Refactoring Boundary

Tim Wood[1] and Sophia Drossopoulou[2]

1 **Imperial College London** t.wood12@imperial.ac.uk
2 **Imperial College London** s.drossopoulou@imperial.ac.uk

Abstract

We argue that the limit of the propagation of the heap effects of a source code modification is determined by the aliasing structure of method parameters in a trace of the method calls that cross a boundary which partitions the heap. Further, that this aliasing structure is sufficient to uniquely determine the state of the part of the heap which has not been affected. And we give a definition of what it means for a part of the heap to be unaffected by a source code modification. This can be used to determine the correctness of a refactoring.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Refactoring, Object Oriented

Digital Object Identifier 10.4230/OASIcs.ICCSW.2013.119

1 Introduction

A refactoring is a transformation applied to the source code of a program that preserves the meaning of the program. Most refactorings however do affect the internal behaviour of a program, for example, new classes and methods may be introduced, conditionals may be replaced with polymorphism and classes may be in-lined. Determining if a particular transformation is indeed meaning preserving is a challenge.

According to our proposal, the heap is partitioned into an affected part and an unaffected part. Moreover we give a sufficient condition that determines that the effects of the code modification do not spread from the affected part into the unaffected part — which we call a *bounded* modification.

Motivation: A programmer modifying a program wants to avoid breaking existing functionality, even functionality that they may not be aware of. A programmer wants to convince other programmers to accept their patch, and wants to show that their change is safe. A programmer wants to precisely characterise the difference between two versions of a program.

Contributions: We argue that a good way to characterise the difference between two versions of a program is to partition the heap into an affected part and an unaffected part, and require that executions correspond in the unaffected part. We propose a precise definition of this novel property. We propose a novel sufficient condition for such a correspondence of heaps — that there will be a correspondence in the unaffected part whenever the aliasing structure of stack frames witnessed at the heap partition boundary is isomorphic between executions of the two versions. We anticipate that this sufficient condition will provide a basis for automated tools that can check if a modification is bounded

Our proposal gives a novel general definition of refactoring correctness in terms of the heap effect of the modification. Moreover it offers a wider definition of refactoring correctness than other approaches, depending on what unaffected partition is picked. For example, if all I/O occurs in the unaffected partition then our approach is similar to traditional definitions

2013 Imperial College Computing Student Workshop (ICCSW'13).
Editors: Andrew V. Jones, Nicholas Ng; pp. 119–127

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of refactoring correctness, but other choices of partition are possible that allow differences in program I/O. This notion of correctness does not require any additional specification of the program, nor does it require the programmer to limit themselves to previously known-correct refactorings with established pre-conditions.

Related Work: Typically the correctness of refactorings is defined as I/O equivalence and is checked by ad-hoc pre-condition checking[10][1] or by composing smaller refactorings[12]. State based encapsulation provides a method for reasoning about the equivalence of classes[9]. Program slicing can be used to establish the correctness of some statement reordering refactorings[2]. Graph transformations have been suggested as a means to reason about refactorings in general[11]. Regression verification uses bounded model checking to verify the equivalence of some loop and recursion free programs[3]. Program verification tools in conjunction with automated theorem provers can be used to check some programs for equivalence[4][6], or library versions for backward compatibility[13]. Semantics aware trace analysis[5] and BCT[7] use traces captured at runtime to attempt to isolate the causes of regression failures. Guru[8] uses sequences of messages sends to define equivalence of differently factored method implementations in an object-oriented system.

Structure: In **section 2** we give a motivating example, and describe what it means for the effect of a refactoring to be bounded. In **section 3** we give a sufficient condition for determining if a part of the heap is unaffected between executions of two versions of the program. Then in **section 4** we describe exactly how partitioning and trace comparison works. In **section 5** we give several interesting properties of bounded modifications, and in **section 6** sketch a proof of those properties.

2 Bounded Effect

An object oriented program p is modified to produce a program q. The heap is partitioned into two parts Φ^m and Φ^u. We say that Φ^m *bounds the modification* if for any execution in p there is a corresponding execution in q such that the heaps are equivalent within Φ^u. In other words, the heap effect of a bounded modification does not spread beyond Φ^m.

```
1  class FiFo {
2   Node f;
3   void add(final Object o) {
4    if(f == null) { f=new Node(o); }
5    else { f.add(o); }}
6   Object remove() {
7    Object r=f.value(); f=f.next(); return
         r;}}
8
9  class Node {
10  Object o; Node n;
11  Node(Object o) { this.o=o; }
12  Node next() { return n; }
13  Object value() { return o; }
14  void add(final Object o) {
15   if(n == null) { n=new Node(o); }
16   else { n.add(o); }}}
```

Listing 1 Program p — fifo queue with recursive add.

```
1  class FiFo {
2   Node f, l;
3   void add(final Object o) {
4    if(f == null) { l=f=new Node(o); }
5    else { l=l.add(o); }}
6   Object remove() {
7    Object r=f.value(); f=f.next(); return
         r;}}
8
9  class Node {
10  Object o; Node n;
11  Node(Object o) { this.o=o; }
12  Node next() { return n; }
13  Object value() { return o; }
14   Node add(Object o) {
15   return n=new Node(o); }}
16
```

Listing 2 Program q — fifo queue with last element pointer. The modified parts are highlighted.

```
1  class Test { Fifo fi = new Fifo(); Object o1 = new Object(), o2 = new Object();
2   void test() { fi.add(o1); fi.add(o2); fi.remove(); fi.remove();}}
```

Listing 3 Test program for the FiFo queue.

We shall clarify the meaning of execution, correspondence, partitioning and equivalence in terms of the following example. Listing 1 shows part of a larger program `p`. The code shown is a fifo queue consisting of two classes `FiFo` and `Node`. The code is modified, as in listing 2, to produce a new version `q` where the implementation and representation of the queue has changed. Listing 3 shows a test program that could be run with the code from either Listing 1 or Listing 2. The only values in our language are addresses, which can be compared for equality but are otherwise opaque, and all fields are private.

Consider Figure 1 which shows the stack and heap when an execution of the test program in Listing 3 with `p` is just about to return from the second call to `add` on line 2 of Listing 3. In this figure Φ^{m} is chosen to include only objects of the `FiFo` and `Node` classes.

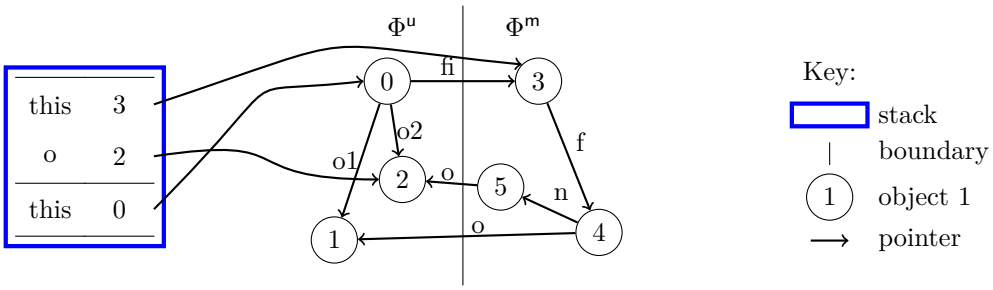

Figure 1 Snapshot of an execution of `p` (Listing 3 and Listing 1). The heap and top two stack frames are shown. The heap is partitioned so that objects at addresses 0,1,2 are in Φ^{u}, and objects 3,4,5 are in Φ^{m}. The top stack frame has a `this` pointer into the Φ^{m} partition, the stack frame below it has a `this` pointer into the Φ^{u} partition.

In Figure 2 we show the stack and heap at a corresponding point in the execution of the test program in Listing 3 with `q`. Notice that when compared to Figure 1 this figure has an additional pointer between objects 10 and 12, this is due to the field `l` which has been added in `q` as shown on line 2 of Listing 2.

We notice that the heap in Figure 1, which we will call h_1, and the heap in Figure 2, which we will call h_2, are *isomorphic* when only the objects within Φ^{u} are considered. We say that h_1 is equivalent to h_2 wrt. Φ^{u}, and in our notation we write this as $h_1 \approx_{\Phi^{\mathsf{u}}} h_2$. In this case, object 0 corresponds with object 7, 1 with 8, and 2 with 9.

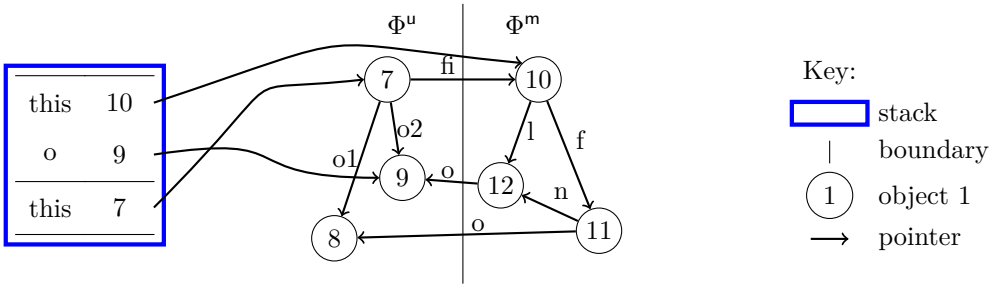

Figure 2 Snapshot of an execution of `q` (Listing 3 and Listing 2). Notice that there is an extra pointer between objects 10 and 12 when compared to Figure 1.

Definition 1 gives the formal meaning of bounded, where we use $\gamma, \gamma_p, \gamma_q$ to range over runtime configurations, and $\gamma \longrightarrow\!\!\!\!\!\rightarrow_p \gamma'$ to mean the multi-step execution of program p from configuration γ to configuration γ'. We write $\mathsf{heap}(\gamma)$ to mean the heap of configuration γ. The judgement $\mathsf{init}(\gamma)$ holds when γ is an initial state, which means it has an empty stack and an expression consisting of a call to the program's main method.

▶ **Definition 1** (bounded). When Φ^{m} *bounds the effect*, for every sequence of states reachable in p there is a sequence of states with equivalent heaps wrt. Φ^{u} reachable in q.

$$\forall \gamma, \gamma_p, \gamma'_p : \mathsf{init}(\gamma) \ \wedge \ \gamma \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{p}} \gamma_p \longrightarrow\!\!\!\!\!\rightarrow_{p} \gamma'_p$$
$$\implies \exists \gamma_q, \gamma'_q : \gamma \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{q}} \gamma_q \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{q}} \gamma'_q \ \wedge \ \mathsf{heap}(\gamma_p) \approx_{\Phi^{\mathsf{u}}} \mathsf{heap}(\gamma_q) \ \wedge \ \mathsf{heap}(\gamma'_p) \approx_{\Phi^{\mathsf{u}}} \mathsf{heap}(\gamma'_q)$$

3 Trace Equivalence

Definition 1 is difficult to establish, it requires considering a potentially infinite number of executions and deep inspection of the heap. In this section we propose a sufficient condition for Definition 1, which can be established by inspection of the stack only at certain points in each execution. We show this sufficient condition in Definition 2. We anticipate that this condition combined with suitable approximation techniques will allow us to make progress in applying static analysis to the problem of checking the correctness of refactorings.

In Definition 2 we will consider traces. A *trace* is the sequence of stack frames observable upon entry/exit of Φ^{u}. Traces for our running example are shown in Figure 3. In this case the elements of the trace correspond to the entry and exit points of the constructor, and the `add` and `remove` methods, of the class `FiFo`. Each trace element contains the address of the method receiver and the address of any parameters. On method return we use `ret` as a synthetic method name, and also to name the return value. For example, on entry to the `add` method we capture the address of `this` and the parameter `o`. On exit of the `remove` method we capture the address of the return value. The ith element of trace $\tau\mathsf{s}$ is written as $\tau\mathsf{s}_i$.

Figure 3 also shows examples of isomorphic and non-isomorphic traces. Two traces are equivalent if they have the same aliasing structure and `null` in the same places. Section 4 describes this in more detail. The judgement $\tau\mathsf{s} \approx \tau\mathsf{s}'$ holds whenever $\tau\mathsf{s}$ and $\tau\mathsf{s}'$ are isomorphic modulo addresses.

Definition 2 gives the formal meaning of trace equivalent executions. It holds whenever, for every initial state, execution in p produces an equivalent trace with execution in q. We write $\gamma \xrightarrow{\tau\mathsf{s}}\!\!\!\!\!\rightarrow_{p} \gamma'$ to mean the multi-step execution of program p from configuration γ to configuration γ' with trace $\tau\mathsf{s}$. Trace concatenation is performed by the $\cdot$ function.

▶ **Definition 2** (Trace Equivalent Executions). p and q are *trace equivalent* iff for every state reachable from an initial state with some trace in p (q), a state is reachable from the same initial state in q (p) with an equivalent trace.

p and q are trace equivalent when:

$$\forall\, \tau\mathsf{s}_p, \tau\mathsf{s}_q, \gamma, \gamma_p, \gamma_q : \mathsf{init}(\gamma) \ \wedge \ \gamma \xrightarrow{\tau\mathsf{s}_p}\!\!\!\!\!\rightarrow_{\mathsf{p}} \gamma_p \ \wedge \ \gamma \xrightarrow{\tau\mathsf{s}_q}\!\!\!\!\!\rightarrow_{\mathsf{q}} \gamma_q$$
$$\implies \left(\begin{array}{l} \left(\exists\, \tau\mathsf{s}'_p, \gamma'_p : \gamma_p \xrightarrow{\tau\mathsf{s}'_p}\!\!\!\!\!\rightarrow_{\mathsf{p}} \gamma'_p \wedge \tau\mathsf{s}_p \cdot \tau\mathsf{s}'_p \approx \tau\mathsf{s}_q \right) \ \vee \\ \left(\exists\, \tau\mathsf{s}'_q, \gamma'_q : \gamma_q \xrightarrow{\tau\mathsf{s}'_q}\!\!\!\!\!\rightarrow_{\mathsf{q}} \gamma'_q \wedge \tau\mathsf{s}_p \approx \tau\mathsf{s}_q \cdot \tau\mathsf{s}'_q \right) \end{array} \right)$$

Lemma 3 relates Definition 1 and Definition 2. It directly relates equivalence of traces to equivalence of heaps wrt. Φ^{u}. In particular it says that whenever executions of p and q produce equivalent traces, then they will also correspond in Φ^{u}. Section 6 sketches a proof of Lemma 3.

▶ **Lemma 3.** *If* p *is trace equivalent with* q *(Definition 2) then* Φ^{m} *bounds the modification (Definition 1).*

4 Partitions and Traces

Heap partition is defined by the function part : Object $\rightarrow\{\Phi^{\mathsf{m}}, \Phi^{\mathsf{u}}\}$, which gives a partition identifier for any object. Objects must not move between partitions during execution. The judgement mod(o) holds iff the object o is modified. An *object is modified* iff its class is modified, and a class is modified iff any of its methods' bodies or its fields differ between p and q. All modified objects must be in the Φ^{m} partition, but the Φ^{m} may also contain non-modified objects[1].

We say that Φ^{u} is entered whenever the this pointer of the top stack frame points into Φ^{m} and a stack frame is pushed or popped leaving a top stack frame whose this pointer points into the Φ^{u}. And conversely for exit of Φ^{u}. For example, in Figure 1 if the top stack frame is popped execution will enter Φ^{u}.

$\tau\mathsf{s}_p$

$\tau\mathsf{s}_{p1}$	Fifo	this:	3	
$\tau\mathsf{s}_{p2}$	ret	ret:	3	
$\tau\mathsf{s}_{p3}$	add	this:	3	o: 1
$\tau\mathsf{s}_{p4}$	ret			
$\tau\mathsf{s}_{p5}$	add	this:	3	o: 2
$\tau\mathsf{s}_{p6}$	ret			
$\tau\mathsf{s}_{p7}$	remove	this:	3	
$\tau\mathsf{s}_{p8}$	ret	ret:	1	
$\tau\mathsf{s}_{p9}$	remove	this:	3	
$\tau\mathsf{s}_{p10}$	ret	ret:	2	

$\tau\mathsf{s}_q$

$\tau\mathsf{s}_{q1}$	Fifo	this:	10	
$\tau\mathsf{s}_{q2}$	ret	ret:	10	
$\tau\mathsf{s}_{q3}$	add	this:	10	o: 8
$\tau\mathsf{s}_{q4}$	ret			
$\tau\mathsf{s}_{q5}$	add	this:	10	o: 9
$\tau\mathsf{s}_{q6}$	ret			
$\tau\mathsf{s}_{q7}$	remove	this:	10	
$\tau\mathsf{s}_{q8}$	ret	ret:	8	
$\tau\mathsf{s}_{q9}$	remove	this:	10	
$\tau\mathsf{s}_{q10}$	ret	ret:	9	

$\tau\mathsf{s}_r$

$\tau\mathsf{s}_{r1}$	Fifo	this:	7	
$\tau\mathsf{s}_{r2}$	ret	ret:	7	
$\tau\mathsf{s}_{r3}$	add	this:	7	o: 4
$\tau\mathsf{s}_{r4}$	ret			
$\tau\mathsf{s}_{r5}$	add	this:	7	o: 6
$\tau\mathsf{s}_{r6}$	ret			
$\tau\mathsf{s}_{r7}$	remove	this:	7	
$\tau\mathsf{s}_{r8}$	ret	ret:	6	
$\tau\mathsf{s}_{r9}$	remove	this:	7	
$\tau\mathsf{s}_{r10}$	ret	ret:	4	

$\tau\mathsf{s}_p \approx^{\beta} \tau\mathsf{s}_q$ when $\beta = (1,8),(3,10),(2,9)$ $\qquad \tau\mathsf{s}_q \not\approx \tau\mathsf{s}_r$

Figure 3 Three traces. The trace $\tau\mathsf{s}_p$ is from an execution of p. The trace $\tau\mathsf{s}_q$ is from an execution of q. The trace $\tau\mathsf{s}_r$ is fictional and does not come from either p or q. Trace $\tau\mathsf{s}_p$ is equivalent to trace $\tau\mathsf{s}_q$ under the address bijection, β, shown. Trace $\tau\mathsf{s}_r$ is not equivalent to either $\tau\mathsf{s}_p$ or $\tau\mathsf{s}_q$, there is no bijection between the addresses of $\tau\mathsf{s}_p$ and $\tau\mathsf{s}_r$ that also preserves the aliasing structure of the traces.

Each element of a trace contains the values of method parameters, or method return values, from the top stack frame whenever the partition is crossed. Only the address values, and associated parameter names, actually present in the stack frame are captured, no information from the heap is needed. Since the actual addresses used by any two executions can vary unpredictably we cannot compare traces directly. Instead we say that traces are equivalent if there exists a bijection β between the addresses present in each trace, that preserves the structure of the trace. Figure 3 shows an example of trace equivalence and non-equivalence.

We write $\tau\mathsf{s}_i(x)$ to mean the value at the location associated with the variable x in the ith element of trace $\tau\mathsf{s}$, or $\perp$ if the variable is not defined in that trace element, and $\tau\mathsf{s}_i(\mathsf{meth})$ to mean the name of the method associated with the ith element of trace $\tau\mathsf{s}$.

[1] For example, we may refactor some code to keep a list sorted for improved search performance. The code of list would not be modified, but its state in the heap would be affected.

The equivalence relation $\approx$ over traces, $\tau\mathsf{s}, \tau\mathsf{s}'$, holds if the traces contain the same number of elements, elements at the same position in the sequence are calls to methods with the same identifier, there is a structure preserving bijection between the addresses in each trace, and both traces have null at the same locations. Therefore, whenever $\tau\mathsf{s} \approx \tau\mathsf{s}'$ holds, whatever is an alias in $\tau\mathsf{s}$ is an alias in $\tau\mathsf{s}'$ and vice versa.

▶ **Definition 4.** The relation $\approx$ holds whenever two traces contain the same sequence of method names, and when the aliasing structure of both traces is precisely the same.

$$\approx^{\beta} \subseteq \mathsf{Trace} \times \mathsf{Trace}$$

$$\tau\mathsf{s} \approx^{\beta} \tau\mathsf{s}' \overset{\text{def}}{\iff} \forall i \in \mathbb{N}, x \in \mathrm{Id}_v : (\tau\mathsf{s}_i(\mathsf{meth}) = \tau\mathsf{s}'_i(\mathsf{meth}) \ \wedge \ \beta(\tau\mathsf{s}_i(x)) = \tau\mathsf{s}'_i(x)) \ \wedge$$
$$\beta(\mathsf{null}) = \mathsf{null} \ \wedge \ (\forall a, a' : \beta(a) = \beta(a') \implies a = a')$$

$$\tau\mathsf{s} \approx \tau\mathsf{s}' \overset{\text{def}}{\iff} \exists \beta : \tau\mathsf{s} \approx^{\beta} \tau\mathsf{s}'$$

5 Properties of Bounded modifications

In this section we describe some properties of executions when part has been picked such that Definition 1 holds, i.e. Φ^{m} bounds the modification of interest. The usefulness of definition 1 is that it helps a programmer to reason about heaps, stacks and executions. We now describe some properties of heaps, stacks and executions that follow from Lemma 3 and hold for bounded modifications.

The structure of executions wrt. the partitioning is a sequence of execution steps in one partition, followed by a sequence of execution steps in the other partition, then back to the original partition, and so on in an interleaved fashion as shown in Figure 4.

Corollary 5 describes the sequences of states that the Φ^{u} partition reaches when execution is in Φ^{m}. We write $\mathsf{part}(\gamma)$ to mean the partition of the object pointed to by the this pointer of the topmost stack frame of state γ. Figure 4 illustrates the preservation of Φ^{u} heap equivalence, when execution is in Φ^{m}, that the corollary describes. In particular, between corresponding parts of the executions in Φ^{m} all of the heaps are equivalent wrt. the Φ^{u} partition.

▶ **Corollary 5.** *When Φ^{m} bounds the modification, every state with execution inside Φ^{m} reachable with some trace in p, and every state reachable with some equivalent trace in q have equivalent heaps wrt. Φ^{u}.*

$$\forall \gamma, \gamma_p, \gamma_q, \tau\mathsf{s}_p, \tau\mathsf{s}_q : \mathsf{init}(\gamma) \ \wedge \ \gamma \xrightarrow{\tau\mathsf{s}_p}\!\!\!\!\!\rightarrow_{\mathsf{p}} \gamma_p \ \wedge \ \gamma \xrightarrow{\tau\mathsf{s}_q}\!\!\!\!\!\rightarrow_{\mathsf{q}} \gamma_q \ \wedge \ \tau\mathsf{s}_p \approx \tau\mathsf{s}_q \ \wedge \ \mathsf{part}(\gamma_p) = \Phi^{\mathsf{m}}$$
$$\implies \mathsf{heap}(\gamma_p) \approx_{\Phi^{\mathsf{u}}} \mathsf{heap}(\gamma_q)$$

Corollary 6 describes the sequences of states that the Φ^{u} partition reaches for execution in Φ^{u}. We write $\xrightarrow{\tau}_p$ to mean a single execution step in program p that produces the non-empty trace τ, and we write $\xrightarrow{\epsilon}_p$ to mean a single execution step in program p that produces an empty trace. We use the equivalence $\gamma \approx_{\Phi^{\mathsf{u}}} \gamma'$ to mean that the part of the heap consisting only of objects in Φ^{u} and the topmost stack frames of γ and γ' are isomorphic modulo addresses. Figure 4 illustrates the lockstep equivalence wrt. Φ^{u} when execution is within Φ^{u}.

In particular, between corresponding parts of the executions in Φ^{u}, the same number of steps are executed in p as q, and corresponding pairs of states are equivalent wrt. Φ^{u}. Intuitively, the two executions proceed in lockstep for as long as execution remains in Φ^{u}.

▶ **Corollary 6.** *When Φ^{m} bounds the modification, for every state reachable n steps after entering Φ^{u} with some trace in p, there is an equivalent state wrt. Φ^{u} reachable n steps after*

each entry to Φ^{u} *that is reachable with an equivalent trace in* q*, provided that none of the* n *steps in* p *crosses out of* Φ^{u}*.*

$$\begin{array}{l}\forall \gamma, n, \gamma_p, \gamma_{p1} \ldots \gamma_{pn}, \gamma_q, \gamma'_q, \tau\mathsf{s}_p, \tau_p, \tau\mathsf{s}_q, \tau'_q : \exists \gamma_{q1} \ldots \gamma_{qn} : \\ \quad \mathsf{init}(\gamma) \ \wedge \ \mathsf{part}(\gamma_{p1}) = \Phi^{\mathsf{u}} \ \wedge \ \tau\mathsf{s}_p \cdot \tau_p \approx \tau\mathsf{s}_q \cdot \tau_q \\ \quad \gamma \xrightarrow{\tau\mathsf{s}_p}\!\!\!\!\rightarrow_{\mathsf{p}} \gamma_p \xrightarrow{\tau_p}_{\mathsf{p}} \gamma_{p1} \xrightarrow{\epsilon}_{\mathsf{p}} \ldots \xrightarrow{\epsilon}_{\mathsf{p}} \gamma_{pn} \ \wedge \ \gamma \xrightarrow{\tau\mathsf{s}_q}\!\!\!\!\rightarrow_{\mathsf{q}} \gamma_q \xrightarrow{\tau_q}_{\mathsf{q}} \gamma'_q \\ \Longrightarrow \\ \quad \gamma'_q = \gamma_{q1} \ \wedge \ \gamma_{q1} \xrightarrow{\epsilon}_{\mathsf{q}} \ldots \xrightarrow{\epsilon}_{\mathsf{q}} \gamma_{qn} \ \wedge \ \bigwedge_{i=1}^{n} \gamma_{pi} \approx_{\Phi^{\mathsf{u}}} \gamma_{qi}\end{array}$$

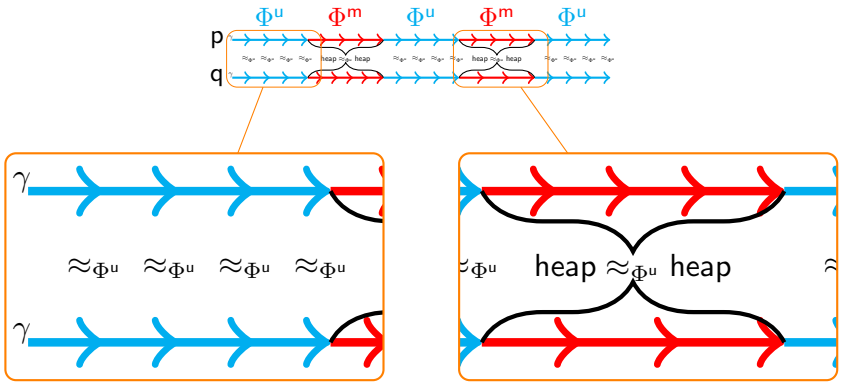

Figure 4 Two executions from initial state γ are shown. The top execution is of program p and the bottom execution is of q. The modification between p and q is assumed to be bounded by Φ^{m}. The leftmost magnified area shows the step wise correspondence of execution steps when execution is within Φ^{u}. The pairs of states are stepwise equivalent wrt. Φ^{u}, in accordance with Corollary 6. The rightmost magnified area shows the preservation of heap equivalence wrt. Φ^{u} whilst execution is in Φ^{m}, in accordance with Corollary 5.

6 Proof Sketch

We provide two auxiliary lemmas that support Lemma 3. Lemma 7 says that for as long as an execution remains outside of a partition, it will not affect the state of any heap objects inside of the partition. We write $\mathsf{heap}(\gamma)_\phi$ to mean the the heap of state γ with objects outside ϕ deleted (leaving dangling pointers if necessary). We justify this lemma by noting that: privacy of fields prevents manipulation of the state of a partition unless the this pointer is pointing inside that partition; entering the partition would cause the trace to be non-empty.

▶ **Lemma 7.** *As long as the execution is outside the partition the heap inside the partition is preserved.*

$$\forall p, \gamma, \gamma', \phi : \gamma \xrightarrow{\epsilon}\!\!\!\!\rightarrow_p \gamma' \ \wedge \ \mathsf{part}(\gamma) \neq \phi \implies \mathsf{heap}(\gamma)_\phi = \mathsf{heap}(\gamma')_\phi$$

Lemma 8 says that the behaviour of an execution while it is in a partition ϕ is not affected by the state of the heap outside of that partition. We write γ_ϕ to mean the state which is the same as γ, but with any objects not in ϕ deleted from the heap. In particular that, unless execution leaves ϕ, the heap inside the partition will reach the same heap states wrt. the partition dependent only on the state inside the partition. We justify this lemma by noting that: field privacy prevents observation of the state of a partition unless execution is

in the partition; execution is deterministic; and the continuations of γ and γ_ϕ are the same. Therefore, only the aliasing structure of pointers inside the partition (including those that point out of the partition), and the code of the classes of objects inside the partition, affects the execution inside the partition.

▶ **Lemma 8.** *The state of the heap outside a partition does not affect execution inside the partition. In particular, if all the objects outside of the partition ϕ were deleted from the heap, execution will still reach equal states wrt. ϕ as long as execution does not leave ϕ.*

$$\forall p, \phi, \gamma, \gamma' \exists \gamma'' : \mathsf{part}(\gamma) = \phi \ \wedge \ \gamma \xrightarrow{\epsilon} \!\!\!\!\rightarrow_p \gamma' \implies \gamma_\phi \xrightarrow{\epsilon} \!\!\!\!\rightarrow_p \gamma'' \ \wedge \ \gamma'_\phi = \gamma''$$

7 Conclusion

We have argued that information about aliasing in partition crossing calls is sufficient to check the propagation of heap effects between areas of the heap. We have presented a definition of what it means for the heap effect of a program modification to be contained within a programmer defined heap partition.

We intend to continue this work by performing a proof of the lemmas given in this paper. We will then try to extend this work to characterise program modifications that are not behaviour preserving.

References

1 Fabian Bannwart and Peter Müller. Changing programs correctly: Refactoring with specifications. *FM 2006: Formal Methods*, pages 492–507, 2006.

2 Ran Ettinger. Program sliding. In James Noble, editor, *ECOOP 2012 Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 713–737. Springer Berlin Heidelberg, 2012.

3 B. Godlin and O. Strichman. Regression verification. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 466 –471, july 2009.

4 Chris Hawblitzel, Ming Kawaguchi, Shuvendu K Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. *International Conference on Automated Deduction*, June 2013.

5 Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 453–464, New York, NY, USA, 2009. ACM.

6 Shuvendu Lahiri, Ken McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Foundations of Software Engineering*. ACM, 2013.

7 Leonardo Mariani, Fabrizio Pastore, and Mauro Pezze. Dynamic analysis for diagnosing integration faults. *IEEE Trans. Softw. Eng.*, 37(4):486–508, July 2011.

8 Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *ACM SIGPLAN Notices*, volume 31, pages 235–250. ACM, 1996.

9 David Naumann and Anindya Banerjee. State based encapsulation for modular reasoning about behaviour-preserving refactorings. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-oriented Programming*. Springer State-of-the-art Surveys, 2012.

10 William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, 1992.

11 Javier Perez, Yania Crespo, Berthold Hoffmann, and Tom Mens. A case study to evaluate the suitability of graph transformation tools for program refactoring. *Software Tools for Technology Transfer - Special Section on GraBaTs 08*, 12(3-4):183–199, 2009. (c) Springer, 2009.

12 Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege Moor. Stepping stones over the refactoring rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag.

13 Yannick Welsch and Arnd Poetzsch-Heffter. Verifying backwards compatibility of object-oriented libraries using boogie. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 35–41, New York, NY, USA, 2012. ACM.