent procedures replaced by calls to the same procedure
$\mathcal{K}$	as above plus selected isomorphic store parts taken to be equal
$\mathcal{A}$	as above plus procedures treated modularly

Table 5.1: Characteristics of the $\{\mathcal{C}, \mathcal{V}, \mathcal{R}, \mathcal{K}, \mathcal{A}\}$ semantics.

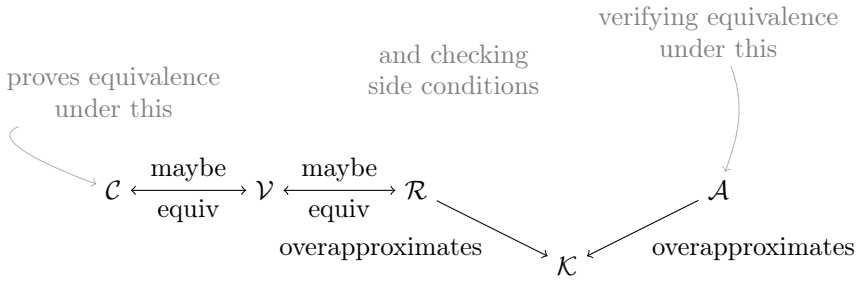

Figure 5.1: The semantics of $\mathcal{BL}$. Semantics $\mathcal{V}$ and $\mathcal{C}$ are equivalent if all of the procedure contracts are valid. Semantics $\mathcal{R}$ is observably equivalent to $\mathcal{V}$ if all the procedures in a mapping $\mathcal{E}$ are indeed equivalent (section 3.3). Semantics $\mathcal{K}$ soundly underapproximates $\mathcal{R}$ by requiring some isomorphic stores to be equal (section 3.4). Semantics $\mathcal{A}$ overapproximates $\mathcal{K}$ by treating procedure call abstractly allowing for modular verification (section 3.5).

5.1 Replacing equivalent calls

It is sound to replace a call to a procedure by a call to an equivalent procedure. If we can prove equivalence and mutual termination (section 5.4) of all the procedure pairs in a partial mapping $\mathcal{E}$ under semantics $\mathcal{R}$, then we may conclude that any procedure pair from $\mathcal{E}$ contains equivalent procedures (Lemma 5.1.1). Please see section 3.3 for more discussion and motivation.

In the $\mathcal{R}$ semantics calls to each procedure are replaced according to the mapping $\mathcal{E}$, e.g. each call to $\mathtt{f}_1$ by a call to $\mathcal{E}(\mathtt{f}_1)$. This is modelled by replacing the $\mathcal{V}$ semantics rule BODV by the $\mathcal{R}$ semantics rule BODA, fig. 2.2. It is perhaps useful to think of this in terms of extensionality: that if things differ only in ways which are unobservable we may soundly consider them the same. Here we apply extensionality to equivalent procedures, and in section 5.2 we apply extensionality to isomorphic stores. Procedure equivalence, $\mathtt{f}_3 \lessapprox \mathtt{f}_4$, is defined in definition 3.1.1 (page 35). Mutual termination, $mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4)$, is defined below in definition 5.4.1 (page 55).

Lemma 5.1.1 (Replace equivalent calls)
Given a mapping between procedure names $\mathcal{E}$:

$$\forall(f_1, f_2) \in \mathcal{E} : (\forall(f_3, f_4) \in \mathcal{E} : f_3 \lessapprox f_4 \wedge mt_{\mathcal{R}}(f_3, f_4)) \implies f_1 \lessapprox f_2$$

Full proof on page 139.

Proof Outline. By induction over the derivation of the execution. Replacing a procedure call in the execution with a call to an equivalent procedure results in a new execution with an isomorphic final state. Then by transitivity of isomorphism.

Two auxiliary lemmas are used. Firstly: whenever a statement is executed to completion under both semantics, if the initial stores are isomorphic then the final stores will also be isomorphic. Secondly: whenever there is a terminating execution from an initial store under the $\mathcal{V}$ semantics, there is a terminating execution from every isomorphic initial store under the $\mathcal{R}$ semantics. □

There is some subtlety here. We seem to first hypothesise that the procedures in $\mathcal{E}$ are equivalent, and use this assumption to justify replacing procedure calls according to the mapping. Then we apparently use those modified procedures to justify the hypothesis! The potential for circularity demanded some care when deciding how to set up our formal system and proof.

The potential circularity is avoided by using two different semantics, $\mathcal{R}$ semantics and $\mathcal{V}$ semantics, and through the inductive structure of our argument. Only terminating executions are relevant so there is a well founded tree of procedure calls in any given execution.

First, the base cases. These correspond to procedure executions which return without reaching a procedure call. Since no procedure call is reached, no replacement of procedure calls according to $\mathcal{E}$ is relevant and the $\mathcal{R}$ semantics and $\mathcal{V}$ semantics are otherwise identical. Hence no justification for replacing procedure calls is needed for these base cases, and verifying procedure equivalence under $\mathcal{R}$ semantics is the same as verifying it under $\mathcal{V}$ semantics.

Now the inductive step. Procedure equivalence is verified by a tool under the $\mathcal{R}$ semantics for all procedures pairs in $\mathcal{E}$ and from all permissible initial stores. Thus, for any procedure execution in the $\mathcal{V}$ semantics call tree that will be replaced, say a call to procedure $p \in \mathcal{E}$ from some store ϕ, we also verify equivalence of the pair $(p, \mathcal{E}(p))$ from initial store ϕ under $\mathcal{R}$ semantics. Our justification for the replacement of procedure calls is checked by verifying equivalence of the pairs in $\mathcal{E}$ at all lower depths in

the call tree. Since all relevant chains of procedure calls eventually reach a base case, where no renaming occurs, then all replacements are justified.

For a concrete example of why termination is important, consider fig. 5.2. Procedures `g` and `g'` are not equivalent. Procedure `g'(x)` terminates only from stores where $x = \texttt{null}$, and `g(x)` also terminates in this case. But when $x = \texttt{null}$ procedure `g(x)` has an effect on the heap whereas procedure `g'(x)` does not. Procedures `f` and `f'` are trivially equivalent since procedure `f'` terminates for no initial stores.

Now consider executing under $\mathcal{R}$ semantics with $\mathcal{E} = \{(g, g'), (f, f'), (f', f'), (g', g')\}$ but for the moment *ignoring the requirement for mutual termination*, the procedures are rewritten as illustrated by fig. 5.3. Under the $\mathcal{R}$ semantics procedures `f` and `f'` remain equivalent. However, under the $\mathcal{R}$ semantics procedures `g` and `g'` become equivalent! When the call to `f` in procedure `g` is replaced by a call to procedure `f'`, we find that procedure `g` no longer terminates for any initial store. The mutual termination requirement in lemma 5.1.1 excludes applying RIE in such cases.

```
1  f(x) {
2     if(x) {
3       f(x.n);
4     }
5  }
6
7  g(x) {
8     f(x)
9     x.v := new;
10 }
```

```
1  f'(x) {
2     f'(x)
3  }
4
5  g'(x) {
6     if(x) {
7        f'(x)
8     }
9  }
```

Figure 5.2: Non terminating procedures

```
11 f(x) {
12    if(x) {
13      f'(x.n);
14    }
15 }
16
17 g(x) {
18    f'(x)
19    x.v := new;
20 }
```

```
10 f'(x) {
11    f'(x)
12 }
13
14 g'(x) {
15    if(x) {
16       f'(x)
17    }
18 }
```

Figure 5.3: Non terminating procedures under $\mathcal{R}$ semantics

5.2 Angelic allocation

Lemma 5.2.1 states that in order to prove that procedures are equivalent under the $\mathcal{R}$ semantics it is sufficient to prove that they are equivalent under the $\mathcal{K}$ semantics. The $\mathcal{K}$ semantics differs from the $\mathcal{R}$ semantics in only one rule: COMV is replaced by COMK. The rule COMK admits only executions that are related by the identity isomorphism at the points identified by the strategy *misos*, see section 3.4.

$\mathcal{K}$ semantics can be thought of as refining the non-deterministic behaviour exhibited by the $\mathcal{R}$ semantics in the way that is most useful for verifying procedure equivalence. This is similar to Hoare's angelic non-determinism for termination, but we apply the idea to memory allocation. Here, the predicate $misos(tr_1, tr_2) \subseteq id$ constrains the non-determinism of allocation throughout the entire program.

$$\frac{misos(tr_1, tr_2) \subseteq id \qquad \phi_1, s_1 \rightsquigarrow_{\mathcal{K}}^{tr_1} \phi_3 \qquad \phi_2, s_2 \rightsquigarrow_{\mathcal{K}}^{tr_2} \phi_4}{\phi_1, s_1 \parallel \phi_2, s_2 \rightsquigarrow_{\mathcal{K}}^{tr_1, tr_2} \phi_3 \parallel \phi_4}\ \text{COMK}$$

Figure 5.4: Procedure composition in the $\mathcal{K}$ semantics

Lemma 5.2.1 (Discard non equal isomorphisms)

$$\forall f_{1,2} : f_1 \overset{\mathcal{K}}{\approx} f_2 \implies f_1 \overset{\mathcal{R}}{\approx} f_2$$

Full proof on page 144.

Proof Outline. Given any two executions, tr_1, tr_2 related by an isomorphism at some points, by lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism), it is possible to pick a third isomorphic execution tr_3 which is related to tr_1 by the inverse isomorphism at those points. By transitivity of isomorphism we find that executions tr_3 and tr_2 are related by the identity at those points (an isomorphism composed with its inverse is the identity isomorphism). Since the $\mathcal{R}$ semantics and $\mathcal{K}$ semantics are identical other than their treatment of $\parallel$ we can easily show, by induction over the derivation of each execution, that tr_3, tr_2 are executions under both semantics. □

5.3 Abstraction

The proof obligation $\texttt{f}_1 \overset{\mathcal{A}}{\approx} \texttt{f}_2$ is discharged by our verification tool. Given this, lemma 5.3.1 allows us to conclude that the procedures $\texttt{f}_1$ and $\texttt{f}_2$ are equivalent under the $\mathcal{K}$ semantics also.

The $\mathcal{A}$ semantics differs from the $\mathcal{K}$ semantics in the rules for procedure call: fig. 3.5 (Procedure call in the $\mathcal{V}$ semantics and $\mathcal{A}$ semantics) vs fig. 5.5 (Procedure call in the $\mathcal{K}$ semantics); and the rules for procedure composition: fig. 3.6 (Procedure composition in the $\mathcal{A}$ semantics) vs fig. 5.4 (Procedure composition in the $\mathcal{K}$ semantics).

$$\frac{mkframe(\phi_1, \texttt{f}, x_1 \dots x_n), \texttt{body f} \rightsquigarrow_{\mathcal{L}}^{tr} \phi_3 \qquad \mathcal{L} \neq \mathcal{A}}{\phi_1, \texttt{call f}(x_1 \dots x_n) \rightsquigarrow_{\mathcal{L}}^{(\texttt{call f}(x_1 \dots x_n), \phi_1, \phi_3^{ctx})} \phi_3^{ctx}}\ \text{CALLV}$$

Figure 5.5: Procedure call in the $\mathcal{K}$ semantics

Lemma 5.3.1 (Modular procedure equivalence)
Given a partial mapping between procedure names $\mathcal{E}$,

$$\forall f_{1,2} : f_1 \overset{\mathcal{A}}{\approx} f_2 \implies f_1 \overset{\mathcal{K}}{\approx} f_2$$

Full proof on page 147.

Proof Outline. Straightforwardly because $\forall s_1, \phi_{1,3}, tr_1 : \phi_1, s_1 \rightsquigarrow_{\mathcal{K}}^{tr_1} \phi_3 \implies \phi_1, s_1 \rightsquigarrow_{\mathcal{A}}^{tr_1} \phi_3$ (lemma A.1.3). □

5.4 Termination

Mutual termination is known to be important if a transitive procedure equivalence relation is required [23, 26]. As shown in section 5.1, a transitive procedure equivalence relation is necessary in order to exclude trivial replacement of procedures by non-terminating ones. Theorem 3.2.1, obliges a tool using RIE to establish mutual termination under the $\mathcal{R}$ semantics.

Definition 5.4.1 defines mutual termination of procedures. Procedures mutually terminate iff whenever one procedure has a terminating execution from some initial store, the other procedure also has a terminating execution from every equivalent initial store.

Definition 5.4.1 (Mutual Termination) *Where*

$$
\begin{aligned}
mt_{\mathcal{L}}(\mathtt{f}_3, \mathtt{f}_4) \overset{def}{\iff} \forall \phi_{1\ldots3} : \phi_1 \approx \phi_2 \implies \\
\Big(\phi_1, \mathtt{body}\ \mathtt{f}_3 \rightsquigarrow_{\mathcal{L}} \phi_3 \implies \exists \phi_4 : \phi_2, \mathtt{body}\ \mathtt{f}_4 \rightsquigarrow_{\mathcal{L}} \phi_4\Big) \wedge \\
\Big(\phi_1, \mathtt{body}\ \mathtt{f}_4 \rightsquigarrow_{\mathcal{L}} \phi_3 \implies \exists \phi_4 : \phi_2, \mathtt{body}\ \mathtt{f}_3 \rightsquigarrow_{\mathcal{L}} \phi_4\Big)
\end{aligned}
$$

5.5 Proof of theorem 3.2.1

The proof of theorem 3.2.1 follows directly from lemma 5.1.1 (Replace equivalent calls), lemma 5.2.1 (Discard non equal isomorphisms), and lemma 5.3.1 (Modular procedure equivalence).

Chapter 6

RIE based equivalence verification tool

This chapter describes the implementation of our RIE based procedure equivalence tool[1]. The tool is implemented on top of the Boogie [3] intermediate verification language.

Our procedure equivalence tool takes a pair of programs and uses RIE to automatically verify if the programs are equivalent. If it can prove that all of the similarly named procedures are pair-wise equivalent it outputs success, otherwise it outputs failure. In order to verify that a pair of procedures are equivalent our tool checks that given equal initial heaps the procedures result in isomorphic final calling contexts. Directly checking that a pair of unboundedly large heaps are isomorphic is intractable, and the underlying SMT solver has no built in support for deciding heap isomorphism.

Isomorphism can be considered a kind of extensional property of heaps. Values are *intensionally equal* when they are defined in the same way, whereas they are *extensionally equal* when they have the same properties. For example, the heaps $h_1 = h_0[(5, \mathtt{f}) \mapsto 7][(5, \mathtt{g}) \mapsto 8]$ and $h_2 = h_0[(5, \mathtt{g}) \mapsto 8][(5, \mathtt{f}) \mapsto 7]$ are not intensionally equal, but they are extensionally equal since $\forall a, f : h_1(a, f) = h_2(a, f)$.

An extensional notion of equivalence is powerful because it naturally accounts for reordering of store updates, as discussed extensively by Benton [11]. The SMT solver deals primarily in intensional equality, and is not very good at establishing isomorphism between heaps. However, there is an effective axiomatisation of extensional heap equality. RIE allows us to establish isomorphism using only extensional heap equality.

The remaining implementation challenges are reasoning about procedure call effects and framing across procedure calls. RIE helps with both of these challenges, but some further ideas and implementation choices are required to create an effective SMT based tool. Our tool is implemented on top of the intermediate verification language Boogie [3], and uses a standard Dafny [37] style heap encoding. This chapter describes some interesting features of the implementation and also gives some example procedure pairs that our tool can verify.

6.1 Boogie Preliminaries

In this section we give a brief overview of the important parts of the Boogie language as used by our tool. The section will cover functions, axioms, procedures, procedure contracts, and assertions.

Boogie functions are mathematical, so may not have side effects. Functions have zero or more arguments and return a value. We write predicate to mean a function that returns a boolean. A function

[1] Available at `https://github.com/lexicalscope/ape`

may optionally be given an implementation. Properties of functions may be postulated using axioms. Figure 6.1 shows three example functions and an axiom. The function `Double` is declared and given an implementation, it takes a single argument of type `int` and returns the result of multiplying that argument by `2`. The predicate `APredicate` and function `Mystery` are given no implementation. However, their behaviour is partially defined by the axiom on line 6, which states that, for all values of type `int`, whenever `APredicate` holds then `Mystery` will return a positive value.

```
1 function Double(x:int) : int { x*2 }
2
3 function APredicate(x:int) : bool;
4
5 function Mystery(x:int) : int;
6 axiom (∀x:int :: APredicate(x)⟹Mystery(x) > 0);
```

Figure 6.1: An example of a Boogie function declaration, function implementation and axiom.

Boogie procedures are imperative and work on global and local variables. A procedure can take zero or more arguments. All local variables must be declared at the start of the procedure. Procedures may be annotated with preconditions and postconditions which we describe further below. Figure 6.2 shows an example of a Boogie procedure called `AProcedure`. The procedure takes three arguments, and returns a single value. The `$` symbols on variables names are a convention we use to distinguish between variables originating in the source $\mathcal{BL}$ program and variables introduced by our encoding into Boogie.

The procedure in fig. 6.2 has a precondition and postcondition. The precondition requires that whenever the procedure is called the predicate `P` must hold for the arguments of the procedure. The postcondition asserts that whenever the procedure terminates, that the predicate `Q` must hold for the values `$h_pre` and `$h_post`. Boogie attempts to verify procedures by assuming the precondition then proving that after all terminating executions of the procedure body the postcondition holds. If a postcondition is marked as `free` then Boogie will not attempt to check that it holds, for example line 20. Free postconditions can be used to state properties of procedures that are already know. In our tool free postconditions are used to express properties derivable from the semantics of $\mathcal{BL}$ and that thus do not need to be verified again.

```
1  function P(Heap,Roots,Ref):bool;
2  function Q(Heap,Roots,Ref):bool;
3  function R(Ref):bool;
4  function S(Ref):bool;
5  function T(Heap):bool;
6
7  procedure AProcedure($h_pre:Heap, $roots:Roots, x:Ref) returns ($h_post:Heap)
8     requires P($h_pre,$roots,x); // precondition
9     ensures Q($h_pre,$h_post); // postcondition
10 {
11    assume R(x);
12    ... // implementation statements go here
13    $h := call AnotherProcedure($h, $roots, x:Ref);
14    assert S(x);
15    ... // more implementation statements
16 }
17
18 procedure AnotherProcedure($h_pre:Heap, $roots:Roots, x:Ref) returns ($h_post:Heap)
19    requires P($h_pre,$roots,x);
20    ensures free T($h_post);
```

Figure 6.2: Example Boogie procedure declarations and implementation.

Boogie is a modular verification system. An example of a procedure call is shown on line 13. When verifying `AProcedure` boogie may try to establish that the precondition of `AnotherProcedure` holds in the prestate of line 13, if it does so it may assume that the postcondition of `AnotherProcedure` holds in the poststate of line 13.

Boogie also supports assume and assert statements. Assert statements are similar to postconditions, Boogie attempts to prove that whenever a procedure execution reaches an assertion, such as on line 14, the property asserted holds. Assume statements, such as on line 11, are used to restrict the set of executions that Boogie considers during verification. If an assume statement does not hold for a particular execution then that execution is discarded — no assertions or postconditions are checked for discarded executions.

6.2 Isomorphism using extensional equality

RIE uses angelic allocation to allow isomorphism to be established by checking only for extensional equality. Angelic allocation means allowing the verifier to attempt to prove equivalence under the most fortuitous choice of memory allocation locations. Specifically we are interested in allocators that result in heaps related by the identity isomorphism. For soundness it is necessary that equivalence is proved for at least one execution pair from each initial state (theorem 3.2.1). So our tool is free not to prove equivalence for many other possible executions, i.e. those with less fortuitous choices of allocation location.

The procedure pair to be verified is self composed several times. Once for each way that the procedure pair allocations can be equated. This is followed by an assertion that at least one of the execution pairs must produce isomorphic final states. The assertion consists of a disjunction. Together the executions and assertion correspond to angelic non-determinism for allocation. In other words, the tool produces a composite trace consisting of executions of the procedure pair for every allocation strategy and uses a disjunction to ask the solver to show the required property for at least one execution. The assertion ensures that verification is successful only if at least one branch of the disjunction holds for every (terminating) initial state, which easily guarantees our soundness condition.

A simple example of a procedure pair that our tool can verify is in fig. 6.3. In this case there are two useful ways in which the allocations could be equated. Either $t_0 = t_1 \wedge u_0 = u_1$ or alternatively $t_0 = u_1 \wedge u_0 = t_1$. Any allocator that allocates addresses such that $t_0 = u_1 \wedge u_0 = t_1$ will produce equal, and therefore isomorphic, heaps at the conclusion of the procedures. Here t_0 means the memory address allocated into variable t in version _0 of the procedure.

```
1  procedure DoubleAllocation_0(x) {
2    t := new();
3    u := new();
4
5    x.f := t;
6    x.g := u;
7  }
```

```
8  procedure DoubleAllocation_1(x) {
9    t := new();
10   u := new();
11
12   x.g := t;
13   x.f := u;
14 }
```

Figure 6.3: Our tool verifies this simple example with reordered allocations

Boogie code for our angelic allocation encoding is shown in fig. 6.4. A copy of the initial heap is made for each procedure for each allocator, lines 14 and 33. Then the procedure pair is executed once for each allocator. A single procedure allocator is a permutation of allocations, for pairs of procedures this corresponding to all the ways in which the allocations could be equated. Allocators are modelled abstractly, by equating pairs of allocations, rather than by using concrete addresses. The assumptions on

lines 24 and 43 model the allocators. The assertion on line 53 uses a disjunction to produce the angelic nature of the verification.

Our encoding allows the verifier to consider a different choice of allocation strategy from each initial store. Which means that initial stores that take different control flow branches may be verified under different allocation strategies. The assertion, line 53, is checked for every initial store. Theorem 3.2.1 shows that it is indeed sufficient to prove equivalence for a single allocation strategy from each initial store, and that it is sound for the tool to select any set of isomorphisms that correspond to such an allocation strategy[2]

[2]In principle it is possible to use information obtained from an execution under one allocation strategy in the proof of the assertion under a different allocation strategy. Our tool doesn't currently do this explicitly, although it is possible that the solver does internally. Using the information explicitly potentially opens an avenue of attack on the limit described in section 1.8, which it would be interesting to pursue further.

```
1  procedure DoubleAllocation_DoubleAllocation($h:Heap, $roots:Roots, x:Ref)
2     ...
3  {
4     // One new heap variable for each allocator for each procedure
5     var $h_0$0, $h_0$1, $h_1$0, $h_1$1:Heap;
6     ...
7     // Variable for allocations, here each procedure body has two allocations
8     var $a#0_0$0,$a#1_0$0:Ref; // procedure 0 allocator 0
9     var $a#0_1$0,$a#1_1$0:Ref; // procedure 1 allocator 0
10    var $a#0_0$1,$a#1_0$1:Ref; // procedure 0 allocator 1
11    var $a#0_1$1,$a#1_1$1:Ref; // procedure 1 allocator 1
12
13    // restore heaps for allocator 0
14    $h_0$0 := $h;
15    $h_1$0 := $h;
16
17    // allocate everything up front
18    $h_0$0,$a#0_0$0 := $allocate($h_0$0); ...
19    $h_0$0,$a#1_0$0 := $allocate($h_0$0); ...
20    $h_1$0,$a#0_1$0 := $allocate($h_1$0); ...
21    $h_1$0,$a#1_1$0 := $allocate($h_1$0); ...
22
23    // assume allocator 0
24    assume $a#0_0$0 == $a#0_1$0;
25    assume $a#1_0$0 == $a#1_1$0;
26
27    // the following is not Boogie syntax, it is shorthand for what the tool does:
28    // which is to inline the body of the named procedures with variables renamed to be unique.
29    inline_procedure DoubleAllocation_0;
30    inline_procedure DoubleAllocation_1;
31
32    // restore heaps for allocator 1
33    $h_0$1 := $h;
34    $h_1$1 := $h;
35
36    // allocate everything up front
37    $h_0$1,$a#0_0$1 := $allocate($h_0$1); ...
38    $h_0$1,$a#1_0$1 := $allocate($h_0$1); ...
39    $h_1$1,$a#0_1$1 := $allocate($h_1$1); ...
40    $h_1$1,$a#1_1$1 := $allocate($h_1$1); ...
41
42    // assume allocator 1
43    assume $a#1_0$1 == $a#0_1$1;
44    assume $a#0_0$1 == $a#1_1$1;
45
46    // the following is not Boogie syntax, it is shorthand for what the tool does:
47    // which is to inline the body of the named procedures with variables renamed to be unique.
48    inline_procedure DoubleAllocation_0;
49    inline_procedure DoubleAllocation_1;
50
51    // assert that at least one of the allocators
52    // results in isomorphic final heaps
53    assert
54         $Isomorphism($h_0$0, $h_1$0, $roots) ∨  // allocator 0
55         $Isomorphism($h_0$1, $h_1$1, $roots);   // allocator 1
56 }
```

Figure 6.4: Angelic Allocation encoded in Boogie, by inlining multiple copies of the procedures. In this case each procedure has two allocations.

6.3 Extensional Equality

Directly proving isomorphism is challenging for an SMT based tool due to the natural existential quantification of the bijection between addresses. Instead of giving our tool a direct definition of isomorphism, we use an uninterpreted function and several axioms. This allows us to provide multiple indirect routes for proving isomorphism, which the solver can then explore.

RIE allows our tool to prove general isomorphism by trying to prove isomorphism with the identity bijection under angelic allocation assumptions (section 6.2), thus avoiding the existential quantification. Proving that store parts are related by the identity bijection corresponds to extensionality. The requirement to solve extensional equality strongly influences the choice of heap encoding.

Figure 6.5 shows how extensional heap equality and isomorphism are related in our encoding. The axiom on line 11 allows the solver to prove isomorphism of a pair of heaps by proving extensional equality of the pair. The roots represent values on the stack of the calling context. Extensional equality is on line 2, and is standard. Isomorphism is abstracted by the function on line 8. The tool also has other axioms allowing it to prove isomorphism in alternative ways.

```
1  // Extensional Equality Of Heaps
2  function $Heap#Equal($h_0, $h_1:Heap) : bool
3  {
4     (∀$a:Ref :: {...} (∀<alpha> $f:Field alpha ::  $Read($h_0, $a, $f) = $Read($h_1, $a, $f)))
5  }
6
7  // Isomorphism is uninterpreted
8  function $Isomorphism($h_0, $h_1:Heap, $roots:Roots) : bool;
9
10 // extensionally equal heaps are isomorphic
11 axiom (∀$h_0,$h_1:Heap, $roots:Roots :: {$Isomorphism($h_0, $h_1, $roots)}
12    $Roots#Allocated($roots, $h_0) ∧ $Heap#Equal($h_0, $h_1)
13 ⟹
14    $Isomorphism($h_0, $h_1, $roots));
```

Figure 6.5: Axiom in Boogie encoding that permits extensionally equal heaps to be considered isomorphic

Our definition of isomorphism, definition 2.2.1, is restricted to the reachable part of the store. This allows for equivalence of procedures that allocate different amounts and shapes of garbage. Figure 6.6 shows an example pair of procedures that our tool can verify. Procedure `DifferentShapedGarbage_0` allocates different garbage than procedure `DifferentShapedGarbage_1`. Note that on line 6 the garbage is made reachable from the procedure parameter, then subsequently made unreachable again.

```
1  procedure DifferentShapedGarbage_0(x) {
2    t := new();
3    u := new();
4    t.f := x;
5    x.f := x;
6    x.g := t;
7    x.g := new();
8  }
```

```
9  procedure DifferentShapedGarbage_1(x) {
10   t := new();
11   x.f := x;
12   x.g := new();
13 }
```

Figure 6.6: Our tool can verify that this pair of procedures, that allocate different shapes and amounts of garbage, are equivalent.

In order to prove equivalences of garbage creating procedures we provide an axiom which allows isomorphism to be proved from extensional equality of the reachable part of the heap. Figure 6.7 shows the

```
1  function $Heap#ReachableEqual($h_0, $h_1:Heap, $roots:Roots) : bool
2  {
3    $Heap#ReachableEqual'($h_0, $h_1, $roots) ∧
4    $Heap#ReachableEqual'($h_1, $h_0, $roots)
5  }
6
7  function $Heap#ReachableEqual'($h_0, $h_1:Heap, $roots:Roots) : bool
8  {
9    (∀$a:Ref :: {...}
10     (∀<alpha> $f:Field alpha ::
11         $Reachable($h_0, $roots, $a)⟹$Read($h_0, $a, $f) = $Read($h_1, $a, $f)))
12 }
13
14 function $Reachable($h:Heap, $roots:Roots, $a:Ref) : bool
15 {
16    (∃$r:Ref :: {...} $Root($roots, $r) ∧ $Reach($h, $r, $a))
17 }
18
19 // Reachabillity is uninterpreted
20 function $Reach($h:Heap, $a:Ref, $b:Ref) : bool;
21
22 axiom (∀$h:Heap, $a:Ref :: $Reach($h, $a, $a));
23 axiom (∀$h:Heap, $a,$b:Ref :: {...} $Reach($h, $a, $b) ∧ $NoInboundEdges($h,$b)⟹$a = $b);
24 axiom (∀$h:Heap, $a,$b:Ref :: {...} $Reach($h,$a,$b)⟹
25   $a = $b ∨ (∃$c:Ref,$f:Field Ref :: $Reach($h, $a, $c) ∧ $Edge($h, $c, $f, $b)));
26 axiom (∀$h:Heap, $a,$b:Ref :: {...} $Reach($h,$a,$b)⟹
27   $a = $b ∨ (∃$c:Ref,$f:Field Ref :: $Edge($h, $a, $f, $c) ∧ $Reach($h, $c, $b)));
28
29 // extensionally equal reachable heaps are isomorphic
30 axiom (∀$h_0, $h_1:Heap, $roots:Roots :: {...}
31   $Roots#Allocated($roots, $h_0) ∧ $Heap#ReachableEqual($h_0, $h_1, $roots)
32  ⟹
33   $Isomorphism($h_0, $h_1, $roots));
```

Figure 6.7: Extensionally equal reachable heaps are considered isomorphic

Boogie encoding. Interesting points to note are:

- That the equivalence of the reachable set for the heaps is not expressed directly, rather this is inferred from the fact that the roots are equal and that everything reachable from them in either heap must be equal, line 11. Directly requiring the solver to prove the reachable sets are the same doesn't work well.

- The tool does not normally need to prove if a particular object is reachable. Rather it may overapproximate the reachable objects. Often it is necessary to prove disjointness of certain heap regions. For example, when identifying garbage objects, or when trying to prove that a property is preserved over a procedure call. We give axioms that enable the tool to establish a lack of reachability by showing either that there are no outgoing or no ingoing edges to a particular heap region. More sophisticated support for framing would increase the completeness of the tool, see section 6.5.

6.4 Procedure Calls

Our tool is modular, so does not examine the bodies of called procedures when verifying the caller. Instead a summary of the called procedure is used. As is standard in modular verification, a procedure summary is an assertion that abstracts the procedure's pre and post store relation [28]. In program equivalence

```
1  procedure TreeCopy_0(t,r)
2    modifies {r};
3  {
4    if(t != null) { if(r != null) {
5      rr := new();
6      rl := new();
7
8      tr := t.r;
9      TreeCopy_0(tr, rr);
10
11     tl := t.l;
12     TreeCopy_0(tl, rl);
13
14     n := new();
15     n.r := rr.v;
16     n.l := rl.v;
17     r.v := n;
18 }}}
```

```
19 procedure TreeCopy_1(t,r)
20 modifies {r};
21 {
22   if(t != null) { if(r != null) {
23     rl := new();
24     rr := new();
25     n := new();
26
27     tl := t.l;
28     TreeCopy_1(tl, rl);
29
30     tr := t.r;
31     TreeCopy_1(tr, rr);
32
33     n.r := rr.v;
34     n.l := rl.v;
35     r.v := n;
36 }}}
```

Figure 6.8: Our tool can verify this pair of procedures that recursively copy a tree. This is the same example as in section 1.5.

verification the standard procedure summary is generalised to pairs of procedures and called a mutual summary [25]. A mutual summary abstracts the pre and post store relation of a pair of procedures.

Because our objective is fully automated procedure equivalence verification, we do not expect the programmer to necessarily have manually annotated the program with procedure summaries. Therefore our tool must automatically produce a summary of the mutual behaviour of called procedures which is sufficiently powerful that the calling procedures can be verified. This is the same approach taken by Symdiff [25] and Rêve [21], but differs from RVT [24] which uses uninterpreted functions, and SCORE which interleaves the programs. In our tool the programmer may supplement the automatic procedure summaries with additional annotations.

A challenge, particular to equivalence verification of imperative procedures, is that even when the global state of two procedures have become different it may still be necessary to use the equivalence of a pair of calls in the verification. For example, fig. 6.8 shows a pair of recursive procedures whose equivalence our tool can verify. The recursive call that copies the right of the tree on lines line 9 and line 31 happen from stores that are not fully equivalent. The tool can use the fact that the two calls have the same observable effects in order to show that `TreeCopy_0` and `TreeCopy_1` are equivalent overall.

The situation is further complicated by the difference in the order of the calls, in `TreeCopy_0` the first call copies the right of the tree, whereas in `TreeCopy_1` the first call copies the left of the tree. If our tool can show that the store parts reachable from the parameters of a pair of calls are isomorphic then it may assume that the effects of the calls are related. Maintaining knowledge of which parts of the store are isomorphic across procedure calls (which may have different effects) requires some support for procedure effect framing. We discuss how our implementation deals with this below.

Procedure calls are handled using a mechanism similar to mutual summaries [25]. Figure 6.9 shows the encoding of mutual procedure summaries in our tool. An uninterpreted *abstraction function*, line 65, abstracts calls to each procedure. The abstraction function is assumed as a free postcondition of the procedure, line 69 and relates the pre and post stores of the call. A free postcondition is assumed but not checked by the verifier. The summary axiom, line 83, can be applied on pairs of pre and post stores that are related by the abstraction function. In this way we provoke the solver into searching the execution traces for pairs of calls that can be related, whilst ensuring consistency by restricting the isomorphism to

be the identity.

During a Simplify [18] 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, our tool controls the proof search by a combination of the logical meaning of the axioms and the quantifier triggers.

The synthetic parameter `$strat` of `TreeCopy_0` (line 68) and `TreeCopy_1` (line 75) is introduced to prevent the proof search from trying to establish equivalence between procedure calls occurring under different allocation correspondences. For example, in fig. 6.4, the inlining of `DoubleAllocation` on line 29 and line 49 will contain recursive calls to `DoubleAllocation_0` and `DoubleAllocation_1` respectively — but it is not useful to find relations between these calls as they pertain to different allocations correspondences. Specifically, the disjunction on line 53 asserts nothing about the relationship between those heaps. The parameter `$strat` represents the allocation correspondence in effect for that call, and the mutual summary trigger (lines 84 and 85 of fig. 6.9) restricts instantiation of the quantifiers to calls which occurred under the same allocation correspondence.

The automatically generated summary axiom, line 83, states that if the procedures are called from equal store parts they have identical effects. The reachable region of the store is expressed in the same way as in fig. 6.5. The summary of the procedures is enhanced with framing axioms that help our tool prove that regions of the heap that are disjoint from the procedure effects may maintain their isomorphisms across the call. The most important of these is on line 71.

RIE relates procedure calls using the $\mathcal{U}$ predicate (fig. 3.6). The behaviour of equivalent procedure calls is modelled in terms of the observable write effects, as well as a relation between the post stores. In the Boogie code the predicate `SameDiff` (line 91) allows the verifier to use knowledge about the equivalence of called procedures in some cases where the reachable set is too approximate.

```
65 function $abs_TreeCopy_0($strat:int, $h_pre:Heap, t:Ref,r:Ref, $h_post:Heap):bool;
66 function $abs_TreeCopy_1($strat:int, $h_pre:Heap, t:Ref,r:Ref, $h_post:Heap):bool;
67
68 procedure TreeCopy_0($strat:int, $h_pre:Heap, t:Ref,r:Ref) returns ($h_post:Heap)
69  free ensures $abs_TreeCopy_0($strat, $h_pre, t, r, $h_post);
70  ...
71  free ensures (∀<alpha> $a:Ref,$f:Field alpha ::
72   $a≠$Null∧$Allocated($h_pre,$a)∧$Read($h_pre,$a,$f)!=$Read($h_post,$a,$f)
73   ⟹$ReachableFromParams($h_pre, t, r, $a));
74
75 procedure TreeCopy_1($strat:int, $h_pre:Heap, t:Ref,r:Ref) returns ($h_post:Heap)
76  free ensures $abs_TreeCopy_1($strat, $h_pre, t, r, $h_post);
77  ...
78  free ensures (∀<alpha> $a:Ref,$f:Field alpha ::
79   $a≠$Null∧$Allocated($h_pre,$a)∧$Read($h_pre,$a,$f)!=$Read($h_post,$a,$f)
80   ⟹$ReachableFromParams($h_pre, t, r, $a));
81
82 axiom (∀$strat:int,$h_pre_0,$h_post_0,$h_pre_1,$h_post_1:Heap,
83         t_0,r_0,t_1,r_1:Ref ::
84  {$abs_TreeCopy_0($strat,$h_pre_0, t_0, r_0, $h_post_0),
85   $abs_TreeCopy_1($strat,$h_pre_1, t_1, r_1, $h_post_1)}
86  $abs_TreeCopy_0($h_pre_0, t_0, r_0, $h_post_0)∧
87  $abs_TreeCopy_1($h_pre_1, t_1, r_1, $h_post_1)
88  ∧$Heap#EqualFromParams($h_pre_0, t_0, r_0, $h_pre_1, t_1, r_1)
89   ⟹$SameDiff($h_pre_0, $h_post_0, $h_pre_1, $h_post_1));
90
91 function $SameDiff($h_pre_0, $h_post_0, $h_pre_1, $h_post_1:Heap) : bool {
92  (∀<alpha> $a:Ref,$f:Field alpha ::
93   ($Read($h_pre_0, $a, $f)≠$Read($h_post_0, $a, $f)
94   ⟹$Read($h_post_0, $a, $f)==$Read($h_post_1, $a, $f))∧
95   ($Read($h_pre_1, $a, $f)≠$Read($h_post_1, $a, $f)
96   ⟹$Read($h_post_0, $a, $f)==$Read($h_post_0, $a, $f))) }
```

Figure 6.9: Mutual summary of the `TreeCopy_0` and `TreeCopy_1` procedures. Written in Boogie.

6.5 Discussion

The number of copies of the procedure pair is equal to the number of permutations of the allocations in one procedure (whichever procedure has the most allocations). The number of permutations grows factorially. Hence permuting the allocations is only practical for relatively small numbers of allocation sites. However, this may not be as restrictive as it may seem. In practice many interesting procedures contain only small numbers of allocations that are not in a loop or recursive procedure call. In some cases it may be possible to split the procedures into chunks or abstract some 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, if we obtained the types of the objects being allocated we would be able to eliminate all permutations that aligned objects of different types.

Framing of procedure calls is important in verifying equivalence for many examples. Our tool has a fairly naïve approach to framing and disjointness of heap regions, and this restricts the class of examples it can currently deal with. However, our techniques, and choice of Dafny [37] style heap encoding, should be amenable to working along side a more powerful framing methodology. Improving our tool with better framing support would likely significantly improve its completeness.

We have not developed any axioms that allow isomorphism to be proved for rearrangements of objects which were allocated prior to procedure entry, such as the example in section 1.7. However, the way our tool encodes the calling context as a set of roots and the fact that isomorphism is uninterpreted should allow such axioms to be added to the tool once we understand what they should be.

We considered many alternative approaches to establishing isomorphism. The natural approach using existentials does not work very well. We investigated several approaches using $\forall$ 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 for disjoint heap writes and heap effects of procedure calls to commute was an important design force: many approaches required extensive additional axioms to handle the various cases, whereas our current approach of enumerating allocators seems to handle many cases naturally.

Chapter 7

Related Work

In this section we discuss the history and current state of research in program equivalence. In particular we detail the current state-of-the-art in fully automatic procedure equivalence verifiers. While one could always go about proving two programs equivalent by proving their full functional correctness against the same specification, we focus on works which attempt to take advantage of both programs to simplify the problem.

7.1 Program Equivalence and Applications

The formal study of program equivalence arguably pre-dates the study of functional correctness. In his 1969 paper [27], introducing the Hoare logic, Hoare identified that "Many [previous] axiomatic treatments of computer programming [1, 29, 59] tackle the problem of proving the equivalence, rather than the correctness, of algorithms". To the present day, practical approaches to proving the equivalence of imperative programs rely on the programs under comparison having structural similarities. Many works focus on methods to account for particular structural differences. The importance of program structure in proving program equivalence was observed by Dijkstra in 1972 [17], where he motivates the problem of proving program equivalence by observing that programmers are often called upon to modify existing programs:

> It is a programmer's everyday experience that for a given problem to be solved by a given algorithm, the program for a given machine is far from uniquely determined. In the course of the design process he has to select between alternatives; once he has a correct program, he will often be called to modify it, for instance because it is felt that an alternative program would be more attractive as far as the demands that the computations make upon the available equipment resources are concerned.
>
> ... But I do not intend to tackle it in this general form. On the contrary instead of starting with two arbitrarily given programs (say: independently conceived by two different authors) I am concerned with alternative programs that can be considered as products of the same mind and then the question becomes: how can we conceive (and structure) those two alternative programs so as to ease the job of comparing the two?

Since then program equivalence has become important in several strands of research. In particular, key developments in the study of program equivalence have come about as a result of research on the problems of non-interference in secure information flow and compiler translation validation.

Non-interference is a property of programs which have both secret and public inputs, programs have the non-interference property if the values of the secret inputs do not influence (interfere with) the public outputs of the program. Compiler *translation validation* attempts to provide assurances that the program output by a compiler is correct wrt. the input program. Translation validation differs from other approaches to compiler correctness in that it only concerns itself with the input and output programs of particular runs of the compiler, and does not attempt to instrument or prove the correctness of the compiler implementation itself.

7.1.1 Secure information flow

Secure information flow was identified as being related to program equivalence [30], then non-interference was formalised as a safety property over pairs of program traces, called a *2-safety property* [7, 55]. Subsequently, the concept of reducing 2-safety properties to safety properties (over a single program trace) using a variety of techniques has been explored [10, 38] and generalised, particular via product programs [4, 6] and similar [51, 56, 61].

A *product program* is an interleaving or combining of a pair of programs into one program, such that useful program invariants can be formulated at interesting points in the combined program. For example, it may be useful to align two loop heads and formulate an inductive coupling invariant that relates the loop counters or other interesting state.

Later work generalises the techniques developed in this field to questions of relations between programs [2, 5]. Including proving program continuity [6], proving abstraction and refinement relationships between programs [4], compiler correctness [61], vectorising loops [4], and proving that programs conform to differential privacy policies [8]. The work of Ettinger [20] uses program slices to justify reordering statements during refactoring, a process which is in some ways the reverse of product program construction.

A Semantic Approach to Secure Information Flow[30] (2000) (Joshi and Leino) discusses several notions of program equivalence in the course of defining secure information flow semantically. Semantic equivalence of programs is defined for a relational semantics and a weakest precondition semantics. In the relational semantics program equivalence is equality on the relations. In the weakest precondition semantics, programs are equivalent if forall predicates the programs' weakest liberal preconditions correspond and the programs terminate from the same initial states.

Secure Information Flow by Self-Composition[7] (2004) (Barthe, D'Argenio, and Rezk) compose a program with itself to allow single program analysis tools to be applied to the question of determining non-interference.

Secure Information Flow As a Safety Problem[55] (2005) (Terauchi and Aiken) formalises the notion of a 2-safety property. A 1-safety property is expressed over 1 trace of a program, whereas a 2-safety property is expressed over 2 traces of the program. They prove that any 2-safety problem can be reduced to a 1-safety problem by using self-composition, and apply this idea to the secure information flow problem.

Relational Verification Using Product Programs[6] (2011) (Barthe, Crespo, and Kunz) defines product programs which reduce relational verification to standard verification by producing a single product program from a pair of programs. Product programs are constructed by applying structural transformations so that structurally different programs can be aligned at certain useful statements, such as at loop entry. The programs execute out of lock step between these alignment points. The transformation rules allow loops and conditionals to be merged, or unrolled.

"Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification"[4] (2013) (Barthe, Crespo, and Kunz) generalises the notion of product program to the question of whether one program is an abstraction or refinement of another program. These are called asymmetric products and can also be applied to non-deterministic programs. A standard product program can be used for questions of the form: do all traces of the product program have a certain property. Asymmetric products can answer questions of the form: for every execution of the first program does there exist a related execution of the second program.

7.1.2 Compiler translation verification

Compiler translation validation [33, 36, 51, 56, 61] is inherently a program equivalence question. *Relational Hoare Logic* [11] (see below) was proposed in the context of proving the correctness of compiler transforms. Several variations of product programs have also been applied [61], as well as related techniques on the graph representations of the program [51, 56].

Covac: Compiler validation by program analysis of the cross-product[61] (2008) (Zaks and Pnueli) present a deductive framework for proving program equivalence by reducing the programs to their cross-product, and a tool for translation validation of intra-procedural LLVM optimisations. Loops are expected to correspond across versions. The cross-product is a simultaneous execution of the programs, where reads and writes are performed in-sync. Inductive invariants must be generated for the cross-product. Optimisations are correct if the optimized version has the same observable effects as the original.

Proving Optimizations Correct Using Parameterized Program Equivalence[33] (2009) (Kundu, Tatlock, and Lerner) describes a tool for proving compiler optimizations correct by proving the equivalence of parameterized programs. Parameterized programs are abstractions, with some lines of code being summarized by properties such as "does not use or define a particular variable". This means that their tool can prove the correctness of compiler optimisations for classes of programs (that have the same abstraction) rather than single instances of programs. If their tool can prove the optimisation correct for every class of program for which it is defined, then the tool proves the optimisation correct. Equivalence is proved by establishing a bisimulation between the control flow graphs, with some enhancements to deal with changes in loop structure.

Equality-based Translation Validator for LLVM[51] (2011) (Stepp, Tate, and Lerner) apply the technique of Equality Saturation to Translation validation for LLVM. Equality Saturation is an intensional equality based technique. Equality Saturation is performed on a Program Expression Graph (PEG), which is a pure functional representation of a program. In their tool a combined PEG is constructed containing the nodes of the original and optimised versions of the program. Equality Saturation infers equality between PEG nodes by repeatedly applying equality axioms. The goal is to infer that the program entry nodes are equal. This process is somewhat similar to SMT solver E-graphs. Store update reordering is handled by the axiom $p \neq q \implies load(store(\sigma, q, v), p) = load(\sigma, p)$. Aliasing information is pre-computed. The tool can generate proofs from the equality saturated PEG.

Evaluating Value-graph Translation Validation for LLVM[56] (2011) (Tristan, Govereau, and Morrisett) also address translation validation tool for LLVM, but they propose comparing programs by normalisation of the value graph. This is also an intensional equality based technique. The goal is to apply transformations to the value graphs, until the value graphs an intensionally equal. Sharing of nodes between the graphs is supported. Programs are first translated into a functional (or Monadic) form. Then normalisation is applied with the goal that the graphs share all nodes. Graphs can only be transformed to

share all nodes if the orginal programs where semantically equivalent. The normalisation rules are derived from the optimisations that the compiler may perform.

7.1.3 Relational Hoare Logic

Relational Hoare Logic [5, 11, 12, 14, 58] (RHL) was proposed by Benton, in 2004 [11], 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 a pair of program states. RHL has been extended by various rules to account for differences in structure between the programs. Benton adds additional rules to the logic that are inspired by the compiler optimisations under examination, for example a rule for eliminating while loops when the condition is known to be false. The connection between RHL and secure information flow is also noted in Benton's 2004 paper.

In 2007 **Benton** [12], introduces an RHL with a notion of heap regions. Bijections are used to relate heap regions between program versions. The semantics of effect systems are characterised in terms of the store relations preserved by the effects. The paper caters for dynamic allocation by using regions to abstract from explicit references. A partial bijection between locations in each region is used. Disjoint bijections can be combined, but no other notion of compatibility between bijections is used. References to integers are supported. The effect system information can be used to reason about reordering of program statements in an extensional manner. A masking rule can eliminate some effects which are not observable, allowing more program equivalences to be validated. Proofs of equivalence of programs with commuting computations, duplicate computation and several others are given. A related approach is taken by Yang [58] in the same year, where separation logic assertions are used to express the pre and post condition relations of the RHL quadruple.

Barthe, Crespo, and Kunz [5] pointed out that RHL is closely related to the idea of product programs. RHL rules that deal with difference in program structure have close analogues in product program construction rules. RHL can be used to give a relational (or 2-safety) meaning to the proof of a 1-safety property on a product program. The concept of left and right programs appears in both ideas. Many rules in RHL and product program construction allow one program to take a step whilst the other executes a `skip`, and so have a "left" and a "right" form depending on which program takes a step. These asynchronous rules are critical in allowing both RHL and product programs to account for structural difference between the programs.

Interestingly the completeness of various RHL variants and product program construction approaches can be related. In particular a self-composition rule can be added to both RHL and product programs to give them equivalent completeness to the standard Hoare Logic [5].

Banerjee, Schmidt, and Nikouei [2] propose a logic for weaving programs so that relational properties of programs can be expressed, bringing together many ideas from the work on product programs and RHL. Their logic accounts for many structural differences in programs such as partially aligned loops. They extend this to support reasoning about encapsulation in dynamically allocating programs; catering for equivalence between programs which vary the representation of objects.

7.2 Fully automatic program verification tools

To our knowledge there are four other tools with the objective of fully automated verification of procedure equivalence for imperative programs: Symdiff [34], RVT [24], SCORE [46], and Rêve [21].

Symdiff [34] uses program verification to prove or provide counter examples of *procedure* equivalence. It supports conditional equivalence [25] which can show partial equivalence over a subset of procedure inputs and construct summaries of interprocedural behavioural differences. Symdiff is built on top of Boogie [3]. Symdiff does not provide built in support for dealing with procedures that differ in memory allocation.

Regression Verification Tool (RVT) [24] can prove equivalence of some C programs and uses the CBMC [16] bounded model checker. RVT generates loop and recursion free program fragments, which are verified with CBMC up to a bound of 1. Using a bound of 1 is sound, since the fragments do not have any loops or recursion. RVT can partially inline the call graph, up to recursion, in order to improve the possibility to find equivalence in cases where code has been moved between procedures. Loops are encoded as recursive functions. Recursive calls are replaced by uninterpreted functions. RVT can deal with unbalance recursive functions by partitioning the inputs of one procedure into base-case or non-base case inputs then inlining one procedure up to a bound [52]. Unbalanced recursive functions make a different number of recursive calls to calculate the result. Automatic mutual termination proofs are also supported [19].

RVT deals with dynamic data structures by generating (symbolic) bounded tree-like data structures as inputs for procedures with pointer arguments. These initial tree-like structures are isomorphic up to some bound. RVT then verifies that those data structures remain isomorphic, at procedure calls and procedure return, up to the same bound — again assuming that they are trees. Our tool does not assume tree-like data structures. The bound is determined by a syntactic over-approximation of the maximum depth of modification in the loop and recursion free fragments. RVT allows the user to specify a condition under which equivalence is checked, this allows the user to ignore output states that are intended to change between program versions.

Rêve [21] uses Horn constraints to verify equivalence of deterministic imperative numerical programs with unbounded integer variables. It automatically 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. In the future Rêve's authors propose extending their tool using a similar approach to RVT for dealing with tree-shaped heap data structures.

SCORE [46] uses abstract interpretation over an interleaving of the programs. A good interleaving is found by searching. SCORE deals with numerical programs. An interesting feature of SCORE is that if the programs are not equivalent it can compute an over-approximation of the semantic difference between the programs. The precision of this over-approximation is related to the size of the syntactic difference between the programs.

7.3 Other automated tools

Here we note several other recent automated tools that tackle problems closely related to program equivalence.

Differential Assertion Checking[35] (2013) (Lahiri, McMillan, Sharma, and Hawblitzel) proposes *differential assertion checking* (DAC) which proves relative correctness between a pair of programs with respect to a set of assertions. The question proposed is: does there exist an environment in which the assertions pass in the old version but fail in the new version. This corresponds to checking for regressions,

i.e. the new program must be at least as correct relative to the assertions as the old was. They give an example where relative memory safety is checked. The new version is memory safe relative to the old version if it accesses fewer memory locations. The paper also details relative specifications for modular DAC, and describes a method for infering inductive relative specifcation of procedures.

Verification Modulo Versions: Towards Usable Verification[39] (2014) (Logozzo, Lahiri, Fahndrich, and Blackshear) introduces verification Modulo Versions (VMV) which aims to reduce the number of alarms reported by static analysis while still providing semantic guarantees. Conditions sufficient or neccessary for the safety of a program are infered. Then these conditions are used to identify regressions in the new version wrt. the old version or to prove relative correctness of the new version wrt. the old version. Regressions occur when the conditions sufficient to avoid safety violations in the old version are insufficient to avoid them in the new version. The new version is relatively correct if the conditions neccessary for safety in the old version are sufficient for safety in the new version. They evaluate their VMV tool against three large C♯ projects.

Compiler Validation via Equivalence Modulo Inputs[36] (2014) (Le, Afshari, and Su) search for compiler bugs by taking an inventive approach to producing semantically equivalent program variations. The variations are then tested to check that the compiled programs are indeed equivalent. The program variations are produced by profiling the program when executed on a particular input value i, then modifying code that is not reached by that input. Since only unreached code is modified the programs should still behave the same for i. If the modified program behaves differently then a compiler bug has been detected. The tool detected 147 bugs in GCC and LLVM.

7.4 Dynamic Allocation, Program Equivalence, and Isomorphism

The formal relationship between program equivalence, isomorphism and dynamic memory allocation has been studied for some time [54], including attempts to develop storeless language semantics (where the store is not observable). Bozga, Iosif, and Laknech [15] make the connection between storeless semantics and heap isomorphism. Several formal works directly tackle the problem of proving program equivalence in the presence of dynamic memory allocation.

Pitts uses a simulation between memory locations when defining a semantic approach to program equivalence [48], the memory model is flat not a heap. In 2005 Benton and Leperchey [13] describes a semantics and relational reasoning principle for a language with dynamic memory allocation and mutable references that may store the addresses of other references. This relational reasoning principle is applied to prove several program equivalences. A denotational semantics involving morphisms between memory locations is established, and this allows the use of equality of locations instead of partial bijections. Also of interest for us is that the issue of ensuring new memory locations are sufficiently fresh arises, as does the relationship between garbage and contextual equivalence. In 2007, Benton, Kennedy, Beringer, and Hofmann uses isomorphism between heap regions when proposing a RHL that supports dynamic allocation [12]. Yang constructs a relational separation logic with support for dynamic allocation [58]. Sumner and Zhang [53] propose a different approach, canonical memory addresses are constructed based on program control flow and syntactic elements.

In 2006 Koutavas and Wand [32] applied bisimulation to the problem of contextual equivalence of languages with dynamic allocation. Their approach is able to reason about encapsulation and relate OO programs where objects have different internal representations. They note the practical problems of a straight-forward denotational equivalence, it is too strong when programs that differ in the number

of memory allocations are considered. They also note the usefulness of using a big-step semantics when reasoning about equivalence of procedures, and adapt their use of bisimulation appropriately. More recently Naumann and Banerjee [42] also worked on the problem of relating different object representations.

Tzevelekos [57] applies three techniques: logical relations, bisimulation and game semantics to the question of dynamic allocation in program equivalence. Following Pitts and Stark [49], Tzevelekos abstracts the idea of dynamic allocation with the more general notion of providing fresh names during execution. In the logical relations technique, bijections (here extended to a more general notion of spans) between names are used to reason about functions that expand the set of names (i.e. dynamically allocate).

In automated tools, as mentioned above, Godlin and Strichman [24] apply tree isomorphisms in the context of automated program equivalence. Ramos and Engler [50] compare explicitly maintained graphs, but their approach restricts itself to executions of bounded size.

Chapter 8

Contributions and Conclusion

We proposed a methodology called RIE for verification of procedure equivalence when the procedures dynamically allocate memory. The methodology is suitable for procedures that allocate different amounts of memory in different orders and create different amounts and shapes of garbage. RIE provides a contextual notion of procedure equivalence in terms of isomorphism between store parts. RIE provides a means to establish isomorphism using only extensional equality, which is suitable for SMT based tools. We proved RIE is sound.

We described a RIE based tool for automatically verifying program equivalence. We showed how RIE can be encoded in the standard single program verifier Boogie. Particularly, the tool demonstrates that RIE can verify procedures where isomorphisms occur out-of-order, which is challenging for approaches that rely on simulation or interleaving. To our knowledge it is the first such tool that targets fully automatic equivalence verification of dynamically allocating unboundedly recursive procedures, without making assumptions about the shape of the datastructures involved.

There is scope to extend the power of our tool. Further work on automatic procedure call framing would allow more examples to be verified. Performance of the tool could be improved by employing additional static analyses to prune the space of possible isomorphisms prior to the verification step. Incorporating recent advances on equivalence verification of programs with unbalanced recursion or structurally dissimilar loops would also increase the range of examples that could be verified. Finally, although RIE theoretically allows for verification of procedures that make equivalent (but different) rearrangements of objects that existed prior to procedure entry, it is not currently clear how to support this in a practical tool.

Bibliography

[1] J. W. de Bakker. *Axiomatics of simple assignment statements.* In: *MR 94* (1968).

[2] Anindya Banerjee, David A. Schmidt, and Mohammad Nikouei. *Relational Logic with Framing and Hypotheses.* In: *FSTTCS.* 2016.

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. *Boogie: A modular reusable verifier for object-oriented programs.* In: *International Symposium on Formal Methods for Components and Objects.* Springer. 2005, pp. 364–387.

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification". In: *Logical Foundations of Computer Science: International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings.* Ed. by Sergei Artemov and Anil Nerode. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–43. ISBN: 978-3-642-35722-0.

[5] Gilles Barthe, Juan Manuel Crespo, and César Kunz. *Product programs and relational program logics.* In: *Journal of Logical and Algebraic Methods in Programming* (2016), pages. ISSN: 2352-2208.

[6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. *Relational Verification Using Product Programs.* In: *Proceedings of the 17th International Conference on Formal Methods.* FM'11. Limerick, Ireland: Springer-Verlag, 2011, pp. 200–214. ISBN: 978-3-642-21436-3.

[7] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. *Secure Information Flow by Self-Composition.* In: *Proceedings of the 17th IEEE Workshop on Computer Security Foundations.* CSFW '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 100–. ISBN: 0-7695-2169-X.

[8] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. *Proving Differential Privacy in Hoare Logic.* In: *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium.* CSF '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 411–424. ISBN: 978-1-4799-4290-9.

[9] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. *When does a refactoring induce bugs? an empirical study.* In: *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on.* IEEE. 2012, pp. 104–113.

[10] Nick Benton. *Abstracting allocation.* In: *Computer Science Logic.* Springer. 2006, pp. 182–196.

[11] Nick Benton. *Simple Relational Correctness Proofs for Static Analyses and Program Transformations.* In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '04. Venice, Italy: ACM, 2004, pp. 14–25. ISBN: 1-58113-729-X.

[12] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. *Relational Semantics for Effect-based Program Transformations with Dynamic Allocation*. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '07. Wroclaw, Poland: ACM, 2007, pp. 87–96. ISBN: 978-1-59593-769-8.

[13] Nick Benton and Benjamin Leperchey. *Relational reasoning in a nominal semantics for storage*. In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 2005, pp. 86–101.

[14] Lennart Beringer. *Relational decomposition*. In: *International Conference on Interactive Theorem Proving*. Springer. 2011, pp. 39–54.

[15] Marius Bozga, Radu Iosif, and Yassine Laknech. *Storeless Semantics and Alias Logic*. In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '03. San Diego, California, USA: ACM, 2003, pp. 55–65. ISBN: 1-58113-667-6.

[16] Edmund Clarke, Daniel Kroening, and Karen Yorav. *Behavioral consistency of C and Verilog programs using bounded model checking*. In: *Design Automation Conference, 2003. Proceedings*. IEEE. 2003, pp. 368–371.

[17] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3.

[18] David Detlefs, Greg Nelson, and James B. Saxe. *Simplify: A Theorem Prover for Program Checking*. In: *J. ACM* 52.3 (May 2005), pp. 365–473. ISSN: 0004-5411.

[19] Dima Elenbogen, Shmuel Katz, and Ofer Strichman. *Proving Mutual Termination*. In: *Form. Methods Syst. Des.* 47.2 (Oct. 2015), pp. 204–229. ISSN: 0925-9856.

[20] Ran Ettinger. *Program Sliding*. In: *ECOOP 2012 Object-Oriented Programming*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 713–737. ISBN: 978-3-642-31056-0.

[21] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. *Automating Regression Verification*. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 349–360. ISBN: 978-1-4503-3013-8.

[22] B. Godlin and O. Strichman. *Regression verification*. In: *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. July 2009, pp. 466–471.

[23] Benny Godlin and Ofer Strichman. *Inference rules for proving the equivalence of recursive procedures*. In: *Acta Informatica* 45.6 (2008), pp. 403–439. ISSN: 1432-0525.

[24] Benny Godlin and Ofer Strichman. *Regression Verification*. In: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. San Francisco, California: ACM, 2009, pp. 466–471. ISBN: 978-1-60558-497-3.

[25] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K Lahiri, and Henrique Rebêlo. *Towards modularly comparing programs using automated theorem provers*. In: *International Conference on Automated Deduction* (June 2013).

[26] Chris Hawblitzel, Ming Kawaguchi, Shuvendu Lahiri, and Henrique Rebêlo. *Mutual summaries and relative termination*. Tech. rep. MSR-TR-2011-112. Microsoft, Oct. 2011.

[27] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782.

[28] C. A. R. Hoare. "Procedures and parameters: An axiomatic approach". In: *Symposium on Semantics of Algorithmic Languages*. Ed. by E. Engeler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1971, pp. 102–116. ISBN: 978-3-540-36499-3.

[29] S. Igarishi. "An axiomatic approach to equivalence problems of algorithms with applications". PhD thesis. Computer Centre, Univ. of Tokyo, 1964.

[30] Rajeev Joshi and K. Rustan M. Leino. *A Semantic Approach to Secure Information Flow*. In: *Sci. Comput. Program.* 37.1-3 (May 2000), pp. 113–138. ISSN: 0167-6423.

[31] Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebelo. *Conditional equivalence, no. MSR-TR-2010-119*. Tech. rep. 2010.

[32] Vasileios Koutavas and Mitchell Wand. *Small Bisimulations for Reasoning About Higher-order Imperative Programs*. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, pp. 141–152. ISBN: 1-59593-027-2.

[33] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. *Proving Optimizations Correct Using Parameterized Program Equivalence*. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 327–337. ISBN: 978-1-60558-392-1.

[34] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. *SYMDIFF: a language-agnostic semantic diff tool for imperative programs*. In: *Proceedings of the 24th international conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, pp. 712–717. ISBN: 978-3-642-31423-0.

[35] Shuvendu Lahiri, Ken McMillan, Rahul Sharma, and Chris Hawblitzel. *Differential Assertion Checking*. In: *Foundations of Software Engineering*. ACM, 2013.

[36] Vu Le, Mehrdad Afshari, and Zhendong Su. *Compiler Validation via Equivalence Modulo Inputs*. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 216–226. ISBN: 978-1-4503-2784-8.

[37] K. Rustan M. Leino. *Dafny: An Automatic Program Verifier for Functional Correctness*. In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal: Springer-Verlag, 2010, pp. 348–370. ISBN: 3-642-17510-4, 978-3-642-17510-7.

[38] K. Rustan M. Leino and Peter Müller. *Verification of equivalent-results methods*. In: *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*. ESOP'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 307–321. ISBN: 3-540-78738-0, 978-3-540-78738-9.

[39] Francesco Logozzo, Shuvendu Lahiri, Manuel Fahndrich, and Sam Blackshear. *Verification Modulo Versions: Towards Usable Verification*. In: *Proceedings of the 35th conference on Programming Languages, Design, and Implementation (PLDI 2014)*. ACM SIGPLAN, June 2014.

[40] Robin Milner. *Fully abstract models of typed λ-calculi*. In: *Theoretical Computer Science* 4.1 (1977), pp. 1–22. ISSN: 0304-3975.

[41] James H Morris. "Lambda-calculus models of programming languages." PhD thesis. MIT, Cambridge, MA, 1968.

[42] David Naumann and Anindya Banerjee. *State Based Encapsulation for Modular Reasoning about Behaviour-Preserving Refactorings*. In: *Aliasing in Object-oriented Programming*. Ed. by Dave Clarke, James Noble, and Tobias Wrigstad. Springer State-of-the-art Surveys, 2012.

[43] George C. Necula and Peter Lee. *The Design and Implementation of a Certifying Compiler*. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: ACM, 1998, pp. 333–344. ISBN: 0-89791-987-4.

[44] William F. Opdyke. "Refactoring Object-Oriented Frameworks". PhD thesis. 1992.

[45] Jihun Park, Miryung Kim, B. Ray, and Doo-Hwan Bae. *An empirical study of supplementary bug fixes*. In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. June 2012, pp. 40–49.

[46] Nimrod Partush and Eran Yahav. *Abstract Semantic Differencing via Speculative Correlation*. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 811–828. ISBN: 978-1-4503-2585-1.

[47] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pasareanu. *Differential symbolic execution*. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. SIGSOFT '08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 226–237. ISBN: 978-1-59593-995-1.

[48] Andrew M. Pitts. *Operational semantics and program equivalence*. In: *Applied Semantics*. Springer, 2002, pp. 378–412.

[49] Andrew M. Pitts and Ian D. B. Stark. "Observable properties of higher order functions that dynamically create local names, or: What's new?" In: *Mathematical Foundations of Computer Science 1993: 18th International Symposium, MFCS'93 Gdańsk, Poland, August 30–September 3, 1993 Proceedings*. Ed. by Andrzej M. Borzyszkowski and Stefan Sokołowski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 122–141. ISBN: 978-3-540-47927-7.

[50] David A. Ramos and Dawson R. Engler. *Practical, Low-effort Equivalence Verification of Real Code*. In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 669–685. ISBN: 978-3-642-22109-5.

[51] Michael Stepp, Ross Tate, and Sorin Lerner. *Equality-based Translation Validator for LLVM*. In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 737–742. ISBN: 978-3-642-22109-5.

[52] Ofer Strichman and Maor Veitsman. *Regression Verification for unbalanced recursive functions*. In: *Proc. of 21st International Symposium on Formal Methods (FM'16)*. 2016.

[53] William N. Sumner and Xiangyu Zhang. *Memory indexing: canonicalizing addresses across executions*. In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 217–226. ISBN: 978-1-60558-791-2.

[54] Robert D. Tennent and Dan R. Ghica. *Abstract Models of Storage*. In: *Higher-Order and Symbolic Computation* 13.1 (2000), pp. 119–129. ISSN: 1573-0557.

[55] Tachio Terauchi and Alex Aiken. *Secure Information Flow As a Safety Problem*. In: *Proceedings of the 12th International Conference on Static Analysis*. SAS'05. London, UK: Springer-Verlag, 2005, pp. 352–367. ISBN: 3-540-28584-9, 978-3-540-28584-7.

[56] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. *Evaluating Value-graph Translation Validation for LLVM*. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 295–305. ISBN: 978-1-4503-0663-8.

[57] Nikos Tzevelekos. *Program equivalence in a simple language with state*. In: *Computer Languages, Systems and Structures* 38.2 (2012), pp. 181–198. ISSN: 1477-8424.

[58] Hongseok Yang. *Relational Separation Logic*. In: *Theor. Comput. Sci.* 375.1-3 (Apr. 2007), pp. 308–334. ISSN: 0304-3975.

[59] Yu I. Yanov. *Logical operator schemes*. In: *Kybernetilca I* (1958).

[60] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. *How do fixes become bugs?* In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 26–36. ISBN: 978-1-4503-0443-6.

[61] Anna Zaks and Amir Pnueli. *Covac: Compiler validation by program analysis of the cross-product*. In: *International Symposium on Formal Methods*. Springer. 2008, pp. 35–51.

Part I

Appendix

Appendix A

Auxiliary Lemmas

This chapter lists the remaining lemmas

A.1 Approximation Lemmas

Note that the following lemmas apply to executions of the shape $\phi_1, s_1 \rightsquigarrow_{\mathcal{K}}^{tr_1} \phi_3$, i.e. not executions involving $\|$. Also note that $\|$ is not a statement and is not allowed to appear in the program text.

Lemma A.1.1 (Sequential $\mathcal{R}$ semantics overapproximate $\mathcal{K}$ semantics)

$$\forall s_1, \phi_{1,3}, tr_1 : \ \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \implies \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{R}}} \phi_3$$

Full proof on page 148.

Lemma A.1.2 ($\mathcal{K}$ semantics overapproximate $\mathcal{R}$ semantics)

$$\forall s_1, \phi_{1,3}, tr_1 : \ \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{R}}} \phi_3 \implies \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3$$

Full proof on page 149.

Lemma A.1.3 ($\mathcal{A}$ semantics overapproximate $\mathcal{K}$ semantics)

$$\forall s_1, \phi_{1,3}, tr_1 : \ \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \implies \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{A}}} \phi_3$$

Full proof on page 149.

Proof Outline. By induction on the derivation of an execution. Since $\|$ is not a statement and cannot appear in the program text, and since all other rules of the $\mathcal{K}$ semantics and $\mathcal{A}$ semantics are identical apart from call. Rule CALLA has a weaker antecedent that rule CALLV so the proof is straightforward. □

A.2 Closure under isomorphism Lemmas

This section lists the remaining closure under isomorphism lemmas

Lemma A.2.1 (ASSUME closed)

$$\begin{aligned}
&\forall L, b, \phi_{1-3}, \pi_{1,2}, tr_1 : \\
&\quad \phi_1, \texttt{assume}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \ \wedge\ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ in(\pi_2) \\
&\quad \Longrightarrow \\
&\quad \exists \phi_4, \pi_3, tr_2 : \ \phi_2, \texttt{assume}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)
\end{aligned}$$

Full proof on page 150.

Lemma A.2.2 (ASSERTT closed)

$$\begin{aligned}
&\forall L, b, \phi_{1-3}, \pi_{1,2}, tr_1 : \\
&\quad \phi_1 \vDash b \ \wedge\ \phi_1, \texttt{assert}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \ \wedge\ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ in(\pi_2) \\
&\quad \Longrightarrow \\
&\quad \exists \phi_4, \pi_3, tr_2 : \ \phi_2, \texttt{assert}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)
\end{aligned}$$

Full proof on page 151.

Lemma A.2.3 (ASSERTF closed)

$$\begin{aligned}
&\forall L, b, \phi_{1-3}, \pi_{1,2}, tr_1 : \\
&\quad \neg(\phi \vDash b) \ \wedge\ \phi_1, \texttt{assert}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \ \wedge\ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ in(\pi_2) \\
&\quad \Longrightarrow \\
&\quad \exists \phi_4, \pi_3, tr_2 : \ \phi_2, \texttt{assert}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)
\end{aligned}$$

Full proof on page 152.

Lemma A.2.4 (Consistent with allocated subset)

$$\begin{aligned}
&\forall \phi_{1,2}, \pi_{1,2} : \\
&\quad \forall (a_1, a_2) \in \pi_1 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \ \wedge\ \pi_2 \subseteq \pi_1 \\
&\quad \Longrightarrow \\
&\quad \forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h)
\end{aligned}$$

Full proof on page 153.

Appendix B

Proofs

B.1 Isomorphism is unique (lemma 2.2.2)

Recall from page 28

Lemma 2.2.2 (Isomorphism is unique)

$$\forall \phi_{1,2}, \pi_{1,2} : \phi_1 \approx_{\pi_1} \phi_2 \ \wedge \ \phi_1 \approx_{\pi_2} \phi_2 \implies \pi_1 = \pi_2$$

Proof B.1.1
Take arbitrary $\phi_{1,2}, \pi_{1,2}$ such that

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{1}$$

$$\phi_1 \approx_{\pi_2} \phi_2 \tag{2}$$

To show, without loss of generality

$$\pi_1 \subseteq \pi_2$$

To show

$$\forall a_{1,2} : (a_1, a_2) \in \pi_1 \implies (a_1, a_2) \in \pi_2$$

By induction on the definition of π_1 and (1)

1. Base case. Take arbitrary i, x such that

$$x \in dom(\phi_1^{\tilde{\sigma}}[i]) \tag{3}$$

$$(\phi_1^{\tilde{\sigma}}[i](x), \phi_2^{\tilde{\sigma}}[i](x)) \in \pi_1 \tag{4}$$

   By (2), (3) and definition 2.2.1

$$(\phi_1^{\tilde{\sigma}}[i](x), \phi_2^{\tilde{\sigma}}[i](x)) \in \pi_2 \tag{5}$$

2. Base case. Take arbitrary v such that

$$v \in \pi_v \tag{6}$$

   By eq. (2) and definition 2.2.1

$$(v, v) \in \pi_2 \tag{7}$$

3. Inductive case. Take arbitrary $(a_1, a_2), f$ such that

$$(a_1, a_2) \in \pi_1 \tag{8}$$
$$(a_1, a_2) \in \pi_2 \tag{9}$$
$$(\phi_1^h(a_1, f), \phi_2^h(a_2, f)) \in \pi_1 \tag{10}$$

By (2), (9)

$$(\phi_1^h(a_1, f), \phi_2^h(a_2, f)) \in \pi_2 \tag{11}$$

By above

$$\pi_1 \subseteq \pi_2 \tag{12}$$

By symmetric argument

$$\pi_2 \subseteq \pi_1 \tag{13}$$

By (12) and (14)

$$\pi_1 = \pi_2 \tag{14}$$

■

B.2 Compatible with a smaller injection (lemma 2.3.2)

Recall from page 29

Lemma 2.3.2 (Compatible with a smaller injection)

$$\forall \pi_{1\ldots3} : \mathit{com}(\pi_1, \pi_2) \ \wedge \ \pi_3 \subseteq \pi_1 \implies \mathit{com}(\pi_3, \pi_2)$$

Proof B.2.1

Take arbitrary $\pi_{1\ldots3}$ such that

$$\mathit{com}(\pi_1, \pi_2) \tag{1}$$
$$\pi_3 \subseteq \pi_1 \tag{2}$$

To show

$$\mathit{com}(\pi_3, \pi_2)$$

Take arbitrary $a_{1\ldots4}$ such that

$$(a_1, a_3) \in \pi_3 \tag{3}$$
$$(a_2, a_4) \in \pi_2 \tag{4}$$

To show

$$a_1 = a_2 \iff a_3 = a_4$$

By (2)

$$(a_1, a_3) \in \pi_1 \tag{5}$$

By above, (1), (4)

$$a_1 = a_2 \iff a_3 = a_4 \tag{6}$$

■

B.3 Can add compatible element to injection (lemma 2.3.3)

Recall from page 29

Lemma 2.3.3 (Can add compatible element to injection)

$$\forall \pi_{1,2}, a_{1,2} : \ com(\pi_1, \pi_2) \ \wedge (a_1, a_2) \in \pi_2 \implies com(\pi_1 \cup (a_1, a_2), \pi_2)$$

Proof B.3.1
Take arbitrary $\pi_{1,2}, a_{1,2}$ such that

$$com(\pi_1, \pi_2) \tag{1}$$
$$(a_1, a_2) \in \pi_2 \tag{2}$$

To show

$$in(\pi_1 \cup (a_1, a_2) \cup \pi_2)$$

By (2) and definition $\cup$ then $(a_1, a_2) \cup \pi_2 = \pi_2$ so

$$\pi_1 \cup (a_1, a_2) \cup \pi_2 = \pi_1 \cup \pi_2 \tag{3}$$

By (1)

$$in(\pi_1 \cup \pi_2) \tag{4}$$

By above, (3)

$$in(\pi_1 \cup (a_1, a_2) \cup \pi_2) \tag{5}$$

■

B.4 Can add disjoint element to injection (lemma 2.3.4)

Recall from page 30

Lemma 2.3.4 (Can add disjoint element to injection)

$$\forall \pi_{1,2}, a_{1,2}:$$
$$com(\pi_1, \pi_2) \ \wedge \ a_1 \notin dom(\pi_1 \cup \pi_2) \ \wedge \ a_2 \notin rng(\pi_1 \cup \pi_2)$$
$$\implies$$
$$com(\pi_1 \cup (a_1, a_2), \pi_2)$$

Proof B.4.1

Take arbitrary $\pi_{1,2}, a_{1,2}$ such that

$$com(\pi_1, \pi_2) \tag{1}$$
$$a_1 \notin dom(\pi_1 \cup \pi_2) \tag{2}$$
$$a_2 \notin rng(\pi_1 \cup \pi_2) \tag{3}$$

To show

$$com(\pi_1 \cup (a_1, a_2), \pi_2)$$

By (2), (3), (1)

$$in(\pi_1 \cup (a_1, a_2)) \tag{4}$$

By cases

1. $(a_1, a_2) \in \pi_1$

   By case, (definition $\cup$)

   $$\pi_1 \cup (a_1, a_2) = \pi_1 \tag{5}$$

   By above, (1)

   $$com(\pi_1 \cup (a_1, a_2), \pi_2) \tag{6}$$

2. $(a_1, a_2) \notin \pi_1$

   Take arbitrary $a_{3\ldots6}$ such that

   $$\{(a_3, a_4), (a_5, a_6)\} \subseteq \pi_1 \cup (a_1, a_2) \cup \pi_2 \tag{7}$$

   To show

   $$a_3 = a_5 \iff a_4 = a_6$$

   By cases without loss of generality

   a) $(a_3, a_4) = (a_1, a_2) \ \wedge (a_5, a_6) = (a_1, a_2)$

      Trivial

   b) $(a_3, a_4) = (a_1, a_2) \ \wedge (a_5, a_6) \neq (a_1, a_2)$

      By case 2b, (7)

      $$(a_5, a_6) \in \pi_1 \cup \pi_2 \tag{8}$$

      By cases

i. $(a_5, a_6) \in \pi_1$
By (4), case 2b, case 2(b)i

$$a_3 = a_5 \iff a_4 = a_6 \tag{9}$$

ii. $(a_5, a_6) \in \pi_2$ By case, (2), (3)

$$a_5 \neq a_1 \tag{10}$$
$$a_6 \neq a_2 \tag{11}$$

By above, case 2b

$$a_5 \neq a_3 \tag{12}$$
$$a_6 \neq a_4 \tag{13}$$

By above, trivially

$$a_3 = a_5 \iff a_4 = a_6 \tag{14}$$

By above, (8)

$$a_3 = a_5 \iff a_4 = a_6 \tag{15}$$

c) $(a_3, a_4) \neq (a_1, a_2) \wedge (a_5, a_6) \neq (a_1, a_2)$
By case, (7)

$$\{(a_3, a_4), (a_5, a_6)\} \subseteq \pi_1 \cup \pi_2 \tag{16}$$

By above, (1)

$$a_3 = a_5 \iff a_4 = a_6 \tag{17}$$

■

B.5 *reach* is a function (lemma 2.4.2)

Recall from page 30

Lemma 2.4.2 (*reach* is a function)

$$\forall \phi, A_{1,2} : (\phi, A_1) \in \mathit{reach} \wedge (\phi, A_2) \in \mathit{reach} \implies A_1 = A_2$$

Proof B.5.1
Take arbitrary $\phi, A_{1,2}$ such that

$$(\phi, A_1) \in \mathit{reach} \tag{1}$$
$$(\phi, A_2) \in \mathit{reach} \tag{2}$$

To show

$$A_1 = A_2$$

First show without loss of generality that

$$A_1 \subseteq A_2$$

By induction on definition 2.4.1 (Reachable addresses)

1. Base case. Take $a \in A_1, i, x$ such that

$$\phi^\sigma[i](x) = a \tag{3}$$

   By definition 2.4.1 (Reachable addresses) and above

$$a \in A_2 \tag{4}$$

2. Inductive case. Take $a_1 \in A_1, a_2 \in A_1, f$ such that

$$a_1 \in A_2 \tag{5}$$
$$\phi^h(a_1, f) = a_2 \tag{6}$$

   By definition 2.4.1 (Reachable addresses) and above

$$a \in A_2 \tag{7}$$

By (4), (7)

$$A_1 \subseteq A_2 \tag{8}$$

By a symmetric argument

$$A_1 \supseteq A_2 \tag{9}$$

By (8), (9) done. ■

B.6 Expressions evaluate to reachable address (lemma 2.4.3)

Recall from page 31

Lemma 2.4.3 (Expressions evaluate to reachable address)

$$\forall \phi, e, a : [\![e]\!]_\phi = a \implies a \in \mathit{reach}(\phi)$$

Proof B.6.1
Take arbitrary ϕ, e, a such that

$$[\![e]\!]_\phi = a \tag{1}$$

To show

$$a \in \mathit{reach}(\phi)$$

By induction on the structure of e, (1), there exists $x, \overline{f}$ such that

$$\phi(x, \overline{f}) = a \tag{2}$$

By above, lemma 2.4.5 (Paths are reachable)

$$a \in \mathit{reach}(\phi) \tag{3}$$

■

B.7 Path to reachable address (lemma 2.4.4)

Recall from page 31

Lemma 2.4.4 (Path to reachable address)

$$\forall \phi, a \in \mathit{reach}(\phi) : \exists i, x, \overline{f} : \phi^h(\phi^{\tilde{\sigma}}[i](x), \overline{f}) = a \ \wedge \ \mathit{acyclic}(\phi, \phi^{\tilde{\sigma}}[i](x), \overline{f})$$

where

$$\mathit{acyclic}(\phi, a, \overline{f}) \overset{def}{\iff} \nexists \overline{g} \cdot \overline{g}' = \overline{f}, \overline{h} \cdot \overline{h}' = \overline{f} : \overline{g} \neq \overline{h} \ \wedge \ \phi^h(a, \overline{g}) = \phi^h(a, \overline{h})$$

Proof B.7.1
By induction on the definition of *reach* ■

B.8 Paths are reachable (lemma 2.4.5)

Recall from page 31

Lemma 2.4.5 (Paths are reachable)

$$\forall \phi, i, x, \overline{f} : \phi^h(\phi^{\tilde{\sigma}}[i](x), \overline{f}) = a \implies a \in \mathit{reach}(\phi)$$

And

$$\forall \phi, a_1 \in \mathit{reach}(\phi), a_2, \overline{f} : \phi^h(a_1, \overline{f}) = a_2 \implies a_2 \in \mathit{reach}(\phi)$$

Proof B.8.1
By induction on the definition on the length of $\overline{f}$ ■

B.9 Calling context reachability smaller (lemma 2.4.6)

Recall from page 31

Lemma 2.4.6 (Calling context reachability smaller)

$$\forall \phi : \mathit{reach}(\phi^{ctx}) \subseteq \mathit{reach}(\phi)$$

Proof B.9.1
Take arbitrary a_1 such that

$$a_1 \in \mathit{reach}(\phi^{ctx}) \tag{1}$$

To show

$$a_1 \in \mathit{reach}(\phi)$$

By induction on definition *reach*

1. $\exists i, x : (\phi^{ctx})^{\tilde{\sigma}}[i](x) = a_1$ By definition $(\phi^{ctx})^{\tilde{\sigma}}[i](x) = (\phi)^{\tilde{\sigma}}[i](x)$
2. $\exists a_2 \in \mathit{reach}(\phi^{ctx}) : (\phi^{ctx})^h(a_2, f) = a_1$ By definition $\phi^h = (\phi^{ctx})^h$ and by I.H.

■

B.10 Isomorphism implies calling context isomorphism (lemma 2.4.7)

Recall from page 31

Lemma 2.4.7 (Isomorphism implies calling context isomorphism)

$$\forall \phi_{1,2} : \phi_1 \approx_{\pi_1} \phi_2 \implies \phi_1^{ctx} \approx_{\pi_2} \phi_2^{ctx} \wedge \pi_2 = \pi_1 \downarrow_{reach(\phi_1^{ctx})} \cup \pi_v$$

Where $\pi_1 \downarrow_{reach(\phi_1^{ctx})}$ *means the relation produced by restricting the domain of* π_1 *to the reachable addresses of* (ϕ_1^{ctx})

Proof B.10.1

Take arbitrary $\phi_{1\ldots4}, \pi_1$ such that

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{1}$$
$$\phi_3 = \phi_1^{ctx} \tag{2}$$
$$\phi_4 = \phi_2^{ctx} \tag{3}$$

To show:

$$\exists \pi_2 : \phi_3 \approx_{\pi_2} \phi_4$$

By cases:

1. $\left|\phi_1^{\tilde{\sigma}}\right| = 0$

   By (1)

   $$\left|\phi_1^{\tilde{\sigma}}\right| = \left|\phi_2^{\tilde{\sigma}}\right| \tag{4}$$

   By above, case, (2), (2)

   $$\phi_1 = \phi_3 \tag{5}$$
   $$\phi_2 = \phi_4 \tag{6}$$

   By above, (1)

   $$\phi_3 \approx_{\pi_1} \phi_4 \tag{7}$$

   Take[1] $\pi_2 = \pi_1$.

   Trivially by above, lemma 2.4.11 (Domain of isomorphism is reachable addresses)

   $$\pi_2 = \pi_1 \downarrow_{reach(\phi_1^{ctx})} \cup \pi_v \tag{8}$$

   By above, (7), esac.

2. $\left|\phi_1^{\tilde{\sigma}}\right| > 0$

   By case, (1), (2), (2)

   $$\left|\phi_1^{\tilde{\sigma}}\right| - 1 = \left|\phi_2^{\tilde{\sigma}}\right| - 1 = \left|\phi_3^{\tilde{\sigma}}\right| = \left|\phi_4^{\tilde{\sigma}}\right| \tag{9}$$

[1] In fact here $\pi_2 = \pi_1 = \emptyset$, but we happen not to need this fact in the proof.

By above, (1), (2), (2)

$$\forall i \leq \left|\phi_3^{\tilde{\sigma}}\right| \colon dom(\phi_3^{\tilde{\sigma}}[i]) = dom(\phi_4^{\tilde{\sigma}}[i]) \tag{10}$$

Take[2]

$$\pi_2 = \pi_1{\downarrow_{reach(\phi_3)}} \cup \pi_v \tag{11}$$

To show[3]

$$\forall i \leq \left|\phi_3^{\tilde{\sigma}}\right|, x \in dom(\phi_3^{\tilde{\sigma}}[i]) \colon \pi_2(\phi_3^{\tilde{\sigma}}[i](x)) = \phi_4^{\tilde{\sigma}}[i](x)$$

Take arbitrary i, x such that

$$i \leq \left|\phi_3^{\tilde{\sigma}}\right| \tag{12}$$

$$x \in dom(\phi_3^{\tilde{\sigma}}[i]) \tag{13}$$

First show $\phi_3^{\tilde{\sigma}}[i](x) \in \pi_2$

By (2), (2), (9), (10), (12), (13)

$$i \leq \left|\phi_1^{\tilde{\sigma}}\right| \wedge x \in dom(\phi_1^{\tilde{\sigma}}[i]) \wedge \phi_1^{\tilde{\sigma}}[i](x) = \phi_3^{\tilde{\sigma}}[i](x) \tag{14}$$

$$i \leq \left|\phi_2^{\tilde{\sigma}}\right| \wedge x \in dom(\phi_2^{\tilde{\sigma}}[i]) \wedge \phi_2^{\tilde{\sigma}}[i](x) = \phi_4^{\tilde{\sigma}}[i](x) \tag{15}$$

By (1), (14)

$$\phi_3^{\tilde{\sigma}}[i](x) \in \pi_1 \tag{16}$$

By definition 2.4.1 (Reachable addresses), (11), above

$$\phi_3^{\tilde{\sigma}}[i](x) \in reach(\phi_3) \tag{17}$$

$$\phi_3^{\tilde{\sigma}}[i](x) \in \pi_2 \tag{18}$$

Next show $\pi_2(\phi_3^{\tilde{\sigma}}[i](x)) = \phi_4^{\tilde{\sigma}}[i](x)$

By (1), (16)

$$\pi_1(\phi_1^{\tilde{\sigma}}[i](x)) = \phi_2^{\tilde{\sigma}}[i](x) \tag{19}$$

By above, (11), (14), (15), (18)

$$\pi_2(\phi_3^{\tilde{\sigma}}[i](x)) = \phi_4^{\tilde{\sigma}}[i](x) \tag{20}$$

By above

$$\forall i \leq \left|\phi_3^{\tilde{\sigma}}\right|, x \in dom(\phi_3^{\tilde{\sigma}}[i]) \colon \pi_2(\phi_3^{\tilde{\sigma}}[i](x)) = \phi_4^{\tilde{\sigma}}[i](x) \tag{21}$$

[2]This means that π_2 is taken to be π_1 with its domain restricted to the addresses reachable in ϕ_3, extended by `null`, `true`, `false` since it is possible that `null` etc. are unreachable in the calling context

[3]We will argue that ϕ_3 is just ϕ_1 with a stack frame removed, anything in the stack of ϕ_3 is also in the stack of ϕ_1, respectively also for ϕ_4 and ϕ_2

To show

$$\forall a \in \pi_2, f : \pi_2(\phi_3^h(a, f)) = \phi_4^h(\pi_2(a), f)$$

Take arbitrary a, f such that

$$a \in \pi_2 \tag{22}$$

By (2), (2)

$$\phi_1^h = \phi_3^h \tag{23}$$
$$\phi_2^h = \phi_4^h \tag{24}$$

By (11), (22)

$$a \in \mathit{reach}(\phi_3) \tag{25}$$

By above, (2), lemma 2.4.6 (Calling context reachability smaller)

$$a \in \mathit{reach}(\phi_1) \tag{26}$$

By above, (1), lemma 2.4.11

$$\pi_1(\phi_1^h(a, f)) = \phi_2^h(\pi_1(a), f) \tag{27}$$

By above, (23), (24)

$$\pi_1(\phi_3^h(a, f)) = \phi_4^h(\pi_1(a), f) \tag{28}$$

By definition *reach*, (25)

$$\phi_3^h(a, f) \in \pi_2 \tag{29}$$

By above, (11), (22), (28)

$$\pi_2(\phi_3^h(a, f)) = \phi_4^h(\pi_2(a), f) \tag{30}$$

By above

$$\forall a \in \pi_2, f : \pi_2(\phi_3^h(a, f)) = \phi_4^h(\pi_2(a), f) \tag{31}$$

To show π_2 is smallest such relation.

By lemma 2.4.6 (Calling context reachability smaller), lemma 2.4.11 (Domain of isomorphism is reachable addresses), (11)

$$\mathit{dom}(\pi_2) = \mathit{reach}(\phi_3) \cup \mathit{dom}(\pi_v) \tag{32}$$

Assume for contradiction exists π_3 such that

$$\pi_3 \subset \pi_2 \tag{33}$$
$$\phi_3 \approx_{\pi_3} \phi_4 \tag{34}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (34)

$$dom(\pi_3) = reach(\phi_3) \cup dom(\pi_v) \tag{35}$$

By above, (32),(33), lemma 2.4.2 (*reach* is a function) contradiction.

By above

$$\nexists \pi_3 \colon \pi_3 \subset \pi_2 \ \wedge \ \phi_3 \approx_{\pi_3} \phi_4 \tag{36}$$

By (9), (10), (21), (31), (36)

$$\phi_3 \approx_{\pi_2} \phi_4 \tag{37}$$

To show

$$\pi_2 = \pi_1{\downarrow_{reach(\phi_1^{ctx})}} \cup \pi_v$$

By lemma 2.4.6 (Calling context reachability smaller), ,

$$reach(\phi_3) \subseteq reach(\phi_1) \tag{38}$$
$$reach(\phi_4) \subseteq reach(\phi_2) \tag{39}$$

By above, lemma 2.4.11 (Domain of isomorphism is reachable addresses)

$$dom(\pi_2) = reach(\phi_3) \cup dom(\pi_v) \tag{40}$$
$$rng(\pi_2) = reach(\phi_4) \cup rng(\pi_v) \tag{41}$$

By above, (11)

$$\pi_2 \subseteq \pi_1 \tag{42}$$

By above, (11), (38)

$$\pi_2 = \pi_1{\downarrow_{reach(\phi_1^{ctx})}} \cup \pi_v \tag{43}$$

By (37), (43), esac.

By above done. ■

B.11 Reach gets smaller (lemma 2.4.8)

Recall from page 32

Lemma 2.4.8 (Reach gets smaller)

$$\begin{aligned}
&\forall \mathcal{L} \neq \mathcal{A}, s, \phi_{1,3}, i \leq j, a_1 \leq |tr|, k, l \in \{2,3\}: \\
&\quad \phi_1, s \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \ \wedge \ k \leq l \\
&\quad \implies \\
&\quad \left(a_1 \in \mathit{reach}(tr_1[j]{\downarrow}l) \implies a_1 \in \mathit{reach}(tr_1[i]{\downarrow}k) \ \vee \ a_1 \notin tr_1[i]{\downarrow}k^h\right) \wedge \\
&\quad \left(a_1 \notin tr_1[j]{\downarrow}l^h \implies a_1 \notin tr_1[i]{\downarrow}k^h\right)
\end{aligned}$$

Proof B.11.1
By induction on the execution $\phi_1, s \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3$.

1. TRANS By case exists $s_{1,2}, tr_3, tr_5, \phi_5$ such that

$$s = s_1; s_2 \tag{1}$$
$$tr_1 = tr_3 \cdot tr_5 \tag{2}$$
$$\phi_1, s_1 \overset{tr_3}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_5 \tag{3}$$
$$\phi_5, s_2 \overset{tr_5}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \tag{4}$$

By above and I.H. twice

$$\begin{aligned}
&\forall i \leq j \leq |tr_3|, k, l \in \{2,3\}, a_1: \\
&\quad \left(a_1 \in \mathit{reach}(tr_3[j]{\downarrow}l) \implies a_1 \in \mathit{reach}(tr_3[i]{\downarrow}k) \ \vee \ a_1 \notin tr_3[i]{\downarrow}k^h\right) \wedge \\
&\quad \left(a_1 \notin tr_3[j]{\downarrow}l^h \implies a_1 \notin tr_3[i]{\downarrow}k^h\right)
\end{aligned} \tag{5}$$

$$\begin{aligned}
&\forall i \leq j \leq |tr_5|, k, l \in \{2,3\}, a_1: \\
&\quad \left(a_1 \in \mathit{reach}(tr_5[j]{\downarrow}l) \implies a_1 \in \mathit{reach}(tr_5[i]{\downarrow}k) \ \vee \ a_1 \notin tr_5[i]{\downarrow}k^h\right) \wedge \\
&\quad \left(a_1 \notin tr_5[j]{\downarrow}l^h \implies a_1 \notin tr_5[i]{\downarrow}k^h\right)
\end{aligned} \tag{6}$$

Take arbitrary $i \leq j \leq |tr_1|, k, l \in \{2,3\}, a_1$ such that

$$k \leq l \tag{7}$$

To show

$$a_1 \in \mathit{reach}(tr_1[j]{\downarrow}l) \implies a_1 \in \mathit{reach}(tr_1[i]{\downarrow}k) \ \vee \ a_1 \notin tr_1[i]{\downarrow}k^h$$

By cases. The cases for different values of k, l are symmetric, so will show the proof for the $k = 2, l = 2$. Assume $a_1 \in \mathit{reach}(tr_1[j]{\downarrow}l)$

a) $i \leq |tr_3| \ \wedge \ j \leq |tr_3|$ Directly from (5)

b) $i > |tr_3| \ \wedge \ j > |tr_3|$ Directly from (6)

c) $i \leq |tr_3| \ \wedge \ j > |tr_3|$ By (6)

$$a_1 \in \mathit{reach}(tr_5[1]{\downarrow}2) \ \vee \ a_1 \notin tr_5[1]{\downarrow}2^h \tag{8}$$

Since $fst(tr_5) = \phi_5 = lst(tr_3)$ and above

$$a_1 \in reach(tr_3[|tr_3|]\downarrow 3) \ \vee \ a_1 \notin tr_3[|tr+3|]\downarrow 3^h \tag{9}$$

By above and (5).

By above

$$a_1 \in reach(tr_1[j]\downarrow l) \implies a_1 \in reach(tr_1[i]\downarrow k) \ \vee \ a_1 \notin tr_1[i]\downarrow k^h \tag{10}$$

To show

$$a_1 \notin tr_1[j]\downarrow l^h \implies a_1 \notin tr_1[i]\downarrow k^h$$

By cases. The cases for different values of k, l are symmetric, so we show the proof for the $k = 2, l = 2$ assume $a_1 \notin tr_1[j]\downarrow l^h$

a) $i \leq |tr_3| \ \wedge \ j \leq |tr_3|$ Directly from (5)

b) $i > |tr_3| \ \wedge \ j > |tr_3|$ Directly from (6)

c) $i \leq |tr_3| \ \wedge \ j > |tr_3|$

By (6)

$$a_1 \notin tr_5[1]\downarrow 2^h \tag{11}$$

Since $fst(tr_5) = \phi_5 = lst(tr_3)$ and above

$$a_1 \notin tr_3[|tr_3|]\downarrow 2^h \tag{12}$$

By above and (5)

By above

$$a_1 \notin tr_1[j]\downarrow l^h \implies a_1 \notin tr_1[i]\downarrow k^h \tag{13}$$

By (10), (13) esac.

2. CONDT by I.H.
3. CONDF by I.H.
4. ASSIGN by lemma 4.1.8 (ASSIGN reduces *reach*)
5. STORE by lemma 4.1.5 (STORE reduces *reach*)
6. NEW by lemma 4.1.10 (NEW doesn't synthesise existing addresses)
7. ASSUME Does not change store
8. ASSERTT Does not change store
9. ASSERTF Does not change store
10. CALLV by I.H.

■

B.12 Effects are reachable (lemma 2.4.10)

Recall from page 32

Lemma 2.4.10 (Effects are reachable)

$$\begin{aligned}&\forall \mathcal{L} \neq \mathcal{A}, s, \phi_{1,2}, a_{1,2}, f: \\ &\quad \phi_1, s \overset{tr}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_2 \ \wedge (a_1, f, a_2) \in \mathit{effect}(\phi_6, \phi_4) \\ &\quad \implies \\ &\quad (a_1 \in \mathit{reach}(\phi_1) \ \vee \ a_1 \notin \phi_1^h) \ \wedge (a_2 \in \mathit{reach}(\phi_1) \ \vee \ a_2 \notin \phi_1^h)\end{aligned}$$

Proof B.12.1
By induction on the derivation of $\phi_1, s \rightsquigarrow^{tr}_{\mathcal{L}} \phi_2$, lemma 2.4.8 (Reach gets smaller) and lemma 2.4.3 (Expressions evaluate to reachable address) ■

B.13 Domain of isomorphism is reachable addresses (lemma 2.4.11)

Recall from page 32

Lemma 2.4.11 (Domain of isomorphism is reachable addresses)

$$\forall \phi_{1,2}, \pi: \ \phi_1 \approx_\pi \phi_2 \implies \mathit{reach}(\phi_1) \cup \mathit{dom}(\pi_v) = \mathit{dom}(\pi)$$

Proof B.13.1
Take arbitrary $\phi_{1,2}, \pi$ such that

$$\phi_1 \approx_\pi \phi_2 \tag{1}$$

To show

$$\mathit{reach}(\phi_1) \cup \mathit{dom}(\pi_v) = \mathit{dom}(\pi)$$

First show that $\mathit{reach}(\phi_1) \subseteq \mathit{dom}(\pi)$
By induction on the definition of *reach*

$$\begin{aligned}&(\forall a \in \mathit{reach}(\phi_1), i, x: \ \phi_1^\sigma[i](x) = a \implies a \in \pi) \ \wedge \\ &\big(\forall a_1 \in \mathit{reach}(\phi_1), a_2 \in \mathit{reach}(\phi), f: \ a_1 \in \pi \ \wedge \ \phi_1^h(a_1, f) = a_2 \implies a_2 \in \pi\big) \\ &\quad \implies \\ &(\forall a \in \mathit{reach}(\phi_1): \ a \in \pi)\end{aligned}$$

1. Base case. Take a, i, x such that

$$\phi_1^\sigma[i](x) = a \tag{2}$$

$$a \in \mathit{reach}(\phi_1) \tag{3}$$

To show

$$a \in \pi$$

By (1)

$$\pi(\phi_1^\sigma[i](x)) = \phi_2^\sigma[i](x) \tag{4}$$

By above

$$a \in \pi \tag{5}$$

2. Inductive case. Take $a_{1,2}, f$ such that

$$a_1 \in \mathit{reach}(\phi_1) \tag{6}$$
$$a_2 \in \mathit{reach}(\phi_1) \tag{7}$$
$$a_1 \in \pi \tag{8}$$
$$\phi_1^h(a_1, f) = a_2 \tag{9}$$

To show

$$a_2 \in \pi$$

By (1), (8), (9)

$$\pi(\phi_1^h(a_1, f)) = \phi_2^h(\pi(a_1), f) \tag{10}$$

By above

$$a_2 \in \pi \tag{11}$$

By (5), (11)

$$\mathit{reach}(\phi_1) \subseteq \mathit{dom}(\pi) \tag{12}$$

By definition 2.2.1 (Isomorphism), (1)

$$\mathit{dom}(\pi_v) \subseteq \mathit{dom}(\pi) \tag{13}$$

By (12), (13)

$$\mathit{reach}(\phi_1) \cup \mathit{dom}(\pi_v) \subseteq \mathit{dom}(\pi) \tag{14}$$

Now show

$$\mathit{reach}(\phi_1) \supseteq \mathit{dom}(\pi)$$

By induction on the definition of $\approx$

$$\begin{aligned}
&(\forall a \in \pi, i, x : \phi_1^\sigma[i](x) = a \implies a \in \mathit{reach}(\phi_1)) \wedge \\
&\big(\forall a \in \pi, f : a \in \mathit{reach}(\phi) \wedge \phi_1^h(a, f) \in \pi \implies \phi_1^h(a, f) \in \mathit{reach}(\phi_1)\big) \\
&\quad \implies \\
&(\forall a \in \pi : a \in \mathit{reach}(\phi_1))
\end{aligned}$$

1. Base case. Take a, i, x such that

$$\phi_1^\sigma[i](x) = a \tag{15}$$
$$a \in \pi \tag{16}$$

By (15) and (1) and definition 2.4.1 (Reachable addresses)

$$a \in \mathit{reach}(\phi_1) \tag{17}$$

2. Inductive case. Take a, f such that

$$a \in \pi \tag{18}$$
$$a \in reach(\phi) \tag{19}$$
$$\phi_1^h(a, f) \in \pi \tag{20}$$
$$\tag{21}$$

By (19) and (20) and definition 2.4.1 (Reachable addresses)

$$a \in reach(\phi_1) \tag{22}$$

By (17), (22)

$$reach(\phi_1) \subseteq dom(\pi) \tag{23}$$

By definition 2.2.1 (Isomorphism), (1)

$$dom(\pi_v) \subseteq dom(\pi) \tag{24}$$

By (23), (24)

$$reach(\phi_1) \cup dom(\pi_v) \subseteq dom(\pi) \tag{25}$$

By (14), (25) done. ■

B.14 Composition of injections is an injection (lemma 2.5.2)

Recall from page 33

Lemma 2.5.2 (Composition of injections is an injection)

$$\forall \pi_{1,2} : in(\pi_1) \ \wedge \ in(\pi_2) \implies in(\pi_1 \circ \pi_2)$$

Proof B.14.1
Take arbitrary $\pi_{1,2}$ such that

$$in(\pi_1) \tag{1}$$
$$in(\pi_2) \tag{2}$$

To show:

$$\forall (a, b), (c, d) \in (\pi_1 \circ \pi_2) : (a = c) \iff (b = d)$$

Take arbitrary $(a, b), (c, d)$ such that

$$(a, b), (c, d) \in (\pi_1 \circ \pi_2) \tag{3}$$

By above, definition 2.5.1 (Isomorphism composition), exists e, f such that

$$(a, e) \in \pi_1 \tag{4}$$
$$(e, b) \in \pi_2 \tag{5}$$
$$(c, f) \in \pi_1 \tag{6}$$
$$(f, d) \in \pi_2 \tag{7}$$

To show:

$$(a = c) \iff (b = d)$$

First show without loss of generality:

$$(a = c) \implies (b = d)$$

Assume

$$a = c \tag{8}$$

By above and (1), (4), (6)

$$e = f \tag{9}$$

By above and (2), (5), (7)

$$b = d \tag{10}$$

By above

$$a = c \implies b = d \tag{11}$$

By symmetric argument

$$b = d \implies a = c \tag{12}$$

■

B.15 Injection composed with inverse is identity (lemma 2.5.3)

Recall from page 33

Lemma 2.5.3 (Injection composed with inverse is identity)

$$\forall \pi_{1\ldots3} : in(\pi_1) \wedge \pi_2 \subseteq \pi_1 \wedge \pi_3 \subseteq \pi_1^{-1} \implies \pi_2 \circ \pi_3 \subseteq id$$

And also

$$\forall \pi : in(\pi) \implies \pi \circ \pi^{-1} = id{\downarrow_{dom(\pi)}}$$

Proof B.15.1

Part One

Take arbitrary $\pi_{1\ldots3}$ such that

$$in(\pi_1) \tag{1}$$
$$\pi_2 \subseteq \pi_1 \tag{2}$$
$$\pi_3 \subseteq \pi_1^{-1} \tag{3}$$

To show

$$\forall (a, b) \in \pi_2 \circ \pi_3 : a = b$$

Take arbitrary (a, b) such that

$$(a, b) \in \pi_2 \circ \pi_3 \tag{4}$$

By above, definition 2.5.1 (Isomorphism composition), exists e such that

$$(a, e) \in \pi_2 \tag{5}$$
$$(e, b) \in \pi_3 \tag{6}$$

By (2), (5)

$$(a, e) \in \pi_1 \tag{7}$$

By (3), (6)

$$(e, b) \in \pi_1^{-1} \tag{8}$$

By above

$$(b, e) \in \pi_1 \tag{9}$$

By (1), (7), (9)

$$a = b \tag{10}$$

Part Two
Take arbitrary π such that

$$in(\pi) \tag{11}$$

To show:

$$\pi \circ \pi^{-1} = id{\downarrow}_{dom(\pi)}$$

By lemma 2.5.3 (Injection composed with inverse is identity) part 1, (11)

$$\forall (a, b) \in \pi \circ \pi^{-1} \colon a = b \tag{12}$$

To show

$$dom(\pi \circ \pi^{-1}) = rng(\pi \circ \pi^{-1}) = dom(\pi)$$

We will show

$$\forall a \in dom(\pi \circ \pi^{-1}) \implies a \in rng(\pi \circ \pi^{-1})$$
$$\forall a \in rng(\pi \circ \pi^{-1}) \implies a \in dom(\pi \circ \pi^{-1})$$
$$\forall a \in dom(\pi) \implies a \in dom(\pi \circ \pi^{-1})$$
$$\forall a \in dom(\pi \circ \pi^{-1}) \implies a \in dom(\pi)$$

To show $\forall a \in dom(\pi \circ \pi^{-1}) \implies a \in rng(\pi \circ \pi^{-1})$

> Take arbitrary a such that
>
> $$a \in dom(\pi \circ \pi^{-1}) \tag{13}$$

To show

$$a \in rng(\pi \circ \pi^{-1})$$

By (12), (13)

$$a \in rng(\pi \circ \pi^{-1}) \tag{14}$$

To show $\forall a \in rng(\pi \circ \pi^{-1}) \implies a \in dom(\pi \circ \pi^{-1})$

Take arbitrary a such that

$$a \in rng(\pi \circ \pi^{-1}) \tag{15}$$

To show

$$a \in dom(\pi \circ \pi^{-1})$$

By (12), (15)

$$a \in dom(\pi \circ \pi^{-1}) \tag{16}$$

To show $\forall a \in dom(\pi) \implies a \in dom(\pi \circ \pi^{-1})$

Take arbitrary a such that

$$a \in dom(\pi) \tag{17}$$

To show

$$a \in dom(\pi \circ \pi^{-1})$$

By (17) exists b such that

$$(a, b) \in \pi \tag{18}$$
$$(b, a) \in \pi^{-1} \tag{19}$$

By above

$$(a, a) \in \pi \circ \pi^{-1} \tag{20}$$

By above

$$a \in dom(\pi \circ \pi^{-1}) \tag{21}$$

To show $\forall a \in dom(\pi \circ \pi^{-1}) \implies a \in dom(\pi)$

Take arbitrary a such that

$$a \in dom(\pi \circ \pi^{-1}) \tag{22}$$

To show

$$a \in dom(\pi)$$

By (22), (12) exists b

$$a, b \in \pi \tag{23}$$
$$b, a \in \pi^{-1} \tag{24}$$

By (23)

$$a \in dom(\pi) \tag{25}$$

Done. ■

B.16 RIE is sound (theorem 3.2.1)

Recall from page 36

Theorem 3.2.1 (RIE is sound)
Given a mapping between procedure names $\mathcal{E}$, and a pair of procedures $(f_1, f_2) \in \mathcal{E}$:
If

$$\forall (f_3, f_4) \in \mathcal{E} : \ mt_{\mathcal{R}}(f_3, f_4) \ \wedge \ f_3 \overset{A}{\approx} f_4$$

Then

$$f_1 \overset{K}{\approx} f_2$$

Proof B.16.1
Given a procedure equivalence mapping $\mathcal{E}$
Take arbitrary $\mathtt{f}_1, \mathtt{f}_2$ where

$$(\mathtt{f}_1, \mathtt{f}_2) \in \mathcal{E} \tag{1}$$
$$\forall (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \ con(\mathtt{f}_3) \subseteq con(\mathtt{f}_4) \tag{2}$$
$$\forall (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \ \mathtt{f}_3 \overset{A}{\approx} \mathtt{f}_4 \tag{3}$$
$$\forall (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \ mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4) \tag{4}$$

To show

$$\mathtt{f}_1 \overset{K}{\approx} \mathtt{f}_2$$

Take arbitrary $\mathtt{f}_{3,4}$ such that

$$(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} \tag{5}$$

To show

$$\mathtt{f}_3 \overset{R}{\approx} \mathtt{f}_4$$

By (3)

$$\mathtt{f}_3 \overset{A}{\approx} \mathtt{f}_4 \tag{6}$$

By above, lemma 5.3.1 (Modular procedure equivalence)

$$\mathtt{f}_3 \overset{K}{\approx} \mathtt{f}_4 \tag{7}$$

By above, lemma 5.2.1 (Discard non equal isomorphisms)

$$\mathtt{f}_3 \lessapprox \mathtt{f}_4 \tag{8}$$

By above

$$\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \mathtt{f}_3 \lessapprox \mathtt{f}_4 \tag{9}$$

By lemma 5.1.1 (Replace equivalent calls) and (1), (4), (9)

$$\mathtt{f}_1 \lessapprox \mathtt{f}_2 \tag{10}$$

Done.

■

B.17 $\mathcal{BL}$ closed under isomorphism (lemma 4.1.3)

Recall from page 46

Lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism)

All statements are closed under isomorphism.

Moreover, given alternative allocation strategy π_2 such that

$$com(\pi_1, \pi_2) \ \wedge \ \forall(a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h)$$

There exists an isomorphic execution $\phi_2, s \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4$ such that

$$tr_1 \approx_{\pi_2} tr_2 \ \wedge \ \forall(a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h)$$

Proof B.17.1
Part 1 — all relevant executions are isomorphic

First we prove that for all semantics, apart from the abstract semantics, all relevant executions are isomorphic.
Take arbitrary $\mathcal{L}, s_1, \phi_{1\ldots4}, tr_{1,2}, \pi_1$ such that

$$\phi_1, s_1 \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \tag{1}$$

$$\phi_2, s_1 \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4 \tag{2}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{3}$$

$$\mathcal{L} \neq \mathcal{A} \tag{4}$$

To show exists $\pi_{3,5}$ such that

$$\phi_3 \approx_{\pi_3} \phi_4$$

$$tr_1 \approx tr_2$$

$$com(\pi_1, \pi_5)$$

$$effect(\phi_1, \phi_3) \approx_{\pi_5} effect(\phi_2, \phi_4)$$

By strong induction on the derivation of the executions.
I.H.

$$\begin{aligned}
&\forall \phi_{5\ldots8}, tr_{3,4}, \pi_6, s_2 :\\
&\phi_5, s_2 \overset{tr_3}{\rightsquigarrow_{\mathcal{L}}} \phi_7 \ \wedge\ \phi_6, s_2 \overset{tr_4}{\rightsquigarrow_{\mathcal{L}}} \phi_8 \ \wedge\ \left|\phi_5, s_2 \overset{tr_3}{\rightsquigarrow_{\mathcal{L}}} \phi_7\right| < \left|\phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3\right| \ \wedge\ \phi_5 \approx_{\pi_6} \phi_6\\
&\implies \exists \pi_{7,8}\\
&\phi_7 \approx_{\pi_7} \phi_2 \ \wedge\ tr_1 \approx tr_2 \ \wedge\ com(\pi_6, \pi_8) \ \wedge\ \mathit{effect}(\phi_5, \phi_7) \approx_{\pi_8} \mathit{effect}(\phi_6, \phi_8)
\end{aligned} \tag{5}$$

By cases.

1. TRANS

   By case, (1), (2), exists $\phi_{5,6}, s_{2,3}, tr_{3\ldots6}$

   $$s_1 = s_2; s_3 \tag{6}$$
   $$tr_1 = tr_3 \cdot tr_5 \tag{7}$$
   $$tr_2 = tr_4 \cdot tr_6 \tag{8}$$
   $$\phi_1, s_2 \overset{tr_3}{\rightsquigarrow_{\mathcal{L}}} \phi_5, s_3 \overset{tr_5}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{9}$$
   $$\phi_2, s_2 \overset{tr_4}{\rightsquigarrow_{\mathcal{L}}} \phi_6, s_3 \overset{tr_6}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \tag{10}$$

   By above, (3), I.H. exists $\pi_{7,6}$

   $$\phi_5 \approx_{\pi_7} \phi_6 \tag{11}$$
   $$tr_3 \approx tr_4 \tag{12}$$
   $$com(\pi_1, \pi_6) \tag{13}$$
   $$\mathit{effect}(\phi_1, \phi_5) \approx_{\pi_6} \mathit{effect}(\phi_2, \phi_6) \tag{14}$$

   By (11), (9), (10), I.H. exists $\pi_{9,8}$

   $$\phi_3 \approx_{\pi_9} \phi_4 \tag{15}$$
   $$tr_5 \approx tr_6 \tag{16}$$
   $$com(\pi_7, \pi_8) \tag{17}$$
   $$\mathit{effect}(\phi_5, \phi_3) \approx_{\pi_8} \mathit{effect}(\phi_6, \phi_4) \tag{18}$$

   To show exists n such that

   $$\begin{aligned}
   &|tr_1| = |tr_2| = n \ \wedge\ \exists \pi'_1 \ldots \pi'_{2n}\\
   &(\forall i \le n :\ tr_1[i]{\downarrow_2} \approx_{\pi'_{2i-1}} tr_2[i]{\downarrow_2})\\
   &(\forall i \le n :\ tr_1[i]{\downarrow_3} \approx_{\pi'_{2i}} tr_2[i]{\downarrow_3})\\
   &(\forall i, j \le 2n :\ com(\pi'_i, \pi'_j))
   \end{aligned}$$

   By (12), (16), (7), (8)

   $$|tr_3| = |tr_4| \tag{19}$$
   $$|tr_5| = |tr_6| \tag{20}$$

By above, (7), (8), take n such that

$$n = |tr_1| = |tr_2| \tag{21}$$

By (12), (16), (7), (8) take $\pi'_1 \dots \pi'_{2n}$ such that

$$(\forall i \leq n : tr_1[i]\downarrow_2 \approx_{\pi'_{2i-1}} tr_2[i]\downarrow_2) \tag{22}$$

$$(\forall i \leq n : tr_1[i]\downarrow_3 \approx_{\pi'_{2i}} tr_2[i]\downarrow_3) \tag{23}$$

By lemma 2.4.8 (Reach gets smaller), lemma 2.4.11 (Domain of isomorphism is reachable addresses), (12), (16)

$$(\forall i, j \leq 2n : com(\pi'_i, \pi'_j)) \tag{24}$$

By above, (7), (8), (12), (16)

$$tr_1 \approx tr_2 \tag{25}$$

By lemma 2.4.10 (Effects are reachable), lemma 2.4.11 (Domain of isomorphism is reachable addresses), eq. (14), eq. (18) exists π_5 such that

$$effect(\phi_1, \phi_3) \approx_{\pi_5} effect(\phi_2, \phi_4) \tag{26}$$

$$com(\pi_1, \pi_5) \tag{27}$$

By above, (15), (25) esac.

2. ASSIGN

   By lemma 4.1.7 (ASSIGN Closed)

3. STORE

   By lemma 4.1.4 (STORE Closed)

4. NEW

   By lemma 4.1.9 (NEW Closed)

5. ASSUME

   By lemma A.2.1 (ASSUME closed)

6. ASSERTT

   By lemma A.2.2 (ASSERTT closed)

7. ASSERTF

   By lemma A.2.3 (ASSERTF closed)

8. CALLV

By case, (1), (2)

$$s_1 = \texttt{call f}(x_1 \dots x_n) \tag{28}$$

$$\mathit{mkframe}(\phi_1, \texttt{f}, x_1 \dots x_n), \texttt{body f} \rightsquigarrow_{\mathcal{L}} \phi_3 \tag{29}$$

$$\mathit{mkframe}(\phi_2, \texttt{f}, x_1 \dots x_n), \texttt{body f} \rightsquigarrow_{\mathcal{L}} \phi_4 \tag{30}$$

By above and I.H. exists $\pi_{3,5}$ such that

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{31}$$

$$\mathit{com}(\pi_1, \pi_5) \tag{32}$$

$$\mathit{effect}(\phi_1, \phi_3) \approx_{\pi_5} \mathit{effect}(\phi_2, \phi_4) \tag{33}$$

And by (3), (31)

$$tr_1 \approx tr_2 \tag{34}$$

Part 2 - there exists an isomorphic execution

Here we show that given isomorphic initial states, there exists an isomorphic execution with the desired properties. We must show this for both the abstract and concrete semantics of $\mathcal{BL}$. In fact, only the rules for procedure call differ between the two semantics.

Take arbitrary $s, \phi_{1\dots3}, tr_1, \pi_{1,2}, \mathcal{L}$ such that

$$\phi_1, s \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{35}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{36}$$

$$\mathit{com}(\pi_1, \pi_2) \tag{37}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \tag{38}$$

To show exists ϕ_4, π_3 such that

$$\phi_2, s \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4$$

$$\phi_3 \approx_{\pi_3} \phi_4$$

$$\mathit{com}(\pi_3, \pi_2)$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h)$$

Proceed by induction on the structure of the derivation of $\phi_1, s \rightsquigarrow_c^{tr_1} \phi_3$
By cases:

- TRANS

  By case, (35), take $s_1, s_2, tr_3, tr_5, \phi_5$ such that

$$s = s_1; s_2 \tag{39}$$

$$tr_1 = tr_3 \cdot tr_5 \tag{40}$$

$$\phi_1, s_1 \overset{tr_3}{\rightsquigarrow_{\mathcal{C}}} \phi_5 \tag{41}$$

$$\phi_5, s_2 \overset{tr_5}{\rightsquigarrow_{\mathcal{C}}} \phi_3 \tag{42}$$

By I.H. (36), (37), (38), (41) exists, ϕ_6,tr_4, π_4 such that

$$\phi_2, s_1 \overset{tr_4}{\underset{\mathcal{C}}{\rightsquigarrow}} \phi_6 \tag{43}$$

$$\phi_5 \approx_{\pi_4} \phi_6 \tag{44}$$

$$com(\pi_4, \pi_2) \tag{45}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_5^h \iff a_2 \in \phi_6^h) \tag{46}$$

By I.H. (44), (45), (46), (42) exists, ϕ_4,tr_6, π_3 such that

$$\phi_6, s_2 \overset{tr_6}{\underset{\mathcal{C}}{\rightsquigarrow}} \phi_4 \tag{47}$$

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{48}$$

$$com(\pi_3, \pi_2) \tag{49}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{50}$$

Take tr_2 such that

$$tr_2 = tr_4 \cdot tr_6 \tag{51}$$

By above, definition 2.1.1 (Semantics of $\mathcal{BL}$), (39), (43), (47)

$$\phi_2, s \overset{tr_2}{\underset{\mathcal{C}}{\rightsquigarrow}} \phi_4 \tag{52}$$

By (52), (48), (49), (50) esac.

- CONDT

  By case, (35), take b, s_a such that

$$\phi_1 \vDash b \tag{53}$$

$$\phi_1, s_a \overset{tr_1}{\underset{\mathcal{C}}{\rightsquigarrow}} \phi_3 \tag{54}$$

  By I.H. (36), (37), (38), (54) exists, ϕ_4, π_3 such that

$$\phi_2, s_a \underset{\mathcal{C}}{\rightsquigarrow} \phi_4 \tag{55}$$

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{56}$$

$$com(\pi_3, \pi_2) \tag{57}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{58}$$

  By lemma 4.2.1 (Isomorphism is assertion preserving), (53), (36)

$$\phi_2 \vDash b \tag{59}$$

  By above, definition 2.1.1 (Semantics of $\mathcal{BL}$), (55), exists tr_2 such that

$$\phi_2, s \overset{tr_2}{\underset{\mathcal{C}}{\rightsquigarrow}} \phi_4 \tag{60}$$

  By (56), (57), (58), (60), esac.

- CONDF

  By case, (35), take b such that

  $$\neg(\phi_1 \vDash b) \tag{61}$$
  $$\phi_1 = \phi_3 \tag{62}$$

  By lemma 4.2.1 (Isomorphism is assertion preserving), (61), (36)

  $$\phi_2 \vDash b \tag{63}$$

  By above, definition 2.1.1 (Semantics of $\mathcal{BL}$), (55), exists tr_2

  $$\phi_2, s \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_2 \tag{64}$$

  By (36), (37), (38), (64), (62) (take $\phi_4 = \phi_2$ and $\pi_3 = \pi_1$) esac.

- ASSIGN

  By lemma 4.1.7 (ASSIGN Closed), lemma A.2.4 (Consistent with allocated subset)

- STORE

  By lemma 4.1.4 (STORE Closed), lemma A.2.4 (Consistent with allocated subset)

- NEW

  By lemma 4.1.9 (NEW Closed)

- ASSUME

  By lemma A.2.1 (ASSUME closed), lemma A.2.4 (Consistent with allocated subset)

- ASSERTT

  By lemma A.2.2 (ASSERTT closed), lemma A.2.4 (Consistent with allocated subset)

- ASSERTF

  By lemma A.2.3 (ASSERTF closed), lemma A.2.4 (Consistent with allocated subset)

- CALLV

  By case, (35), definition 2.1.1 (Semantics of $\mathcal{BL}$), take $\sigma_{1,3}, \mathtt{f}, x_{1\ldots n}$ such that

  $$s = \mathtt{call\ f}(x_1, \ldots, x_n) \tag{65}$$
  $$\sigma_1 = [\mathcal{V}(\mathtt{f}, 1) \mapsto \phi_1(x_1), \ldots, \mathcal{V}(\mathtt{f}, n) \mapsto \phi_1(x_n)] \tag{66}$$
  $$(\phi_1^{\tilde{\sigma}} \cdot \sigma_1, \phi_1^h), \mathcal{B}(\mathtt{f}) \overset{tr_3}{\underset{\mathcal{C}}{\rightsquigarrow}} (\phi_3^{\tilde{\sigma}} \cdot \sigma_3, \phi_3^h) \tag{67}$$

  By definition 2.2.1 (Isomorphism), (36)

  $$\forall i : \pi_1(\phi_1(x_i)) = \phi_2(x_i) \tag{68}$$

  By definition 2.1.1 (Semantics of $\mathcal{BL}$), (66)

  $$dom(\sigma_1) \subseteq dom(\phi_1^{\tilde{\sigma}}[|\phi_1^{\tilde{\sigma}}|]) \tag{69}$$

By definition 2.2.1 (Isomorphism), (36)

$$dom(\phi_1^{\tilde{\sigma}}[|\phi_1^{\tilde{\sigma}}|]) = dom(\phi_2^{\tilde{\sigma}}[|\phi_2^{\tilde{\sigma}}|]) \tag{70}$$

By (69), (70), (36) take σ_2 such that

$$\sigma_2 = [\mathcal{V}(\mathtt{f}, 1) \mapsto \phi_2(x_1), \ldots, \mathcal{V}(\mathtt{f}, n) \mapsto \phi_2(x_n)] \tag{71}$$

By above

$$dom(\sigma_2) \subseteq dom(\phi_2^{\tilde{\sigma}}[|\phi_2^{\tilde{\sigma}}|]) \tag{72}$$

By (69), (70), (36), lemma 4.1.6 (Isomorphism preserves expression)

$$\pi(\phi_1(x_1)) = \phi_2(x_1) \ \wedge \ \ldots \ \wedge \ \pi(\pi_2(x_n)) = \phi_2(x_n) \tag{73}$$

By above, definition 2.2.1 (Isomorphism), (36), (66), (70), (71)

$$(\phi_1^{\tilde{\sigma}} \cdot \sigma_1, \phi_1^h) \approx_{\pi_1} (\phi_2^{\tilde{\sigma}} \cdot \sigma_2, \phi_2^h) \tag{74}$$

By above, I.H exists, (67), (74)

$$(\phi_2^{\tilde{\sigma}} \cdot \sigma_2, \phi_2^h), \mathcal{B}(\mathtt{f}) \overset{tr_4}{\rightsquigarrow_{\mathcal{C}}} (\phi_4^{\tilde{\sigma}} \cdot \sigma_4, \phi_4^h) \tag{75}$$

$$(\phi_3^{\tilde{\sigma}} \cdot \sigma_3, \phi_3^h) \approx_{\pi_5} (\phi_4^{\tilde{\sigma}} \cdot \sigma_4, \phi_4^h) \tag{76}$$

$$com(\pi_5, \pi_2) \tag{77}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{78}$$

By lemma 2.4.7 (Isomorphism implies calling context isomorphism) take π_3 such that

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{79}$$

$$\pi_3 \subseteq \pi_5 \tag{80}$$

By above, lemma 2.3.2 (Compatible with a smaller injection), (77), (80)

$$com(\pi_3, \pi_2) \tag{81}$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (75) exists tr_2 such that

$$\phi_2, s \overset{tr_2}{\rightsquigarrow_{\mathcal{C}}} \phi_4 \tag{82}$$

By (78), (79), (81), (82), esac.

- CALLA

  By case, the above case CALLV, and lemma A.1.3 ($\mathcal{A}$ semantics overapproximate $\mathcal{K}$ semantics)

By above, exists ϕ_4, π_3 such that

$$\phi_2, s \overset{tr_2}{\rightsquigarrow_{\mathcal{C}}} \phi_4 \tag{83}$$

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{84}$$

$$com(\pi_3, \pi_2) \tag{85}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{86}$$

By **part 1** and lemma A.1.3 ($\mathcal{A}$ semantics overapproximate $\mathcal{K}$ semantics). Executions $\phi_1, s \rightsquigarrow_{\mathcal{L}}^{tr_1} \phi_3$ and $\phi_2, s \rightsquigarrow_{\mathcal{L}}^{tr_2} \phi_4$ are isomorphic. ■

B.18 STORE Closed (lemma 4.1.4)

Recall from page 47

Lemma 4.1.4 (STORE Closed)

$$\forall e_{1,2}, f\colon e_1.f := e_2 \textit{ is closed under isomorphism}$$

Proof B.18.1
Part one - all such executions are isomorphic

Take arbitrary $s_1, \mathcal{L}, e_{1,2}, f, \phi_{1-4}, \pi_1, tr_{1,2}$ such that

$$\mathcal{L} \neq \mathcal{A} \tag{1}$$
$$s_1 = e_1.f := e_2 \tag{2}$$
$$\phi_1, s_1 \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \tag{3}$$
$$\phi_2, s_1 \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4 \tag{4}$$
$$\phi_1 \approx_{\pi_1} \phi_2 \tag{5}$$

To show, there exists $\pi_{3,5}$ such that:

$$\phi_3 \approx_{\pi_3} \phi_4$$
$$\textit{effect}(\phi_1, \phi_3) \approx_{\pi_5} \textit{effect}(\phi_2, \phi_4)$$
$$com(\pi_1, \pi_5)$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (2), (3)

$$\phi_3 = (\phi_1^\sigma, \phi_1^h[([\![e_1]\!]_{\phi_1}, f) \mapsto [\![e_2]\!]_{\phi_1}]) \tag{6}$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (2), (4)

$$\phi_4 = (\phi_2^\sigma, \phi_2^h[([\![e_1]\!]_{\phi_2}, f) \mapsto [\![e_2]\!]_{\phi_2}]) \tag{7}$$

By lemma 4.1.6 (Isomorphism preserves expression), (5), (3), (6)

$$\pi_1([\![e_1]\!]_{\phi_1}) = [\![e_1]\!]_{\phi_2} \tag{8}$$
$$\pi_1([\![e_2]\!]_{\phi_1}) = [\![e_2]\!]_{\phi_2} \tag{9}$$

Take π_3 such that

$$\pi_3 = \pi_1{\downarrow_{reach(\phi_3)}} \cup \pi_v \tag{10}$$

By (6), (7)

$$\phi_1^{\tilde{\sigma}} = \phi_3^{\tilde{\sigma}} \tag{11}$$
$$\phi_2^{\tilde{\sigma}} = \phi_4^{\tilde{\sigma}} \tag{12}$$

By above, (5)

$$\left|\phi_3^{\tilde{\sigma}}\right| = \left|\phi_4^{\tilde{\sigma}}\right| \tag{13}$$
$$\forall i\colon dom(\phi_3^{\tilde{\sigma}}[i]) = dom(\phi_4^{\tilde{\sigma}}[i]) \tag{14}$$

By above, definition 2.4.1 (Reachable addresses), (5), (10)

$$\forall i, x \in \phi_3^{\tilde{\sigma}}[i] \colon \phi_3^{\tilde{\sigma}}[i] \in \pi_3 \ \wedge \ \pi_3(\phi_3^{\tilde{\sigma}}[i](x)) = \phi_4^{\tilde{\sigma}}[i](x) \tag{15}$$

Take arbitrary a, g such that

$$a \in \pi_3 \tag{16}$$

To show

$$\pi_3(\phi_3^h(a, g)) = \phi_4^h(\pi_3(a), g)$$

By cases

1. $g = f \ \wedge \ a = [\![e_1]\!]_{\phi_1}$
   By case, (6)

   $$\phi_3^h(a, g) = [\![e_2]\!]_{\phi_1} \tag{17}$$

   By (9)

   $$[\![e_2]\!]_{\phi_1} \in \pi_1 \tag{18}$$

   By definition 2.4.1 (Reachable addresses), (6), case[a]

   $$[\![e_2]\!]_{\phi_1} \in \mathit{reach}(\phi_3) \tag{19}$$

   By (10), (17), (18), above

   $$[\![e_2]\!]_{\phi_1} \in \pi_3 \tag{20}$$

   By above, (9), (16)

   $$\pi_3([\![e_2]\!]_{\phi_1}) = [\![e_2]\!]_{\phi_2} \tag{21}$$

   Substitute, by above, (17)

   $$\pi_3(\phi_3^h(a, g)) = [\![e_2]\!]_{\phi_2} \tag{22}$$

   By (7)

   $$\phi_4^h([\![e_1]\!]_{\phi_2}, g) = [\![e_2]\!]_{\phi_2} \tag{23}$$

   By (8), (16)

   $$\pi_3([\![e_1]\!]_{\phi_1}) = [\![e_1]\!]_{\phi_2} \tag{24}$$

   Substitute by above, case

   $$\pi_3(a) = [\![e_1]\!]_{\phi_2} \tag{25}$$

   Substitute by above, (23)

   $$\phi_4^h(\pi_3(a), g) = [\![e_2]\!]_{\phi_1} \tag{26}$$

   Substitute by above, (22)

   $$\phi_4^h(\pi_3(a), g) = \pi_3(\phi_3^h(a, g)) \tag{27}$$

2. otherwise

   By case, (6)

$$\phi_1^h(a,g) = \phi_3^h(a,g) \tag{28}$$

   By above, (5), (10), (16)

$$\pi_3(\phi_3^h(a,g)) = \phi_2^h(\pi_3(a),g) \tag{29}$$

   By (5), (10)

$$in(\pi_3) \tag{30}$$

   By above, case, (8)

$$\pi_3(a) \neq [\![e_1]\!]_{\phi_2} \ \vee \ f \neq g \tag{31}$$

   By above, (7)

$$\phi_2^h(\pi_3(a),g) = \phi_4^h(\pi_3(a),g) \tag{32}$$

   Substitute by (29), (32)

$$\pi_3(\phi_3^h(a,g)) = \phi_4^h(\pi_3(a),g) \tag{33}$$

[a] if a is in π_3, then $a \in reach(\phi_3)$, so $\phi_3^h(a,g) \in reach(\phi_3)$, of course perhaps there is no such a

By (16)-(33)

$$\forall a \in \pi_3 \colon \pi_3(\phi_3^h(a,g)) = \phi_4^h(\pi_3(a),g) \tag{34}$$

By lemma 4.1.5 (STORE reduces *reach*), (10)

$$reach(\phi_3) = dom(\pi_3) \tag{35}$$

Proceed by contradiction

   Assume exists π_4 such that

$$\phi_3 \approx_{\pi_4} \phi_4 \tag{36}$$

$$\pi_4 \subset \pi_3 \tag{37}$$

   By (5), (10)

$$in(\pi_3) \tag{38}$$

   By above, (37)

$$dom(\pi_4) \subset dom(\pi_3) \tag{39}$$

   By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (36)

$$reach(\phi_3) = dom(\pi_4) \tag{40}$$

   By above (35), (39), contradiction

By above

$$\nexists \pi_4 : \phi_3 \approx_{\pi_4} \phi_4 \ \wedge \ \pi_4 \subset \pi_3 \tag{41}$$

By (7), (13), (14), (15), (34), (41)

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{42}$$

By (6)

$$\mathit{effect}(\phi_1, \phi_3) = [(\llbracket e_1 \rrbracket_{\phi_1}, f) \mapsto \llbracket e_2 \rrbracket_{\phi_1}] \tag{43}$$

By (7)

$$\mathit{effect}(\phi_2, \phi_4) = [(\llbracket e_1 \rrbracket_{\phi_2}, f) \mapsto \llbracket e_2 \rrbracket_{\phi_2}] \tag{44}$$

Take π_5

$$\pi_5 = [\llbracket e_1 \rrbracket_{\phi_1} \mapsto \llbracket e_1 \rrbracket_{\phi_2}, \llbracket e_2 \rrbracket_{\phi_1} \mapsto \llbracket e_2 \rrbracket_{\phi_2}] \tag{45}$$

By (43), (44), (44)

$$\mathit{effect}(\phi_1, \phi_3) \approx_{\pi_5} \mathit{effect}(\phi_2, \phi_4) \tag{46}$$

By (5), (8), (9), (45)

$$\mathit{com}(\pi_1, \pi_5) \tag{47}$$

By (42), (46), (47)

Part two - there exists an isomorphic execution

Take arbitrary $s_1, L, e_{1,2}, f, \phi_{1-3}, \pi_{1,2}, tr_1$ such that

$$s_1 = e_1.f := e_2 \tag{48}$$

$$\phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\Sigma}} \phi_3 \tag{49}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{50}$$

$$\mathit{com}(\pi_1, \pi_2) \tag{51}$$

$$\mathit{in}(\pi_2) \tag{52}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \tag{53}$$

To show, there exists ϕ_4, π_3, tr_2 such that:

$$\phi_2, e_1.f := e_2 \overset{tr_2}{\rightsquigarrow_{\Sigma}} \phi_4$$

$$\phi_3 \approx_{\pi_3} \phi_4$$

$$\mathit{com}(\pi_2, \pi_3)$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h)$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (48), (49)

$$\phi_3 = (\phi_1^\sigma, \phi_1^h[(\llbracket e_1 \rrbracket_{\phi_1}, f) \mapsto \llbracket e_2 \rrbracket_{\phi_1}]) \tag{54}$$

By lemma 4.1.6 (Isomorphism preserves expression), (50), (49), (54)

$$\pi_1(\llbracket e_1 \rrbracket_{\phi_1}) = \llbracket e_1 \rrbracket_{\phi_2} \tag{55}$$
$$\pi_1(\llbracket e_2 \rrbracket_{\phi_1}) = \llbracket e_2 \rrbracket_{\phi_2} \tag{56}$$

By above, take ϕ_4 such that

$$\phi_4 = (\phi_2^\sigma, \phi_2^h[(\llbracket e_1 \rrbracket_{\phi_2}, f) \mapsto \llbracket e_2 \rrbracket_{\phi_2}]) \tag{57}$$

By above, definition 2.1.1

$$\phi_1, s_1 \overset{tr_1}{\rightsquigarrow_\Sigma} \phi_3 \tag{58}$$

By above, part one, exists π_3 such that

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{59}$$

By lemma 4.1.5 (STORE reduces *reach*), lemma 2.3.2 (Compatible with a smaller injection), (51), (59)

$$com(\pi_2, \pi_3) \tag{60}$$

By lemma 4.1.5 (STORE reduces *reach*), (53)

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{61}$$

By (58), (59), (60) (61) done ■

B.19 STORE reduces *reach* (lemma 4.1.5)

Recall from page 47

Lemma 4.1.5 (STORE reduces *reach*)

$$\forall \phi_{1,3}, e_{1,2}, f : \phi_1, e_1.f := e_2 \hookrightarrow \phi_3 \implies reach(\phi_3) \subseteq reach(\phi_1)$$

Proof B.19.1
Take arbitrary $\phi_{1,3}, e_{1,2}, f$ such that

$$\phi_1, e_1.g := e_2 \hookrightarrow \phi_3 \tag{1}$$

By above, definition 2.1.1 (Semantics of $\mathcal{BL}$)

$$\phi_3 = (\phi_1^\sigma, \phi_1^h[(\llbracket e_1 \rrbracket_{\phi_1}, g) \mapsto \llbracket e_2 \rrbracket_{\phi_1}]) \tag{2}$$

To show

$$reach(\phi_3) \subseteq reach(\phi_1)$$

Take arbitrary a such that

$$a \in \mathit{reach}(\phi_3) \tag{3}$$

To show

$$a \in \mathit{reach}(\phi_1)$$

By (3), lemma 2.4.4 (Path to reachable address), take $i, x, \overline{f}$ such that

$$\phi_3^h(\phi_3^\sigma[i](x), \overline{f}) = a \tag{4}$$

$$\mathit{acyclic}(\phi_3, \phi_3^\sigma[i](x), \overline{f}) \tag{5}$$

By cases

1. $\exists \overline{f_1}, \overline{f_2} \colon \phi_3^h(\phi_3^\sigma[i](x), \overline{f_1}) = [\![e_1]\!]_{\phi_1} \ \wedge \ \overline{f} = \overline{f_1} \cdot g \cdot \overline{f_2}$

   By induction on length $\overline{f_1}$, case, (2), (4), (5)

   $$\phi_1^h(\phi_1^\sigma[i](x), \overline{f_1}) = [\![e_1]\!]_{\phi_1} \tag{6}$$

   By lemma 2.4.3 (Expressions evaluate to reachable address), (2)

   $$[\![e_2]\!]_{\phi_1} \in \mathit{reach}(\phi_1) \tag{7}$$

   By case, (2)

   $$\phi_3^h(\phi_3^\sigma[i](x), \overline{f_1} \cdot g) = [\![e_2]\!]_{\phi_1} \tag{8}$$

   By above, induction on length $\overline{f_2}$, case, (2), (4)

   $$\phi_3^h([\![e_2]\!]_{\phi_1}, \overline{f_2}) = a \tag{9}$$

   By above, (5), (2)

   $$\phi_1^h([\![e_2]\!]_{\phi_1}, \overline{f_2}) = a \tag{10}$$

   By above, lemma 2.4.3 (Expressions evaluate to reachable address), (7)

   $$a \in \mathit{reach}(\phi_1) \tag{11}$$

2. otherwise

   By induction on length $\overline{f}$, case, (2), (4)

   $$\phi_1^h(\phi_1^\sigma[i](x), \overline{f}) = a \tag{12}$$

   By above, lemma 2.4.5 (Paths are reachable)

   $$a \in \mathit{reach}(\phi_1) \tag{13}$$

By (11), (13) done. ■

B.20 Isomorphism preserves expression (lemma 4.1.6)

Recall from page 48

Lemma 4.1.6 (Isomorphism preserves expression)

$$\forall \phi_{1,2}, \pi, v, e \colon \phi_1 \approx_\pi \phi_2 \land v = [\![e]\!]_{\phi_1} \implies \pi(v) = [\![e]\!]_{\phi_2}$$

Proof B.20.1

Take arbitrary $\phi_{1,2}, \pi, v, e_1$ such that

$$\phi_1 \approx_\pi \phi_2 \tag{1}$$
$$v = [\![e_1]\!]_{\phi_1} \tag{2}$$

To show

$$\pi(v) = [\![e_1]\!]_{\phi_2}$$

By induction on the definition of e_1

1. $e_1 = b$

   By case, (1), lemma 4.2.1 (Isomorphism is assertion preserving), definition 2.1.1 (Semantics of $\mathcal{BL}$)

   $$[\![e_1]\!]_{\phi_1} = [\![e_1]\!]_{\phi_2} \tag{3}$$

   By cases

   a) $v = \texttt{true}$

      By case, definition 2.2.1 (Isomorphism), (1)

      $$\pi(\texttt{true}) = \texttt{true} \tag{4}$$

      By case, above, (3)

      $$\pi(v) = [\![e_1]\!]_{\phi_2} \tag{5}$$

   b) $v = \texttt{false}$

      By case, definition 2.2.1 (Isomorphism), (1)

      $$\pi(\texttt{false}) = \texttt{false} \tag{6}$$

      By case, above, (3)

      $$\pi(v) = [\![e_1]\!]_{\phi_2} \tag{7}$$

2. $e_1 = x$

   By case, (1), (2), definition 2.2.1 (Isomorphism)

   $$\pi(\phi_1(x)) = \phi_2(x) \tag{8}$$

   Substitute by above, case, (2),

   $$\pi(v) = [\![e_1]\!]_{\phi_2} \tag{9}$$

3. $e_1 = e_2.f$

   By I.H., definition 2.1.1 (Semantics of $\mathcal{BL}$), (2), exists a such that

   $$a = [\![e_2]\!]_{\phi_1} \tag{10}$$
   $$\pi(a) = [\![e_2]\!]_{\phi_2} \tag{11}$$

   By above, definition 2.1.1 (Semantics of $\mathcal{BL}$), (2), case

   $$[\![e_2.f]\!]_{\phi_1} = \phi_1^h([\![e_2]\!]_{\phi_1}, f) \tag{12}$$

   By (11)

   $$a \in \pi \tag{13}$$

   By above, definition 2.2.1 (Isomorphism), (1)

   $$\pi(\phi_1^h(a, f)) = \phi_2^h(\pi(a), f) \tag{14}$$

   Substitute by above, (11)

   $$\pi(\phi_1^h(a, f)) = \phi_2^h([\![e_2]\!]_{\phi_2}, f) \tag{15}$$

   Substitute by above, (10)

   $$\pi(\phi_1^h([\![e_2]\!]_{\phi_1}, f)) = \phi_2^h([\![e_2]\!]_{\phi_2}, f) \tag{16}$$

   Substitute by above, (12)

   $$\pi([\![e_2.f]\!]_{\phi_1}) = \phi_2^h([\![e_2]\!]_{\phi_2}, f) \tag{17}$$

   By above, definition 2.1.1 (Semantics of $\mathcal{BL}$), case

   $$\pi([\![e_2.f]\!]_{\phi_1}) = [\![e_2.f]\!]_{\phi_2} \tag{18}$$

   Substitute by above, case, (2)

   $$\pi(v) = [\![e_1]\!]_{\phi_2} \tag{19}$$

4. $e_1 = \texttt{null}$

   By definition 2.1.1 (Semantics of $\mathcal{BL}$), case

   $$[\![e_1]\!]_{\phi_1} = \texttt{null} \tag{20}$$
   $$[\![e_1]\!]_{\phi_2} = \texttt{null} \tag{21}$$

   By definition 2.2.1 (Isomorphism), (1)

   $$\pi(\texttt{null}) = \texttt{null} \tag{22}$$

   By (20), (21), (22)

   $$\pi(v) = [\![e_1]\!]_{\phi_2} \tag{23}$$

By above case analysis

$$\pi(v) = [\![e_1]\!]_{\phi_2} \tag{24}$$

■

B.21 ASSIGN Closed (lemma 4.1.7)

Recall from page 48

Lemma 4.1.7 (ASSIGN Closed)

$$\forall x, e : x := e \text{ is closed under isomorphism}$$

Proof B.21.1
Part one - all relevant executions are isomorphic

Take arbitrary $\mathcal{L}, x, e, \phi_{1-4}, \pi_1, tr_{1,2}$ such that

$$\phi_1, x := e \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{1}$$

$$\phi_2, x := e \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \tag{2}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{3}$$

To show exists $\pi_{3,5}$ such that

$$\pi_3 : \phi_3 \approx_{\pi_3} \phi_4$$
$$\mathit{effect}(\phi_1, \phi_3) \approx_{\pi_5} \mathit{effect}(\phi_2, \phi_4)$$
$$\mathit{com}(\pi_1, \pi_5)$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (1)

$$\phi_3 = (\phi_1^\sigma[x \mapsto [\![e]\!]_{\phi_1}], \phi_1^h) \tag{4}$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (2)

$$\phi_4 = (\phi_2^\sigma[x \mapsto [\![e]\!]_{\phi_2}], \phi_2^h) \tag{5}$$

By (1), exists v_1 such that

$$v_1 = [\![e]\!]_{\phi_1} \tag{6}$$

By above, (3), lemma 4.1.6 (Isomorphism preserves expression) exists v_2 such that

$$v_2 = [\![e]\!]_{\phi_2} = \pi_1(v_1) = \pi_1([\![e]\!]_{\phi_1}) \tag{7}$$

By (4), lemma 2.4.3 (Expressions evaluate to reachable address)

$$[\![e]\!]_{\phi_1} \in \mathit{reach}(\phi_1) \tag{8}$$

Take π_3

$$\pi_3 = \pi_1 \downarrow_{\mathit{reach}(\phi_3)} \cup \pi_v \tag{9}$$

By (4), (5)

$$\mathit{dom}(\phi_3^{\tilde{\sigma}}[|\phi_3^{\tilde{\sigma}}|]) = \mathit{dom}(\phi_1^{\tilde{\sigma}}[|\phi_1^{\tilde{\sigma}}|]) \cup \{x\} \tag{10}$$

$$\mathit{dom}(\phi_4^{\tilde{\sigma}}[|\phi_4^{\tilde{\sigma}}|]) = \mathit{dom}(\phi_2^{\tilde{\sigma}}[|\phi_2^{\tilde{\sigma}}|]) \cup \{x\} \tag{11}$$

By above, (3)

$$dom(\phi_3^{\tilde{\sigma}}[|\phi_3^{\tilde{\sigma}}|]) = dom(\phi_4^{\tilde{\sigma}}[|\phi_4^{\tilde{\sigma}}|]) \tag{12}$$

By above, (3), (4), (5)

$$|\phi_3^{\tilde{\sigma}}| = |\phi_4^{\tilde{\sigma}}| \tag{13}$$

$$\forall i\colon\ dom(\phi_3^{\tilde{\sigma}}[i]) = dom(\phi_4^{\tilde{\sigma}}[i]) \tag{14}$$

Take arbitrary y, i such that

$$i \leq |\phi_3^{\tilde{\sigma}}| \ \wedge\ y \in dom(\phi_3^{\tilde{\sigma}}[i]) \tag{15}$$

By above, (13), (14)

$$i \leq |\phi_4^{\tilde{\sigma}}| \ \wedge\ y \in dom(\phi_4^{\tilde{\sigma}}[i]) \tag{16}$$

By cases

1. $i = |\phi_3^{\tilde{\sigma}}| \ \wedge\ y = x$

   By case, (4), (7), (5)

   $$\pi_1(\phi_3^{\tilde{\sigma}}[i](y)) = \phi_4^{\tilde{\sigma}}[i](y) \tag{17}$$

   By above, definition 2.4.1 (Reachable addresses), (8), (9)

   $$\pi_3(\phi_3^{\tilde{\sigma}}[i](y)) = \phi_4^{\tilde{\sigma}}[i](y) \tag{18}$$

2. otherwise

   By case, (4), (5), (15), (16)

   $$\phi_3^{\tilde{\sigma}}[i](y) = \phi_1^{\tilde{\sigma}}[i](y) \tag{19}$$

   $$\phi_4^{\tilde{\sigma}}[i](y) = \phi_2^{\tilde{\sigma}}[i](y) \tag{20}$$

   By above, (3)

   $$\pi_1(\phi_3^{\tilde{\sigma}}[i](y) = \phi_4^{\tilde{\sigma}}[i](y) \tag{21}$$

   By above, definition 2.4.1 (Reachable addresses), (9)

   $$\pi_3(\phi_3^{\tilde{\sigma}}[i](y) = \phi_4^{\tilde{\sigma}}[i](y) \tag{22}$$

By above

$$\forall i \leq |\phi_3^{\tilde{\sigma}}| \ \wedge\ y \in dom(\phi_3^{\tilde{\sigma}}[i])\colon\ \pi_3(\phi_3^{\tilde{\sigma}}[i](y) = \phi_4^{\tilde{\sigma}}[i](y) \tag{23}$$

Take arbitrary a, f such that

$$a \in dom(\pi_3) \tag{24}$$

By (4), (5)

$$\phi_1^h = \phi_3^h \tag{25}$$
$$\phi_2^h = \phi_4^h \tag{26}$$

By (9), (24)

$$a \in dom(\pi_1) \tag{27}$$

By above, (3)

$$\pi_1(\phi_1^h(a, f)) = \phi_2^h(\pi_1(a), f) \tag{28}$$

By above, (25), (26)

$$\pi_1(\phi_3^h(a, f)) = \phi_4^h(\pi_1(a), f) \tag{29}$$

By above

$$\phi_3^h(a, f) \in dom(\pi_1) \tag{30}$$

By above, definition 2.4.1 (Reachable addresses), (9), (24)

$$\phi_3^h(a, f) \in dom(\pi_3) \tag{31}$$

By above, (9), (29)

$$\pi_3(\phi_3^h(a, f)) = \phi_4^h(\pi_1(a), f) \tag{32}$$

By above, (9), (24)

$$\pi_3(\phi_3^h(a, f)) = \phi_4^h(\pi_3(a), f) \tag{33}$$

By above

$$\forall a \in dom(\pi_3), f : \pi_3(\phi_3^h(a, f)) = \phi_4^h(a, f) \tag{34}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (3)

$$dom(\pi_1) = reach\phi_1 \cup dom(\pi_v) \tag{35}$$

By above, lemma 4.1.8 (ASSIGN reduces *reach*), (1), (9)

$$dom(\pi_3) = reach\phi_3 \cup dom(\pi_v) \tag{36}$$

Proceed by contradiction

Assume exists π_4 such that

$$\phi_3 \approx_{\pi_4} \phi_4 \tag{37}$$
$$\pi_4 \subset \pi_3 \tag{38}$$

By (3), (9)

$$in(\pi_3) \tag{39}$$

By above, (38)

$$dom(\pi_4) \subset dom(\pi_3) \tag{40}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (37)

$$reach(\phi_3) \cup dom(\pi_v) = dom(\pi_4) \tag{41}$$

By above, (49), (40), contradiction

By above

$$\nexists \pi_4 : \phi_3 \approx_{\pi_4} \phi_4 \ \wedge \ \pi_4 \subset \pi_3 \tag{42}$$

By (13), (14), (23), (34), (42).

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{43}$$

By (4), (5)

$$\mathit{effect}(\phi_1, \phi_3) = \mathit{effect}(\phi_2, \phi_4) = \emptyset \tag{44}$$

By above, trivially

$$\mathit{effect}(\phi_1, \phi_3) \approx_{\emptyset} \mathit{effect}(\phi_2, \phi_4) \tag{45}$$

$$com(\pi_1, \emptyset) \tag{46}$$

By (43), (45), (46), done.

Part two - there exists an isomorphic execution

Take arbitrary $\mathcal{L}, x, e, \phi_{1-3}, \pi_{1,2}, tr_1$ such that

$$\phi_1, x := e \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{47}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{48}$$

$$com(\pi_1, \pi_2) \tag{49}$$

$$in(\pi_2) \tag{50}$$

$$\phi_3 = (\phi_1^{\sigma}[x \mapsto [\![e]\!]_{\phi_1}], \phi_1^h) \tag{51}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \tag{52}$$

To show

$$\exists \phi_4, \pi_3, tr_2 : \phi_2, x := e \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge \ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge \ com(\pi_2, \pi_3)$$

By (47), (48), exists v_1 such that

$$v_1 = [\![e]\!]_{\phi_1} \tag{53}$$

By above, lemma 4.1.6 (Isomorphism preserves expression) exists v_2 such that

$$v_2 = [\![e]\!]_{\phi_2} = \pi_1(v_1) = \pi_1([\![e]\!]_{\phi_1}) \tag{54}$$

By above take ϕ_4 such that

$$\phi_4 = (\phi_2^\sigma[x \mapsto [\![e]\!]_{\phi_2}], \phi_2^h) \tag{55}$$

By above, definition 2.1.1 (Semantics of $\mathcal{BL}$)

$$\phi_2, x := e \overset{tr_2}{\rightsquigarrow} \phi_4 \tag{56}$$

By above, (48), (47), part one, exists π_3 such that

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{57}$$

By above, lemma 2.4.11 (Domain of isomorphism is reachable addresses)

$$\pi_3 = \pi_1{\downarrow_{reach(\phi_3)}} \cup \pi_v \tag{58}$$

By lemma 4.1.8 (ASSIGN reduces *reach*), (49), (58)

$$com(\pi_2, \pi_3) \tag{59}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{60}$$

By (57), (59), (60), done. ■

B.22 ASSIGN reduces *reach* (lemma 4.1.8)

Recall from page 48

Lemma 4.1.8 (ASSIGN reduces *reach*)

$$\forall \phi_{1,3}, e, x : \ \phi_1, x := e \hookrightarrow \phi_3 \implies reach(\phi_3) \subseteq reach(\phi_1)$$

Proof B.22.1
Take arbitrary $\phi_{1,3}, e, x$ such that

$$\phi_1, x := e \hookrightarrow \phi_3 \tag{1}$$

By above, definition 2.1.1 (Semantics of $\mathcal{BL}$)

$$\phi_3 = (\phi_1^\sigma[x \mapsto [\![e]\!]_{\phi_1}], \phi_1^h) \tag{2}$$

To show

$$reach(\phi_3) \subseteq reach(\phi_1)$$

Take arbitrary a such that

$$a \in reach(\phi_3) \tag{3}$$

To show

$$a \in \mathit{reach}(\phi_1)$$

By (3), lemma 2.4.4 (Path to reachable address), take $i, y, \overline{f}$ such that

$$\phi_3^h(\phi_3^\sigma[i](y), \overline{f}) = a \tag{4}$$

By cases

1. $x = y$

   By lemma 2.4.3 (Expressions evaluate to reachable address), (2)

   $$[\![e]\!]_{\phi_1} \in \mathit{reach}(\phi_1) \tag{5}$$

   By case, (2)

   $$\phi_3^{\tilde{\sigma}}[i](x) = [\![e]\!]_{\phi_1} \tag{6}$$

   By above, case, (2), (4)

   $$\phi_3^h([\![e]\!]_{\phi_1}, \overline{f}) = a \tag{7}$$

   By above, (2)

   $$\phi_1^h([\![e]\!]_{\phi_1}, \overline{f}) = a \tag{8}$$

   By above, lemma 2.4.3 (Expressions evaluate to reachable address), (5)

   $$a \in \mathit{reach}(\phi_1) \tag{9}$$

2. otherwise

   By case, (2), (4)

   $$\phi_1^h(\phi_1^\sigma[i](x), \overline{f}) = a \tag{10}$$

   By above, lemma 2.4.5 (Paths are reachable)

   $$a \in \mathit{reach}(\phi_1) \tag{11}$$

By (9), (11) done. ■

B.23 NEW Closed (lemma 4.1.9)

Recall from page 48

Lemma 4.1.9 (NEW Closed)

$$\forall x : x := \texttt{new}() \textit{ is closed under isomorphism}$$

Proof B.23.1

Part one — all relevant executions are isomorphic

Take arbitrary $\mathcal{L}, x, \phi_{1-4}, \pi_1, tr_{1,2}$ such that

$$\phi_1, x := \texttt{new}() \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{1}$$

$$\phi_2, x := \texttt{new}() \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \tag{2}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{3}$$

To show exists $\pi_{3,5}$ such that

$$\pi_3 : \phi_3 \approx_{\pi_3} \phi_4$$
$$\mathit{effect}(\phi_1, \phi_3) \approx_{\pi_5} \mathit{effect}(\phi_2, \phi_4)$$
$$\mathit{com}(\pi_1, \pi_5)$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (1), take a_1 such that

$$a_1 \notin \mathit{dom}(\phi_1^h) \tag{4}$$

$$a_1 \neq \texttt{null} \tag{5}$$

$$\phi_3 = (\phi_1^\sigma[x \mapsto a_1], \texttt{alloc}(\phi_1^h, a_1)) \tag{6}$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (2), take a_2 such that

$$a_2 \notin \mathit{dom}(\phi_2^h) \tag{7}$$

$$a_2 \neq \texttt{null} \tag{8}$$

$$\phi_4 = (\phi_2^\sigma[x \mapsto a_2], \texttt{alloc}(\phi_2^h, a_2)) \tag{9}$$

Take π_3

$$\pi_3 = \pi_1 \downarrow_{\mathit{reach}(\phi_3)} \cup \pi_v \cup \{a_1 \mapsto a_2\} \tag{10}$$

By (3)

$$\mathit{in}(\pi_1) \tag{11}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (3)

$$\mathit{dom}(\pi_1) = \mathit{reach}(\phi_1) \cup \mathit{dom}(\pi_v) \tag{12}$$

By lemma 4.3.1 (Isomorphism is an equivalence relation) (symmetry), lemma 2.4.11 (Domain of isomorphism is reachable addresses), (3)

$$\mathit{rng}(\pi_1) = \mathit{reach}(\phi_2) \cup \mathit{dom}(\pi_v) \tag{13}$$

By definition 2.4.1 (Reachable addresses), (4), (7), (12), (13)

$$a_1 \notin \mathit{dom}(\pi_1) \tag{14}$$

$$a_2 \notin \mathit{rng}(\pi_1) \tag{15}$$

By above, (10), (11)

$$in(\pi_3) \tag{16}$$

By (6), (9)

$$dom(\phi_3^{\tilde{\sigma}}[|\phi_3^{\tilde{\sigma}}|]) = dom(\phi_1^{\tilde{\sigma}}[|\phi_1^{\tilde{\sigma}}|]) \cup \{x\} \tag{17}$$
$$dom(\phi_4^{\tilde{\sigma}}[|\phi_4^{\tilde{\sigma}}|]) = dom(\phi_2^{\tilde{\sigma}}[|\phi_2^{\tilde{\sigma}}|]) \cup \{x\} \tag{18}$$

By above, (3)

$$dom(\phi_3^{\tilde{\sigma}}[|\phi_3^{\tilde{\sigma}}|]) = dom(\phi_4^{\tilde{\sigma}}[|\phi_4^{\tilde{\sigma}}|]) \tag{19}$$

By above, (3), (6), (9)

$$|\phi_3^{\tilde{\sigma}}| = |\phi_4^{\tilde{\sigma}}| \tag{20}$$
$$\forall i\colon dom(\phi_3^{\tilde{\sigma}}[i]) = dom(\phi_4^{\tilde{\sigma}}[i]) \tag{21}$$

Take arbitrary y, i such that

$$i \leq |\phi_3^{\tilde{\sigma}}| \ \wedge \ y \in dom(\phi_3^{\tilde{\sigma}}[i]) \tag{22}$$

By above, (20), (21)

$$i \leq |\phi_4^{\tilde{\sigma}}| \ \wedge \ y \in dom(\phi_4^{\tilde{\sigma}}[i]) \tag{23}$$

By cases

1. $i = |\phi_3^{\tilde{\sigma}}| \ \wedge \ y = x$
   By case, (6), (9), (10), (16)
   $$\pi_3(\phi_3^{\tilde{\sigma}}[i](y)) = \phi_4^{\tilde{\sigma}}[i](y) \tag{24}$$
2. otherwise
   By case, (6), (9), (22), (23)
   $$\phi_3^{\tilde{\sigma}}[i](y) = \phi_1^{\tilde{\sigma}}[i](y) \tag{25}$$
   $$\phi_4^{\tilde{\sigma}}[i](y) = \phi_2^{\tilde{\sigma}}[i](y) \tag{26}$$
   By above, (3)
   $$\pi_1(\phi_3^{\tilde{\sigma}}[i](y) = \phi_4^{\tilde{\sigma}}[i](y) \tag{27}$$
   By definition 2.4.1 (Reachable addresses), (10), (16)
   $$\pi_3(\phi_3^{\tilde{\sigma}}[i](y) = \phi_4^{\tilde{\sigma}}[i](y) \tag{28}$$

By above

$$\forall i \leq |\phi_3^{\tilde{\sigma}}| \ \wedge \ y \in dom(\phi_3^{\tilde{\sigma}}[i])\colon \pi_3(\phi_3^{\tilde{\sigma}}[i](y) = \phi_4^{\tilde{\sigma}}[i](y) \tag{29}$$

Take arbitrary a, f such that

$$a \in dom(\pi_3) \tag{30}$$

1. $a = a_1$

   By case, (10), (16), (30)

   $$\pi_3(a) = a_2 \tag{31}$$

   By case, above, (6), (9)

   $$\phi_3^h(a, f) = \texttt{null} \tag{32}$$
   $$\phi_4^h(\pi_3(a), f) = \texttt{null} \tag{33}$$

   By (10), (16)

   $$\pi_3(\texttt{null}) = \texttt{null} \tag{34}$$

   By above, (16), (32), (33)

   $$\pi_3(\phi_3^h(a, f)) = \phi_4^h(\pi_3(a), f) \tag{35}$$

2. $a \neq a_1$

   By case, (10), (16), (30)

   $$\pi_3(a) \neq a_2 \tag{36}$$

   By case, (10), (30)

   $$a \in dom(\pi_1) \tag{37}$$
   $$\pi_1(a) = \pi_3(a) \tag{38}$$

   By above, (6), (9), (12), (36)

   $$\phi_1^h(a, f) = \phi_3^h(a, f) \tag{39}$$
   $$\phi_2^h(\pi_1(a), f) = \phi_4^h(\pi_3(a), f) \tag{40}$$

   By case, (36), (37), (3)

   $$\pi_1(\phi_1^h(a, f)) = \phi_2^h(\pi_1(a), f) \tag{41}$$

   By above, (39), (40)

   $$\pi_1(\phi_3^h(a, f)) = \phi_4^h(\pi_3(a), f) \tag{42}$$

   By above

   $$\phi_3^h(a, f) \in dom(\pi_1) \tag{43}$$

   By above, definition 2.4.1 (Reachable addresses), (10), (30)

   $$\phi_3^h(a, f) \in dom(\pi_3) \tag{44}$$

   By above, (10), (42), case

   $$\pi_3(\phi_3^h(a, f)) = \phi_4^h(\pi_3(a), f) \tag{45}$$

By above

$$\forall a \in dom(\pi_3), f : \pi_3(\phi_3^h(a, f)) = \phi_4^h(a, f) \tag{46}$$

By definition 2.4.1 (Reachable addresses), (6)

$$a_1 \in reach(\phi_3) \tag{47}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (3)

$$dom(\pi_1) = reach(\phi_1) \cup dom(\pi_v) \tag{48}$$

By above, lemma 4.1.10 (NEW doesn't synthesise existing addresses), (1), (10), (47)

$$dom(\pi_3) = reach(\phi_3) \cup dom(\pi_v) \tag{49}$$

Proceed by contradiction

> Assume exists π_4 such that
>
> $$\phi_3 \approx_{\pi_4} \phi_4 \tag{50}$$
> $$\pi_4 \subset \pi_3 \tag{51}$$
>
> By above, (51)
>
> $$dom(\pi_4) \subset dom(\pi_3) \tag{52}$$
>
> By lemma 2.4.11 (Domain of isomorphism is reachable addresses), (50)
>
> $$reach(\phi_3) = dom(\pi_4) \cup dom(\pi_v) \tag{53}$$
>
> By above, (16), (52), contradiction

By above

$$\nexists \pi_4 : \phi_3 \approx_{\pi_4} \phi_4 \ \wedge \ \pi_4 \subset \pi_3 \tag{54}$$

By (16), (20), (21), (29), (46), (54)

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{55}$$

By (6), (9)

$$effect(\phi_1, \phi_3) = \{(a_1, f, \texttt{null}) \mid f \in Fid\} \tag{56}$$
$$effect(\phi_2, \phi_4) = \{(a_2, f, \texttt{null}) \mid f \in Fid\} \tag{57}$$

Take π_5

$$\pi_5 = [a_1 \mapsto a_2, \texttt{null} \mapsto \texttt{null}] \tag{58}$$

By above, (56), (56), (58)

$$effect(\phi_1, \phi_3) \approx_{\pi_5} effect(\phi_2, \phi_4) \tag{59}$$

By (4), (7), (3)

$$com(\pi_1, \pi_5) \tag{60}$$

By (55), (59), (60), done.

Part two — isomorphic execution exists

Take arbitrary $\mathcal{L}, x, \phi_{1-3}, \pi_{1,2}, tr_1$ such that

$$\phi_1, x := \texttt{new}() \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \tag{61}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{62}$$

$$com(\pi_1, \pi_2) \tag{63}$$

$$in(\pi_2) \tag{64}$$

$$(\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h)) \tag{65}$$

To show exists ϕ_4, π_3, tr_2 such that

$$\phi_2, x := e \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4$$

$$\phi_3 \approx_{\pi_3} \phi_4$$

$$com(\pi_2, \pi_3)$$

$$(\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h))$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (61), take a_1 such that

$$a_1 \notin dom(\phi_1^h) \tag{66}$$

$$a_1 \neq \texttt{null} \tag{67}$$

$$\phi_3 = (\phi_1^\sigma[x \mapsto a_1], \texttt{alloc}(\phi_1^h, a_1)) \tag{68}$$

To show exists a_2 such that

$$a_2 \notin dom(\phi_2^h)$$

$$a_2 \neq \texttt{null}$$

$$a_1 \in \pi_2 \implies a_2 = \pi_2(a_1)$$

$$a_1 \notin \pi_2 \implies a_2 \notin rng(\pi_2)$$

By cases

1. $a_1 \in \pi_2$

   Because $Addr$ is infinite and enumerable, by case, (65), there exists some address a_2 such that

   $$a_2 \notin dom(\phi_2^h) \tag{69}$$

   $$a_2 \neq \texttt{null} \tag{70}$$

   $$a_2 = \pi_2(a_1) \tag{71}$$

2. $a_1 \notin \pi_2$

Because *Addr* is infinite and enumerable there exists some address a_2 such that

$$a_2 \notin dom(\phi_2^h) \cup rng(\pi_2) \tag{72}$$

$$a_2 \neq \texttt{null} \tag{73}$$

By above take a_2 such that

$$a_2 \notin dom(\phi_2^h) \tag{74}$$

$$a_2 \neq \texttt{null} \tag{75}$$

$$a_1 \in \pi_2 \implies a_2 = \pi_2(a_1) \tag{76}$$

$$a_1 \notin \pi_2 \implies a_2 \notin rng(\pi_2) \tag{77}$$

Take ϕ_4 such that

$$\phi_4 = (\phi_2^\sigma[x \mapsto a_2], \texttt{alloc}(\phi_2^h, a_2)) \tag{78}$$

By above, definition 2.1.1 (Semantics of $\mathcal{BL}$), (74), (75), take tr_2 such that

$$\phi_2, x := e \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4 \tag{79}$$

By part one, (61), (79), exists π_3 such that

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{80}$$

$$\pi_3 = \pi_1 \downarrow_{reach(\phi_3)} \cup \pi_v \cup \{a_1 \mapsto a_2\} \tag{81}$$

By cases

1. $a_1 \in \pi_2$
   By case, (76)

   $$a_2 = \pi_2(a_1) \tag{82}$$

   By lemma 2.3.2 (Compatible with a smaller injection), (63), (81)

   $$com(\pi_3 \setminus (a_1, a_2), \pi_2) \tag{83}$$

   By lemma 2.3.3 (Can add compatible element to injection), above, case, (82)

   $$com(\pi_3, \pi_2) \tag{84}$$

2. $a_1 \notin \pi_2$
   By case (77)

   $$a_2 \notin rng(\pi_2) \tag{85}$$

   By lemma 2.3.2 (Compatible with a smaller injection), (63), (81)

   $$com(\pi_3 \setminus (a_1, a_2), \pi_2) \tag{86}$$

   By above, case, (85), lemma 2.3.4 (Can add disjoint element to injection)

   $$com(\pi_3, \pi_2) \tag{87}$$

By above

$$com(\pi_2, \pi_3) \tag{88}$$

Take arbitrary (a_3, a_4) such that

$$(a_3, a_4) \in \pi_2 \tag{89}$$

To show:

$$a_3 \in \phi_3^h \iff a_4 \in \phi_4^h$$

Without loss of generality, show $a_3 \in \phi_3^h \implies a_4 \in \phi_4^h$

Assume

$$a_3 \in \phi_3^h \tag{90}$$

By above, (68)

$$a_3 \in \phi_1^h \vee a_3 = a_1 \tag{91}$$

By cases

1. $a_3 = a_1$

   By (89), (88), (81)

   $$a_4 = a_2 \tag{92}$$

   By above, lemma 2.4.11 (Domain of isomorphism is reachable addresses), (80)

   $$a_4 \in \phi_4^h \tag{93}$$

2. $a_3 \in \phi_1^h$

   By case, (65)

   $$a_4 \in \phi_2^h \tag{94}$$

   By above, (78)

   $$a_4 \in \phi_4^h \tag{95}$$

By above, (90)

$$a_3 \in \phi_3^h \implies a_4 \in \phi_4^h \tag{96}$$

By symmetric argument

$$a_4 \in \phi_4^h \implies a_3 \in \phi_3^h \tag{97}$$

By above

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_3^h \iff a_2 \in \phi_4^h) \tag{98}$$

By (80), (88), (98), done. ■

B.24 NEW doesn't synthesise existing addresses (lemma 4.1.10)

Recall from page 49

Lemma 4.1.10 (NEW doesn't synthesise existing addresses)

$$\begin{aligned} &\forall \phi_{1,3}, x, a : \\ &\quad \phi_1, x := \mathit{new} \hookrightarrow \phi_3 \ \wedge \\ &\qquad \implies \mathit{reach}(\phi_3) \subseteq \mathit{reach}(\phi_1) \cup \{a, \mathit{null}\} \end{aligned}$$

Proof B.24.1

Take arbitrary $\phi_{1,3}, x, a_1$ such that

$$\phi_1, x := \texttt{new} \hookrightarrow \phi_3 \tag{1}$$

$$\phi_3 = (\phi_1^\sigma[x \mapsto a_1], \texttt{alloc}(\phi_1^h, a_1)) \tag{2}$$

To show

$$\mathit{reach}(\phi_3) \subseteq \mathit{reach}(\phi_1) \cup \{a_1, \texttt{null}\}$$

Take arbitrary a_2 such that

$$a_2 \in \mathit{reach}(\phi_3) \tag{3}$$

To show

$$a_2 \in \mathit{reach}(\phi_1) \cup \{a_1, \texttt{null}\}$$

By lemma 2.4.4 (Path to reachable address), (3), take arbitrary $i, y, \overline{f}$ such that

$$\phi_3^h(\phi_3^{\tilde{\sigma}}[i](y), \overline{f}) = a_2 \tag{4}$$

$$\mathit{acyclic}(\phi_3, \phi_3^{\tilde{\sigma}}[i](y), \overline{f}) \tag{5}$$

By cases

1. $i = \left|\phi_3^{\tilde{\sigma}}\right| \ \wedge \ y = x$

   By cases

   a) $\left|\overline{f}\right| = 0$

      By case, (2), (4)

      $$\phi_3^h(\phi_3^{\tilde{\sigma}}[i](y), \overline{f}) = \phi_3(x) = a_1 = a_2 \tag{6}$$

      By definition 2.4.1 (Reachable addresses), above

      $$a_2 \in \{a_1, \texttt{null}\} \tag{7}$$

b) otherwise
By case, (2), (4)

$$\phi_3^h(\phi_3^{\tilde{\sigma}}[i](y), \overline{f}) = \texttt{null} = a_2 \tag{8}$$

By above

$$a_2 \in \{a_1, \texttt{null}\} \tag{9}$$

2. otherwise
By case, (2)

$$\phi_3^{\tilde{\sigma}}[i](y) = \phi_1^{\tilde{\sigma}}[i](y) \tag{10}$$

By induction on $\overline{f}$, definition 2.1.1 (Semantics of $\mathcal{BL}$) (def. `alloc`), above, (2)

$$\forall \overline{g} \colon \overline{g} \leq \overline{f} \implies \phi_1^h(\phi_1^{\tilde{\sigma}}[i](y), \overline{g}) = \phi_3^h(\phi_3^{\tilde{\sigma}}[i](y), \overline{g}) \tag{11}$$

By above, case, (4)

$$\phi_1^h(\phi_1^{\tilde{\sigma}}[i](y), \overline{f}) = a_2 \tag{12}$$

By above, lemma 2.4.5 (Paths are reachable)

$$a_2 \in \mathit{reach}(\phi_1) \tag{13}$$

By above

$$a_2 \in \mathit{reach}(\phi_1) \cup \{a_1, \texttt{null}\} \tag{14}$$

■

B.25 Isomorphism is assertion preserving (lemma 4.2.1)

Recall from page 49

Lemma 4.2.1 (Isomorphism is assertion preserving)

$$\forall \phi_1, \phi_2, b \colon \phi_1 \approx \phi_2 \implies (\phi_1 \vDash b \iff \phi_2 \vDash b)$$

Proof B.25.1
Take arbitrary ϕ_1, ϕ_2, π, b, such that

$$\phi_1 \approx_\pi \phi_2 \tag{1}$$

Proceed by induction on the structure of b.

- Bases cases:

  1. $b =$ x=y

a) $\phi_1(x) = \phi_1(y)$
By definition 2.2.1 (Isomorphism):

$$\pi(\phi_1(x)) = \phi_2(x) \tag{2}$$
$$\pi(\phi_1(y)) = \phi_2(y) \tag{3}$$

Hence $\phi_2(x) = \phi_2(y)$

b) $\neg(\phi_1(x) = \phi_1(y))$
By definition 2.2.1 (Isomorphism)

$$\pi(\phi_1(x)) = \phi_2(x) \tag{4}$$
$$\pi(\phi_1(y)) = \phi_2(y) \tag{5}$$

Hence $\neg(\phi_2(x) = \phi_2(y))$

2. $b = \mathtt{x}$

a) $\phi_1(x) = \mathtt{true}$
By definition 2.2.1 (Isomorphism)

$$\pi(\phi_1(x)) = \phi_2(x) \tag{6}$$
$$\pi(\mathtt{true}) = \mathtt{true} \tag{7}$$

Hence $\phi_2(x) = \mathtt{true}$

b) $\phi_1(x) = \mathtt{false}$
By definition 2.2.1 (Isomorphism)

$$\pi(\phi_1(x)) = \phi_2(x) \tag{8}$$
$$\pi(\mathtt{false}) = \mathtt{false} \tag{9}$$

Hence $\phi_2(x) = \mathtt{false}$

- Inductive cases.

1. $b = \mathtt{!}\, b'$
I.H. $\phi_1 \vDash b' \iff \phi_2 \vDash b'$
To show: $\phi_1 \vDash \mathtt{!}\, b' \iff \phi_2 \vDash \mathtt{!}\, b'$
By I.H. and by def. !

2. $b = b' \mathtt{\&\&} b''$
I.H. $(\phi_1 \vDash b' \iff \phi_2 \vDash b') \wedge (\phi_1 \vDash b'' \iff \phi_2 \vDash b'')$
By I.H. and by def. &&

■

B.26 Isomorphism is an equivalence relation (lemma 4.3.1)

Recall from page 49

Lemma 4.3.1 (Isomorphism is an equivalence relation)
It is transitive

$$\forall \phi_{1\ldots3}, \pi_{1,2} : \phi_1 \approx_{\pi_1} \phi_2 \ \wedge \ \phi_2 \approx_{\pi_2} \phi_3 \implies \phi_1 \approx_{\pi_1 \cdot \pi_2} \phi_3$$

It is reflexive

$$\forall \phi : \phi \approx_{id} \phi$$

It is symmetric

$$\forall \phi_{1,2}, \pi : \phi_1 \approx_{\pi} \phi_2 \implies \phi_2 \approx_{\pi^{-1}} \phi_1$$

Proof B.26.1
We write $a \in \pi$ to mean that a is in the domain of π. Now we prove that $\approx$ is an equivalence in three parts, first we prove reflexivity, then symmetry, and then finally transitivity.

Reflexivity
To Show:

$$\forall \tilde{\sigma}, h : \exists \pi : (\tilde{\sigma}, h) \approx_{\pi} (\tilde{\sigma}, h)$$

Consider arbitrary $\tilde{\sigma}, h$ and taking π to be the identity function restricted to the reachable addresses of ϕ:

$$\pi = id{\downarrow_{reach(\phi)}} \cup \pi_v \tag{1}$$

By def. cardinality

$$|\tilde{\sigma}| = |\tilde{\sigma}| \tag{2}$$

By *dom* is a function

$$\forall i : 1 \leq i \leq |\tilde{\sigma}| \implies dom(\tilde{\sigma}_i) = dom(\tilde{\sigma}_i) \tag{3}$$

By *dom* is a function, and (1)

$$\forall i \leq |\tilde{\sigma}| : \forall x \in dom(\tilde{\sigma}) : \pi(\tilde{\sigma}[i](x)) = \tilde{\sigma}[i](x) \tag{4}$$

By (1)

$$in(\pi) \tag{5}$$

To show:

$$\forall a \in dom(\pi), f : \pi(h(a, f)) = h(\pi(a, f))$$

Consider arbitrary a, f such that

$$a \in dom(\pi) \tag{6}$$

By above, (1) and definition 2.4.1 (Reachable addresses)

$$\phi^h(a, f) \in dom(\pi) \tag{7}$$

By above and (1)

$$h(a, f) = \pi(h(a, f)) = h(\pi(a), f) \tag{8}$$

To show π is the smallest such relation. Proceed by contradiction.

Assume $\exists \pi'$ such that

$$\pi' \subset \pi \tag{9}$$

$$(\tilde{\sigma}, h) \approx_{\pi'} (\tilde{\sigma}, h) \tag{10}$$

By (1)

$$dom(\pi) = reach(\tilde{\sigma}, h) \cup dom(\pi_v) \tag{11}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses) and (10)

$$dom(\pi') = reach(\tilde{\sigma}, h) \cup dom(\pi_v) \tag{12}$$

By lemma 2.4.2 (*reach* is a function), (9), (11), (12), contradiction

By above

$$\pi \text{ is the smallest such relation} \tag{13}$$

By (2), (3), (4), (5), (8), and (13), done.

Symmetry
To show:

$$\forall \phi_{1,2}, \pi : \ \phi_1 \approx_\pi \phi_2 \implies \phi_2 \approx_{\pi^{-1}} \phi_1$$

Consider arbitrary $\phi_{1,2}, \pi$ such that

$$\phi_1 \approx_\pi \phi_2 \tag{14}$$

To show:

$$\phi_2 \approx_{\pi^{-1}} \phi_1$$

By (14) and symmetry =:

$$\left|\phi_2^{\tilde{\sigma}}\right| = \left|\phi_1^{\tilde{\sigma}}\right| \tag{15}$$

$$\forall i : dom(\phi_2^{\tilde{\sigma}}[i]) = dom(\phi_1^{\tilde{\sigma}}[i]) \tag{16}$$

By (14) ($in(\pi)$)

$$in\left(\pi^{-1}\right) \tag{17}$$

To show:

$$\forall i, x : \ \pi^{-1}(\phi_2^{\tilde{\sigma}}[i](x)) = \phi_1^{\tilde{\sigma}}[i](x)$$

Take i, x, a_1 such that

$$a_1 = \phi_2^{\tilde{\sigma}}[i](x) \tag{18}$$

By (15), (16) there exists a_2 such that

$$a_2 = \phi_1^{\tilde{\sigma}}[i](x) \tag{19}$$

By (14), (18), (19)

$$(a_2, a_1) \in \pi \tag{20}$$

By above and definition π^{-1}

$$(a_1, a_2) \in \pi^{-1} \tag{21}$$

By above and (19)

$$\pi^{-1}(a_1) = \phi_1^{\tilde{\sigma}}[i](x) \tag{22}$$

To show:

$$\forall a_1 \in \pi^{-1}, f\colon \pi^{-1}(\phi_2^h(a_1, f)) = \phi_1^h(\pi^{-1}(a_1), f)$$

Take arbitrary a_1, f such that

$$a_1 \in \pi^{-1} \tag{23}$$

By above, (17) exists a_2

$$(a_1, a_2) \in \pi^{-1} \tag{24}$$
$$(a_2, a_1) \in \pi \tag{25}$$

By (25), (14)

$$\pi(\phi_1^h(a_2, f)) = \phi_2^h(\pi(a_2), f) \tag{26}$$

By above

$$\phi_1^h(a_2, f), \phi_2^h(\pi(a_2), f) \in \pi \tag{27}$$

By above and (17)

$$\phi_2^h(\pi(a_2), f), \phi_1^h(a_2, f) \in \pi^{-1} \tag{28}$$

By above and (24), (25) (subst $\pi^-1(a_1)$ for a_2 and a_1 for $\pi(a_2)$)

$$\phi_2^h(a_1, f), \phi_1^h(\pi^{-1}(a_1), f) \in \pi^{-1} \tag{29}$$

To show π^{-1} is the smallest such relation. Proceed by contradiction.

> Assume exists π_2 such that
>
> $$\phi_2 \approx_{\pi_2} \phi_1 \tag{30}$$
> $$\pi_2 \subset \pi^{-1} \tag{31}$$
>
> By lemma 2.4.11 (Domain of isomorphism is reachable addresses) and (30)
>
> $$dom(\pi_2) = reach(\phi_2) \cup dom(\pi_v) \tag{32}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses) and (14)

$$dom(\pi^{-1}) = reach(\phi_2) \cup dom(\pi_v) \tag{33}$$

By (17), (31), (32), (33) and lemma 2.4.2 (*reach* is a function)

$$dom(\pi_2) \subset dom(\pi^{-1}) \tag{34}$$

By (32), (33), (34), contradiction.

By above

$$\nexists \pi_2 \colon \pi_2 \subset \pi^{-1} \ \wedge \ \phi_2 \approx_{\pi_2} \phi_1 \tag{35}$$

By above and (15), (16), (17), (22), (29) done.

Transitivity

To show:

$$\forall \phi_{1\ldots3} \colon (\exists \pi_1 \colon \phi_1 \approx_{\pi_1} \phi_2) \ \wedge \ (\exists \pi_2 \colon \phi_2 \approx_{\pi_2} \phi_3, h_3) \implies \exists \pi_3 \colon \phi_1 \approx_{\pi_3} \phi_3$$

Take arbitrary $\phi_{1\ldots3}.\pi_{1,2}$ such that

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{36}$$

$$\phi_2 \approx_{\pi_2} \phi_3 \tag{37}$$

To show:

$$\exists \pi_3 \colon \phi_1 \approx_{\pi_3} \phi_3$$

By def. $\approx$ and above

$$\left|\phi_1^{\tilde{\sigma}}\right| = \left|\phi_2^{\tilde{\sigma}}\right| = \left|\phi_3^{\tilde{\sigma}}\right| \tag{38}$$

$$\forall i \colon dom(\phi_1^{\tilde{\sigma}}[i]) = dom(\phi_2^{\tilde{\sigma}}[i]) = dom(\phi_3^{\tilde{\sigma}}[i]) \tag{39}$$

$$in(\pi_1) \tag{40}$$

$$in(\pi_2) \tag{41}$$

By symmetry $\approx$, lemma 2.4.2 (*reach* is a function) and (36), (37)

$$rng(\pi_1) = reach(\phi_2) \tag{42}$$

$$dom(\pi_2) = reach(\phi_2) \tag{43}$$

By above, lemma 2.5.2 (Composition of injections is an injection), and (40), (41)

$$\exists \pi_3 \colon \pi_3 = \pi_1 \circ \pi_2 \tag{44}$$

$$in(\pi_1 \circ \pi_2) \tag{45}$$

$$\pi_3(\texttt{null}) = \texttt{null} \ \wedge \ \pi_3(\texttt{true}) = \texttt{true} \ \wedge \ \pi_3(\texttt{false}) = \texttt{false} \tag{46}$$

By (44) take π_3 such that

$$\pi_3 = \pi_1 \circ \pi_2 \tag{47}$$

To show

$$\left(\forall i, x, a \colon a = \phi_1^{\tilde{\sigma}}[i](x) \implies \pi_3(a) = \phi_3^{\tilde{\sigma}}[i](x)\right)$$

Take i, x, a such that

$$a = \phi_1^{\tilde{\sigma}}[i](x) \tag{48}$$

To show

$$\pi_3(a) = \phi_3^{\tilde{\sigma}}[i](x))$$

By (36), (37), (39), (48)

$$\pi_1(\phi_1^{\tilde{\sigma}}[i](x)) = \phi_2^{\tilde{\sigma}}[i](x) \tag{49}$$
$$\pi_2(\phi_2^{\tilde{\sigma}}[i](x)) = \phi_3^{\tilde{\sigma}}[i](x) \tag{50}$$

By substitution in the above

$$\pi_2(\pi_1(\phi_1^{\tilde{\sigma}}[i](x))) = \phi_3^{\tilde{\sigma}}[i](x) \tag{51}$$

By above and (47)

$$\pi_3(\phi_1^{\tilde{\sigma}}[i](x))) = \phi_3^{\tilde{\sigma}}[i](x) \tag{52}$$

To show

$$\big(\forall a \in \pi_3, f\colon\ \pi_3(\phi_1^h(a, f)) = \phi_3^h(\pi_3(a), f)\big)$$

Take arbitrary a_1, f such that

$$a_1 \in \pi_3 \tag{53}$$

To show

$$\pi_3(\phi_1^h(a_1, f)) = \phi_3^h(\pi_3(a_1), f)$$

By (47), (53) take a_2, a_3 such that

$$\pi_1(a_1) = a_2 \tag{54}$$
$$\pi_2(a_2) = a_3 \tag{55}$$
$$\pi_3(a_1) = a_3 \tag{56}$$

By (36), (54)

$$\pi_1(\phi_1^h(a_1, f)) = \phi_2^h(\pi_1(a_1), f) \tag{57}$$

By (37), (55)

$$\pi_2(\phi_2^h(a_2, f)) = \phi_3^h(\pi_2(a_2), f) \tag{58}$$

By above and (54), (55), (56) (subst $\pi_1(a_1)$ for a_2, and $\pi_3(a_1)$ for $\pi_2(a_2)$)

$$\pi_2(\phi_2^h(\pi_1(a_1), f)) = \phi_3^h(\pi_3(a_1), f) \tag{59}$$

Substituting (57) into above

$$\pi_2(\pi_1(\phi_1^h(a_1, f)) = \phi_3^h(\pi_3(a_1), f) \tag{60}$$

By above and (47)

$$\pi_3(\phi_1^h(a_1, f)) = \phi_3^h(\pi_3(a_1), f) \tag{61}$$

To show π_3 is smallest. Proceed by contradiction

Assume exists π_4 such that

$$\pi_4 \subset \pi_3 \tag{62}$$

$$\phi_1 \approx_{\pi_4} \phi_3 \tag{63}$$

By lemma 2.4.11 (Domain of isomorphism is reachable addresses)

$$dom(\pi_4) = reach(\phi_1) \cup dom(\pi_v) \tag{64}$$

By (47), (36), lemma 2.4.11 (Domain of isomorphism is reachable addresses)

$$dom(\pi_3) = reach(\phi_1) \cup dom(\pi_v) \tag{65}$$

By (45), (62)

$$dom(\phi_3) \subset dom(()\phi_4) \tag{66}$$

By (64), (65), (66) contradiction

By above

$$\nexists \pi_4 : \ \pi_4 \subset \pi_3 \ \wedge \ \phi_1 \approx_{\pi_4} \phi_3 \tag{67}$$

By above and (38), (39), (45), (52), (61), done. ■

B.27 Replace equivalent calls (lemma 5.1.1)

Recall from page 52

Lemma 5.1.1 (Replace equivalent calls)
Given a mapping between procedure names $\mathcal{E}$:

$$\forall(\mathtt{f}_1, \mathtt{f}_2) \in \mathcal{E} : \ (\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \ \mathtt{f}_3 \lessapprox \mathtt{f}_4 \ \wedge \ mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4)) \implies \mathtt{f}_1 \lessapprox \mathtt{f}_2$$

Proof B.27.1

Auxiliary lemma one - whenever a statement is executed to completion under both semantics, if the

initial stores are isomorphic then the final stores will also be isomorphic.

$$\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \mathtt{f}_3 \mathrel{\mathbb{R}} \mathtt{f}_4 \ \wedge \ mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4)$$
$$\implies$$
$$\Big(\forall s_1, \phi_{1\ldots4} : \phi_1, s_1 \rightsquigarrow \phi_3 \ \wedge \ \phi_2, ren_{\mathcal{E}}(s_1) \rightsquigarrow \phi_4 \ \wedge \ \phi_1 \approx \phi_2 \implies \phi_3 \approx \phi_4 \quad \Big)$$

Note that `body` is not a statement, and cannot appear in the program text.
Assume

$$\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \mathtt{f}_3 \mathrel{\mathbb{R}} \mathtt{f}_4 \tag{1}$$
$$\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4) \tag{2}$$

By induction over the size of the derivation of the execution. Take arbitrary $s_1, \phi_{1\ldots4}$ such that

$$\phi_1, s_1 \rightsquigarrow \phi_3 \tag{3}$$
$$\phi_2, ren_{\mathcal{E}}(s_1) \rightsquigarrow \phi_4 \tag{4}$$
$$\phi_1 \approx \phi_2 \tag{5}$$

To show

$$\phi_3 \approx \phi_4$$

By cases

- $s_1 = \mathtt{call}\ \mathtt{f}_3(x_1 \ldots x_n) \ \wedge \ \exists \mathtt{f}_4 : (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E}$

  By definition

  $$ren_{\mathcal{E}}(s_1) = \mathtt{call}\ \mathtt{f}_4(x_1 \ldots x_n) \tag{6}$$

  By (3) exists ϕ_9, ϕ_{11}

  $$\phi_9 = mkframe(\phi_1, \mathtt{f}_3, x_1 \ldots x_n) \tag{7}$$
  $$\phi_9, \mathtt{body}\ \mathtt{f}_3 \rightsquigarrow \phi_{11} \tag{8}$$
  $$\phi_3 = (\phi_1^{\tilde{\sigma}}, \phi_{11}^{h}) = \phi_{11}^{ctx} \tag{9}$$

  By (4) exists ϕ_{10}, ϕ_{12}

  $$\phi_{10} = mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n) \tag{10}$$
  $$\phi_{10}, \mathtt{body}\ \mathtt{f}_4 \rightsquigarrow \phi_{12} \tag{11}$$
  $$\phi_4 = (\phi_2^{\tilde{\sigma}}, \phi_{12}^{h}) = \phi_{12}^{ctx} \tag{12}$$

  By definition and (5), (7), (10)

  $$\phi_9 \approx \phi_{10} \tag{13}$$

  By (2), (11), exists ϕ_{14} such that

  $$\phi_{10}, \mathtt{body}\ \mathtt{f}_3 \rightsquigarrow \phi_{14} \tag{14}$$

By above, (11), (1)

$$\phi_{12} \approx \phi_{14} \tag{15}$$

By (14), rule BODA

$$\phi_{10}, ren_{\mathcal{E}}(\mathcal{B}(\mathtt{f}_3)) \twoheadrightarrow_{\mathcal{R}} \phi_{14} \tag{16}$$

By (8)

$$\phi_9, \mathcal{B}(\mathtt{f}_3) \twoheadrightarrow_{\mathcal{V}} \phi_{11} \tag{17}$$

By IH, (8), (13), (16)

$$\phi_{11} \approx \phi_{14} \tag{18}$$

By lemma 4.3.1 (Isomorphism is an equivalence relation), (15), (18)

$$\phi_{11} \approx \phi_{12} \tag{19}$$

By lemma 2.4.7 (Isomorphism implies calling context isomorphism), (9), (12), (19)

$$\phi_3 \approx \phi_4 \tag{20}$$

- $s_1 = \mathtt{call}\ \mathtt{f}_3(x_1 \ldots x_n) \ \wedge\ \nexists \mathtt{f}_4 : (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E}$

  By application of IH

- $s_1 = s_3; s_4$

  By application of IH twice

- $s_1 = \mathtt{if}(b)\{s_2\}$

  By application of IH

- otherwise

  The $\mathcal{V}$ semantics and $\mathcal{A}$ semantics are identical for all other cases. And $ren_{\mathcal{E}}(s_1) = s_1$. So these cases follow directly from lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism).

By above, done.

Auxiliary lemma two Whenever there is a terminating execution from an initial store under the $\mathcal{V}$ semantics, there is a terminating execution from every isomorphic initial store under the $\mathcal{R}$ semantics
To simplify the proof, we assume that all procedure pairs in $\mathcal{E}$ have the same parameter names. In practice it is straightforward to rename the procedure parameters to ensure this is the case.

$$\begin{aligned}
&\forall (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \ \mathtt{f}_3 \lessapprox \mathtt{f}_4 \ \wedge\ mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4) \\
&\implies \\
&\Big(\forall \phi_{1\ldots3} : \ \phi_1 \approx \phi_2 \ \wedge\ \phi_1, s_1 \twoheadrightarrow_{\mathcal{V}} \phi_3 \implies \exists \phi_4 : \ \phi_2, ren_{\mathcal{E}}(s_1) \twoheadrightarrow_{\mathcal{R}} \phi_4 \quad \Big)
\end{aligned}$$

Assume

$$\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \mathtt{f}_3 \lessapprox \mathtt{f}_4 \tag{21}$$

$$\forall(\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : mt_{\mathcal{R}}(\mathtt{f}_3, \mathtt{f}_4) \tag{22}$$

By induction over the derivation of the execution. Take arbitrary $s_1, \phi_{1\ldots3}$ such that

$$\phi_1 \approx \phi_2 \tag{23}$$

$$\phi_1, s_1 \twoheadrightarrow \phi_3 \tag{24}$$

To show

$$\exists\phi_4 : \phi_2, ren_{\mathcal{E}}(s_1) \twoheadrightarrow_{\mathcal{R}} \phi_4$$

By cases

- $s_1 = \mathtt{call}\ \mathtt{f}_3(x_1 \ldots x_n) \wedge \exists \mathtt{f}_4 : (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E}$

  By definition and case

  $$ren_{\mathcal{E}}(s_1) = \mathtt{call}\ \mathtt{f}_4(x_1 \ldots x_n) \tag{25}$$

  To show

  $$\exists\phi_4 : \phi_2, \mathtt{call}\ \mathtt{f}_4(x_1 \ldots x_n) \twoheadrightarrow_{\mathcal{R}} \phi_4$$

  By calculation:

  $$\Big(\exists\phi_4 : \phi_2, \mathtt{call}\ \mathtt{f}_4(x_1 \ldots x_n) \twoheadrightarrow_{\mathcal{R}} \phi_4\Big)$$

  By definition 2.1.1 (Semantics of $\mathcal{BL}$)

  $$\iff \Big(\exists\phi_4 : mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n), \mathtt{body}\ \mathtt{f}_4 \twoheadrightarrow_{\mathcal{R}} \phi_4\Big)$$

  By (22)

  $$\iff \Big(\exists\phi_4 : mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n), \mathtt{body}\ \mathtt{f}_3 \twoheadrightarrow_{\mathcal{R}} \phi_4\Big)$$

  By definition 2.1.1 (Semantics of $\mathcal{BL}$)

  $$\iff \begin{pmatrix} \exists\phi_4 : (mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n), \_) \in con(\mathtt{f}_3) \wedge \\ mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n), ren_{\mathcal{E}}(\mathcal{B}(\mathtt{f}_3)) \twoheadrightarrow_{\mathcal{R}} \phi_4 \end{pmatrix}$$

  To show

  $$\begin{aligned} &\exists\phi_4 : (mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n), \_) \in con(\mathtt{f}_3) \wedge \\ &\quad mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n), ren_{\mathcal{E}}(\mathcal{B}(\mathtt{f}_3)) \twoheadrightarrow_{\mathcal{R}} \phi_4 \end{aligned}$$

  By definition *mkframe*, (23) and that the procedures have the same parameter names:

  $$mkframe(\phi_1, \mathtt{f}_3, x_1 \ldots x_n) \approx mkframe(\phi_2, \mathtt{f}_4, x_1 \ldots x_n) \tag{26}$$

By (24), case, definition 2.1.1 (Semantics of $\mathcal{BL}$) (CALLV)

$$mkframe(\phi_1, \mathtt{f}_3, x_1 \dots x_n), \_) \in con(\mathtt{f}_3) \tag{27}$$

$$mkframe(\phi_1, \mathtt{f}_3, x_1 \dots x_n), \mathcal{B}(\mathtt{f}_3) \rightsquigarrow \phi_3 \tag{28}$$

By (26), (27), and that definition 2.1.1 (Semantics of $\mathcal{BL}$) requires contracts do not distinguish between isomorphic stores

$$(mkframe(\phi_2, \mathtt{f}_4, x_1 \dots x_n), \_) \in con(\mathtt{f}_3) \tag{29}$$

By IH. (26), (28), exists ϕ_4 such that

$$mkframe(\phi_2, \mathtt{f}_4, x_1 \dots x_n), ren_{\mathcal{E}}(\mathcal{B}(\mathtt{f}_3)) \rightsquigarrow \phi_4$$

- $s_1 = \mathtt{call}\ \mathtt{f}_3(x_1 \dots x_n) \wedge \nexists \mathtt{f}_4 : (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E}$

  By application of IH

- $s_1 = s_3; s_4$

  By application of IH twice

- $s_1 = \mathtt{if}(b)\{s_2\}$

  By application of IH

- otherwise

  The $\mathcal{V}$ semantics and $\mathcal{R}$ semantics are identical for all other cases. And $ren_{\mathcal{E}}(s_1) = s_1$. So these cases follow directly from lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism).

Done.

Now prove main lemma

Take arbitrary $\mathtt{f}_1, \mathtt{f}_2$ such that

$$(\mathtt{f}_1, \mathtt{f}_2) \in \mathcal{E} \tag{30}$$

$$\forall (\mathtt{f}_3, \mathtt{f}_4) \in \mathcal{E} : \mathtt{f}_3 \lessapprox \mathtt{f}_4 \tag{31}$$

To show

$$\mathtt{f}_1 \lessapprox \mathtt{f}_2$$

Take $\phi_{1\dots4}$ such that

$$\phi_1 \approx \phi_2 \tag{32}$$

$$\phi_1, \mathtt{body}\ \mathtt{f}_1 \parallel \phi_2, \mathtt{body}\ \mathtt{f}_2 \rightsquigarrow \phi_3 \parallel \phi_4 \tag{33}$$

To show

$$\phi_3^{ctx} \approx \phi_4^{ctx}$$

By (33)

$$\phi_1, \texttt{body f}_1 \rightsquigarrow \phi_3 \tag{34}$$

$$\phi_2, \texttt{body f}_2 \rightsquigarrow \phi_4 \tag{35}$$

By above and definition 2.1.1

$$(\phi_1, \_) \in \mathit{Con}(\texttt{f}_3) \tag{36}$$

$$\phi_1, \mathcal{B}(\texttt{f}_1) \rightsquigarrow \phi_3 \tag{37}$$

$$(\phi_2, \_) \in \mathit{Con}(\texttt{f}_4) \tag{38}$$

$$\phi_2, \mathcal{B}(\texttt{f}_2) \rightsquigarrow \phi_4 \tag{39}$$

By above, (30), (31), and auxiliary lemma two exists $\phi_{5,6}$

$$\phi_1, \mathit{ren}_{\mathcal{E}}(\mathcal{B}(\texttt{f}_1)) \rightsquigarrow \phi_5 \tag{40}$$

$$\phi_2, \mathit{ren}_{\mathcal{E}}(\mathcal{B}(\texttt{f}_2)) \rightsquigarrow \phi_6 \tag{41}$$

By above, (30), (31), auxiliary lemma one, (32)

$$\phi_3 \approx \phi_5 \tag{42}$$

$$\phi_4 \approx \phi_6 \tag{43}$$

By (36), (38), (40), (41)

$$\phi_1, \texttt{body f}_1 \rightsquigarrow \phi_5 \tag{44}$$

$$\phi_2, \texttt{body f}_2 \rightsquigarrow \phi_6 \tag{45}$$

By above, (30), (31)

$$\phi_5 \approx \phi_6 \tag{46}$$

By (42), (43), (46), lemma 4.3.1 (Isomorphism is an equivalence relation)

$$\phi_3 \approx \phi_4 \tag{47}$$

Done. ■

B.28 Discard non equal isomorphisms (lemma 5.2.1)

Recall from page 54

Lemma 5.2.1 (Discard non equal isomorphisms)

$$\forall f_{1,2} \colon f_1 \lessapprox f_2 \implies f_1 \gtrapprox f_2$$

Proof B.28.1
Take arbitrary $\texttt{f}_{1,2}$ such that

$$\texttt{f}_1 \lessapprox \texttt{f}_2 \tag{1}$$

To show

$$\mathtt{f}_1 \approxeq \mathtt{f}_2$$

Take arbitrary $\phi_{1\ldots4}, tr_{1,2}, \pi_1$ such that

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{2}$$

$$\phi_1, \mathtt{body\ f}_1 \parallel \phi_2, \mathtt{body\ f}_2 \overset{tr_1, tr_2}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \parallel \phi_4 \tag{3}$$

To show

$$\phi_3^{ctx} \approx \phi_4^{ctx}$$

By definition 2.1.1 (Semantics of $\mathcal{BL}$), (3)

$$\phi_1, \mathtt{body\ f}_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \tag{4}$$

$$\phi_2, \mathtt{body\ f}_2 \overset{tr_2}{\rightsquigarrow_{\mathcal{K}}} \phi_4 \tag{5}$$

By above and lemma A.1.2 ($\mathcal{K}$ semantics overapproximate $\mathcal{R}$ semantics)

$$\phi_1, \mathtt{body\ f}_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \tag{6}$$

$$\phi_2, \mathtt{body\ f}_2 \overset{tr_2}{\rightsquigarrow_{\mathcal{K}}} \phi_4 \tag{7}$$

By definition 3.4.1 (Selected isomorphic points), take π_2 such that[4]

$$misos(tr_1, tr_2) = \pi_2 \tag{8}$$

By (8), (2) and definition 3.4.1 (Selected isomorphic points)

$$com(\pi_1, \pi_2) \tag{9}$$

$$\forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \tag{10}$$

By above, lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism), (6) there exists ϕ_5 such that

$$\phi_2, \mathtt{body\ f}_1 \overset{tr_3}{\rightsquigarrow_{\mathcal{K}}} \phi_5 \tag{11}$$

$$tr_1 \approx_{\pi_2} tr_3 \tag{12}$$

By (8) take P such that

$$P = mpts(tr_1, tr_2) \tag{13}$$

To show

$$\forall (i, j, X, Y) \in P : tr_3[i] \approxeq_{id}^{X,Y} tr_2[j]$$

Take arbitrary $(i, j, X, Y) \in P$

[4] note that $misos(tr_1, tr_2)$ is always defined, although it may give the empty set

By (8) exists π_3 such that

$$tr_1[i] \approx^{X,Y}_{\pi_3} tr_2[j] \tag{14}$$

$$\pi_3 \subseteq \pi_2 \tag{15}$$

By (12) exists π_4 such that

$$tr_1[i] \approx^{X,X}_{\pi_4} tr_3[j] \tag{16}$$

$$com(\pi_4, \pi_2) \tag{17}$$

By above and definition 3.4.1 (Selected isomorphic points)

$$\pi_4 \subseteq \pi_2 \tag{18}$$

By

$$dom(\pi_4) = dom(\pi_3) \tag{19}$$

By above and (15), (18)

$$\pi_4 = \pi_3 \tag{20}$$

By above, (14), (16), and symmetry lemma 4.3.1 (Isomorphism is an equivalence relation)

$$tr_3[j] \approx^{X,X}_{\pi_3^{-1}} tr_1[i] \approx^{X,Y}_{\pi_3} tr_2[j] \tag{21}$$

By above, transitivity lemma 4.3.1 (Isomorphism is an equivalence relation)

$$tr_3[j] \approx^{X,Y}_{id} tr_2[j] \tag{22}$$

By above and definition 3.4.1 (Selected isomorphic points)

$$misos(tr_3, tr_2) \subseteq id \tag{23}$$

By (7), (11), (23), and COMK

$$\phi_1, \texttt{body f}_1 \parallel \phi_2, \texttt{body f}_2 \overset{tr_3,tr_2}{\rightsquigarrow} \phi_5 \parallel \phi_4 \tag{24}$$

By above, (1)

$$\phi_5^{ctx} \approx \phi_4^{ctx} \tag{25}$$

By (12), lemma 4.3.1 (Isomorphism is an equivalence relation)

$$\phi_3^{ctx} \approx \phi_5^{ctx} \tag{26}$$

By (25), (26), lemma 4.3.1 (Isomorphism is an equivalence relation)

$$\phi_3^{ctx} \approx \phi_4^{ctx} \tag{27}$$

Done. ■

B.29 Modular procedure equivalence (lemma 5.3.1)

Recall from page 54

Lemma 5.3.1 (Modular procedure equivalence)
Given a partial mapping between procedure names $\mathcal{E}$,

$$\forall f_{1,2} \colon f_1 \overset{\mathcal{A}}{\approx} f_2 \implies f_1 \overset{\mathcal{K}}{\approx} f_2$$

Proof B.29.1
Take $\mathtt{f}_{1,2}$ such that

$$\mathtt{f}_1 \overset{\mathcal{A}}{\approx} \mathtt{f}_2 \tag{1}$$

To show

$$\mathtt{f}_1 \overset{\mathcal{K}}{\approx} \mathtt{f}_2$$

Take arbitrary $\phi_{1\ldots4}, tr_{1,2}$ such that

$$\phi_1 \approx \phi_2 \tag{2}$$

$$\phi_1, \mathtt{body\ f_1} \parallel \phi_2, \mathtt{body\ f_2} \overset{tr_1, tr_2}{\underset{\mathcal{K}}{\rightsquigarrow}} \phi_3 \parallel \phi_4 \tag{3}$$

$$misos(tr_1, tr_2) \subseteq id \tag{4}$$

To show

$$\phi_3^{ctx} \approx \phi_4^{ctx}$$

By (3)

$$\phi_1, \mathtt{body\ f_1} \overset{tr_1}{\underset{\mathcal{K}}{\rightsquigarrow}} \phi_3 \tag{5}$$

$$\phi_2, \mathtt{body\ f_2} \overset{tr_2}{\underset{\mathcal{K}}{\rightsquigarrow}} \phi_4 \tag{6}$$

By above, lemma A.1.3 ($\mathcal{A}$ semantics overapproximate $\mathcal{K}$ semantics)

$$\phi_1, \mathtt{body\ f_1} \overset{tr_1}{\underset{\mathcal{A}}{\rightsquigarrow}} \phi_3 \tag{7}$$

$$\phi_2, \mathtt{body\ f_2} \overset{tr_2}{\underset{\mathcal{A}}{\rightsquigarrow}} \phi_4 \tag{8}$$

In order to be able to apply COMA on the above, we need to show:

$$\mathcal{U}(tr_1, tr_2)$$

Take arbitrary $\pi_1, \mathtt{f}, \phi_{5\ldots8}$ such that

$$(\mathtt{call\ f}(x_1 \ldots x_n), \phi_5, \phi_7) \in tr_1 \tag{9}$$

$$(\mathtt{call\ f}(y_1 \ldots y_n), \phi_6, \phi_8) \in tr_2 \tag{10}$$

$$\phi_5 \overset{\{x_1 \ldots x_n\}, \{y_1 \ldots y_n\}}{\underset{\pi_1}{\approx}} \phi_6 \tag{11}$$

To show

$$\exists \pi_2 : \mathit{com}(\pi_1, \pi_2) \wedge \mathit{effect}(\phi_5, \phi_7) \approx_{\pi_2} \mathit{effect}(\phi_6, \phi_8)$$

By (5), (6), (9), (10), exists $\phi_{9\ldots12}$ such that

$$\phi_9, \mathcal{B}(\texttt{f}) \twoheadrightarrow \phi_{11} \tag{12}$$
$$\phi_{10}, \mathcal{B}(\texttt{f}) \twoheadrightarrow \phi_{12} \tag{13}$$
$$\phi_9 \approx_{\pi_1} \phi_{10} \tag{14}$$
$$\phi_5^h = \phi_9^h \tag{15}$$
$$\phi_6^h = \phi_{10}^h \tag{16}$$
$$\phi_7^h = \phi_{11}^h \tag{17}$$
$$\phi_8^h = \phi_{12}^h \tag{18}$$

By (15), (16), (17), (18)

$$\mathit{effect}(\phi_5, \phi_7) = \mathit{effect}(\phi_9, \phi_{11}) \tag{19}$$
$$\mathit{effect}(\phi_6, \phi_8) = \mathit{effect}(\phi_{10}, \phi_{12}) \tag{20}$$

By lemma 4.1.3 ($\mathcal{BL}$ closed under isomorphism), definition 4.1.1 (Isomorphic Executions), and (12), (13) exists π_2 such that

$$\mathit{com}(\pi_1, \pi_2) \wedge \mathit{effect}(\phi_9, \phi_{11}) \approx_{\pi_2} \mathit{effect}(\phi_{10}, \phi_{12}) \tag{21}$$

By above, (19), (20)

$$\mathit{effect}(\phi_5, \phi_7) \approx_{\pi_2} \mathit{effect}(\phi_6, \phi_8) \tag{22}$$

By above

$$\mathcal{U}(tr_1, tr_2) \tag{23}$$

By (4), (7), (8), (23) and COMA

$$\phi_1, \texttt{body f}_1 \parallel \phi_2, \texttt{body f}_2 \overset{tr_1, tr_2}{\rightsquigarrow} \phi_3 \parallel \phi_4 \tag{24}$$

By above, (1)

$$\phi_3^{ctx} \approx \phi_4^{ctx} \tag{25}$$

■

B.30 Sequential $\mathcal{R}$ semantics overapproximate $\mathcal{K}$ semantics (lemma A.1.1)

Recall from page 81

Lemma A.1.1 (Sequential $\mathcal{R}$ semantics overapproximate $\mathcal{K}$ semantics)

$$\forall s_1, \phi_{1,3}, tr_1 : \ \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \implies \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{R}}} \phi_3$$

Proof B.30.1
By induction. Since $\|$ is not a statement and cannot appear in the program text, and since all other rules of the $\mathcal{R}$ semantics and $\mathcal{K}$ semantics are identical.

■

B.31 $\mathcal{K}$ semantics overapproximate $\mathcal{R}$ semantics (lemma A.1.2)

Recall from page 81

Lemma A.1.2 ($\mathcal{K}$ semantics overapproximate $\mathcal{R}$ semantics)

$$\forall s_1, \phi_{1,3}, tr_1 : \ \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{R}}} \phi_3 \implies \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3$$

Proof B.31.1
By induction. Since $\|$ is not a statement and cannot appear in the program text, and since all other rules of the $\mathcal{R}$ semantics and $\mathcal{K}$ semantics are identical.

■

B.32 $\mathcal{A}$ semantics overapproximate $\mathcal{K}$ semantics (lemma A.1.3)

Recall from page 81

Lemma A.1.3 ($\mathcal{A}$ semantics overapproximate $\mathcal{K}$ semantics)

$$\forall s_1, \phi_{1,3}, tr_1 : \ \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \implies \phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{A}}} \phi_3$$

Proof B.32.1
By induction on the derivation of an execution. Since $\|$ is not a statement and cannot appear in the program text, and since all other rules of the $\mathcal{K}$ semantics and $\mathcal{A}$ semantics are identical apart from call. Proof for call follows.
Take arbitrary $\phi_1, s_1, tr_1, \phi_3$ such that

$$\phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{K}}} \phi_3 \tag{1}$$

$$s_1 = \texttt{call f}(x_1 \dots x_n) \tag{2}$$

In order to apply CALLA, need to show there exists ϕ_2, σ, h_2 such that

$$dom(h_2) \supseteq dom(\phi_1^h) \qquad \phi_2 = (\phi_1^{\tilde{\sigma}} \cdot \sigma, h_2)$$

$$(mkframe(\phi_1, \texttt{f}, x_1 \dots x_n), \phi_2) \in Con(\texttt{f})$$

$$\phi_3 = \phi_2^{ctx}$$

By case, CALLV, exists ϕ_4 such that

$$(mkframe(\phi_1, \texttt{f}, x_1 \dots x_n), \phi_4) \in Con(\texttt{f}) \tag{3}$$

$$\phi_3 = \phi_4^{ctx} \tag{4}$$

Take

$$\phi_2 = \phi_4 \tag{5}$$

There are no instructions in $\mathcal{K}$ semantics that reduce the size of the heap (lemma 2.4.8), so by above and induction on the derivation of (1)

$$dom(h_2) \supseteq dom(\phi_1^h) \tag{6}$$

By (3), (5)

$$mkframe(\phi_1, \mathtt{f}, x_1 \dots x_n), \phi_4) \in Con(\mathtt{f}) \tag{7}$$

By (4), (5)

$$\phi_3 = \phi_2^{ctx} \tag{8}$$

By (6), (6), (6) and CALLA

$$\phi_1, s_1 \overset{tr_1}{\rightsquigarrow_{\mathcal{A}}} \phi_3 \tag{9}$$

■

B.33 ASSUME closed (lemma A.2.1)

Recall from page 81

Lemma A.2.1 (ASSUME closed)

$$\begin{aligned}
&\forall L, b, \phi_{1-3}, \pi_{1,2}, tr_1 : \\
&\quad \phi_1, \mathtt{assume}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \ \wedge\ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ in(\pi_2) \\
&\quad \Longrightarrow \\
&\quad \exists \phi_4, \pi_3, tr_2 :\ \phi_2, \mathtt{assume}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)
\end{aligned}$$

Proof B.33.1

Take arbitrary $L, b, \phi_{1-3}, \pi_{1,2}, tr_1$ such that

$$\phi_1, \mathtt{assume}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{1}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{2}$$

$$com(\pi_1, \pi_2) \tag{3}$$

$$in(\pi_2) \tag{4}$$

To show

$$\exists \phi_4, \pi_3, tr_2 :\ \phi_2, \mathtt{assume}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)$$

By definition 2.1.1, (1)

$$\phi_1 = \phi_3 \tag{5}$$

$$\phi_1 \vDash b \tag{6}$$

By lemma 4.2.1, eq. (2), eq. (6)

$$\phi_2 \vDash b \tag{7}$$

By above, take $tr_2 = (\texttt{assume}\ b, \phi_2, \phi_2)$ such that

$$\phi_2, \texttt{assume}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_2 \tag{8}$$

Taking $\phi_4 = \phi_2$ and $\pi_3 = \pi_1$, by (2), (3), (8)

$$\phi_2, \texttt{assume}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \tag{9}$$

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{10}$$

$$com(\pi_3, \pi_2) \tag{11}$$

■

B.34 ASSERTT closed (lemma A.2.2)

Recall from page 82

Lemma A.2.2 (ASSERTT closed)

$$\begin{aligned}
&\forall L, b, \phi_{1-3}, \pi_{1,2}, tr_1 : \\
&\quad \phi_1 \vDash b \ \wedge\ \phi_1, \mathit{assert}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \ \wedge\ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ in(\pi_2) \\
&\quad \Longrightarrow \\
&\quad \exists \phi_4, \pi_3, tr_2 :\ \phi_2, \mathit{assert}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)
\end{aligned}$$

Proof B.34.1
Take arbitrary $L, b, \phi_{1-3}, \pi_{1,2}, tr_1$ such that

$$\phi_1 \vDash b \tag{1}$$

$$\phi_1, \texttt{assert}\ b \overset{tr_1}{\rightsquigarrow_{\mathcal{L}}} \phi_3 \tag{2}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{3}$$

$$com(\pi_1, \pi_2) \tag{4}$$

$$in(\pi_2) \tag{5}$$

To show

$$\exists \phi_4, \pi_3, tr_2 :\ \phi_2, \texttt{assert}\ b \overset{tr_2}{\rightsquigarrow_{\mathcal{L}}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)$$

By definition 2.1.1, (2), (1)

$$\phi_1 = \phi_3 \tag{6}$$

By lemma 4.2.1, eq. (3), eq. (1)

$$\phi_2 \vDash b \tag{7}$$

By above, take $tr_2 = (\texttt{assert}\ b, \phi_2, \phi_2)$ such that

$$\phi_2, \texttt{assert}\ b \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_2 \tag{8}$$

Taking $\phi_4 = \phi_2$ and $\pi_3 = \pi_1$, by (3), (4), (8)

$$\phi_2, \texttt{assert}\ b \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4 \tag{9}$$

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{10}$$

$$com(\pi_3, \pi_2) \tag{11}$$

■

B.35 ASSERTF closed (lemma A.2.3)

Recall from page 82

Lemma A.2.3 (ASSERTF closed)

$$\begin{aligned}
&\forall L, b, \phi_{1-3}, \pi_{1,2}, tr_1 : \\
&\quad \neg(\phi \vDash b) \ \wedge\ \phi_1, \mathit{assert}\ b \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \ \wedge\ \phi_1 \approx_{\pi_1} \phi_2 \ \wedge\ com(\pi_1, \pi_2) \ \wedge\ in(\pi_2) \\
&\quad \Longrightarrow \\
&\quad \exists \phi_4, \pi_3, tr_2 :\ \phi_2, \mathit{assert}\ b \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)
\end{aligned}$$

Proof B.35.1
Take arbitrary $L, b, \phi_{1-3}, \pi_{1,2}, tr_1$ such that

$$\neg(\phi \vDash b) \tag{1}$$

$$\phi_1, \texttt{assert}\ b \overset{tr_1}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_3 \tag{2}$$

$$\phi_1 \approx_{\pi_1} \phi_2 \tag{3}$$

$$com(\pi_1, \pi_2) \tag{4}$$

$$in(\pi_2) \tag{5}$$

To show

$$\exists \phi_4, \pi_3, tr_2 :\ \phi_2, \texttt{assert}\ b \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \phi_4 \ \wedge\ \phi_3 \approx_{\pi_3} \phi_4 \ \wedge\ com(\pi_2, \pi_3)$$

By definition 2.1.1, (1), (2)

$$\texttt{error} = \phi_3 \tag{6}$$

By lemma 4.2.1, eq. (3), eq. (1)

$$\neg(\phi_2 \vDash b) \tag{7}$$

By above, take $tr_2 = (\texttt{assert}\ b, \phi_2, \texttt{error})$ such that

$$\phi_2, \texttt{assert}\ b \overset{tr_2}{\underset{\mathcal{L}}{\rightsquigarrow}} \texttt{error} \tag{8}$$

Taking $\phi_4 = \texttt{error}$ and $\pi_3 = \emptyset$, by definition 2.3.1, (3), (8)

$$\phi_2, \texttt{assert } b \overset{tr_2}{\rightsquigarrow_\Sigma} \phi_4 \tag{9}$$

$$\phi_3 \approx_{\pi_3} \phi_4 \tag{10}$$

$$com(\pi_3, \pi_2) \tag{11}$$

■

B.36 Consistent with allocated subset (lemma A.2.4)

Recall from page 82

Lemma A.2.4 (Consistent with allocated subset)

$$\begin{aligned}
&\forall \phi_{1,2}, \pi_{1,2} : \\
&\quad \forall (a_1, a_2) \in \pi_1 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \land \pi_2 \subseteq \pi_1 \\
&\quad \Longrightarrow \\
&\quad \forall (a_1, a_2) \in \pi_2 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h)
\end{aligned}$$

Proof B.36.1

Take arbitrary $\phi_{1,2}, \pi_{1,2}$ such that

$$\forall (a_1, a_2) \in \pi_1 : (a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \tag{1}$$

$$\pi_2 \subseteq \pi_1 \tag{2}$$

Take arbitrary (a_1, a_2) such that

$$(a_1, a_2) \in \pi_2 \tag{3}$$

By above, (2)

$$(a_1, a_2) \in \pi_1 \tag{4}$$

By above, (1)

$$(a_1 \in \phi_1^h \iff a_2 \in \phi_2^h) \tag{5}$$

■