Detecting Full Initialization Points of Objects to Support Code Refactorings

Girish Maskeri Rama
Infosys Technologies, India
Email: girish_rama@infosys.com

Raghavan Komondoor
Indian Institute of Science, Bangalore
Email: raghavan@iisc.ac.in

Abstract—A common need of refactorings that involve object allocations is to determine precisely the program points at which objects allocated at a given allocation site become fully initialized. In this paper we formalize the notion of full initialization points (FIPs) of allocation sites, and present a static analysis to determine precisely these program points. While this analysis can benefit several allocation-site refactorings, to demonstrate its usefulness we select two specific refactorings in this paper – object sharing refactoracting and immutability refactoring. By introducing code to cache and share objects at the FIPs suggested by our analysis, object-sharing refactoring was able to obtain a mean memory savings of 11.4% on a set of real Java benchmarks. Immutability refactoring guided by our analysis achieved a mean runtime speedup of 1.6X compared to performing the same refactoring using a baseline approach.

I. INTRODUCTION

Several allocation-site based refactorings have been proposed in literature such as refactoring code for object-sharing [1], immutability refactoring [2], and refactoring to introduce design patterns (especially creational patterns such as abstract factory and builder) [3], [4]. An allocation site is the focus of these refactorings, and has a direct bearing on the code that is analyzed and modified as part of these refactorings. For instance, the object-sharing refactoring takes an allocation site and inserts code at an appropriate program point after the allocation site to share objects created at that allocation site [1]. Immutability refactoring takes an allocation site, transforms the class of the allocated object into an immutable class and changes all field updates of the allocated objects after the allocation statement into method calls that preserve immutability [2]. The refactoring to introduce the builder pattern takes an allocation site that creates a complex composite object, identifies all the statements after the allocation site that build the composite object, and moves them into a separate builder class [4], [5].

A common thread that runs across these refactorings is the need to identify the locations in the program after an allocation site by which any object freshly allocated at that site becomes fully initialized.

A. Full Initialization Points

Though objects should ideally be initialized in the constructor, it is commonplace to find code that initializes each object after it is allocated [6], [7]. For example, consider the code snippet shown in Listing 1. This shows only the code for method ‘foo()’; we have omitted other details such as the definition for class ‘ClassA’ for brevity. The allocation site at Line 2 returns a new object of class ‘ClassA’ and assigns to variable variable1. But the newly allocated object is not yet fully initialized at Line 2. The attribute field of the object allocated at Line 2 is set subsequently at Line 5.

Listing 1. Deferred initialization example

```java
public ClassA foo(int i){
    ClassA variable1 = new ClassA(); // allocation site 's'
    variable1.field=0;
    if(i/2 > 1){
      variable1.field=1;
    }
    return variable1;
}
```

Hence, the point after Line 6 (and not the point after Line 2) is the full initialization point for the allocation site in Line 2.

A more challenging form of deferred initialization, termed loop deferred initialization, is when the allocation site is inside a loop, and an object created in one iteration is initialized in a subsequent iteration. For instance, consider the code in Listing 2 where the allocation site s is enclosed in a loop.

Listing 2. Deferred initialization inside loop

```java
public ClassA foo(int i){
    ClassA variable = null;
    ClassA variable2 = null;
    for(i=0;i<n;i++){
      variable = new ClassA(); // allocation site 's'
      if(i % 2 == 0)
        variable2 = variable1;
      variable2.field= i;
    }
    return variable;
}
```

In this example, it may appear that the point after Line 8 is the full initialization point for the allocation site s. However, this is actually not the case, because after an object is created in the first iteration of the loop, and after control crosses Line 8 in the first iteration and then comes back again to Line 8 in the next iteration of the loop, at this point the object created in the first iteration gets mutated. In other words, the first time Line 8 was crossed (in the first iteration), the most recently allocated object at site s is not yet fully initialized.
In other words Line 8 is not a full initialization point at all for the allocation site $s$.

Another candidate location for the full initialization point is the point after Line 10 (i.e., outside the loop). The objects created inside the loop indeed are fully initialized by the time execution crosses this point. However, as will become clear in Section II-A, many allocation-site refactorings need an additional property from full initialization points: any object created at an allocation site must cross a full initialization point and hence become fully initialized before the same site is visited again in the run. The point after Line 10 does not satisfy this property. In fact, in the example in Listing 2, there does not exist any suitable full initialization point at all for the allocation site $s$.

In other cases, a given allocation site can have multiple FIPs. For instance in Listing 1, the location after Line 5 and the location after Line 6 both are FIPs (because there are two paths from $s$, with each of these locations serving as an FIP along one of the paths).

To summarize informally, a program point $p$ can be considered to be a full initialization point (FIP) for an allocation site $s$ if in any run of the program, and for any object $o$ that is allocated during any visit to the site $s$ during the run: (a) No more mutations to $o$ (or objects reachable from $o$) occur after execution crosses point $p$ for the first time after object $o$ is created, and (b) between the time $o$ is created and the time $p$ is visited for the first time after $o$ is created, the allocation site $s$ is not visited again. We formalize the notion of an FIP more fully in Section III.

This notion of FIP is substantially different from other extant notions of object initialization [8], [9], [10], [7], [6], [11]. These approaches ignore deferred initializations inside loops, as in Listing 2, which in fact interfere with many allocation-site refactorings. Most of these approaches also do not consider objects reachable from a “root” object as being implicitly part of the state of the root object, which is something that allocation-site based refactorings typically rely on.

B. Contributions

In summary, this paper makes the following contributions:

- We define a novel concept termed full-initialization points (FIPs) to characterize the points in the program where objects allocated at a given allocation site $s$ become fully initialized. This notion can support various allocation site based refactorings, such as object sharing refactoring and immutability refactoring.
- We present a novel and conservative static analysis to detect FIPs for a given allocation site.
- Guided by our implementation of our static analysis, we could successfully apply object-sharing refactoring on 17 allocation sites across 5 real-world benchmarks. We observed 24% reduction in heap footprint in the best case, and a geometric mean of 11.40% of memory savings across all benchmarks.
- We applied an improved version of immutability refactoring, based on the notion of FIPs, on 5 real-world benchmarks. We observed 8X speedup in the best case, and a geometric mean of 1.6X speedup across all benchmarks, over standard immutability refactoring [2].

The rest of the paper is structured as follows. Section II motivates the use of FIPs in allocation site based refactorings and discusses the challenges in identifying FIPs. Section III provides our basic definitions, including the main definitions relating to FIPs. Section IV describes our approach. Section V briefly discusses our implementation, while Section VI discusses our empirical evaluation in detail. Discussion on the results, limitations and threats to validity is presented in Section VII. Section VIII discusses related work, while Section IX concludes the paper.

II. USE OF FIPS IN REFACTORINGS

We now discuss applications of our notion of FIPs in the context of a couple of allocation site based refactorings.

A. Object sharing refactoring

The goal of object-sharing refactoring [1] is to reduce the heap memory usage by sharing isomorphic objects allocated at a given allocation site. Intuitively, two objects are isomorphic if they are of the same type, have identical values in corresponding primitive fields, and are such that corresponding reference fields themselves point to isomorphic objects. Objects allocated at a given allocation site are cached, and a reference to any newly created object $o$ is replaced with a reference to an object that is isomorphic to $o$ if such an object is available in the cache. The newly allocated object $o$ then gets garbage collected, resulting in memory savings. The program point in the program at which a search is made in the cache for an object that is isomorphic to the most recently allocated object at an allocation site $s$ is referred to as the hash-consing point for site $s$.

Listing 4 shows the object-sharing refactoring performed for the allocation site in Line 2 in Listing 1. Here, the point after Line 6 in Listing 1 has been chosen as the hash-consing point. The newly added code at the hash-consing point appears in a different font in Listing 4.

```
main() {
    ClassA variable2 = foo(1);
    ClassA variable3 = foo(4);
    ClassA variable4 = foo(4);
    print(variable2.field);
    print(variable3.field);
    print(variable4.field);
}
```

Listing 3. Main method
In the rest of this discussion, we discuss how certain properties of FIPs, namely, soundness and optimality have an impact on the efficacy of object-sharing refactoring.

1) Soundness of FIPs: a program point is a sound FIP if it satisfies the definition of FIP, which was given informally in Section I-A and will be given more formally in Section III. Object sharing refactoring is correct only if the hash-consing point(s) are after FIP(s) in the program paths. For instance, in the refactored code shown in Listing 4, the hash-consing point is immediately after an FIP (the one in Line 6 in Listing 1). Object-sharing refactoring can in general become incorrect if the hash-consing point is encountered before the first FIP in a path. For instance, say hash-consing is done immediately after Line 3 in Listing 1. Consider the routine `main` in Listing 3, which calls `foo` in Listing 1 three times. Originally, the three print statements in `main` would print 0, 4, and 4, respectively. Whereas, with incorrect hash-consing right after Line 3 in Listing 1, `main` would print 4, 4 and 4.

2) Optimality of FIPs: An optimal FIP for an allocation site s is intuitively one that appears as close to s as possible while being sound. For instance in Listing 1 the point after Line 6 is an optimal FIP for allocation site s. As another illustration, consider the example in Listing 5. The point after Line 4 is an optimal FIP for the allocation site s. The point after Line 5 is an FIP, but not an optimal FIP for the site s. If hash-consing is done after Line 5 at a non-optimal FIP (but still within the loop body), then no memory savings will result. This is because each allocated object escapes into the collection `arr`, meaning none of them can be garbage collected.

3) Impact of loops: Hashconsing cannot be done profitably inside loops when there exists a loop deferred initialization for the allocation site. For instance, in the example in Listing 2, if a point after Line 10 (i.e., outside the loop) is selected as an FIP, then no benefit is obtained after object-sharing refactoring. Only a reference to the last object allocated in the loop is still available at this point, all other objects allocated cannot be shared. Therefore, almost no memory savings would result by performing hashconsing at this point.

B. Immutability refactoring

Immutability refactoring [2] is a refactoring whose primary objective is to make fields of classes read-only. For example consider the sample code in Listing 6, where class `ClassA` is mutable. The refactored code where class `ClassA` is made immutable, is shown in Listing 7. The field `attribute1` has been marked as a final (i.e., read-only) field and the mutator method `setAttr()` has been modified as follows: It first obtains a deep copy of the the received object using the deepClone() API method (Line 7) in the cloner cloning library [12]. It then updates this fresh copy using the original setter method that has now been renamed as `setAttr_orig` and then returns this updated copy. The net effect of these code changes is that objects of class `ImmutableClassA` become immutable. We can observe that in the calling code (see Lines 18-20 in Listing 7) the method `setAttr()` is invoked repeatedly in a loop. Thus, the deep cloning operation would be repeatedly performed in the refactored code (see Line 7 in Listing 7). However, all this deep copying, which is in general an expensive operation, can drastically reduce performance.

We therefore propose a scheme for a developer to use the concept of FIPs to make further code changes after the immutability refactoring that can reduce the extent of deep-cloning that is performed. We can observe that Lines 18-20 in Listing 7 is in fact a deferred initialization of the objects allocated at s and our analysis would identify the point after Line 19 as an FIP. Our proposal is to move mutation code starting from the allocation site up to the FIPs to inside the constructor. That is, modify the loop in Lines 18-20 as follows: “for (b: someBcollection) setAttr_orig(b);”. Then move this (modified) loop into the constructor that is invoked
### III. Notation and Problem Statement

In this section we will formally define the notions of FIP, optimal FIP and loop deferred initialization, which we had informally introduced in Section I.

Recall, as discussed in Section II, that the motivation behind the concept of FIPs is to assist in finding program locations where code changes related to allocation site refactorings need to be introduced. In general, an object allocated at a site can be mutated anywhere in the program. Therefore, an FIP can appear anywhere in the program. In such cases, an FIP may not be useful to the developer, because they might be reluctant to introduce code changes far away from the allocation site that is being refactored. Such code changes made to the program harder to understand and maintain in the future. To deal with this situation we introduce the concept of a local full initialization points (LFIP). Let $f_s$ be the function containing the allocation site $s$. Intuitively, an LFIP is a FIP if the function $f_s$ were to be considered as if it is the entire program. In other words every LFIP of $s$ definitely occurs within $f_s$. More formally:

**Definition 1.** Local Full Initialization Point (LFIP): Given an allocation site $s$, a point $p$ that is in the function $f_s$ that contains $s$ is said to be a LFIP for $s$, if in every trace $r$ that goes through $s$ and then goes through $p$, letting $t_1$ be the time instant when control returns from the constructor at site $s$, and letting $t_2$ be the first time $r$ crosses point $p$ after time $t_1$, and letting $o$ be the object allocated at time $t_3$ at site $s$:

- r1: there is no mutations to the object $o$ or to objects reachable from $o$ at time $t_2$ after time $t_2$, and before the trace leaves function $f_s$.
- r2: if $t_3$ is any time instant between $t_1$ and $t_2$ such that $o$ or the objects reachable from $o$ immediately before time $t_3$ is mutated at time $t_3$, then trace $r$ does not revisit site $s$ between $t_1$ and $t_3$ and before the trace leaves function $f_s$.

Note that by a “trace”, we refer to the sequence of statements visited in some run of the program.

The clause r1 ensures that all mutations to an object $o$ are completed before $p$ is visited for the first time after $o$ is allocated. The clause r2 ensures that there is no run where an object $o_1$ allocated at this site, or any object reachable from $o_1$, gets mutated after a subsequent visit to the same allocation site in the same run (a different object $o_2$ would be allocated during this subsequent visit).

**Definition 2.** Local Loop Deferred Initialization (LLDI): An allocation site $s$ inside a loop $l$ in the function $f_s$ that contains the site $s$ is said to undergo LLDI if there is some trace $r$ such that $r$ goes through $s$ at time $t_1$ as well as at a later time $t_2$, and the object allocated at $s$ at time $t_1$ or objects
reach from \( o \) immediately before time \( t_2 \) get mutated in the trace after time \( t_2 \) before the trace leaves function \( f_s \).

As an illustration of this notion, the allocation site \( s \) in Listing 2 is present in a loop, and undergoes LLDI. Whereas, the allocation site \( s \) in Listing 5, although being present inside a loop, does not undergo LLDI.

It is easy to see that one or more LFIPs necessarily exist for the allocation site \( s \) if the site does not undergo LLDI. We also argue later (in Section IV-C) that the absence of LLDI suffices to prove that Clause r2 in Definition 1 holds for any LFIP of \( s \).

**Definition 3.** Optimal Local Full-Initialization point (OLFIP) Given an allocation site \( s \) and its corresponding set of LFIPs \( P \), a LFIP \( p_i \in P \) is said to be a optimal local full-initialization point (OLFIP) if it is the first LFIP encountered after \( s \) in at least one trace.

### A. Other assumptions and notations

Our analysis requires a conservative points-to analysis, such as that of Milanova et al. [13]. This constructs a static points-to graph. A symbolic object is an abstraction of heap allocated objects. The vertices in a points-to graph are variables and symbolic objects. Edges represent points-to relations among these vertices. In particular, \( P_v(g) \) denotes the set of symbolic objects that variable \( g \) may point to, while \( P_f(o_1, h) \) denotes the set of symbolic objects that field \( h \) of symbolic object \( o_1 \) may point to.

Let \( \text{Mod}(m) \) denote the set of symbolic objects that are potentially modified by a statement \( m \). If \( m \) is a call statement, it denotes the set of symbolic objects that are potentially modified by the called method or its callees transitively. This can be computed using points-to information [13].

Let \( \text{Alloc}(s) \) denote the set of symbolic objects allocated at a given allocation site \( s \). Given an allocation site \( s \), let \( O_s \) be the set of symbolic objects that are transitively reachable from each object in \( \text{Alloc}(s) \) in the static points-to graph.

Let \( F = \{m_1, m_2, ..., m_n\} \) be the set of statements in \( f_s \). A control flow graph (CFG) is a graph whose nodes are statements and edges represent flow of control [14]. We will use the notation \( m \rightarrow n \) to mean that statement \( n \) is a direct successor of \( m \) in the control flow graph (CFG), and \( m \leadsto n \) to mean that \( n \) is reachable from \( m \) in the CFG. Let \( \text{Edges} \) be a set containing all the edges in the CFG of \( f_s \). While CFG represents the control flow for a given procedure, ICFG (Inter-Procedural Control Flow Graph) represents control flow for an entire program.

### IV. OUR APPROACH

Given a site \( s \) the problem is to identify LFIPs for \( s \) using static analysis. The analysis should be sound and the identified LFIPs should be as close to OLFIPs as possible.

Given an allocation site \( s \), our approach consists of two major steps: **Step 1.** Check whether there exists at least one LFIP. **Step 2.** If LFIPs exist, identify LFIPs as close to the optimal LFIPs as possible.

#### A. Step 1: Detecting local loop-deferred initialization

We perform LLDI checking first because one of more LFIPs exists only if there is no LLDI. Say we have a putfield statement \( u = \text{“}p.q = r\text{”} \) that might modify objects reachable from an object allocated at the given site \( s \). Let \( l \) be some common loop \( l \) in function \( f_s \) that contains \( s \) and \( u \) or contains \( s \) and method call that directly or transitively calls a method that contains \( u \). The key intuition is that we want to check if some value that is present in some memory location when control is at the header of \( l \) flows into \( ‘p’ \) in statement \( u \) eventually. If this does not happen, then LLDI cannot exist, because \( ‘p’ \) cannot have a value coming from a previous iteration.

Let \( M \) be the set of all putfield statements in the program. If \( u \) is a putfield statement, then let \( L(u) \) be a set that contains a loop \( l \) in the function \( f_s \) such that (a) \( l \) contains the allocation site \( s \), and (b) the body of loop \( l \) contains \( u \) directly or contains a call that directly or transitively calls a method that contains \( u \). \( L(u) \) could be empty.

Algorithm 1 depicts the pseudo-code to conservatively detect LLDI.

**Algorithm 1:** checkLocalLoopDeferredInitialization

```plaintext
input : Allocation site \( s \), function \( f_s \)
output: true if LLDI exists, false otherwise

1. \( U = \{m \mid m \in M \land \text{Mod}(m) \cap O_s \neq \emptyset\} \)
2. foreach \( \langle u, l \rangle \mid u \in U \land l \in L(u) \) do
3. if checkIfUninitialized\((u, l)\) then
4. \[ \text{return true;} \]
5. return false;
```

The objective of Algorithm 1 is to check if there exists a path from the header of \( l \) to \( u \) such that some value in some memory location when control is at the header eventually flows into the variable \( ‘p’ \) in statement \( u \). This checking is delegated to the subprocedure checkIfUninitialized\((u, l)\), which is designed as a context sensitive backward data flow analysis.

1. **Procedure checkIfUninitialized\((u, l)\):** An access path is either a variable, or a variable followed by a sequence of field references; e.g., ‘v’, ‘v.f1’, ‘v.f1.f2’, etc. Our idea is to start with the access path ‘\( p \)’ at statement \( u \), propagate this access path in the backward direction along the inter-procedural control graph. The access paths are rewritten as they are propagated. The idea is that the access paths obtained at any program point by the analysis contain the values that eventually flow into ‘\( p \)’ at statement \( u \). Now, if
any such access path is obtained at the header of loop \(l\), the procedure returns `true`.

For instance, consider the code in Listing 2. The putfield statement is in Line 8. The analysis starts with the access path ‘variable2’ at the point before Line 8. When it goes through Line 7, the ‘variable2’ is rewritten to ‘variable1’. Along the path from Line 8 onward to Line 6, ‘variable2’ is transferred directly. Therefore, both ‘variable1’ and ‘variable2’ reach Line 5. Line 5 blocks the propagation of ‘variable1’ (because variable ‘variable1’ is being initialized here to a fresh value). Therefore, ‘variable2’ reaches the header of loop in Line 4. Since some access path has reached this header, the procedure returns `true`. Conversely, in Listing 5, when ‘variable1’ is propagated up from Line 4, no access path reaches the loop header in Line 2 (due to the initialization in Line 3). Therefore, the procedure returns `false`.

More formally, our dataflow `lattice` is such that each element of it is a set of access paths. The “join” operation used at join points (in the backward direction) is a set union operation. The “transfer” function for a statement of the form ‘\(v = w\)’ is \(\lambda d.d[w/v]\). Here, \(d\) is a set of access paths, while \(d[w/v]\) denotes replacement of \(v\) with \(w\) in all access paths in \(d\). The transfer function for ‘\(v = w.g\)’ is \(\lambda d.d[w,g/v]\). For ‘\(v = \text{new}\)’ it is, \(\lambda d.\{ap_i | ap_i \in d \land \neg \text{prefix}(v, ap_i)\}\). The predicate `prefix(ap_i, ap_j)` is true if \(ap_i\) is (textually) a prefix of \(ap_j\).

The transfer function for the putfield statement ‘\(r.f = v\)’ is more complicated, and is as depicted in Figure 1.

\[
\lambda d. \left\{ \begin{array}{l}
\{ap_i | ap_i \in d \land \neg \text{prefix}(r.f, ap_i) \} \\
\lor \\
\{ap_i \in d \land \text{prefix}(w.f, ap_i) \lor \text{mayalias}(w, r) \land ap_i = ap_j[v/w.f]\}
\end{array} \right.
\]

Figure 1. Transfer function for putfield ‘\(r.f = v\)’

Note that this transfer function relies on a “may alias” query, which can be answered using the static points-to graph. When a fix-point solution is reached by the dataflow analysis, the procedure `checkIfUninitialized` returns `true` if a non-empty set of access paths is available at the header of the loop \(l\), and returns `false` otherwise.

Note that the algorithm can in general go into non-termination in the presence of statements such as ‘\(v = v.f\)’ inside loops. Therefore, whenever our approach generates any access path with a field name that appears more than once in the access path, we terminate the analysis and conservatively return `true`.

**B. Step 2: Identifying LFIPs**

Given an allocation site \(s\), if LFIPs exist as per Step 1, the goal of this step is to identify LFIPs as close to the optimal LFIPs as possible. Our algorithm to detect LFIPs for a given allocation site \(s\) is shown in Algorithm 2.

**Algorithm 2: IdentifyLFIPs**

**input**: Allocation site \(s\) in function \(f_s\) and CFG of \(f_s\)

**output**: Set of program points in \(f_s\) identified as LFIPs for \(s\).

1. \(LFIPs = \{\}\)
2. \(Reachable = \{m \in F | s \rightarrow m\}\)
3. \(logicalMutators = \{m | m \in F, \text{Mod}(m) \cap O_s \neq \emptyset\}\)
4. \(ModAnticipable = \{m' | m, m' \in F, m \in \text{logicalMutators}, (m' \in \text{logicalMutators}) \lor m' \rightarrow m \land s \rightarrow m'\}\)
5. \(LFIPs = \{(m_1, m_2) | (m_1, m_2) \in \text{Edges} \land m_1 \in \{\text{Reachable} \land \text{ModAnticipable}\} \land m_2 \in \{\text{Reachable} \land \text{ModAnticipable}\}\}\)
6. return \(LFIP\)

This algorithm is an intra-procedural traversal of the CFG of method \(f_s\). An illustration of this step is shown in Figure 2 which shows the CFG of the method \(f_s\). In Line 2 in Algorithm 2 we create the set `Reachable`, which contains all statements in the set \(f_s\) that are reachable from \(s\) in the CFG of method \(f_s\). Subsequently, in Line 3 we identify all `logical mutator statements` for the allocation site \(s\). That is, all statements that directly or via callees mutate any symbolic object in the set \(O_s\). These are represented as shaded circles in Figure 2.

In Line 4 we construct the set `ModAnticipable` by collecting all statements in \(f_s\) that are encountered in a backward traversal from any of the logical mutator statements. Any
program point before any statement in the set \textit{ModAnticipable} cannot be an LFIP. Hence, any CFG edge whose source is in the set \textit{ModAnticipable} and whose target is in the set \textit{Reachable} is added to the set \textit{LFIP}. That is, in Figure 2 all outgoing edges from a statement in the heavily shaded region to a statement in the lightly shaded region are LFIPs; these edges are marked with shaded rectangular boxes in the figure. The key intuition is that these edges represent (earliest possible) program points after which we will never encounter a mutating statement along a path.

C. Proof of correctness

Our approach is sound, in that all program points returned by Algorithm 2 are necessarily LFIPs. For this, we need to show that every CFG edge \( l \) in the set returned by Algorithm 2 is necessarily an LFIP. For this, we need to show that \( l \) satisfies clauses r1 and r2 in Definition 1. We argue this informally in the interest of space.

It is easy to see that Algorithm 2 ensures that Clause r1 is satisfied. An important assumption here is that pointer analysis ensures that the set ‘Mod(m)’ overapproximates the set of symbolic objects that are modified by stament \( m \) and by all possible direct and transitive callees of \( m \) in case \( m \) is a call statement.

From the definition of LLDI (Definition 2), it is easy to see that Clause r2 is satisfied if the allocation site \( s \) does not undergo LLDI. Therefore, it suffices for us to show that Algorithm 1 detects LLDI conservatively. Intuitively, this holds because an LLDI can occur only if a value flows into the base variable \( p \) of a put-field statement \( u \) along a path from the header of a loop \( l \) that contains the allocation site \( s \) and contains either \( u \) or a call statement that directly or transitivity invokes the method that contains \( u \). However, if such a value flow occurs, the access paths at the loop header’s point that could contain this value must appear in the dataflow analysis solution at the loop header, due to the soundness of the transfer functions in Section IV-A.

Therefore, checkIfUninitialized\((u, l)\) will return \textit{true}, causing Algorithm 1 to detect the LLDI.

V. IMPLEMENTATION

We have implemented our algorithm to detect LFIPs, specified in Algorithm 2, using the Wala static analysis framework [15]. We used the class ‘ExplodedControlFlowGraph’ of Wala to obtain the control-flow graphs (CFGs) of all the methods in the program. We implemented the standard loop detection algorithm [16] that is based on identifying dominators within Wala to identify loops and loop headers. As mentioned in Section IV-A, instead of implementing our LLDI backward analysis directly, we implemented it as a simple wrapper over the null-dereference analysis of Madhavan et al. [17]. Given a query, which is a dereference of a pointer ‘\( q \)’ at a statement \( w \), and given any program point \( r \), the null-dereference analysis computes a \textit{weakest pre-condition} (i.e., a formula in the variables of the program) such that if this formula is \textit{false} whenever execution reaches \( r \), then ‘\( q \)’ cannot possibly have value null when the same execution later reaches statement \( w \). The correlation between our backward analysis and the null-dereference analysis is as follows: Given the put-field statement \( u = ‘p.q = r’ \), and the header \( h \) of a loop \( l \), a value that resides in some access path at location \( h \) can flow into variable ‘\( p \)’ at statement \( u \) only if the null-dereference analysis determines as a weakest pre-condition at point \( h \) some formula that is not \textit{false} given the dereference of ‘\( p \)’ at statement \( u \) as the query. Therefore, in our implementation, procedure checkIfUninitialized\((u, l)\) returns true conservatively if the null-dereference analysis computes a non-\textit{false} value as weakest pre-condition at point \( h \).

Algorithm 2 is directly implemented using the various APIs provided by Wala. To identify logical mutators we used ‘com.ibm.wala.ipa.modref.ModRef’ class provided by Wala that implements the “mod” analysis. We used 1-object sensitivity analysis of Milanova [13] for the pointer analysis.

VI. EXPERIMENTS

Our objective is to empirically validate the precision of our static analysis that identifies LFIPs, and to estimate the usefulness of our entire approach in the context of object sharing, and immutability refactoring. A candidate way to validate precision could be to measure how close the identified LFIPs are to the optimal LFIPs. However, determining this directly would not be appropriate for the following two reasons: (a) It would be infeasible to manually identify all the optimal FLIPs for all the allocation sites – there is no gold standard and (b) Even if there exists a gold standard, measuring how close the LFIPs are to optimal LFIPs in terms of lines of code, while appropriate for immutability refactoring, does not make much sense for allocation site refactorings such as object-sharing refactoring.

As discussed in Section II-A2, what matters more is whether the LFIP is before the statement at which the allocated objects escape or not. Therefore measuring actual memory savings upon performing object sharing refactoring using the identified LFIPs as hash consing points is a suitable indirect indicator of the precision of our approach. In the context of immutability refactoring, there is no obvious way to measure precision of our approach. Therefore, we only evaluate the potential usefulness of our approach to developers.

A. Experimental results for object sharing refactoring

In this section we first describe our experimental methodology and then evaluate the precision and usefulness of our approach.

1) Experimental methodology: We considered 5 benchmark systems, pmd, luindex, hsqldb, antlr (from the Dacapo 2006 benchmarks [18]) and pdfbox [19]. We used the estimates from the approach of Rama et al. [1] to select
the allocation site to manually refactor. We ranked each allocation site based on the estimated savings and selected those sites which have a ranking of less than or equal to 5 and whose estimated savings is greater than 1%.

In this way, we selected in total 17 sites across these 5 benchmark systems for object-sharing refactoring. We ran the analysis described in this paper on each of the selected allocation sites and identified the corresponding LFIPs. On an average our analysis took 19.2 seconds per allocation site with maximum time of 35.4 seconds to analyze allocation site at Line 278 in file PDFStreamEngine.java.

2) RQ1: What is the precision of LFIP detection?: As discussed in the beginning of this section, for this RQ we measure the precision of LFIP detection by measuring the savings in memory when object sharing refactoring is performed for each allocation site with LFIPs as the hashconsing points.

We introduced hashconsing at least at one LFIP for each of the 17 sites. However, for some of the suggested LFIPs, we did not do hashconsing at those points at all, because it was obvious that doing so would give no savings (e.g., because that LFIP was just before a statement that terminated the program). Also, for safety of object-sharing, hashconsing at LFIP is a necessary but not sufficient condition [1]. Hence, we additionally checked if the functional output after refactoring is the same as that for the original program using test cases.

We ran both the original as well as the refactored code using the ‘medium’ as well as the ‘large’ input that come with the Dacapo benchmarks. However, except for Hsqljdbc, the medium and large inputs are the same for all the Dacapo benchmarks we considered. For PDFBox we used cweb.pdf, which is a 28 page document provided along with the PDFBox distribution, as the test input. We measured the tenured heap size consumed at the end of the run for the original code as well as the refactored code.

Each application was run 5 times and the median values are considered. Experimental results were obtained on a dual-core laptop with Intel i54310 2.60 GHz processor, running windows 8. The JVM was configured to use max heap size of 2GB out of which 2MB was reserved for nursery.

The results are shown in Table I. The second column in the table indicates the number of allocation sites considered for each benchmark. The next two columns depict tenured heap consumption by the original program (OTH) and refactored program (RTH), respectively, under the ‘medium’ input. The fifth column measures the memory savings, as a percentage, due to the object-sharing refactoring. The remain columns show similar results, but from the ‘large’ inputs. (For pdfbox we used the same input as both the medium input and the large input.)

We can observe that savings in tenured heap is observed in all the benchmarks ranging, from 2.5 % to 23.5%. The geometric mean of these savings is 11.4%. Note that except for Hsqljdbc, the default and large inputs are exactly the same for all other benchmarks; hence the numbers are the same.

Note that, while memory was saved, the running time did increase marginally in most of the benchmarks. The running time of the original code for pmd, pdfbox, luindex, hsqljdbc and antlr is 7ms, 1.28s, 1.82s, 8.18s and 610ms respectively. The running time after refactoring for these benchmarks is 7ms, 1.35s, 2.05s, 8.68s and 620ms respectively.

In addition, we also quantitatively compared the usefulness of hashconsing at FIP instead of naively at the end of the method. In 11 out of 17 cases allocated objects escape before the end of the method. So object-sharing at end of the method would not have yielded any benefit.

In 15 out of 17 cases our approach was fully precise in that it identified the most optimal LFIP. Note that in these 15 cases an identified LFIP is immediately after the allocation site and we were able to manually verify that there are no further mutations to the allocated objects.

3) RQ2: Would our tool be useful to developers during object-sharing refactoring?: Our approach is useful if it would be difficult for developers to locate the LFIPs manually. In two cases out of 17, the LFIPs were located several statements after the allocation site. The first of these two sites is at line 464 in file ‘BaseParser.java’ in pdfbox benchmark; one of the LFIPs for this site is at line 653, almost 200 lines after the allocation site. For the other site, which is at Line 278 in file ‘PDFStreamEngine.java’ in pdfbox, an LFIP is 9 lines after the allocation site.

In the remaining 15 cases, the LFIPs occurred either right after the allocation site $s$, or at the end of the containing method $f_s$. However, even in these cases if a developer did not have our tool, they would still have to look at the entire function body as well as the called methods to confirm that these trivial points are indeed correct.

none of the 17 allocation sites that we have used for our experiments were contained in a loop that was present in the same method that contained the site. Hence, trivially none of these sites underwent LLDI.

B. Experimental results for immutability refactoring

In this section, we first discuss our experimental methodology, and then structure our empirical evaluation in the form of two research questions (RQs).

1) Methodology: For the immutability refactoring application we considered 5 benchmark systems - pmd, luindex, antlr, pdfbox and sablecc [20].

As a first step, we implemented and used a lightweight dynamic analysis that estimates runtime overhead due to deep-copies in mutator calls. Then, we selected 5 allocation sites per benchmark from the ones with high overhead estimates. We produced two versions of each benchmark: the first version refactored as per the original Kjolstad approach using their tool [21], and the second where we refactored
<table>
<thead>
<tr>
<th>System</th>
<th>#sites</th>
<th>MS OTH (KB)</th>
<th>MS RTH (KB)</th>
<th>MS %saving</th>
<th>LS OTH (KB)</th>
<th>LS RTH (KB)</th>
<th>LS %saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmd</td>
<td>4</td>
<td>132368</td>
<td>101230</td>
<td>23.5%</td>
<td>132368</td>
<td>101230</td>
<td>23.5%</td>
</tr>
<tr>
<td>pdlbox</td>
<td>3</td>
<td>10732</td>
<td>8980</td>
<td>16.3%</td>
<td>10732</td>
<td>8980</td>
<td>16.3%</td>
</tr>
<tr>
<td>luindex</td>
<td>4</td>
<td>24724</td>
<td>21138</td>
<td>14.5%</td>
<td>24724</td>
<td>21138</td>
<td>14.5%</td>
</tr>
<tr>
<td>lsqldb</td>
<td>3</td>
<td>987822</td>
<td>962134</td>
<td>2.5%</td>
<td>1977801</td>
<td>1950961</td>
<td>1.3%</td>
</tr>
<tr>
<td>antlr</td>
<td>3</td>
<td>3121</td>
<td>2689</td>
<td>13.8%</td>
<td>3121</td>
<td>2689</td>
<td>13.8%</td>
</tr>
</tbody>
</table>

Table I

<table>
<thead>
<tr>
<th>System</th>
<th>#sites</th>
<th>orig (ms)</th>
<th>KD (ms)</th>
<th>our (ms)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>pmd</td>
<td>2</td>
<td>388</td>
<td>3368</td>
<td>405</td>
<td>8.30</td>
</tr>
<tr>
<td>luindex</td>
<td>2</td>
<td>1927</td>
<td>2832</td>
<td>2330</td>
<td>1.20</td>
</tr>
<tr>
<td>antlr</td>
<td>2</td>
<td>479</td>
<td>904</td>
<td>837</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Table II

VII. DISCUSSION

Experimental results indicate that the notion of LFIPs is very useful for allocation site refactorings. We would not have been able to obtain mean memory savings of 11.4% on our benchmarks if we had not used the concept of FIPs. Similarly, our extension to immutability refactoring that makes use of FIPs, achieved a mean runtime speedup of 1.6X compared to performing the same refactoring using a baseline approach that did not use FIPs. This indicates that several other allocation site refactorings could also potentially benefit from the notion of FIP.

A. Limitations and threats to validity

The efficacy of our approach is dependent on the precision of the pointer analysis and the scalability of the underlying Wala framework. For instance, we were not able to run 1-object sensitive pointer analysis on the benchmark pmd because of the scalability limitation of Wala. For pmd we had to use the 1-callstring approach, which is less precise. For all the benchmark systems, we had to manually choose the “entry point” methods (i.e., main methods) from which to start the analysis. If we chose too few, important allocation sites were no longer reachable, whereas if we chose too many, Wala would not scale.

Though refactoring is guided by the LFIPs identified by our tool, it is performed manually as it is a complex task in itself and is out of scope of this paper. For instance, object-sharing refactoring mandates that we introduce ‘equals()’ and ‘hashcode()’ methods for each class whose instances are cached, which is non-trivial. For immutability refactoring, moving mutator code inside constructor, while taking care of all data and control dependencies between statements is non-trivial.

VIII. RELATED WORK

Object initialization has been studied widely in the literature in the context of checking that a field is initialized (explicitly) before use, and in the context of establishing

using our approach discussed in Section II-B. We omitted allocation sites from consideration if (a) Kjolstad's tool did not work, or (b) manual immutability refactoring seemed complex, or (c) testcases failed after refactoring. In this way, in total we evaluated our approach on 14 allocation sites. On an average our analysis took 20.8 seconds per allocation site with maximum time of 34.2 seconds to analyze allocation site at Line 33 in file LocalScope.java in pmd benchmark.

2) RQ1: How useful are LFIPs in improving the immutability refactoring?: One of the contribution of this paper is a proposal to use LFIPs to mitigate the expense of immutability refactoring. Hence, for this RQ we measure the reduction in running time.

For this experiment, we ran each benchmark thrice, as follows: (1) in its original form (without any immutability refactoring), (2) in its immutability-refactored form produced by Kjolstad et al., and (3) in the form produced by us following our proposal. Table II summarizes the results. The inputs used in the runs were the ‘medium’ runs for the Dacapo benchmarks, the file cweb.pdf (mentioned earlier) for pdlbox, and a grammar file minibasic.sablecc for sablecc that comes with the SableCC distribution. Columns 3, 4, and 5 show the running times from the three forms of each benchmark mentioned above. It is notable that our proposal gives significant benefit over the standard immutability refactoring of Kjolstad et al. (KD). The last column indicates our speedup over theirs; the best case speedup was 8X (with pmd), while the geometric mean of the speedups is 1.6X.

3) RQ2: Would our tool be useful to developers during immutability refactoring?: Our tool would be useful if it is difficult for developers to manually identify the mutating statements (as well as logical mutators) and discern the window of code to inspect and move the code to the constructor. For the 14 sites that were considered in this experiment, none of the LFIPs were either immediately after the allocation site or at the end of the method containing the site. Detailed artifacts corresponding to both our experiments are available from the location ‘http://www.csa.iisc.ac.in/raghavan/FIPs’.

VIII. RELATED WORK

Object initialization has been studied widely in the literature in the context of checking that a field is initialized (explicitly) before use, and in the context of establishing
object invariants that may not hold immediately after allocation but hold after (delayed) initialization. Many previous approaches [7], [8], [9], [10], [11] use type systems (which involve programmer annotations) for this purpose.

Unkel and Lam propose a fully automated static analysis approach to identify program points where objects from an allocation site get initialized. But their notion of a full initialization point is simplistic, in that it simply considers the point where an object escapes into some other structure as the full initialization point. It ignores subsequent mutations to these objects after the point of escape. It also ignores mutations of objects that are reachable from the “root” objects allocated at the site s. Note that they do not suggest that their approach be used to support allocation-site based refactorings. Their finding is that delayed initialization is common in practice.

Our previous work [1] studied only object-sharing refactoring and the concept of full initialization points of an allocation site was not introduced in that paper.

IX. Conclusion

In this paper we introduced a novel notion of the program points at which objects allocated at a given allocation site are fully initialized. We also presented a static analysis to detect these points. This analysis was implemented in Wala and empirically evaluated using two sample allocation-site refactorings – object-sharing refactoring, and refactoring for immutability. Results show that our analysis is reasonably precise in identifying LFIPs, and yields good results with these two refactorings, on a set of large benchmark programs.

As part of future work we plan to improve the precision of the algorithm to identify LFIPs by using a more precise ‘Mod’ analysis. Hashconsing at even optimal FIPs would not be beneficial if allocated objects escape before the FIPs. Hence, extending the notion of FIPs to also take this into account could be beneficial for object-sharing refactoring. Finally, identifying the window of code in which objects are fully initialized could be important for other allocation-site refactorings also. Hence, exploring the use of FIPs for other allocation-site refactorings appears to be a fruitful direction for future work.

REFERENCES


