Static Escape Analysis in Pharo: Towards Minimizing Object Allocations

Faouzi Mokhefi[1], Stéphane Ducasse[1], Pablo Tesone[1] and Luc Fabresse[2]

[1]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
[2]Institut Mines Telecom Nord Europe, Douai, Nord, France

Abstract

Object-oriented programming idioms, such as boxing numbers or using design patterns like builders and commands, often create numerous short-lived objects. These objects put pressure on garbage collectors and are ideal candidates for static optimizations such as object inlining [1]. Escape analysis identifies these short-lived objects, offering insights for future optimizations—both manual and automatic. Traditionally, escape analysis has been applied to statically-typed languages. However, dynamically-typed languages introduce additional uncertainty, particularly due to highly polymorphic call sites.

In this paper, we present ESCAPHA, the first implementation of a context-sensitive, flow-insensitive, interprocedural escape analysis for the dynamically-typed language Pharo. We applied ESCAPHA to various Pharo packages, and it successfully identified instances of short-lived object creation and potential locations for optimization. Our approach demonstrates that static analysis can identify a small but relevant number of optimizable allocation sites. We found 280 candidates after analyzing 24,000 methods for a static call graph depth of 10. The paper also discusses the limitations of our analysis, particularly regarding the complexity of the call graph, and explains how we heuristically manage the potential explosion of analysis paths.

Keywords

Static Escape Analysis, static call graph, interprocedural analysis

1. Introduction

Object-oriented programming idioms, such as integer boxing, encapsulation, reification, and design patterns (*e.g.,* Builder, Command, Visitor) [2], often result in the creation of many short-lived or transient objects. Moreover, well-structured object-oriented design, as promoted by design patterns, encourages frequent object creation and extensive use of late-bound calls. These objects typically have a short lifespan, limited to the execution scope of a single method.

Object-oriented programming languages typically rely on garbage collection (GC) algorithms to automatically reclaim memory no longer used by the application. The abundance of short-lived objects increases the workload of these algorithms, as more allocated objects must be traced. Generational garbage collectors [3, 4] have been designed to handle such object lifecycles effectively. However, avoiding object allocation altogether can further reduce the pressure on the GC.

This situation has led to the development of optimizations, with varying levels of automation, based on escape analysis [5, 6, 7]. Escape analysis is a static compiler optimization technique used to determine whether the lifetime of data exceeds its static scope [8]. In the context of object-oriented programming (OOP), it identifies objects that are allocated only within a local scope and whose contents can be optimized without affecting the program's correctness. Most existing escape analysis approaches are designed for statically-typed languages [7], or, in the case of the PyPy Python implementation, focus specifically on avoiding integer boxing [9]. Similarly, Google's TurboFan JavaScript compiler uses escape analysis to inline temporary objects.

This motivates our research into discovering optimization opportunities through static analysis for the dynamically-typed object-oriented programming language Pharo [10]. In Pharo, computation occurs primarily through the creation of many small objects that communicate via message passing. The challenge of analyzing Pharo stems from lexical closures existing as first-class entities. This means they

IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland
*Corresponding author.

can be stored and passed as arguments, and capture their defining environment. From this perspective, Pharo combines object-oriented programming with strong functional programming characteristics. This has a significant impact on escape analysis, as analyzing closures locally is insufficient to determine whether a variable escapes its scope.

In this article, we address the following objectives:

1. **Identify allocation sites** that potentially produce multiple objects which do not outlive the lifetime of their originating method's receiver, using static analysis.
   Research questions:
   - Can static analysis identify non-escaping variables in Pharo?
   - What are the key parameters that define the analysis search space (such as number of implementors, call stack depth, etc.)?
   - What variations in analysis strategies are possible?
2. **Evaluate the relevance and impact** of the proposed approach on performance. *Research questions:*
   - How should such an approach be evaluated?
   - Are there heuristics that can make the analysis tractable?
   - If the analysis is expensive but yields significant benefits, can it be performed in a batch mode statically?

In this paper, we present ESCAPHA (Section 3), the first implementation of a context-sensitive escape analysis for the dynamically-typed language: Pharo. ESCAPHA builds an alias graph that tracks variables referring to newly created objects. It performs interprocedural analysis to handle aliases involving method arguments. A major challenge in such static analysis for dynamically-typed languages is the high number of polymorphic call sites, which leads to large and complex call graphs. This complexity can negate the performance benefits of subsequent optimizations. To address this, the proposed analysis limits the depth of graph exploration and adopts an open-world assumption.

We applied ESCAPHA to various Pharo packages, where it successfully identified instances of short-lived object creation and potential optimization opportunities. We also discuss the limitations of the analysis, particularly regarding the complexity of the call graph, and explain how we heuristically manage the explosion of possible paths (Section 5). Our approach demonstrates that static analysis can identify a small but relevant number of optimizable allocation sites. We identified 280 candidates among 24,000 methods, for a call graph depth of 10.

The outline of the paper is as follows. We begin by discussing the challenges of escape analysis in the specific context of Pharo and present a motivating example (Section 2.1). We then describe the proposed analysis and examine its potential to reduce allocation rates. In Section 4, we experimentally evaluate the effectiveness of the analysis, focusing on its complexity and strategies to mitigate the explosion in call graph size during static analysis.

2. Challenges of Identifying Escaping Variables

Heap allocation is significantly slower and more resource-intensive than stack allocation or register use [11]. Moreover, adhering to good object-oriented design principles and best practices often results in a higher rate of object allocation at runtime.

From the perspective that compilers should address such inefficiencies through optimization, this work focuses on identifying short-lived objects within methods that could be eliminated through stack allocation or object inlining. Object inlining [1] is an optimization technique in which referenced objects of class B are merged into their referencing objects of class A, effectively embedding the fields and methods of B into A.

In the remainder of this section, we detail a motivating example. Then, we discuss some specific challenges that arise when applying static analysis to Pharo, which extensively uses polymorphic methods.

2.1. Motivating Example

Listing 1 shows a code snippet taken from Pharo's graph algorithms library. Each time the method heuristicFrom:to: is called, it instantiates two objects used only within it.

1. The local variable dijkstra (line 4) refers to an AIDijkstra object created during each iteration of the AIstar algorithm.
2. The local variable parameters (line 10) refers to an OrderedCollection (a generic list) used only as an intermediary to store model parameters. These parameters are then extracted and passed to the Dijkstra object.

```
1  AIstar >> heuristicFrom: startModel to: endModel
2      | dijkstra addEdges pathD parameters |
3       ...
4      dijkstra := AIDijkstra new.
5      dijkstra nodes: (nodes first model to: nodes last model).
6      dijkstra edges: addEdges
7          from: [ :each | each first ]
8          to: [ :each | each second ]
9          weight: [ :each | each third ].
10     parameters := OrderedCollection new.
11     parameters add: startModel model.
12     parameters add: endModel model.
13     dijkstra start: parameters first.
14     dijkstra end: parameters second.
15     dijkstra run.
16     pathD := (dijkstra findNode: endModel model) pathDistance.
17     ^ pathD
```

Listing 1: Two objects local to their originating method

In this example, allocating heap memory for these objects is unnecessary because their references are stored in local variables and the objects do not outlive the method in which they are created. One possible optimization is to allocate them on the stack (within their method's activation context) instead of on the heap.

Escape analysis helps identify such unnecessary object allocations and highlights candidates for optimization. This analysis determines whether object inlining is possible. For instance, it could replace the use of OrderedCollection with a direct, method-local implementation inside heuristicFrom:to:. Similarly, it can evaluate whether the functionality of the AIDijkstra object and its used methods can be embedded directly into its caller. Such inlining requires verifying that none of the invoked methods create aliases to the object's internal state. For example, the methods from:to:width:, start:, end:, run, and findNode: must be analyzed to ensure no references to inlined objects escape their scope.

2.2. Static Analysis on Dynamically-typed Languages: The case of Pharo

Pharo is a highly dynamically-typed, pure object-oriented, and reflective language where computation is primarily expressed through extensive message passing. One of its defining characteristics is the pervasive use of lexical closures. These closures are first-class entities: they can be stored, passed as arguments, and executed later, all while capturing their lexical environment. This functional feature is deeply integrated into the language. For example, Pharo does not provide built-in control structures such as loops or conditionals. Instead, such constructs are expressed through message sends (for example, do:, ifTrue:), which developers can redefine or extend. It is common for developers to redefine these messages when creating Domain Specific Languages (DSLs).

Pharo also supports powerful reflective capabilities. Developers frequently use methods such as perform:with: to dynamically construct and invoke messages, often based on metadata such as method annotations [12].

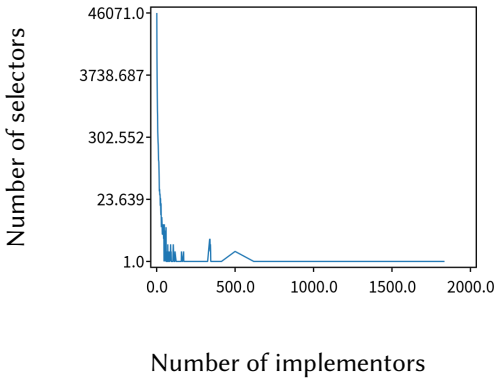

Figure 1: Method selector implementors distribution

Listing 2 illustrates typical Pharo code that exhibits these dynamic features:

- **Polymorphism:** Use of the do: iterator. Pharo's standard library includes 52 different implementations of do:, and developers can define additional ones [10].
- **Reflection:** Use of perform:with: to dynamically invoke a method based on metadata retrieved at runtime.
- **Closures:** Use of multiple lexical closures to capture and manage control and data flow.

```
stSpotterProcessorsFor: aSpotterStep

    (Pragma allNamed: #stSpotterOrder: from: self class to: Object)
        do: [ :aPragma |
            self perform: aPragma methodSelector with: aSpotterStep.
            aSpotterStep processors
                ifNotEmpty: [ :list | list last order: (aPragma argumentAt: 1) ] ]
```

Listing 2: Example of reflective and closure uses

These dynamic features, including closures, polymorphism, and reflection, present significant challenges for static analysis. Closures may escape their lexical scope, which makes local reasoning insufficient. A global analysis is often required to determine whether a variable escapes. To illustrate the impact of polymorphism on static analysis, we present statistics on method name reuse in Pharo 13. As shown in Table 1, the system contains 140,463 compiled methods. Of these, 63,217 have unique names, representing 45% of the total. Figure 1 presents the distribution of the method implementors.

This indicates that a name in the code usually corresponds to two methods or more. Note that the selector initialize is a major outlier with 1,879 methods, typically invoked via self initialize.

Table 1
Method name distribution in stock Pharo 13

Kind	Number	Comments
# Compiled methods	140,463	(100%)
# Possible call-site selectors	63,217	(45%)
# selectors with 1 implementor	46,071	(33%)
# selectors with 2 implementors	7,817	(5%)
# selectors with 3 implementors	1,569	(1.1%)

These dynamic characteristics limit the assumptions that static analysis tools can make about specific message sends or method names. As a result, conventional static analysis techniques cannot be directly applied to Pharo without significant adaptation.

2.3. Summary

Some short-lived objects are repeatedly allocated on the heap but remain confined to the method in which they are created, presenting clear opportunities for optimization through stack allocation or object inlining. Escape analysis plays a crucial role in identifying these candidates for optimization. However, performing static analysis in Pharo poses significant challenges due to its pervasive use of polymorphism, lexical closures, and runtime reflection. These dynamic features introduce uncertainties that hinder conventional static reasoning and often require more sophisticated or whole-program analysis techniques. In the next section, we introduce our solution: ESCAPHA.

3. ESCAPHA: a Simple Static Escape Analysis

In this section, we provide a brief definition of the vocabulary, the meta-model, and the static analysis we developed. It takes as input a single method. The goal, however, is to apply it to method batches. The analysis yields false positives that necessitate manual inspection for a definitive conclusion. We obtain a set of non-escaping variables, along with the analysis snapshots for each method to be reused.

3.1. Illustration

The following case scenario 3 exemplifies the awaited results of the analysis.

```
1  A >> foo: arg
2
3      | newObject outputStream outputStreamCopy |
4      newObject := Object new.
5      outputStream := Stream new.
6      self bar: newObject.
7      outputStreamCopy := outputStream returnSelf.
8      newObject printOn: outputSteam.
9
10 A >> bar: anObject
11     ^ anObject copy
12
13 Object >> printOn: aStream
14     ...
15
16 A >> printOn: stream
17     ...
18
19 Object >> copy
20     ...
21
22 Stream >> returnSelf
23     ^ self
```

Listing 3: Simplified example

The Method foo: has one argument that is discarded from the followed variables set because it is the root of the call graph. It creates two objects $Object, Stream \in O$, held in newObject, outputStream respectively. We visit the ast node of the method and treat every statement that references one of the followed variables p $\in P$. In this example, the message sends AST nodes of interest are in lines $L6, L7, L8, L11 \in I$. Line 8 message corresponds to two potential compiled methods, Object»printOn: , A»printOn:, the rest are monomorphic call sites.

The resulting call graph associated with Listing 3 is composed of 6 nodes: two nodes for the message printOn: implementations, two for bar: and returnSelf each. The nodes related to bar: refer back to

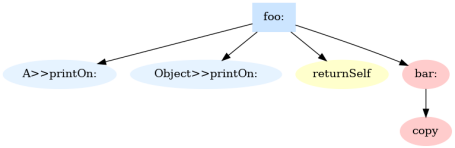

Figure 2: static call graph for listing 3

their caller. The call graph could be recursively expanded further to cover the message copy at a greater depth. The local variables and parameters form a points-to graph in Figure 3.

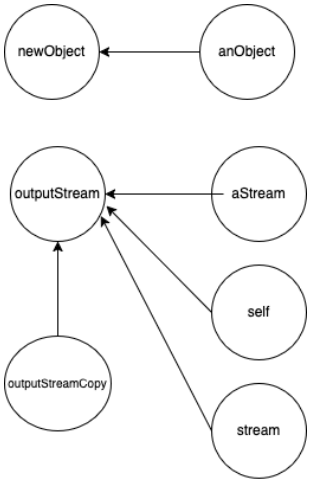

Figure 3: Points-to graph of listing 3

outputStream is aliased by outputStreamCopy assumed to be a return of method returnSelf. outputStreamCopy is aliased by variables aStream from Object»printOn: and stream from A»printOn:

According to escape conditions 3.2.2, all variables outputStream, outputStreamCopy, newObject escape.

3.2. Meta Model

Our analyzer takes as *input* a single method for individual analysis and identifies a set of variables that are guaranteed to be of *non-escaping*.

To achieve this, we combine escape analysis with points-to analysis, focusing on the data flow of references while neglecting control flow. We denote the abstraction of runtime features of a given program:

- O: the set of all heap objects potentially allocated during execution, also referred to as pointer targets. Each is associated with the AST statement responsible for its allocation.
- P: the set of alias variables that reference heap objects in O, each corresponding to a local variable or an argument. Given a reference p, the set $P(p)$ denotes the heap objects it may point to; there must exist an $o \in O$ such that o is the first element in a points-to chain starting from p. The analysis primarily only follows these variables' aliases and infers their types instead of deeply inferring all arbitrary expressions.
- I: the set of all method invocations, also called analysis contexts. Each call site where at least one argument is in P is associated with one or more contexts, particularly in the case of polymorphic call sites. This is further discussed under virtual call resolution.

An object $o \in O$, may be instantiated using different selectors according to the class definition, with new message or other known constructors. Our approach only focuses on the subset problem by

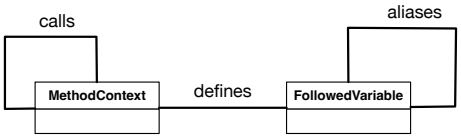

Figure 4: meta model diagram of ESCAPHA

considering this specific message and other derivatives that begin with the prefix new such as newFrom: because they are definite allocations. The impediment of this selectiveness is that it overlooks many objects and data structures, such as Arrays (e.g. Array from: aCollection) that are frequently used but are created otherwise.

Virtual call resolution is the process of identifying all possible target methods that a virtual call may invoke at runtime. It involves linking the arguments at the call site with the parameters of the declared method. In dynamically-typed languages, achieving precision in target methods is difficult due to the loss of type information caused by message sends and variable reassignment. This feature introduces context sensitivity into the points-to analysis.

3.2.1. Input

Our analyzer takes as *input* a single method within the context of a class hierarchy. For the analysis to be meaningful, the method must include at least one explicit allocation. We only consider allocations found at the root method analyzed.

3.2.2. Escape Conditions

The analysis considers a variable as *escaping* (*i.e.,* outliving its method context) if it or any of its aliases (other references to the same memory location) satisfies one of the following conditions:

1. Returned and propagated up to the top level of the call stack.
2. Assigned to an instance variable, global variable, or shared variable.
3. Passed to or referenced within a lexical closure that may be evaluated later in the call graph, starting from its defining method.
4. Stored in a collection, making it difficult for the analyzer to track individual elements.
5. Used in reflective or primitive operations, such as instVarAt:put:.
6. Involved in a serialization process.

3.2.3. Output

It is a set of *non-escaping* variables, along with the analysis history. This history is useful for deriving statistical insights from the results.

Internally, as illustrated in Figure 4, our analyzer takes a method as input and creates a corresponding representative entity in the model. We call it an MethodContext. It then attaches additional objects to this method context:

- (1) Other MethodContext instances for each message send within the input method. It represents one of the possible implementations for a given message.
- (2) A list of temporaries, referred to as FollowedVariables, possibly referencing a heap-allocated object. The analysis binds the caller's followed variables to the arguments of the callee's method context [13]. This allows the analysis to determine whether a variable escapes the scope of the root method.

3.3. Analysis Steps

In this section, we summarize the analysis in the following steps and pseudo-algorithm 1. As presented before, the input is a compiled method that is then visited in its AST form.

1. Create a root method context to statically represent runtime properties of the given method for the analysis. The entry method is set as the root node of the static call graph. Normally, this method contains the allocations to be studied.
2. Represent dynamically allocated objects as entities in the points to graph. Then, only select in the root method definition, local variables that are assigned these objects to append as nodes to the graph and to follow during the points-to phase.
3. Select call-sites whose arguments are followed and appear in the points-to graph. Since 67% of the selectors in Pharo have multiple implementations, as shown in Figure 1. We need to consider each of them in the static call graph. The
   Virtual call resolution determines possible implementations using type propagation or retrieves all implementations when type information is insufficient. This involves examining the receiver class hierarchy.
4. Store the call graph nodes in the working list following the sequential order of execution, whilst neglecting the control flow.
5. For each element of the working list we need to:
   - Create a new method context.
   - Create its followed variables.
   - Link the new method context to the existing method context in the abstract static call graph.
   - Unify the formal arguments of the new method declaration with the arguments at the call-site of its caller.
   - Verify and propagate the escape condition of the variables.
6. Check escape constraints.
7. Pop the next element in the working list and repeat steps from (4).

3.4. Type Propagation

Since type compatibility between the assigned and assignee is not required in dynamically-typed languages, we cannot rely on this information to predict a variable's type. We take the example of an assignment. On the right-hand side, a message send return value is conservatively assumed to be of unknown type. This increases the likelihood of inconclusive program analysis.

To address this issue, we record the types of variables starting from their explicit allocation sites (e.g., via the *new* message or class-side messages) and propagate this information to all aliasing variables. We assess the impact of this heuristic in the evaluation 4.

We chose this approach for its simplistic aspect and low impact on analysis time, comprad to type inference tools [14]. These traditional models provide complete type lattices for all variables. Our recorded types are simply stored as the class of the object's explicit declaration in the allocation statement. It is then associated with the representation of the entity $p \in P$ referencing the object in question $o \in P(p)$ in the points-to graph.

Now that we have defined our approach to find unnecessary allocations. We evaluate it in the next section.

4. Evaluation

To better understand ESCAPHA's behavior on real-world programs and evaluate the impact of the proposed precision-enhancing features, we conducted a series of experiments under different settings.

Algorithm 1 ESCAPHA: ESCAPE ANALYSIS PSEUDO-ALGORITHM

Require: Method m
Ensure: Set of non-escaping variables NE

```
C ← createAbstractContext(m)                    ▷ Create root MethodContext
WorkList ← [C]
Visited ← ∅
Escaping ← ∅
while WorkList is not empty do
    ctx ← pop(WorkList)
    if ctx ∈ Visited then
        continue
    end if
    Visited ← Visited ∪ {ctx}
    for all message sends s in ctx do
        Targets ← virtualCallResolution(s, H)
        for all t ∈ Targets do
            calleeCtx ← createAbstractContext(t)
            bindFollowedVariables(ctx, calleeCtx)
            add(calleeCtx, WorkList)
        end for
    end for
    for all variables v in ctx do
        if checkEscapeConditions(v) then
            Escaping ← Escaping∪ aliasesOf(v)
        end if
    end for
end while
NE ← { v | v is local in m and v ∉ Escaping }
return NE
```

4.1. Methodology

First, we select a subset of Pharo system methods ($SubMethods$) that include explicit allocations using the new message. This subset consists of 24,000 compiled methods.

We then analyze various parameters of the approach, such as call graph depth, worklist size, and the number of implementors per call site, by varying some parameters while keeping others fixed to assess their individual effects.

To evaluate the scalability of the algorithm with respect to the number of initial method parameters, we use execution time as the performance metric.

4.2. Approximating Precision and Recall

The preselected $SubMethods$ for the analysis are not annotated. Instead of doing so, we select fewer packages to annotate and calculate the precision of the analysis in finding true positives at this scale.

The difficult part of this approach is the validation of the found samples. An automatic validation raises the question of how to ensure program execution will reach the optimization point? Crafting such applications to tackle the specific running scenario is a challenge that we do not address here.

Although unit test methods are most common among the results and are immediately available to refactor into optimised code, the standard methods are found too.

Instead, we do a manual validation of the escape conditions described in section 3.2.2. We write specific unit tests to evaluate the proposed changes on candidate allocation sites found in test methods,

or visually navigate through the static call graph. The latter is the default approach we take for standard methods.

We apply stack allocation, object fusion, or object deletion at the end of program execution, and confirm that the resulting code runs without errors. A clear direction for future work is to leverage automatic code refactorings for validation.

Threats to Validity: This evaluation involves manual checks, which are very time-consuming; therefore, the results are incomplete and may include errors. Furthermore, we typically run the experiment within a single Pharo image, and we maintain the detailed analysis context information, which can lead to longer garbage collection times and influence the time complexity described in the tables below 2 3 4.

4.3. Intraprocedural Analysis

We begin by running the intraprocedural analysis on $SubMethods$, limiting the call graph depth to 1 and restricting the maximum number of elements in the worklist to 5000 methods. This yields 54 candidate allocation sites from the global set GS, where GS is defined as the union of Nod from each local method in the initial subset of Pharo methods. Here, Nod denotes the set of assignment nodes whose right-hand side is an allocation. For optimization purposes, 25 of these sites are located in test methods (see the first line of table 2).

Depth	Analysis time	Candidates
1	9 min 14 s (39 ms)	54
2	47 min 41 s (204 ms)	189
3	3 h 29 min (895 ms)	199
4	27 h 11 min (270712995 ms)	203
$\geq$5	> 27 h	—

Table 2
Variation of call graph depth impact on runtime and resulting positive candidates with fixed input $SubMethods$.

4.4. Heuristics Analysis

In this section, we aim to determine when an analysis run should be discarded. Among the possible parameters for the open-world assumptions, we define three that act as limitations: call graph depth, number of implementors, and worklist length.

Call Graph Depth Setting the depth to 1 effectively disables interprocedural analysis. When the depth exceeds 2, summary information becomes necessary, as the static call graph may revisit nodes. Since the graph is constructed on the fly, such revisits can form closed cycles and eventually lead to indirect recursion.

Number of Implementors For messages with a large number of implementors, this parameter controls the maximum number to consider. For instance, a simple analysis reveals that in Pharo 13, 1448 selectors have more than 10 implementations. This parameter controls the number of implementors considered at a given call site.

Worklist Length The worklist for a given root method is limited to a maximum size, which restricts the expansion of the call graph. This is particularly useful for handling long methods. Let's suppose large methods are methods exceeding 10 lines of code; we have 14407 of them in a Pharo image alone.

These help us assess whether a thorough analysis is worthwhile. We mainly explore variations across call graph depth and input batch size.

Total methods	Analysis time	Candidates
4 800	6 min 32 s	103
7 200	12 min 32 s	138
9 600	16 min 35 s	147
12 000	19 min 46 s	158
14 400	1 h 32 min 43 s	166
16 800	35 min 57 s	173
19 200	41 min 10 s	177
21 600	46 min 48 s	182
24 000	2 h 35 min 38 seconds	189

Table 3
Runtime and resulting positive candidates - variable number of input methods for call graph depth of 2

Table 2 presents the results obtained by keeping the input ($SubMethods$) fixed while varying the depth of the call graph. We observe the following:

- As the call graph depth increases, the analysis takes more time. Beyond a depth of 3, the execution time becomes excessively high.
- The activation of interprocedural property significantly enhances the outcome of the analysis, even at one higher degree of depth.

4.5. Varying The Input Method Sample With Fixed Depth

In this section, we report the performance of the algorithm with type propagation enabled.

Table 3 shows the results of the analysis with the call graph depth of 2, performed on an increasing number of methods, ranging from 4800 to 24000.

The relationship between the input batch size and analysis time is not linear. The analysis time does not increase at a constant rate per additional method, and there are two very significant jumps (at 14,400 and 24,000 methods) that suggest specific methods causing extreme slowdowns. For it to be proportional, doubling the methods would roughly double the time, which is not consistently happening here. The results highlight the influence of methods with multiple implementors. It shows poor scalability due to non-linear behavior with particular performance drops for higher workloads.

4.6. Varying Depth for a Small Input Method Sample

We intentionally select a small subset of 50 methods, which were previously identified as containing optimization candidates. We then vary the call graph depth to observe at which point the results stabilize.

We observe from the table 4:

- Interprocedural analysis offers clear benefits. The number of identified candidates increases from 12 to 43 (with 50 being the maximum possible).
- Beyond a call graph depth of 3, the results reach a plateau.
- It is important to note that the analysis is biased, as each method has unique characteristics. This makes it difficult to generalize findings from a randomly selected subset.
- The simple type propagation reduces time complexity and filters fewer true positive candidates.

We can deduce from 4 and 2 that a call graph depth of 3 is enough to get escape information for a reasonable execution time, so we follow the analysis with these optimal parameters. For validation, we restrain the input batch of the analysis to include a few packages for annotation.

5. Discussions and Pharo specific Analysis Concerns

5.1. Field Sensitivity

In this section, we discuss an effort to improve the precision of the escape analysis by incorporating field sensitivity. This is part of our future work and is not yet fully implemented.

Many objects are embedded as fields within other objects. We are particularly interested in cases where the outcome of the escape analysis for a field object depends on the escape status of its container. For example, consider the case where a point object pt1 is stored in a line object line1.
In the current implementation, the point is assigned to an instance variable of the Line class and is therefore conservatively marked as escaping.
In future work, we aim to refine this behavior by analyzing the escape status of the container object. If the container (in this case, the line) does not escape, it may be possible to conclude that the contained object (the point) also does not escape. This requires a deeper analysis of how instance variables are used and propagated.

```
pt1 := Point new.
pt1 setX: 1 setY: 1.
line1 := Line new.
line1 addPoint: pt1.
```

Listing 4: Simplified example of non-escaping object assigned to instance variable

We conducted an additional exploratory phase of the analysis to increase the effectiveness of candidate detection. In this phase, we introduced field sensitivity by treating fields as tracked variables. The motivation behind this approach is that, by analyzing field usage, we may identify true positives that were previously missed due to conservative assumptions about instance variable assignments.

A representative example is found in the HoneyGinger library (see Example 5), where each HGSimulator instance holds a set of actions and particles in its instance variables. These objects are used exclusively within the scope of the program execution and are never exposed externally. According to the definition of escape, such objects should not be considered escaping. However, the current analysis conservatively marks them as escaping.

Depth	Analysis time		Candidates	
n	w/ TP	w/o TP	w/ TP	w/o TP
1	5 s	7 s	12	12
2	20 s	56 s	43	38
3	1 min 1 s	2 min 9 s	48	38
4	1 min 53 s	3 min 30s	49	39
5	3 min 33 s	1 h 1 min	48	37
6	7 min 50 s	18 min 09 s	48	39
7	14 min 39 s	45 min 1 s	48	37
8	48 min 40 s	—	48	—
9	10 h 1 min	—	47	—

Table 4
Analysis time and candidate counts across depths, with and without type propagation (tp: Type propagation).

```
. . .
simulator := HGSimulator new.

simulator addInitializer:
   (HGVelocityInitialization with: HG2dPoint zero).
simulator addInitializer: (HGTemperatureInitialization with: 270).

simulator addAction: (HGForceInitialization with: HG2dPoint zero).

simulator addParticleAt: 500 @ 500 temperature: 390 mass: 100.0.

World activeHand showTemporaryCursor: HGSimulator.
. . .
```

Listing 5: Program with inlineable embedded objects

Our approach to addressing the misclassification of non-escaping variables as escaping is summarized as follows:

1. The first step examines all references to the instance variable within its defining class. The same escape conditions applied to temporary variables and blocks are reused here. The analysis targets only the referenced instance variable within its referencing methods. This step determines whether the analysis should proceed, which is the case when the instance variable escapes solely through its accessor.
2. The second step constructs a static call graph in the reverse direction compared to the standard analysis flow. Specifically, it collects all senders of the accessor identified in the previous step. These message sends are treated as variable references to the instance variable.
3. Finally, the analysis inspects all uses of the accessor in the collected sender methods and resumes interprocedural analysis in the usual forward execution direction, applying the standard escape constraint-solving logic.

We could not measure the improvement brought by this approach due to Pharo-specific constraints.

5.2. Cascade and Indirect Object Allocation

Object creation can occur in several ways: through direct use of new, via factory methods, or as part of a message cascade. Each of these cases requires dedicated static analysis strategies.

1. The standard form of allocation is explicit, using new or similar primitives, with the result assigned to an aliasing variable. Tracking such objects involves adding an abstraction to the points-to graph.
2. A second category includes indirect allocations, such as those performed by factory methods. These allocations are initiated by message sends—often not primitive—that eventually call a primitive such as new or basicNew. While our focus is on identifying escaping objects, other approaches use runtime execution data to locate such allocations [15]. In our analysis, we restrict this category to class-side methods, which are most likely to act as indirect allocation sites.
3. In cascaded messages, an object created in the first message should be tracked as an alias of self in the subsequent messages. The return value of the cascade depends on the final message. In listing 6 for instance, if the last message is yourself, the result of the initial allocation is returned. In such cases, the variable on the left-hand side of the assignment should be associated with the created object.

```
receiver := FixtureEscapeAnalysis new
                        anotherMessage: 5;
```

```
yourself.
```

Listing 6: Cascade message assignment

Another challenge involves indirect aliasing. For example, in a message send where an argument itself is the result of an allocating message, aliasing becomes interprocedural. To broaden the scope of the analysis, we must recognize allocation sites beyond assignments of the form variable:= Object new.

Proposition: Cascades and non-assigned allocating messages should be modeled as abstract entities. They contain the object's type and a unique identifier in the analysis.

6. Related Work

6.1. Points-to/Alias Analysis

Escape analysis is heavily influenced by and built on top of points-to analysis. Milanova *et al.,*. [16] explored different levels of context-sensitivity for points-to analysis, showing that object-sensitivity provides better precision for object-oriented languages than call-site sensitivity. Our approach takes into consideration these insights to apply to Pharo's programming language model.

Early work [17] highlighted naming schemes for alias variables. Their approach involved retaining the original variable names or generating synthetic names during the analysis. In our implementation, we chose the latter representation to compose a unique identifier for the memory location in different execution contexts. Steensgaard presented an almost linear time points-to analysis algorithm based on type inference. It is flow-insensitive, context-insensitive due to the monomorphic characteristic of the subject type system, and interprocedural. The efficiency lies in the representation of multiple potential runtime locations by a single graph component. Including all elements of composite objects, such as struct objects in C.

In [18], Rak-amnouykit *et al.,* provided a hybrid evaluation of traditional static points-to analysis with concrete evaluation using the Python interpreter. Static analysis tools typically require access to the source code of external libraries to understand their behavior and propagate information through the program. When this code is missing, PoTo attempts a concrete evaluation of such expressions in their enclosing import environment and infers concrete types in the new 3-address code representation. When processing call statements or field access statements, the analysis checks if the object being operated on is concrete. If it is a concrete object, the analysis attempts concrete evaluation of the operation, returning a new concrete object and adding it to the points-to set. This provides more information during constraint resolution and improves program coverage.

6.2. Escape Analysis

[19] proposes a framework of escape analysis and demonstrates an application on Java programs, introducing a context-sensitive, flow-sensitive algorithm to identify the non-escaping objects in methods and threads efficiently. This enables stack allocations and synchronization removal and serves as the basis for subsequent work on escape analysis for object-oriented programming languages.

Wang *et al.,* apply optimizations for the existing escape analysis for the Go programming language (Golang) [20]. They transform the code with situations considered escape, mainly caused by passing pointer arguments holding a reference to variables in the points-to set. It bypasses Golang's escape analysis mechanism by converting the pointer through intermediate types. This breaks the compiler's ability to trace the pointer back to the original object based solely on this function call. They evaluate real-world projects before and after code optimisations using memory usage and time consumption metrics.

Kotzmann *et al.,* present a new intraprocedural and interprocedural algorithm for escape analysis in the context of dynamic compilation, where the compiler has to cope with dynamic class loading and deoptimization [5]. It operates on an intermediate representation in SSA form. It introduces equi-escape

sets for the efficient propagation of escape information between related objects. The analysis is used for scalar replacement of fields and synchronization removal, as well as for stack allocation of objects and fixed-sized arrays.

The all-or-nothing approach taken by most Escape Analysis algorithms prevents all these optimizations as soon as there is one branch where the object escapes, no matter how unlikely this branch is at runtime. [7] presents a new, practical algorithm that performs control flow sensitive partial escape analysis in a dynamic Java compiler. It allows escape analysis, scalar replacement, and lock elision to be performed on individual branches. The algorithm is implemented on top of the Just-in-time compiler.

6.3. Tracing JITs

In dynamically-typed languages, Bolz *et al.,*. [21] demonstrated how trace-based JIT compilers can achieve similar optimizations through specialization.

7. Conclusion

After outlining the motivations and challenges of static analysis, we focused on the specific difficulties of performing such analysis in a purely static setting within a dynamically-typed language like Pharo.

We introduced ESCAPHA, an analysis designed to identify non-escaping variables. It represents a first implementation of an object- and context-sensitive, flow-insensitive escape analysis for the Pharo language.

We discussed the impact of type refinement, field nesting, and the various forms of object creation. Additionally, we reported empirical results concerning worklist size and call graph depth effects on static analysis on different batch sizes.

For the future, we would like to experiment with combining type inference tools, for example, RoelTyper [14]. It helps in finding types of instance variables statically, thereby we expect a higher precision. We also want to include more forms of allocation into the points-to graph initial nodes. We mentioned in 5.2 allocations found in cascades, factory methods, and builders.

8. Acknowledgments

We acknowledge the support of the Region Hauts de France for the PhD of the first author.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] J. Dolby, A. Chien, An automatic object inlining optimization and its evaluation, in: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, 2000, pp. 345–357.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[3] D. Ungar, Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, ACM SIGPLAN Notices 19 (1984) 157–167. doi:10.1145/390011.808261.

[4] R. Jones, A. Hosking, E. Moss, The garbage collection handbook: the art of automatic memory management, CRC Press, 2016.

[5] T. Kotzmann, H. Mössenböck, Escape analysis in the context of dynamic compilation and deoptimization, in: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution

Environments, VEE '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 111–120. URL: https://doi.org/10.1145/1064979.1064996. doi:10.1145/1064979.1064996.

[6] T. Kotzmann, H. Mossenbock, Run-time support for optimizations based on escape analysis, in: International Symposium on Code Generation and Optimization (CGO'07), 2007, pp. 49–60. doi:10.1109/CGO.2007.34.

[7] L. Stadler, T. Würthinger, H. Mössenböck, Partial escape analysis and scalar replacement for java, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 165–174. URL: https://doi.org/10.1145/2544137.2544157. doi:10.1145/2544137.2544157.

[8] B. Blanchet, Escape analysis: correctness proof, implementation and experimental results, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, Association for Computing Machinery, New York, NY, USA, 1998, p. 25–37. URL: https://doi.org/10.1145/268946.268949. doi:10.1145/268946.268949.

[9] H. Ardö, C. F. Bolz, M. FijaBkowski, Loop-aware optimizations in pypy's tracing jit, SIGPLAN Not. 48 (2012) 63–72. URL: https://doi.org/10.1145/2480360.2384586. doi:10.1145/2480360.2384586.

[10] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: http://books.pharo.org.

[11] A. Appel, Z. Shao, An empirical and analytic study of stack vs. heap cost for languages with closures, Journal of Functional Programming (1996).

[12] I. Thomas, S. Ducasse, P. Tesone, G. Polito, Pharo: a reflective language - analyzing the reflective api and its internal dependencies, Journal of Computer Languages (2024). doi:10.1016/j.scico.2014.02.016.

[13] O. Agesen, The cartesian product algorithm, in: W. Olthoff (Ed.), Proceedings ECOOP '95, volume 952 of *LNCS*, Springer-Verlag, Aarhus, Denmark, 1995, pp. 2–26.

[14] R. Wuyts, RoelTyper, a fast type reconstructor for Smalltalk, 2005. URL: http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/, http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/.

[15] S. Jordan Montaño, G. Polito, S. Ducasse, P. Tesone, Evaluating finalization-based object lifetime profiling, in: International Symposium on Memory Management (ISMM '24), 2024.

[16] A. Milanova, A. Rountev, B. Ryder, Parameterized object sensitivity for points-to and side-effect analyses for java, ACM SIGSOFT Software Engineering Notes 27 (2002). doi:10.1145/566172.566174.

[17] M. Hind, M. Burke, P. Carini, J.-D. Choi, Interprocedural pointer alias analysis, ACM Trans. Program. Lang. Syst. 21 (1999) 848–894. URL: https://doi.org/10.1145/325478.325519. doi:10.1145/325478.325519.

[18] I. Rak-amnouykit, A. Milanova, G. Baudart, M. Hirzel, J. Dolby, Poto: A hybrid andersen's points-to analysis for python, 2024. URL: https://arxiv.org/abs/2409.03918. arXiv:2409.03918.

[19] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, S. Midkiff, Escape analysis for java, in: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '99, Association for Computing Machinery, New York, NY, USA, 1999. URL: https://doi.org/10.1145/320384.320386. doi:10.1145/320384.320386.

[20] C. Wang, M. Zhang, Y. Jiang, H. Zhang, Z. Xing, M. Gu, Escape from escape analysis of golang, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 142 – 151. URL: https://doi.org/10.1145/3377813.3381368. doi:10.1145/3377813.3381368.

[21] C. F. Bolz, A. Cuni, M. FijaBkowski, M. Leuschel, S. Pedroni, A. Rigo, Allocation removal by partial evaluation in a tracing jit, in: PERM'11 - Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2011, pp. 43–52. doi:10.1145/1929501.1929508.