An Efficient Data Structure for Must-Alias Analysis

George Kastrinis
University of Athens
Athens, Greece
gkastrinis@di.uoa.gr

George Balatsouras
University of Athens
Athens, Greece
gbalats@di.uoa.gr

Kostas Ferles
University of Texas
Austin, USA
kferles@cs.utexas.edu

Nefeli Prokopaki-Kostopoulou
University of Athens
Athens, Greece
nefelipk@gmail.com

Yannis Smaragdakis
University of Athens
Athens, Greece
smaragd@di.uoa.gr

Abstract

A must-alias (or "definite-alias") analysis computes sets of expressions that are guaranteed to be aliased at a given program point. The analysis has been shown to have significant practical impact, and it is actively used in popular research frameworks and commercial tools. We present a custom data structure that speeds up must-alias analysis by nearly two orders of magnitude (while computing identical results). The data structure achieves efficiency by encoding multiple alias sets in a single linked structure, and compactly representing the aliasing relations of arbitrarily long expressions. We explore the data structure's performance in both an imperative and a declarative setting and contrast it extensively with prior techniques. With our approach, must-alias analysis can be performed efficiently, over large Java benchmarks, in under half a minute, making the analysis cost acceptable for most practical uses.

CCS Concepts • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**;

Keywords Alias analysis, Datalog, Must analysis, data structure

ACM Reference Format:
George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An Efficient Data Structure for Must-Alias Analysis. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179519

CC'18, February 24–25, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179519

1 Introduction

Most sophisticated static analyses need to model the program heap—a task that can prove to be a challenge. An established tool to achieve scalable heap modeling is *pointer analysis*. There are two major directions one could follow. Either compute sets of heap objects that program expressions (e.g., variables) may point to—i.e., perform *points-to analysis*—or compute program expressions that may refer to the same heap object—i.e., perform *alias analysis*.

Each kind comes in two major flavors. If the analysis overapproximates feasible behaviors we get a *May* analysis. If it underapproximates feasible behaviors we get a *Must* analysis. As a result, when a *Must-alias* analysis reports aliasing between program expressions, these are guaranteed to always be aliases during execution. In order to produce useful results soundly, a must-alias analysis typically has to employ flow-sensitivity, i.e., compute information per program point, taking into account the inherent control-flow of the program.

A must-analysis for pointers is invaluable for automatic optimizations, as well as for bug detection that traditionally has a high false-warnings rate. Nikolić and Spoto [16] report that a must-alias analysis significantly increases the precision of both a null-reference detector (46% fewer warnings) and a non-termination detector (11% fewer warnings). Earlier work has reported similar benefits [13]. Furthermore, results can form the base of more complex reasoning. For example, the Doop framework employs a must-alias analysis for performing "strong updates" at instructions that write to heap objects. Earlier work has used must-alias analysis to similar benefit [6, 12].

In this work, we present a data structure that can dramatically speed up the performance of must-alias analysis. The insights behind the data structure are quite general. First, must-alias sets are *equivalence classes* (must-alias is a symmetric, transitively-closed relation, unlike typical may-alias). A naive implementation will explicitly compute each pair in these equivalence classes. In contrast, an optimized implementation will encode aliasing implicitly: as membership in the same sub-structure. This is a technique also employed in past static analysis approaches in different settings—e.g., in

the use of union-find trees in Steensgaard-style [19] points-to analysis. Additionally, aliasing can be implicitly extended to longer access paths and this inference should be readily computable in the course of the analysis. For instance, if two program expressions `x` and `y.next` are aliases, then so are all their extensions—e.g., `x.prev` and `y.next.prev`. Such "derivative" relationships should be represented compactly. In our data structure, we represent complex program expressions implicitly until expansion is needed and up to the extent that must-alias information exists for them.

Our data structure is effectively a symbolic abstraction of the program's heap—as a directed graph. We invent for each variable a graph node: an abstract object that represents "the object that the variable points to". Although several abstractions of the heap have appeared in the literature, ours is distinguished by several elements—e.g., a mere "load" operation introduces a new abstract object. An abstract object in our structure represents at most one concrete object, unlike traditional abstractions that map multiple concrete objects to one abstract. Whenever a must-alias inference is made, the corresponding abstract objects are merged: the two abstract objects have to correspond to the same concrete one. Access paths are represented implicitly, as regular paths that follow object fields through our symbolic heap. All operations over the graph arising during a must-alias analysis (esp. the intersection of graphs) are performed highly efficiently.

We implement the data structure in two settings: imperatively, in Java, with destructive updates (upon aliasing, abstract objects are collapsed together) and purely functionally (upon aliasing, abstract objects are related in an associative structure). The latter is suitable for a declarative implementation, in the Datalog language. We show that the data structure yields large performance improvements compared to an explicit representation of alias pairs. The imperative version achieves a speedup of up to two orders of magnitude, with the declarative implementation nearly matching it in most cases. As a result, the running time of a realistic must-alias analysis becomes small—a few tens of seconds for large benchmarks and the full Java library.

Overall, our work:

- describes the primitive operations (e.g., set intersection) that a must-alias analysis needs to perform;
- presents an efficient data structure for representing must-alias analysis inferences and efficiently encodes operations over that structure;
- applies the new data structure on an must-alias analysis implemented in an existing framework and quantifies the benefits in different implementation settings.

2 Background and Example

We illustrate some basic concepts of must-alias reasoning with a small example, also used in later sections to illustrate our data structures and algorithms.

```
1  class A {
2    A next;
3    B member;
4
5    A(A next, B member) {
6      this.next = next;
7      this.member = member;
8    }
9
10   void foo(A a) {
11     member.container = a;
12   }
13 }
14
15 class B {
16   A container;
17   B(A container) {
18     this.container = container;
19   }
20 }
21
22 public class Test {
23   public static void main(String[] args) {
24     B b1 = new B(null);
25     A a1 = new A(null, b1);
26     A a2;
27     if (args != null)
28       a2 = new A(null, b1);
29     else
30       a2 = new A(a1, b1);
31     b1.container = a2;
32     a1.foo(a1);
33   }
34 }
```

Figure 1. Simple illustration of must-alias inferences.

Consider the small Java program in Figure 1. Even at this size, inspecting the program requires human effort. A must-alias analysis can provide useful information to tools and humans alike. The output consists of must-alias pairs: expressions that are guaranteed to point to the same object–denoted by ∼. (More precise definitions follow in Section 3.) For instance, `a1.member` and `b1` form an alias pair for almost the entire body of method `main`. Alias pairs are established by direct variable assignments (which are plentiful in a compiler intermediate language, although less so in original source code), as well as heap stores and loads. A must-alias analysis has to report aliases only when they are guaranteed to hold, and needs to invalidate them on store instructions or method calls that may change the fields of objects pointed by subexpressions in an alias pair. In Figure 1, `b1.container` is an alias for `a2` on (i.e., right after) line 31. However, the analysis needs to recognize that line 32 invalidates that alias pair. Line 32 instead establishes an aliasing relationship between `b1.container` (as well as `a1.member.container`) and `a1`. The analysis remains sound (i.e., safely under-approximate)

if the earlier `b1.container` ∼ `a2` alias pair is invalidated, regardless of whether the new alias pair (`b1.container` ∼ `a1`) is established, via inter-procedural reasoning, on line 32.

Other aliasing relationships hold throughout the program. Establishing them often requires some inter-procedural reasoning—e.g., to see the aliasing effects of the constructor call on lines 25, 28, or 30. Constructors feature prominently in the example, since they are one of the best sources of must-alias information in a typical program.

3 Must-Alias Analysis Needs

Before presenting our optimized data structure for a must-alias analysis, it is important to ponder upon the properties of such an analysis and the algorithmic needs that arise if one is to implement it with performance in mind.

We consider a "must alias" relation defined on access paths, i.e., expressions of the form "`var(.fld)*`". There can be several formulations of such must-alias analyses, and our goal is to identify properties common to all. A concrete reference point, however, is the must-alias analysis of the Doop Java bytecode analysis framework [3]—our experimental evaluation will compare against this analysis. In the Doop analysis, the meaning of the MustAlias(*i, ctx, ap1, ap2*) relation is that access path *ap1* aliases access path *ap2* (i.e., they are guaranteed to point to the same heap object, or to both be `null`) right after program instruction *i*, executed under calling context *ctx* (effectively, a sequence of callers), provided that the instruction is indeed executed under *ctx* at program run-time. The two access paths are said to form an *alias pair*.

Representing Equivalence Relations. The first observation regarding the must-alias relation is that it is an equivalence relation. Consider once again the example in Figure 1. At line 32, a must-alias relationship is established between various access paths. A naive implementation would have to explicitly record each aliasing pair, i.e. `b1.container` with `a2`, with `a1.member.container` and with `a1` and so on enforcing that the relation stays reflexive, symmetric and transitively closed. In our simple example, for four program expressions we would need to record twelve distinct pairs (ignoring the trivial pairs of each expression with itself).

Since must-alias is an equivalence relation, it induces a partitioning of the space of access paths: every access path can only belong in one alias (equivalence) class. This means that we can represent the contents of each class compactly, by grouping together all aliased access paths. An access path can denote that it belongs in a certain alias class, e.g., by having a unique identifier, or by being a member in a linked data structure. The goal is to represent an alias class using linear space and time (in the number of its access paths) instead of enumerating all pairs of aliased access paths (and taking up quadratic space and time).

It is important to note that the concrete (i.e., dynamic) "alias" relation is also an equivalence relation, but most *may*-alias relations in the static analysis literature are *not*. For instance, in a typical subset-based pointer analysis, access path *ap1* may-alias *ap2* by pointing to the same abstract object (among others). Similarly, *ap2* may-alias *ap3*. However, it may not be the case that *ap1* and *ap3* may-alias: the common elements in the points-to sets of *ap1* and *ap2* may not be among the common elements in the points-to sets of *ap2* and *ap3*. This highly influences all data structure operations. Notably, the main algorithm that we will describe (intersection of data structures when joining control-flow paths) is not present in a may analysis.

Extending Access Paths. A less obvious observation concerns the representation of aliasing in extended access paths. A naive implementation would, once again, have to represent aliases explicitly. For instance, two aliased program variables `x` and `y` will also induce alias pairs `x.f` and `y.f`, as well as `x.g` and `y.g`, `x.f.g` and `y.f.g`, etc., up to the maximum access path length (and modulo valid field accesses). This is an exponential number, $\Omega(c^k)$, of aliased access paths, for c valid fields and access path length limit of k. The access path length can be easily limited (e.g., $k = 3$ does not restrict the vast majority of useful alias inferences), so the burden is not insurmountable, but it can still be significant.

Ideally, we would like a data structure that only explicitly maintains the initial aliasing relationship and can implicitly derive the aliasing of all extended access paths.

Algorithms. Once we have a data structure that satisfies the above requirements, what algorithms should we implement efficiently on this data structure? The basic algorithms behind most must-alias analysis inferences are straightforward. The analysis needs to copy alias classes (equivalence classes), add a single access path, remove a single access path, or rename variables in a set of alias classes. The only case that introduces some complexity is the one dealing with multiple predecessors of an instruction in the control-flow graph.

In a *must*-alias analysis setting, in order for an alias pair to be valid at the instruction where multiple control-flow paths meet, it should hold in each path. The operation we need here is that of taking the intersection of alias classes from many different sets (one for each predecessor instruction). In our running example, on line 31, we can infer that `a2.member` ∼ `b1` since it holds in both paths, but not that `a2.next` ∼ `a1`.

Notably, in contrast to a typical data structure for equivalence classes (e.g., union-find trees), unions of (non-singleton) equivalence classes do not arise: if an expression is newly aliased with others, it is because it is no longer aliased with its previous aliases. The corresponding operation is a single access path addition and removal (from a different class). Conversely, intersections of alias classes are central to our structure.

4 An Optimized Data Structure and Algorithms

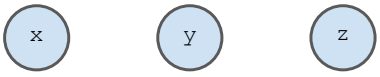
Based on the above requirements, we propose an *alias graph* data structure (and associated algorithms) for representing all alias sets of access paths that hold at a certain program point. In a typical must-alias analysis, a program point is a possibly context-qualified instruction. Each such *instruction-context combination* maintains an alias graph and the analysis updates it until fixpoint. An updated alias graph depends on the earlier graph for the same program point, on the graphs of its predecessor instructions, and on the current instruction's semantics.

We begin with a description of the easier case: how the current instruction affects the alias graph. This will also help illustrate the data structure.

The intuition is that an alias graph abstractly represents local variables and the heap, with abstract objects as placeholders for concrete objects. Nodes (abstract objects) are alias classes, edges are field-points-to relationships. Every abstract object, however, corresponds to (at most) a single concrete object at the current program point: our data structure is isomorphic with a part of the concrete heap. This property is true only because the data structure represents definite (must) aliasing.

We illustrate with simple examples. It is worth noting once again that every program instruction will maintain a different alias graph. The following examples focus on the situation at a certain instruction.

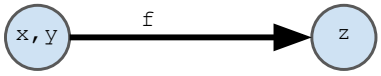
All variables conceptually begin with their own node in the graph. (In practice, such nodes need not be represented, unless connected to others.) The node represents "the object that the variable points to at this program point".



Aliasing can be induced by various program operations (e.g., Move, Load, and Store), as seen in our earlier model. Since we are interested in must-alias, two aliased variables have to point to the same object—their nodes can be merged if a Move instruction, `x = y`, is encountered:

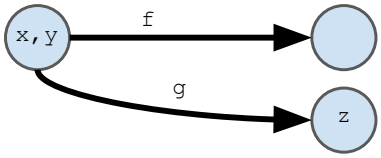
This collapsing of nodes is responsible for compact encoding of equivalence relations: two variables are computed to be aliases iff[1] they belong in the same node of the alias graph.

If the next instruction is a Store, `x.f = z`, the previous graph will get propagated—i.e., copied. Subsequently, the Store will add an edge to the graph, signifying that the field, `f`, of the object pointed by `x` will point to the object that `z` points to:



A subsequent Load operation, `z = y.g`,[2] will inherit the alias graph of its predecessor and will modify it. Variable `z` is removed from its old node (`z` no longer points to this abstract object), a new node for `z` is created, and the nodes are linked, to indicate that `z` now points to the same object as `y.g`. The empty, former node of `z` will be garbage collected if no other paths can reach it in the alias graph.

On the other hand, when an access path can still reach the emtpy node, the empty node provides useful information. It represents "the object that an access path points to at this program point". Empty here doesn't describe the lack of information–just the lack of a *local variable* pointing to this abstract object.



The Load operation shows that our alias graph, although intended to abstractly represent a real heap, behaves quite differently: a load from a field can introduce new objects, as well as update fields of existing objects.

Generally, the alias graph captures compactly all aliasing relationships among access paths. Maintaining the graph across program instructions is simple, as in the above examples. Graph manipulation merely has to observe some invariants:

- Two variables are in the same graph node iff the analysis reports them to be aliased. (Since alias classes are disjoint, the variables in different nodes are also disjoint.)
- A path in the graph represents a set of access paths, starting from a non-empty node (that denotes the base variables of the access paths) and extended with the field labels along the path's edges. If two paths in the graph reach the same node, all access paths they represent must be aliased.

For illustration, consider Figure 2, which shows the alias graph after line 31 of our example in Section 2.

The graph concisely represents a set of alias relationships that hold at that program point: `b1.container`

[1] The statement refers to the must-alias analysis results, not to actual aliasing during program execution.

[2] The example sequence of actions described is contracted for brevity. Our implementation works on a static single assignment (SSA) intermediate form, so this exact scenario will never arise, since `z` has had its value read earlier and its single assignment has to dominate its use.

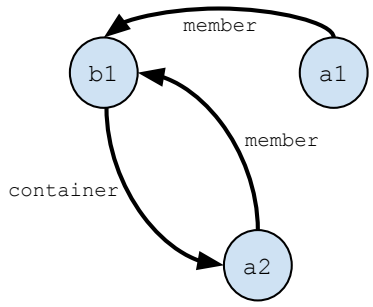


Figure 2. Example alias graph data structure.

~ a2, a1.member ~ b1, and a2.member ~ b1. An infinite number of other alias pairs are represented implicitly: a1.member ~ a2.member, a1.member.container ~ a2, a1.member.container.member ~ b1, a1.member.container.member ~ a1.member, etc.

Overall, the alias graph satisfies both of our requirements of Section 3 for an efficient representation. Equivalence relations are represented compactly: an alias class with n members does not need $O(n^2)$ space and time for its computation. Instead, it is represented implicitly, as all the variables in a node ($O(n)$ space) and all alias graph paths that can reach a node. Similarly, long (and even infinite) access paths are represented implicitly as graph paths. The implicit representation is sufficient for any specific queries (i.e., "are two given program expressions aliases?") and for subsequent aliasing computations, per the algorithms we detail next.

4.1 Main Algorithms

Most of the required must-alias analysis actions (per the discussion of Section 3) over our data structure are straightforward, consisting of copying, additions and removals of variables and edges, and variable renamings. Standard mappings for efficient indexing are required: each target of a directed edge needs to be able to quickly retrieve its source, and each program variable needs to quickly map to the node in which it appears.

For instance, according to the earlier definition of the data structure, finding all aliases of an access path is simple (but requires a transitive computation—our graph is a condensed representation of alias classes):

Algorithm: `all-aliases(ap)`

- Find the node for the base variable of access path ap, traverse in the forward direction the labeled edges that match each of the fields of ap to reach a target node.
- Any graph path that reaches the same node corresponds to an aliased access path, from a base variable adding the fields labeling the edges. (I.e., traverse $k-1$ directed edges backwards to find access paths of length up to k.)

For instance, in Figure 2, we can find all aliases of length 3 of access path a2.member by traversing edge member from node a2 (thus reaching the node containing b1) and finding all paths of length 2 that can reach the same node, also including the variable(s) in the starting node of the path (e.g., b1.container.member).

The more interesting algorithm, as suggested earlier, is that of intersecting alias graphs—necessary for merging alias information from predecessor instructions. This is easy to see as a repeated intersection of two graphs (which is then iterated by intersecting a third with the result, then a fourth, and so on). Note that the graphs do not need to contain a single connected component.

Algorithm: `intersect(`g_1`,`g_2`)`

- The domain of possible nodes for the result of the intersection is the cartesian product of nodes of g_1 and g_2. For every two nodes i, j of g_1 and g_2, respectively, node (i, j), if it exists in the intersection result, will contain the intersection of the variables of i and j.
- Nodes are materialized incrementally, according to the rules below.
  1. For every two nodes i, j of g_1 and g_2, if the intersection of the variables of i and j is non-empty, add to the intersection result a new node (i, j).
  2. (Repeatedly) If node (i, j) exists in the intersection result, then for every label f, if g_1 has an edge $i \rightarrow k$ with label f, and g_2 has an edge $j \rightarrow l$, also with label f, then add to the intersection result (if not already present):
     - a node (k, l) (possibly empty);
     - an edge $(i, j) \rightarrow (k, l)$ with label f.

Note that the first step is of linear complexity in the number of nodes, since empty nodes can be eagerly skipped and indexing from a variable to the, up to one, node that may contain it in a different graph is constant-time.

The algorithm considers all possible pair-wise node combinations and all possible edges out of node intersections. It maintains the property that any aliasing relationship (either variables belonging in the same node, or paths reaching the same node) in the result also exists in both input alias graphs.

Notably, the intersection of two alias graphs can produce nodes with empty variable sets, due to the second step of the algorithm. Empty nodes with no in-edges can be eliminated eagerly. Empty nodes with in-edges are meaningful in the output and need to be maintained. To illustrate, consider the example in Figure 3.

In this case, the empty note denotes that access paths x.f and z.g are still aliased in the intersection alias graph, even though they are no longer aliased with any single-variable access path.

For upper bounds n, v, e in the number of nodes, variables, and edges in the input alias graphs, respectively, the algorithm has a running time asymptotic bound of $O(n+v+e)$, i.e.,

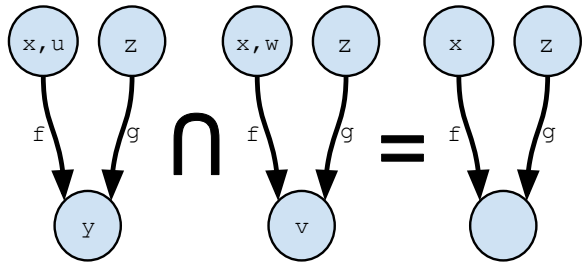


Figure 3. Intersecting alias graphs

linear in all quantities, if one assumes a practically constant-time indexing scheme from a variable to its node. (Proof sketch: Non-empty nodes are fewer than variables and only linear cost is incurred when combining non-empty nodes pair-wise, since each node has a distinct set of variables, used to index into any node that may intersect that set in the other alias graph. Empty nodes arise in the result and are only examined in the input if there is an edge into or out of them, therefore their number is below e. The number of edges in the output is at most that in the input (taken as the union of both input graphs).

Empty nodes with no in-edges are only one instance of nodes that no longer encode useful access path aliasing. Such nodes should be garbage-collected for maximum efficiency, producing a normalized input. The node collection algorithm is as follows:

Algorithm: **gc**(g)

- Any node in g containing a single variable and with no incoming or outgoing edges is eliminated.
- Any node in g containing no variables and with either zero in-edges or one in-edge and zero out-edges is eliminated.

In all cases covered by the above algorithm, the node does not encode alias pairs that would disappear with the node's removal: either there are no two paths (or variable names) that reach the node, or the node does not extend other paths beyond the implicit extension with the same field names that is already captured by the data structure.

4.2 Use in Practice

Having described the individual steps of an analysis using a must-alias data structure, we turn our attention to how it is used in the context of a realistic analysis. Consider the must-alias analysis of the Doop framework, discussed in Section 3. This computes a MustAlias(*i, ctx, ap1, ap2*) relation, i.e., aliased access paths at each program point and calling context. That is, each instruction-context combination is associated with an alias graph. Initially, conceptually every possible variable has its own node. (Implementation-wise, this is represented as an empty graph, requiring no initialization.) Every program instruction is visited and its alias graph is updated based on the instruction semantics and the operations that we described earlier. Specifically, every variable-aliasing instruction merges nodes (creating them if they only existed implicitly, i.e., they were single-variable nodes), every load and store instruction creates nodes and/or edges, every instruction also integrates the alias graphs of its predecessors, intersecting them if the instruction is a control-flow merge point. The visit order is not important for correctness, though it might affect performance (i.e., the number of steps needed before convergence is achieved).

At a call instruction, the Doop analysis creates a new instruction-context pair (unless it exists already) for the first instruction of the callee method and the given context. Maximum context depth is a parameter of the analysis and if reached the call instruction will not be further analyzed. This is a sound approach for a must-alias analysis, since the aim is to compute an *underapproximation* of the alias relationships that are guaranteed to hold. With our data structure, the alias graph at the call site is copied to the first instruction in the callee method and then the usual operations are applied.

The above are repeated until a fixpoint is met. At any given alias graph, the number of non-empty nodes is bounded by the number of local variables in the program text. Empty nodes can arise but gc ensures that the intersection operation, where merging of states takes place, can never produce more modes than the union of its inputs graphs: if an empty node is kept, it is because an incident edge existed in the input, hence a corresponding node appeared in the input.

4.3 Declarative Implementation

As discussed earlier, the alias graph data structure can be employed in a must-alias analysis by maintaining alias graphs per-program-point (i.e., per context-qualified instruction), and updating them (to incorporate information from their predecessor instructions) until fixpoint.

The data structure description we have seen so far considers this update to be a destructive operation. For instance, after a Move instruction, we saw the nodes of two variables getting merged. Similar merging can be induced by information that the analysis discovers while it is executing (i.e., not directly induced by the semantics of the current instruction)—e.g., propagated from predecessors.

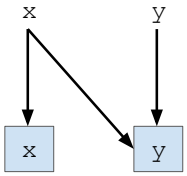
We have also designed and implemented a declarative/purely functional version of the data structure. The main reason for the declarative implementation has been fairness in experimental comparisons. We will compare our optimized implementation against the must-alias analysis of the Doop framework, implemented in Datalog. Thus, it is desirable to also implement a, perhaps not as optimal, declarative version of the data structure in Datalog. This will allow us to isolate the effect of destructive updates from the inherent properties of the data structure.

In the declarative version of the structure, aliased abstract objects are not merged, but instead associated with each other. Schematically, we can consider that each variable has

its own node and points to at least one abstract object—at first the abstract object signifying "whichever object the variable got assigned to" (at its single-assignment site):

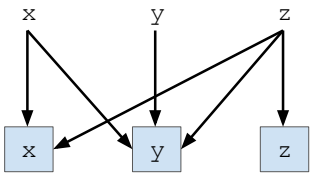
(For the declarative implementation, we will represent abstract objects as squares, to avoid confusion.)

Since we no longer have destructive updates, variables can point to multiple abstract objects. For instance, after a Move instruction, `x = y`, we have:



As before, however, a variable can really point to a single concrete object. Therefore, the two abstract objects that x points to have to be different abstractions of the same concrete object—their equivalence is encoded, but implicitly. It is easy to check that variables x and y are aliased, in the usual sense: they both point to the same abstract object.

If the next operation is another Move, `z = x`, variable z is made to point to both abstract objects that x points to:



Thus, the declarative data structure is less efficient than its imperative counterpart: a quadratic representation of an alias set cannot be precluded, and depends on the order of alias-inducing instructions. In our example, we could then assign z, the variable with the most out-edges, to a new variable, w, then assign w (which now has the most out-edges) to a new variable, and so on. In practice we expect that this effect will be mitigated. If the next instruction assigns y to a new variable, u, then u receives only a single extra edge, maintaining a more compact representation of the alias set. The cost, much as in the imperative structure, is that the alias set is not fully explicit and requires a transitive closure computation to be materialized.

5 Implementation and Experiments

We implemented an analysis that functionally matches the published specification [3] of the must-alias analysis in the Doop framework [5]. Doop is a declarative framework for Java bytecode analysis, with analyses written in the Datalog language. In contrast, our implementation is in Java, since our optimized alias graph data structure has imperative features in its full form. Finally, to also eliminate language-level factors of the Datalog-vs-Java implementations, we also produced an optimized Datalog implementation, based on our purely functional data structure.

The three implementations are functionally equivalent, with very minor variations, due to clear engineering differences: The original Datalog analysis has to bound the access path length for aliases to a finite number, while the optimized data structure implicitly stores aliases for longer access paths.

We use a 64-bit machine with an Intel(R) Xeon(R) E5-2687W v4 (24-cores) CPU at 3.00GHz. The machine has 512GB of RAM. All measurements are single-threaded (though, as is common, Java runs its garbage collector in extra threads) and all executions occupy only a small fraction of the available RAM. We experiment with the DaCapo benchmark programs [4] v.2006-10-MR2 and v.9.12-bach under JDK 1.7.0_45. We use the LogicBlox Datalog engine, v.3.10.14.

Speed across benchmarks. Figure 4 shows the performance effect of our optimized data structure on analyzing the benchmark programs. We bounded the access path length (for the original Datalog analysis) to 3 and the analysis context depth to 2.

Note that the figure is log-scale. Across all benchmarks, the difference between the optimized implementations and the original is typically at least an order of magnitude and often close to two. The speedup of the two optimized implementations (vs. the original) is also shown more explicitly in Figure 5: over half the benchmarks enjoy speedups of over 20x for both the Java and the Datalog optimized implementation. The Java version of the data structure achieves a median speedup of 25.7x (min. 8.4x, max. 68.9x), while the Datalog version has a median speedup of 24.6x (min. 5.4x, max. 47.3x). The analysis time typically drops from over ten minutes to under half a minute.

It is not hard to see why the explicit representation is not competitive. Figure 6 correlates the number of aliased access-path pairs (computed by the original analysis) and execution time. (This applies to context-qualified access paths, in the application and libraries alike, as long as the library code is reachable from application code with the given context depth.) This metric reflects the size (in tuples) of the corresponding relation in the Datalog database. It clearly suggest that maintaining access path relationships explicitly can prove quite costly.

Varying access-path length. To further see the performance advantage of the optimized representation of must-alias information, one can vary the maximum access path length allowed for computations of the original, explicit

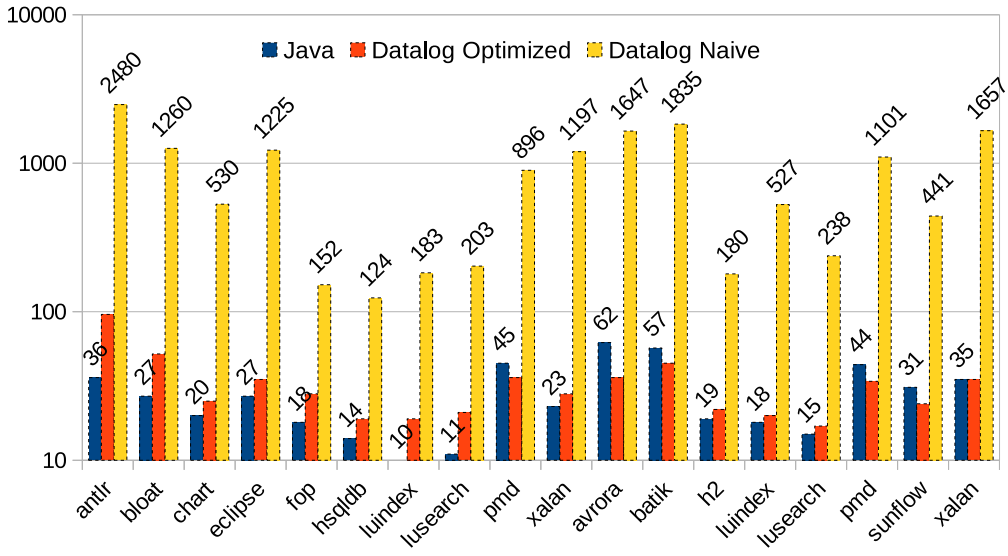


Figure 4. Execution time (sec.) of the analysis. We only show the numbers for the Java and Datalog naive versions, to avoid crowding the chart.

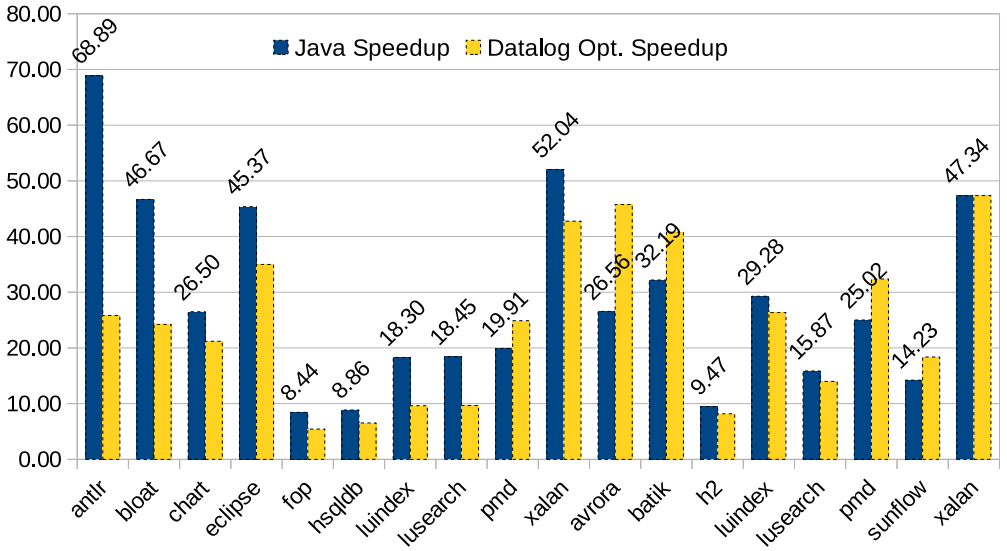


Figure 5. Speedup of the two analyses employing optimized data structure.

(Datalog) implementation. Figure 7 shows how running time varies for maximum access path lengths of 2, 3 (same as in Figure 4), 4 and 5. The numbers are for the xalan benchmark. The speedup readily grows to over 75x for an allowed access path length of 5. The optimized Datalog implementation is shown as a baseline although it should be (and is) largely unaffected by the change of maximum access path length.

Varying context depth. Similar observations can be made by varying the context depth of the analysis. As seen in Figure 8, although the running time of the optimized implementation grows slowly, the running time of the explicit representation of alias relationships gets dramatically higher. For a context depth of 4, the explicit representation did not terminate after one-and-a-half hour.

Recall the two claimed benefits of the optimized representation: long access paths are represented implicitly, and equivalence classes are represented with linear space and time complexity, instead of quadratic. It is the latter factor that comes into play when context depth increases: alias sets

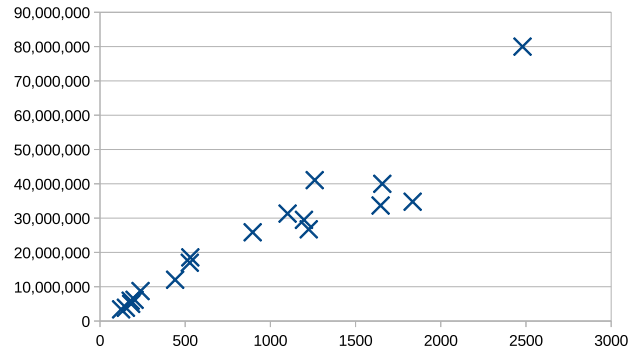


Figure 6. Number of pairs of (instruction-and-context-qualified) access paths that must alias vs. analysis time.

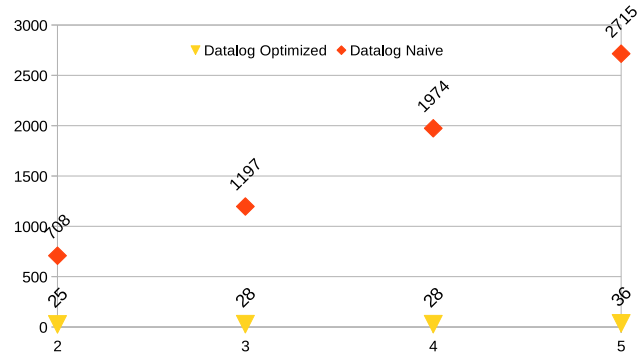


Figure 7. Execution time when varying maximum access-path length. Optimized Datalog running time given as a baseline.

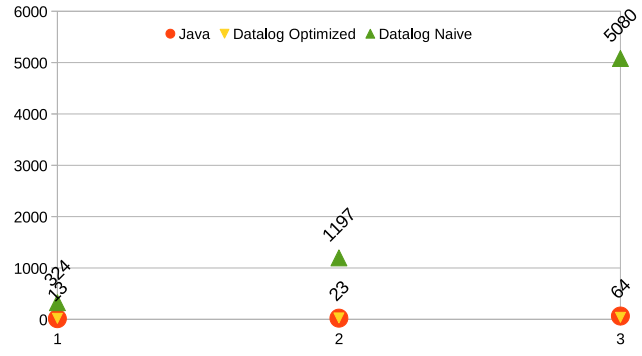


Figure 8. Execution time when varying maximum context depth.

grow in size, by exploiting inter-procedural inference (e.g., aliasing established at the caller and propagated through formal arguments) in addition to local instructions.

In all, the optimized representation fulfills its promise of a much more economical representation of must-alias (equivalence) relations.

6 Related Work

There are several approaches in the literature that present must-analyses in the pointer analysis setting or employ them in a may-analysis. Additionally, there are several approaches that integrate efficient data structures in the representation of points-to information.

Data Structures and Heap Abstractions. Our optimized data structure is (partly) based on the observation that must-alias sets are equivalence classes. This is not the first time that a data structure that efficiently implements equivalence classes has been used to speed up pointer analysis. Most notably, a Steensgaard-style (or *unification-based*) [19] analysis computes may-point-to sets that are equivalence classes. This means that points-to sets are disjoint—if two points-to sets are found to possibly overlap, they get unified. This loses precision (relative to a standard subset-based points-to analysis) but enables the algorithm to use union-find trees for a very efficient representation.

Another optimized data structure often used in pointer analysis is the *constraint graph*: a graph with nodes denoting pointer variables and an edge between nodes `p` and `q` denoting flow (e.g., a direct assignment) from variable `p` to variable `q`. Online cycle elimination by Fähndrich et al. [7] detects cycles in the constraint graph and collapses all nodes in a cycle into a representative node, since such nodes will have identical points-to information. The technique of Nasre [15] extends such constraint graph reasoning based on the observation that if two nodes have the same dominator in the constraint graph, then they are clones: the values flowing to them are (only) those of the dominator node. Several other constraint graph optimizations are applied off-line (i.e., before the points-to analysis runs). Prime examples of such techniques are Rountev and Chandra's [17] and Hardekopf and Lin's [10]. (Hardekopf and Lin have also applied similar ideas in a hybrid online/offline setting [9].) Both of these techniques perform an off-line detection of equivalent points-to sets and use this knowledge to eliminate redundant work in subsequent points-to computations. Our data structure can be seen as somewhat analogous to constraint-graph techniques, in the sense that we do not compute the flow of objects or the fully expanded set of all possible alias pairs. Instead, we compute the "wiring" (i.e., the alias relationships, locally, that the program induces) and keep the alias information in condensed form, until it needs to be queried by a client analysis.

Another conceptual relative of our data structure is the model presented by Madhavan et al. [14] for modular *may* analyses. That model is similar in that it invents abstract nodes for heap objects that resemble ours (without the equivalence-class nature). The Madhavan et al. approach aims to achieve modular reasoning, i.e., to model the heap effects of a method without knowing its calling environment. To do so, the approach creates abstract nodes that represent

concepts such as "whichever object variable x may point to". Our data structure has nodes with a similar meaning, however we also take advantage of the "must" nature of the analysis to merge nodes, every time the same access path can reach both.

Must-Analyses for Aliasing. There are several instances of past work that apply must reasoning in pointer analysis. These mostly serve to paint the landscape of potential applicability of our data structure.

Ma et al. [13] present an algorithm for null-pointer dereference detection using a context-insensitive may-alias and a must-alias analysis; the latter is used to increase the precision of the former, by enabling strong updates when possible.

Nikolić and Spoto [16] present a must-alias analysis that tracks aliases between program expressions and local variables (or stack locations, since they analyze Java bytecode, which is a stack-based representation). The analysis is a generator of constraints, which are subsequently solved to produce the analysis results.

Hind et al. [11] present a collection of pointer analysis algorithms. Among them, the most relevant to this work is a flow-sensitive interprocedural pointer alias analysis. The authors optimistically produce *must* information for pointers to single non-summary objects.

Emami et al. [6] present an approach that simultaneously calculates both must- and may-point-to information for a C analysis. Their empirical results "show the existence of a substantial number of definite points-to relationships, which forms very valuable information"—much in line with our own experience.

Must- information is often computed in conjunction with a client analysis. One of the best examples is the typestate verification of Fink et al. [8], which demonstrates the value of a must-analysis and the techniques that enable it.

An approach for integrating *must* point-to reasoning in an analysis is to propagate such information only at instructions where we know that the given heap allocation target still refers to the last object allocated at that site [1]. Thus, an execution path that may create another object at the same site (such as when reaching the end of the loop) would invalidate any previous must-point-to facts (i.e., it will stop them from propagating any further).

Generally, must-analyses can vary greatly in sophistication and can be employed in an array of different combinations with may-analyses. The analysis of Balakrishnan and Reps [2], which introduces the *recency abstraction*, distinguishes between the most recently allocated object at an allocation site (a concrete object, allowing strong updates) and earlier-allocated objects (represented as a summary node). The analysis additionally keeps information on the size of the set of objects represented by a summary node. At the extreme, one can find full-blown shape analysis approaches, such as that of Sagiv et al. [18], which explicitly maintains must- and may- information simultaneously, by means of three-valued truth values, in full detail up to predicate abstraction: a relationship can definitely hold ("must"), definitely not hold ("must not", i.e., negation of "may"), or possibly hold ("may"). Summary and concrete nodes are again used to represent knowledge, albeit in full detail, as captured by arbitrary predicates whose value is maintained across program statements, at the cost of a super-exponential worst-case complexity.

Jagannathan et al. [12] present an algorithm for must-alias analysis of functional languages. The algorithm adapts must-alias insights to the setting of captured variables. For instance, must-alias information for non-summary objects permits strong updates, which the authors find to improve analysis precision. We employ must-alias analysis results quite similarly in applications of our model analysis.

7 Conclusions

We presented a data structure for the its optimized implementation of must-alias analysis over access paths. The algorithmic improvements afforded by the specialized data structure yield a large performance advantage, often approaching two orders of magnitude.

Acknowledgments.

We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), the Facebook Research and Academic Relations Program, and an Oracle Labs collaborative research grant.

References

[1] R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *Proc. of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 74–84, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199466.

[2] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. of the 14th International Symp. on Static Analysis*, SAS '06, pages 221–239. Springer, 2006.

[3] G. Balatsouras, K. Ferles, G. Kastrinis, and Y. Smaragdakis. A datalog model of must-alias analysis. In *Proc. of the 6th International Workshop on State Of the Art in Program Analysis*, SOAP '17, pages 7–12. ACM, 2017. ISBN 978-1-4503-5072-3. doi: 10.1145/3088515.3088517.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488.

[5] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc.*

of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X.

[7] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. of the 1998 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: 10.1145/277650.277667.

[8] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–144, New York, NY, USA, 2006. ACM. ISBN 1-59593-263-1. doi: http://doi.acm.org/10.1145/1146238.1146254.

[9] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2.

[10] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *Proc. of the 14th International Symp. on Static Analysis*, SAS '07, pages 265–280. Springer, 2007. ISBN 978-3-540-74060-5.

[11] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999. ISSN 0164-0925. doi: 10.1145/325478.325519.

[12] S. Jagannathan, P. Thiemann, S. Weeks, and A. Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 329–341, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268973.

[13] X. Ma, J. Wang, and W. Dong. Computing must and may alias to detect null pointer dereference. In *Proc. of the 3rd International Symp. On Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *ISoLA '08*, pages 252–261. Springer, 2008. ISBN 978-3-540-88478-1. doi: 10.1007/978-3-540-88479-8_18.

[14] R. Madhavan, G. Ramalingam, and K. Vaswani. A framework for efficient modular heap analysis. *Foundations and TrendsÂő in Programming Languages*, 1(4):269–381, 2015. ISSN 2325-1107. doi: 10.1561/2500000020. URL http://dx.doi.org/10.1561/2500000020.

[15] R. Nasre. Exploiting the structure of the constraint graph for efficient points-to analysis. In *Proc. of the 2012 International Symp. on Memory Management*, ISMM '12, pages 121–132. ACM, 2012. ISBN 978-1-4503-1350-6. doi: 10.1145/2258996.2259013.

[16] D. Nikolić and F. Spoto. Definite expression aliasing analysis for Java bytecode. In *Proc. of the 9th International Colloquium on Theoretical Aspects of Computing*, volume 7521 of *ICTAC '12*, pages 74–89. Springer, 2012. ISBN 978-3-642-32942-5. doi: 10.1007/978-3-642-32943-2_6.

[17] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349310.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002. ISSN 0164-0925.

[19] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.