Using Disjoint Reachability for Parallelization

James Jenista, Yong hun Eom, and Brian Demsky

University of California, Irvine

Abstract. We present a disjoint reachability analysis for Java. Our analysis computes extended points-to graphs annotated with reachability states. Each heap node is annotated with a set of reachability states that abstract the reachability of objects represented by the node. The analysis also includes a global pruning step which analyzes a reachability graph to prune imprecise reachability states that cannot be removed with local reasoning alone. We have implemented the analysis and used it to parallelize 9 benchmarks. Our evaluation shows the analysis results are sufficiently precise to parallelize our benchmarks and achieve an average speedup of 16.9×.

1 Introduction

As the number of cores in mainstream processors increases, more software application domains are challenged with developing parallelized implementations in order to harness the available cores. Program analyses such as points-to analysis [1, 2] and shape analysis [3–7] support parallelization tools, particularly in application domains like scientific programs that exhibit highly regular data structures and workloads with much parallelism. In the future, developers will likely need to parallelize programs in the domain of object-oriented desktop and server applications where the program heap is complicated by many levels of abstraction or arbitrary data structure composition.

We believe the key to efficient and general heap analysis in support of parallelizing object-oriented applications is to focus on reachability. Data structures often have a single root object — disjoint reachability analysis can statically extract the property that the objects that comprise a data structure are only reachable from at most one data structure root object. This property is powerful for parallelizing tasks — the combination of (1) the static reachability analysis plus (2) a lightweight dynamic check to determine that the variables live into two tasks reference different root objects suffices to guarantee that the two tasks will not perform conflicting data accesses.

The use of reachability for parallelization is not new; points-to analysis is typically fast and can be scaled to millions of lines of code, but reachability information derived from it is imprecise so it can assist parallelization of only very specific code patterns. A path of edges in a points-to graph can capture that a data structure root object can reach the objects that comprise the data structure, but also admits the possibility that many data structure root objects can reach the same objects.

Reachability from objects is essential and different from the reachability from variables discovered by alias analysis [8–10] — reachability from variables can only express a finite number of disjoint sets. Variable reachability cannot discover that an unbounded set of live data structures do not share objects. This property is often necessary to parallelize computations on data structures.

The strategy of shape analysis is to statically model a potentially unbounded heap with a *shape graph*, where heap objects that share a common pattern of references are summarized with a shape node. While reachability properties can be deduced from a precise shape graph, in the worst case a program's heap shape is difficult to analyze to the effect that the derived reachability properties are much less precise.

Naik and Aiken introduced the concept of disjoint reachability in support of static race detection [11]. We introduce a new static heap analysis, disjoint reachability analysis, for discovering precise reachability properties in Java programs with general heap structures. Our analysis compactly represents the reachability of the many objects abstracted by a single node of a points-to graph by annotating each node with a set of reachability states. The reachability of an object is conservatively approximated by one of the reachability states for the corresponding node in the points-to graph. Shape information is not preserved in general; in this regard, shape analysis and disjoint reachability analysis are complementary. For instance, a compiler might generate tree-specific parallel code when shape analysis discovers that a structure is a valid tree.

Existing heap analyses have primarily either used equivalence classes of objects (e.g. TVLA [5], separation logic [12], storage shape graphs [3]) or access paths [4, 10] to reason about heap references. Disjoint reachability analysis combines aspects of both approaches — it combines an abstraction that decomposes the heap into static regions with reachability annotations that provide access path information. Choi et al. used a combination of storage and paths for alias analysis [13], however that approach focuses only on reachability from variables and is subject to the limitations discussed above.

1.1 Basic Approach

Disjoint reachability analysis extends a standard points-to graph with reachability annotations. The nodes in our points-to graph abstract objects and the edges abstract heap references between objects. While the points-to graph captures some reachability information; nodes in the points-to graph by necessity must abstract many objects. In general it is impossible to determine whether a path of edges in a points-to graph from a node n_{src} to a node n_{dst} represents a path of references from one object or many objects abstracted by n_{src} to an object abstracted by n_{dst}. We therefore annotate nodes with sets of reachability states. A reachability state for an object o contains a tuple for each node n that gives an abstract count of how many objects abstracted by node n can reach o.

Using reachability states only on nodes can make it difficult to precisely propagate changes to reachability states. We therefore also annotate edges with sets of reachability states. The reachability state for an edge abstracts the reachability states for the objects that can be reached through that edge. In addition to enabling the analysis to more precisely propagate changes to reachability states, edge reachability annotations can also serve to refine the reachability information for a node based on the reference used to access an object abstracted by the node.

We evaluate the precision of disjoint reachability analysis in support of a task-level parallelizing approach that relies on reachability properties rather than exploiting heap shape. Out-of-order Java (OoOJava) [14, 15] decomposes a sequential program into tasks and discovers which data structure root objects a task uses to obtain references to other objects. OoOJava then queries disjoint reachability analysis to discover whether the objects reachable by two tasks are disjoint conditionally on whether the root objects of the tasks are distinct at runtime. This information enables parallelization when combined with a constant-time dynamic check that verifies the tasks access distinct root objects. OoOJava reports disjoint reachability results back to the developer to identify unintended sharing that prevents parallelization. We also note disjoint reachability analysis is employed by Bamboo [16], another task-based parallel programming model.

Our analysis is demand-driven — it takes as input a set of allocation sites that are of interest to the analysis client. The analysis then computes the reachability only from the objects allocated at the selected allocation sites to all objects in the program.

1.2 Contributions

The paper makes the following contributions:

- **Disjoint Reachability Analysis:** It presents a new demand-driven analysis that discovers precise disjoint reachability properties for a wide variety of programs.
- **Reachability Abstraction:** It extends the points-to graph abstraction with reachability annotations to precisely reason about reachability properties.
- **Global Pruning:** It introduces a global pruning algorithm to improve the precision of reachability states.
- **Experimental Results:** It presents experimental results for several benchmarks. The results show that the analysis successfully discovers disjoint reachability properties and that it is suitable for parallelizing the benchmarks with significant speedups.

The remainder of the paper is organized as follows. Section 2 presents an example that illustrates how the analysis operates. Section 3 presents the program representation and the reachability graph. Section 4 presents the intraprocedural analysis. Section 5 presents the interprocedural analysis. Section 6 evaluates the analysis on several benchmarks. Section 7 presents related work; we conclude in Section 8. The appendix formalizes the reachability abstraction and overviews the correctness of the analysis.

2 Example

Figure 1 presents an example that constructs several graphs and then updates the vertices in those graphs. The `graphLoop` method populates an array with `Graph` objects. The developer uses the `task` keyword to indicate that the annotated block is worth executing in parallel with the current thread if it is safe to do so without violating the program's sequential semantics. Our analysis will show that each `Vertex` object is reachable from at most one `Graph` object. This information could be used to parallelize the execution of tasks in the loop in Line 8. If a runtime check shows that instances of the task in Line 9 operate on different `Graph` objects, then our static analysis results will imply that task instances operate on disjoint sets of `Vertex` objects. The OoOJava compiler [14] therefore *flags* the allocation site on Line 4 to indicate to the analysis that it needs information about the reachability from `Graph` objects to all objects. In this example, the disjoint reachability results allow the OoOJava compiler to generate a parallel implementation.

2.1 Intraprocedural Analysis

We next examine the analysis of the `graphLoop` method. Our analysis computes a reachability graph for each program point. Figure 2(a) presents the analysis results just after the allocation of the `Graph` and `Vertex` objects referenced by variables `g` and `v1`, respectively. The rectangular heap node $\mathtt{n}_2$ abstracts the most recently allocated `Graph` object and is associated with a flagged allocation site; we call such heap nodes *flagged heap nodes* and shade them in all graphs in this paper. The analysis computes

```
1  public void graphLoop(int nGraphs) { Graph[] a=new Graph[nGraphs];
2     for(int i=0; i<nGraphs; i++) {
3     Graph g=new Graph(); /* Analysis client
4    flags this site */  Vertex v1=new Vertex();
5     g.vertex=v1; Vertex
6    v2=new Vertex();  v2.f=v1;
7    v1.f=v2; a[i]=g; }
8    for(int i=0; i<nGraphs; i++) {  Graph g=a[i];
9    task {  /* This task updates the graph
10   vertices. */ Vertex v=g.vertex; while(!v.marked) { v.marked=true;
11   v.updateVertex(); v=v.f; } } } }
```

Fig. 1. Graph Example

reachability only from flagged nodes. Heap nodes are assigned unique identifiers of form $\mathtt{n}_\mathtt{i}$, where i is an unique integer. The reachability set $\{[\langle \mathtt{n}_2, \mathtt{1}\rangle]\}$ on $\mathtt{n}_2$ indicates the object abstracted by that node has the *reachability state* $[\langle \mathtt{n}_2, \mathtt{1}\rangle]$. The reachability set $\{[\langle \mathtt{n}_2, \mathtt{1}\rangle]\}$ on the `g` edge indicates that the corresponding heap reference can only reach objects whose reachability state is $[\langle \mathtt{n}_2, \mathtt{1}\rangle]$.

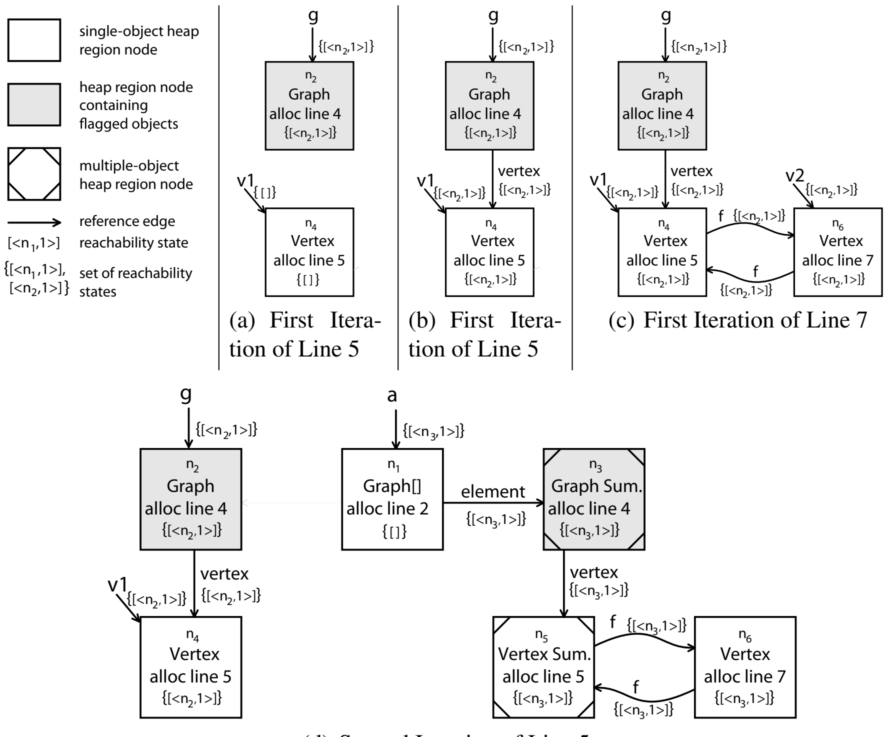
A reachability state for object o contains a set of *reachability tuples*: a reachability tuple consists of a heap node abstracting objects that may reach o and an arity that indicates how many objects abstracted by that node may reach o. The reachability state $[\langle \mathtt{n}_2, \mathtt{1}\rangle]$ means that the object with that reachability state is reachable from at most one `Graph` object abstracted by heap node $\mathtt{n}_2$ and no other flagged objects. In Figure 2(a), node $\mathtt{n}_4$ abstracts the most recently allocated `Vertex` object from Line 5 and has the reachability set $\{[\,]\}$, meaning at this program point the object abstracted by $\mathtt{n}_4$ is not reachable from any flagged objects. Appendix A precisely defines the abstraction.

Figure 2(b) presents the analysis results after the `vertex` field of the `Graph` object is updated to reference the `Vertex` object in Line 5. The set of reachability states for $\mathtt{n}_4$ is updated to $\{[\langle \mathtt{n}_2, \mathtt{1}\rangle]\}$ to reflect that the `Vertex` object is now reachable from at most one `Graph` object. The newly created edge models the reference from the `vertex` field of the `Graph` object. Edges are marked with the field they abstract.

Figure 2(c) presents the analysis results after Line 7. At this point a second `Vertex` object has been allocated and the two `Vertex` objects have references to one another. The heap reference from $\mathtt{n}_4$ to $\mathtt{n}_6$ propagated the reachability set $\{[\langle \mathtt{n}_2, \mathtt{1}\rangle]\}$ to $\mathtt{n}_6$.

The intraprocedural analysis continues until it computes a fixed-point solution. Figure 2(d) presents the analysis results after visiting Line 7 a second time. Node $\mathtt{n}_2$ abstracts the most recently allocated `Graph` object while node $\mathtt{n}_3$ summarizes the `Graph` object from the previous loop iteration. We denote summary heap nodes as rectangles with chords across each corner. The reachability state $[\langle \mathtt{n}_3, \mathtt{1}\rangle]$ has propagated backward across all edges up to the reference from variable `a`. Disjoint reachability analysis maintains the invariant that a reference is annotated with the reachability states of any objects reachable by following that reference. Note we omit the empty reachability state $[\,]$ in some reachability sets for brevity.

Figure 3 presents the analysis results at Line 7 of the example program. These results state that any `Vertex` object is reachable from at most one `Graph` object because (1) the analysis client flagged the only allocation site for `Graph` objects and (2) there is no heap node abstracting `Vertex` objects with a reachability state indicating the con-


single-object heap region node
heap region node containing flagged objects
multiple-object heap region node
reference edge
[<n1,1>] reachability state
{[<n1,1>], [<n2,1>]} set of reachability states
g
{[<n2,1>]}
n2 Graph alloc line 4 {[<n2,1>]}
v1
{[]}
n4 Vertex alloc line 5 {[]}
(a) First Iteration of Line 5
vertex {[<n2,1>]}
n4 Vertex alloc line 5 {[<n2,1>]}
(b) First Iteration of Line 5
v2
f {[<n2,1>]}
n6 Vertex alloc line 7 {[<n2,1>]}
(c) First Iteration of Line 7
a
{[<n3,1>]}
n1 Graph[] alloc line 2 {[]}
element {[<n3,1>]}
n3 Graph Sum. alloc line 4 {[<n3,1>]}
vertex {[<n3,1>]}
n5 Vertex Sum. alloc line 5 {[<n3,1>]}
f {[<n3,1>]}
n6 Vertex alloc line 7 {[<n3,1>]}

(d) Second Iteration of Line 5

Fig. 2. Intraprocedural reachability graph results for several program points.

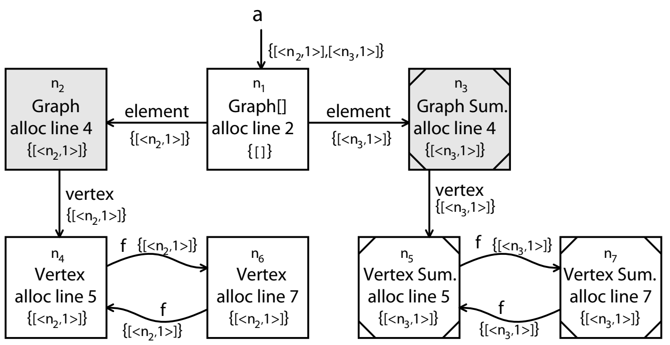

a
{[<n2,1>],[<n3,1>]}
n2 Graph alloc line 4 {[<n2,1>]}
element {[<n2,1>]}
n1 Graph[] alloc line 2 {[]}
element {[<n3,1>]}
n3 Graph Sum. alloc line 4 {[<n3,1>]}
vertex {[<n2,1>]}
vertex {[<n3,1>]}
n4 Vertex alloc line 5 {[<n2,1>]}
f {[<n2,1>]}
n6 Vertex alloc line 7 {[<n2,1>]}
n5 Vertex Sum. alloc line 5 {[<n3,1>]}
f {[<n3,1>]}
n7 Vertex Sum. alloc line 7 {[<n3,1>]}

Fig. 3. Analysis result at Line 7 of `graphLoop`.

trary. Examples of reachability states for a `Vertex` object that is possibly reachable from more than one `Graph` object are $[\langle \mathtt{n}_2, \mathtt{1}\rangle, \langle \mathtt{n}_3, \mathtt{1}\rangle]$ and $[\langle \mathtt{n}_3, \mathtt{MANY}\rangle]$.

Disjoint reachability analysis will determine that a given `Vertex` object is reachable from at most one `Graph` object regardless of the relative shape of `Vertex` objects in the heap. In Section 6 we describe how OoOJava queries disjoint reachability results such as this in order to parallelize benchmarks with a variety of heap structures. In this example, the OoOJava compiler will use the analysis results to generate a simple dynamic check: if the `Graph` object referenced by variable `g` is different from `Graph` objects referenced in previous iterations, then the task in Line 9 may safely execute in parallel with the current thread.

3 Analysis Abstractions

This section presents the analysis abstractions for the input program, elements of the reachability graph, and reachability annotations that extend reachability graphs.

3.1 Program Representation

The analysis takes as input a standard control flow graph representation of each method. Program statements have been decomposed into statements relevant to the analysis: copy, load, store, object allocation, and call site statements.

3.2 Reachability Graph Elements

Our analysis computes a reachability graph for the exit of each program statement. Reachability graphs extend the standard points-to graph representation to maintain object reachability properties. Heap nodes abstract objects in the heap. There are two heap nodes for each allocation site $m \in M$ in the program — one heap node abstracts the single most recently allocated object at the allocation site, and the other is a summary node that abstracts all other objects allocated at the site[1].

In general, analysis clients only need to determine reachability from some subset of the objects in the program. The analysis takes as input a set of allocation sites $M_F \subseteq M$ for objects of interest — the analysis then computes for all objects in the program their reachability from objects allocated at those sites.

The reachability graph G has the set of heap nodes $n \in N = M \times \{0, \texttt{summary}\}$. The analysis client specifies a set of flagged heap nodes $N_F = M_F \times \{0, \texttt{summary}\} \subseteq N$ that it is interested in determining reachability from.

Graph edges $e \in E$ abstract references $r \in R$ in the concrete heap and are of the form $\langle v, n\rangle$ or $\langle n, f, n'\rangle$. The heap node or variable that edge e originates from is given by $\mathrm{src}(e)$ and the heap node that edge e refers to is given by $\mathrm{dst}(e)$. Every reference edge between heap nodes has an associated field $f \in F = \texttt{Fields} \cup \texttt{element}$[2].

The equation $E \subseteq V \times N \cup N \times F \times N$ gives the set of reference edges E in a reachability graph. We define the convenience functions:

$$E_n(v) = \{n \mid \langle v, n\rangle \in E\} \quad (3.1) \qquad E_e(n) = \{\langle n, f, n'\rangle \mid \langle n, f, n'\rangle \in E\} \quad (3.4)$$

$$E_e(v) = \{\langle v, n\rangle \mid \langle v, n\rangle \in E\} \quad (3.2) \qquad E_n(v, f) = \{n' \mid \langle v, n\rangle, \langle n, f, n'\rangle \in E\} \quad (3.5)$$

$$E_n(n) = \{n' \mid \langle n, f, n'\rangle \in E\} \quad (3.3) \qquad E_e(v, f) = \{\langle n, f, n'\rangle \mid \langle v, n\rangle, \langle n, f, n'\rangle \in E\} \quad (3.6)$$

3.3 Reachability Annotations

This section discusses how our analysis augments a points-to graph with reachability annotations; Appendix A formalizes this abstraction. A reachability tuple $\langle n, \mu\rangle$ is a heap node and arity pair where the arity value μ is taken from the set $\{\texttt{0}, \texttt{1}, \texttt{MANY}\}$. The arity μ gives the number of objects from the heap node n that can reach the relevant

[1] Our implementation generalizes this to support abstracting the k most recently allocated objects from the allocation site with single-object heap nodes.

[2] The special field `element` abstracts all references from an array's elements.

object. The arity `0` means the object is not reachable from any objects in the given heap node, the arity `1` means the object is reachable from at most one object in the given heap node, and the arity `MANY` means the object is reachable from any number of objects in the given node. The arities have the following partial order $\mathtt{0} \sqsubseteq \mathtt{1} \sqsubseteq \mathtt{MANY}$.

A reachability state $\phi \in \Phi$ contains exactly one reachability tuple for every distinct flagged heap node. For efficiency, our implementation elides arity-`0` reachability tuples. When we write reachability states, we use brackets to enclose the reachability tuples to make them visually more clear. For example, the reachability state $\phi_n = [\langle \mathtt{n}_3, \mathtt{1}\rangle] \in \Phi_{\mathtt{n}_7}$ that appears on node $\mathtt{n}_7$ in Figure 3 indicates that it is possible for at most one object in heap node $\mathtt{n}_3$, and zero objects from any other flagged heap nodes (i.e. $\mathtt{n}_2$) to reach an object from heap node $\mathtt{n}_7$.

The function $\mathcal{A}^N : N \to \mathcal{P}(\mathcal{P}(M))$ maps a heap node n to a set of reachability states. The reachability of an object abstracted by the heap node n is abstracted by one of the reachability states given by the function $\mathcal{A}^N$. We represent $\mathcal{A}^N$ as a set of tuples of heap nodes and reachability states and define the helper function:

$$\mathcal{A}^N(n) = \{\phi \mid \langle n, \phi\rangle \in \mathcal{A}^N\}. \tag{3.7}$$

When a new reference is created, the analysis must propagate reachability information. Simply using the graph edges to do this propagation would yield imprecise results. To improve the precision of this propagation step, the analysis maintains for each edge the reachability states of all objects that can be reached from that edge. The function $\mathcal{A}^E : E \to \mathcal{P}(\mathcal{P}(M))$ maps a reference edge e to the set of reachability states of all the objects reachable from the references abstracted by e. If there exists a path of heap references $r_1; r_2; r_j$ that leads to an object o with the reachability state ϕ then for each edge e_i that abstracts a reference r_i, $\phi \in \mathcal{A}^E(e_i)$. We represent $\mathcal{A}^E$ as a set of tuples of edges and reachability states and define the helper functions:

$$\mathcal{A}^E(v) = \{\phi \mid \langle\langle v, n\rangle, \phi\rangle \in \mathcal{A}^E\}, \tag{3.8}$$

$$\mathcal{A}^{E^\circ}(v) = \{\langle\langle v, n\rangle, \phi\rangle \mid \langle\langle v, n\rangle, \phi\rangle \in \mathcal{A}^E\}, \tag{3.9}$$

$$\mathcal{A}^E(e) = \{\phi \mid \langle e, \phi\rangle \in \mathcal{A}^E\}. \tag{3.10}$$

In programs that construct unbounded data structures, heap nodes and reference edges must be summarized in the finite heap abstraction. Summarization in basic points-to analysis rapidly loses precision for reachability properties. Disjoint reachability analysis improves the precision of reachability information with $\mathcal{A}^N$ because it can express that an object is reachable from only a single object abstracted by a given summary node. The abstraction $\mathcal{A}^E$ is instrumental for precisely updating the reachability states of heap nodes because it refines path information present in the points-graph alone and allow the analysis to more precisely propagate changes to reachability states than is possible from the points-to graph alone. Another critical aspect of $\mathcal{A}^E$ is that an edge in effect selects some subset of reachability states of a node — the analysis uses this information to refine the reachability states for a node.

4 Intraprocedural Analysis

We begin by presenting the intraprocedural analysis. Section 5 extends this analysis to support method calls. The analysis is structured as a fixed-point computation.

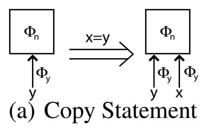
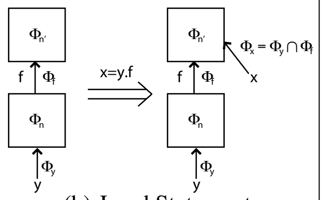
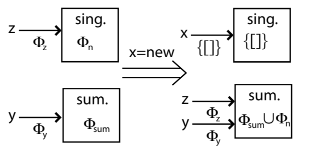
 (a) Copy Statement	$E' = (E - E_e(\mathtt{x})) \cup (\{\mathtt{x}\} \times E_n(\mathtt{y}))$ (4.1) $\mathcal{A}^{E'} = (\mathcal{A}^{E} - \mathcal{A}^{E^{\circ}}(\mathtt{x})) \cup \bigcup_{\langle v,n\rangle \in E_e(\mathtt{y})} (\{\langle \mathtt{x}, n\rangle\} \times \mathcal{A}^{E}(\mathtt{y}))$ (4.2)
 (b) Load Statement	$E' = (E - E_e(\mathtt{x})) \cup (\{\mathtt{x}\} \times E_n(\mathtt{y}, \mathtt{f}))$ (4.3) $\mathcal{A}^{E'} = (\mathcal{A}^{E} - \mathcal{A}^{E^{\circ}}(\mathtt{x})) \cup \bigcup_{\langle n,\mathtt{f},n'\rangle \in E_e(\mathtt{y},\mathtt{f})} \Big(\{\langle \mathtt{x}, n'\rangle\} \times \big(\mathcal{A}^{E}(\langle \mathtt{y}, n\rangle) \cap \mathcal{A}^{E}(\langle n, \mathtt{f}, n'\rangle)\big)\Big)$ (4.4)
 z Φz sing. Φn y Φy sum. Φsum x=new x {[]} sing. {[]} z Φz y Φy sum. Φsum∪Φn (c) Allocation Statement	$\mathcal{A}^{N'} = \{\langle \mathcal{S}_N(n, n_{\mathtt{alloc}}), \mathcal{S}_{\phi}(\phi, n_{\mathtt{alloc}})\rangle \mid \langle n, \phi\rangle \in \mathcal{A}^{N}\} \cup \{\langle n_{\mathtt{alloc}}, \phi_{\mathtt{alloc}}\rangle\}$ (4.5) $\mathcal{A}^{E'} = \{\langle \mathcal{S}_E(e, n_{\mathtt{alloc}}), \mathcal{S}_{\phi}(\phi, n_{\mathtt{alloc}})\rangle \mid \langle e, \phi\rangle \in \mathcal{A}^{E}\} \cup \{\langle\langle \mathtt{x}, n_{\mathtt{alloc}}\rangle, \phi_{\mathtt{alloc}}\rangle\}$ (4.6) $E' = \{\mathcal{S}_E(e, n_{\mathtt{alloc}}) \mid e \in E\} \cup \{\langle \mathtt{x}, n_{\mathtt{alloc}}\rangle\}$ (4.7) $\phi_{\mathtt{alloc}} = \begin{cases} [\langle n_{\mathtt{alloc}}, \mathtt{1}\rangle] & \text{if } n_{\mathtt{alloc}} \text{ is flagged} \\ [] & \text{otherwise} \end{cases}$ (4.8)

Fig. 4. Transfer Functions for Copy, Load, and Allocation Statements

4.1 Method Entry

The method entry transfer function creates an initial reachability graph to abstract the part of the calling methods' heaps that are reachable from parameters. In the example, the initial reachability graph is empty because the method `graphLoop` does not take parameters. Method context generation is explained in detail in Section 5.1.

4.2 Copy Statement

A copy statement of the form `x=y` makes the variable `x` point to the object that `y` references. The analysis always performs strong updates for variables — it discards all the reference edges from variable `x` and then copies all the edges along with their reachability states from variable `y`. Equation 4.1 and Equation 4.2 describe the transformations.

4.3 Load Statement

Load statements of the form `x=y.f` make the variable `x` point to the object that `y.f` references. Existing reference edges for the field are copied to `x` as described by Equation 4.3. Note that this statement does not change the reachability of any object. The reachability on new edges from `x`, as described by Equation 4.4, is the intersection of $\mathcal{A}^{E}(\langle \mathtt{y}, n\rangle)$ and $\mathcal{A}^{E}(\langle n, \mathtt{f}, n'\rangle)$, because `x` can only reach objects that were reachable from both the variable `y` and a heap reference abstracted by the edge $\langle n, \mathtt{f}, n'\rangle$.

4.4 Object Allocation Statement

The analysis abstracts the most recently allocated object from an allocation site as a single-object heap node. A summary node for the allocation site abstracts any objects from the allocation site that are older than the most recent.

The object allocation transfer function merges the single-object node n_{alloc} into the site's summary node. The single-object node n_{alloc} is then the target of a variable assignment. Equations 4.5 through 4.7 describe the basic transformation. We define the helper function $\mathcal{S}_N(n, n_{\text{alloc}})$ to return the corresponding summary node for n_{alloc} if $n = n_{\text{alloc}}$ and n otherwise. We define $\mathcal{S}_E(\langle v, n\rangle, n_{\text{alloc}}) = \langle v, \mathcal{S}_N(n_{\text{alloc}})\rangle$ and $\mathcal{S}_E(\langle n, f, n'\rangle, n_{\text{alloc}}) = \langle \mathcal{S}_N(n, n_{\text{alloc}}), f, \mathcal{S}_N(n', n_{\text{alloc}})\rangle$.

As stated, the single-object node and its reachability information merge with the summary node. We define the helper function $\mathcal{S}_\phi(\phi, n_{\text{alloc}})$ to update a reachability state by rewriting all reachability tuples with n_{alloc} to use the summary node. If both the single-object node and the summary node appeared in the same reachability state before this transform, after rewriting the tuples there will be, conceptually, two summary node tuples in the state. In this case the tuples are combined and the new arity for the summary heap node tuple is computed by $+_\triangle$, which is addition in the domain $\{\mathtt{0}, \mathtt{1}, \mathtt{MANY}\}$. Note that the reachability annotations enable the analysis to maintain precise reachability information over summarizations.

Finally, if the heap node is flagged, the analysis generates the set of reachability states $\{[\langle n_f, \mathtt{1}\rangle]\}$, where n_f is the given heap node, for the new object's node and edge. Otherwise, it generates the set $\{[\,]\}$ with the empty state for the node and edge.

4.5 Store Statement

Store statements of the form `x.f=y` point the `f` field of the object referenced by `x` to the object referenced by `y`. Store statements can change the reachability of objects reachable from `y`. Equation 4.9 describes how a store changes the edge set.

$$E' = E \cup (E_n(\mathtt{x}) \times \{\mathtt{f}\} \times E_n(\mathtt{y})) \tag{4.9}$$

Let $o_{\mathtt{x}}$ be the object referenced by `x` in the concrete heap and $o_{\mathtt{y}}$ be the object referenced by `y`. The new edge from the object $o_{\mathtt{x}}$ to the object $o_{\mathtt{y}}$ can only add new paths from objects that could previously reach $o_{\mathtt{x}}$ to objects that were reachable from $o_{\mathtt{y}}$. In the reachability graph, the heap nodes $n_{\mathtt{x}} \in E_n(\mathtt{x})$ abstract $o_{\mathtt{x}}$ and the heap nodes $n_{\mathtt{y}} \in E_n(\mathtt{y})$ abstract $o_{\mathtt{y}}$. The set of flagged heap nodes containing objects that could potentially reach $o_{\mathtt{x}}$ is given by the set of reachability states:

$$\Psi_{\mathtt{x}} = \mathcal{A}^N(n_{\mathtt{x}}) \cap \mathcal{A}^E(\langle \mathtt{x}, n_{\mathtt{x}}\rangle). \tag{4.10}$$

The reachability states of the objects reachable from $o_{\mathtt{y}}$ is

$$\Psi_{\mathtt{y}} = \mathcal{A}^E(\langle \mathtt{y}, n_{\mathtt{y}}\rangle). \tag{4.11}$$

We define $\cup_\triangle$ to compute the union of two reachability states. When two reachability states are combined, tuples with matching heap nodes merge arity values according to $+_\triangle$. We divide updating the reachability graph into the following steps:

1. **Construct the New Graph:** The analysis first constructs the new edge set as described by Equation 4.9.
2. **Update Reachability States of Downstream Heap Nodes:** The reachability of every object o' reachable from $o_{\mathtt{y}}$ is (i) abstracted by some $\psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}$ and (ii) there exist a path of edges from the heap node that abstracts $o_{\mathtt{y}}$ to the heap node that abstracts o' in which each edge has $\psi_{\mathtt{y}}$ in its reachability state. The newly created edge can make the object o' reachable from the objects that can reach $o_{\mathtt{x}}$ — this set of objects is abstracted by some reachability state $\psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}$. Therefore the new reachability state for o' should be $\psi_{\mathtt{y}} \cup_\triangle \psi_{\mathtt{x}}$. We capture this reachability change with the *change tuple*

set $C_{n_{\mathtt{y}}} = \{\langle \psi_{\mathtt{y}}, \psi_{\mathtt{y}} \cup_{\triangle} \psi_{\mathtt{x}} \rangle \mid \psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}, \psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}\}$. Constraints 4.12 and 4.13 express the path constraint (ii). The analysis uses a fixed point to solve these constraints and then uses Equation 4.14 to update the reachability states of downstream nodes.

$$\Lambda^{\text{node}}(n_{\mathtt{y}}) \supseteq C_{n_{\mathtt{y}}} \tag{4.12}$$

$$\Lambda^{\text{node}}(n') \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(n), \langle n, f, n' \rangle \in E, \phi \in \mathcal{A}^E(\langle n, f, n' \rangle)\} \tag{4.13}$$

$$\begin{aligned}\mathcal{A}^{N'}(n) = \{\phi' \mid \phi \in \mathcal{A}^N(n), \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(n)\} \cup \\ \{\phi \mid \phi \in \mathcal{A}^N(n), \nexists \phi'. \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(n)\}\end{aligned} \tag{4.14}$$

3. **Propagate Reachability from Downstream Nodes to Edges:** The analysis must propagate the reachability changes of objects back to any edge that abstracts a reference that can reach the object. Constraint 4.15 ensures that edges contain reachability change tuples that capture the reachability changes in the incident objects. Constraint 4.16 ensures that the change set contains tuples to re-establish the transitive reachability state property.

$$\Lambda^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(\text{dst}(e)), \phi \in \mathcal{A}^N(\text{dst}(e)), \phi \in \mathcal{A}^E(e)\} \tag{4.15}$$

$$\Lambda^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{edge}}(e'), \phi \in \mathcal{A}^E(e), \text{dst}(e) = \text{src}(e')\} \tag{4.16}$$

4. **Propagate Reachability Changes Upstream of $o_{\mathtt{x}}$:** The reachability states of edges that abstract references that can reach $o_{\mathtt{x}}$ must be updated to reflect the objects they can now reach through the newly created edge. We define the change tuple set $C_{n_{\mathtt{x}}} = \{\langle \psi_{\mathtt{x}}, \psi_{\mathtt{y}} \cup_{\triangle} \psi_{\mathtt{x}} \rangle \mid \psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}, \psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}\}$ that updates the reachability states of edges that can reach o_x. Constraint 4.17 ensures that edges incident to the heap nodes that abstract $o_{\mathtt{x}}$ contain reachability change tuples that capture the reachability states of the newly reachable objects. Constraint 4.18 ensures that the change set contains tuples to re-establish the transitive reachability state property.

$$\Upsilon^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in C_{n_{\mathtt{x}}}, \phi \in \mathcal{A}^E(e), \text{dst}(e) = n_{\mathtt{x}}\} \tag{4.17}$$

$$\Upsilon^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Upsilon^{\text{edge}}(e'), \phi \in \mathcal{A}^E(e), \text{dst}(e) = \text{src}(e')\} \tag{4.18}$$

5. **Update Edge Reachability:** Finally, the analysis generates the reachability states for the edges in the new graph. Equation 4.19 computes the reachability states of all edges that existed before the store operation using the change tuple sets. Equation 4.20 computes the reachability for the newly created edges from the reachability of the edges for `y` with the constraint that every reachability state on the edge must be at least as large as the reachability state for the object $o_{\mathtt{x}}$. We define $\phi \subseteq_{\triangle} \phi'$ if $\forall \langle n, \mu \rangle \in \phi$ there exists a reachability tuple $\langle n, \mu' \rangle \in \phi'$ such that $\mu \sqsubseteq \mu'$.

$$\begin{aligned}\mathcal{A}^{E'}(e) = \mathcal{A}^E(e) \cup \{\phi' \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{edge}}(e), \phi \in \mathcal{A}^E(e)\} \cup \\ \{\phi' \mid \langle \phi, \phi' \rangle \in \Upsilon^{\text{edge}}(e), \phi \in \mathcal{A}^E(e)\}\end{aligned} \tag{4.19}$$

$$\mathcal{A}^{E'}(\langle n_{\mathtt{x}}, \mathtt{f}, n_{\mathtt{y}} \rangle) \subseteq \{\phi \in \mathcal{A}^{E'}(\langle \mathtt{y}, n_{\mathtt{y}} \rangle) \mid \exists \phi' \in \mathcal{A}^{N'}(n_{\mathtt{x}}), \phi' \subseteq_{\triangle} \phi\} \tag{4.20}$$

Strong Updates While in general the analysis performs *weak updates* that simply add edges, under certain circumstances the analysis can perform *strong updates* that also remove edges to increase the precision of the results. Strong updates are possible under either of two conditions. First, when variable x is the only reference to a heap node $n_{\mathtt{x}}$. In this case we can destroy all reference edges from $n_{\mathtt{x}}$ with field f because no other variables can reach $n_{\mathtt{x}}$. Second, when the variable x references exactly one heap node $n_{\mathtt{x}}$ and $n_{\mathtt{x}}$ is a single-object heap node. When this is true x definitely refers to the object in $n_{\mathtt{x}}$ and the existing edges with field f from $n_{\mathtt{x}}$ can be removed.

For strong updates, the analysis first removes edges that the strong update eliminates. It then performs the normal transform as described in this section. Note that when strong updates remove edges, reachability of graph elements may change if the removed edges provided the reachability path. Therefore, reachability states may become imprecise. After a store transform with a strong update occurs, a global pruning step improves imprecise reachability states. Section 4.9 presents the global pruning step.

4.6 Element Load and Store Statements

Our analysis implements the standard pointer analysis treatment of arrays: Array elements are treated as a special field of array objects and always have weak store semantics. The analysis does not differentiate between different indices. This treatment can cause imprecision for operations such as vector removes that move a reference from one array element to another. Our implementation uses a special analysis to identify array store operations that acquire an object reference from an array and then create a reference from a different element of that array to the same object. Because the graph already accounts for this reachability, the effects of such stores can be safely omitted.

4.7 Return Statement

Return statements are of the form `return x` and return the object referenced by x. Each reachability graph has a special `Return` variable that is out of program scope. At a method return the transfer function assigns the `Return` variable to the references of variable x. We assume without loss of generality that the control flow graph has been modified to merge the control flow for all return statements.

4.8 Control Flow Join Points

To analyze a statement, the analysis first computes the join of the incoming reachability graphs. The operation for merging reachability graphs r_0 and r_1 into r_{out} follows below:

1. The set of variables for r_{out} is the set of live variables into the statement.

2. The set of heap nodes for r_{out} is the union of the heap nodes in the input graphs. The union of the reachability states is taken, $\mathcal{A}^N_{\text{out}}(n) = \mathcal{A}^N_0(n) \cup \mathcal{A}^N_1(n)$.

3. The set of reference edges for r_{out} is the union of the reference edges of the input graphs. Reference edges are unique in a reachability graph with respect to source, field, and destination. For a reference edge e, $\mathcal{A}^E_{\text{out}}(e) = \mathcal{A}^E_0(e) \cup \mathcal{A}^E_1(e)$.

4.9 Global Pruning

When strong updates remove edges, the reachability states may become imprecise. The call site transfer function in Section 5 can also introduce imprecise reachability states. Our analysis includes a global pruning step that uses global reachability constraints to prune imprecise reachability states to improve the precision of the analysis results.

The intuition behind global pruning is that multiple abstract states can correspond to the same set of concrete heaps, and the global pruning step generates an equivalent abstraction that locally has more precise reachability states.

Global Reachability Constraints Reachability information must satisfy two reachability constraints that follow from the discussion in Section 3.3.

- **Node reachability constraint:** For each node n, $\forall \phi \in \mathcal{A}^N(n)$, $\forall \langle n', \mu \rangle \in \phi$, if $\mu \in \{\mathtt{1}, \mathtt{MANY}\}$ then there must exist a set of edges $e_1, \ldots, e_m$ such that $\phi \in \mathcal{A}^E(e_i)$ for all $1 \leq i \leq m$ and the set of edges $e_1, \ldots, e_m$ form a path through the reachability graph from n' to n.
- **Edge reachability constraint:** For each edge e, $\forall \phi \in \mathcal{A}^E(e)$ there exists $n \in N$ and $e_1, \ldots, e_m \in E$ such that $\phi \in \mathcal{A}^N(n)$; $\phi \in \mathcal{A}^E(e_i)$ for all $1 \leq i \leq m$; and the set of edges $e_1, \ldots, e_m$ form a path through the reachability graph from e to n.

The first phase of the algorithm generates a reachability graph with the most precise set of reachability states for the nodes. The second phase of the algorithm generates the most precise set of reachability states for the edges.

1. Improve the precision of the node reachability states: The algorithm first uses the node reachability constraint to prune the reachability states of nodes. This phase uses the existing $\mathcal{A}^E$ to prune reachability tuples from imprecise reachability states to generate a more precise $\mathcal{A}^{N'}$ from the previous $\mathcal{A}^N$. The algorithm iterates through each flagged node n_f. The function $\mathcal{A}^{\mathcal{E}}_f(e)$ maps the edge $e \in E$ to the set of reachability states Φ for which each $\phi \in \Phi$ (1) includes a non-zero arity reachability tuple with the node n_f and (2) there exist a path from n_f to e for which every edge along the path contains ϕ in its set of reachability states. We compute $\mathcal{A}^{\mathcal{E}}_f$ using a fixed-point algorithm on the following two constraints:

$$\forall e \in E_e(n_f), \mathcal{A}^{\mathcal{E}}_f(e) \supseteq \mathcal{A}^E(e), \tag{4.21}$$

$$\forall e \in E, e' \in E_e(\mathrm{dst}(e)), \mathcal{A}^{\mathcal{E}}_f(e') \supseteq \mathcal{A}^E(e') \cap \mathcal{A}^{\mathcal{E}}_f(e). \tag{4.22}$$

For each node n and each reachability state $\phi \in \mathcal{A}^N(n)$ the analysis shortens ϕ to remove tuples n_f or n_f^* to generate a new reachability state ϕ' if ϕ does not appear in $\mathcal{A}^{\mathcal{E}}_f(e)$ of any edge e incident to n. This step does not prune n_f or n_f^* from the reachability states of flagged nodes n_f. The analysis then propagates these changes to $\mathcal{A}^E$ of the upstream edges using the same propagation procedure described by Equations 4.15 and 4.16 to generate $\mathcal{A}^E_r$.

2. Improve the precision of the edge reachability states: The algorithm next uses the pruned node reachability states in $\mathcal{A}^{N'}$ and $\mathcal{A}^E_r$ to generate a more precise $\mathcal{A}^{E'}$. The intuition is that an edge can only have a given reachability state if there exists a path from that edge to a node with that reachability state such that all edges along the path contain the reachability state. The analysis starts from every heap node n and propagates the reachability states of $\mathcal{A}^N(n)$ backwards over reference edges. The analysis initializes $\mathcal{A}^{E'} = \{\mathcal{A}^E_r(e) \cap \mathcal{A}^{N'}(n) \mid \forall e \in E, n = \mathrm{dst}(e)\}$. The analysis then propagates reachability information backwards to satisfy the constraint: $\mathcal{A}^{E'}(e) \supseteq \mathcal{A}^E_r(e) \cap \mathcal{A}^{E'}(e')$ for all $e' \in E_e(\mathrm{dst}(e))$. The propagation continues until a fixed-point is reached.

4.10 Static Fields

We have omitted analysis of static fields or globals. We assume that the preprocessing stage creates a special global object that contains all of the static fields and passes it to every call site. Through this semantics-preserving program transformation, static field store and load statements become normal store and load statements, respectively.

5 Interprocedural Analysis

The interprocedural analysis adds a call site transfer function to the intraprocedural analysis. It uses a standard fixed-point algorithm that accomodates recursive call chains and begins by analyzing the `main` method. Our analysis processes each method using one context that summarizes the heaps for all call sites.

We expose only the callee reachable portion of the caller's heap when analyzing a callee method, similar to previous work [17, 18] but with extensions to accommodate reachability properties. A summary of the transform follows:

i) Compute the portion of the heap that is reachable from the callee.
ii) Rewrite reachability states to abstract flagged heap nodes that are not in the callee heap with special out-of-context heap nodes.
iii) Merge this portion of the heap with the callee's current initial graph. If the graph changes, schedule the callee for reanalysis.
iv) Specialize the callee's current analysis result using the caller context.
v) Replace the callee reachable portion of the caller's heap with the specialized callee results.
vi) Merge nodes such that each allocation site has at most one summary heap node and one single object heap node.
vii) Call the global pruning step introduced in Section 4.9 to improve the precision of the caller reachability graph.

5.1 Compute Callee Context Subgraph

For each call site, the analysis computes the subgraph $G_{\text{sub}} \subseteq G$ that is reachable from the call site's arguments. For each incoming edge $\langle n, f, n' \rangle \in E$ into G_{sub} where $n \notin G_{\text{sub}}$ and $n' \in G_{\text{sub}}$, the analysis generates a new *placeholder node* n_p and a new edge $e' = \langle n_p, \mathcal{R}(n') \rangle$ where $\mathcal{A}^E(e') = \mathcal{A}^E(e)$. The placeholder node n_p serves as a proxy flagged node for all reachability nodes in $\mathcal{A}^N(n)$ during global pruning in the callee context. For each incoming edge $\langle v, n' \rangle \in E$ into G_{sub} where $n' \in G_{\text{sub}}$, the analysis generates a new placeholder variable v_p and *placeholder edge* $e_p = \langle v_p, n' \rangle$ where $\mathcal{A}^E(e_p) = \mathcal{A}^E(e)$.

5.2 Out-of-Context Reachability

Summarization presents a problem for any out-of-context flagged heap node $n_f \notin G_{\text{sub}}$ that appears in reachability states of an in-context heap node $n \in G_{\text{sub}}$. The interprocedural analysis uses placeholder flagged nodes to rewrite out-of-context flagged heap nodes in reachability states. Each heap node n_f that appears in $\mathcal{A}^N$ of a placeholder node is (1) outside of the graph G_{sub} and (2) abstracts objects that can potentially reach objects abstracted by the subgraph G_{sub}. The analysis replaces all such nodes in all in-context reachability states with special out-of-context heap nodes for the allocation site. There can be up to two out-of-context heap nodes per an allocation site: one is a summary node and one abstracts the most recently allocated object from the allocation site.

The purpose of these heap nodes is to allow the analysis of the callee context to summarize in-context, single-object heap nodes without affecting out-of-context flagged heap nodes that can reach objects in the callee's reachability graph.

The analysis maps (1) the newest single-object heap node for an allocation site that is out of the callee's context to the special single-object out-of-context heap node and (2) all other nodes for the allocation site that are out of the callee's context for the heap node to the special summary out-of-context heap node. The analysis stores this mapping for use in the splicing step. These special out-of-context nodes serve as placeholders to track changes to the reachability of out-of-context edges.

5.3 Merge Graphs

The analysis merges the subgraphs from all calling contexts using the join operation from Section 4.8 to generate G_{merge}. The analysis of the callee operates on G_{merge}.

5.4 Predicates

The interprocedural analysis extends all nodes, edges, and reachability states in G_{merge} with a set of predicates. These predicates are included to prevent nodes and edges from escaping to the wrong call site and are used to correctly propagate reachability states in the caller. Predicates are comprised of the following atomic predicates, which can be combined with logical disjunction (or):

- Edge e exists with reachability state ϕ in G_{sub} of the caller
- Node n exists with reachability state ϕ in G_{sub} of the caller
- Edge e exists in G_{sub} of caller
- Node n exists in G_{sub} of caller
- *true*

The caller analysis begins by initializing the predicates for all nodes, edges, and reachability states to tautologies. For example, the initial predicate for a node n is that the node n exists in the caller — this prevents node n from escaping to the wrong call site. The initial predicate for a reachability state ϕ on node n is that node n exists in G_{sub} of the caller with reachability state ϕ.

Store operations can change the reachability states of both edges and heap nodes. When the propagation of a change set creates a new reachability state on a node or an edge, the new state inherits the predicate from the previous state on the node or edge, respectively. Object allocation operations can merge single-object heap nodes into the corresponding summary node. In this case, predicates for the nodes are or'ed together. Likewise, if the operation causes two edges to be merged, their predicates are also or'ed together. Duplicated reachability states may also be merged and their predicates are or'ed together.

Newly created nodes or edges are assigned the *true* predicate.

5.5 Specializing the Graph

The algorithm uses G_{sub} to specialize the callee heap reachability graph G_{callee}. The analysis makes a copy of the heap reachability graph G_{callee}. It then prunes all elements of the graph whose predicates are not satisfied by the caller subgraph G_{sub}. The callee predicates of each heap element in G_{callee} are replaced with the caller predicate for the heap element in G_{sub} that satisfied the callee predicate.

If a reachability state contains out-of-context heap nodes, then the analysis uses the stored mapping to translate the out-of-context heap nodes to caller heap nodes. The stored mapping may map multiple heap nodes to the same out-of-context summary heap node. If the arity of the reachability tuple for an out-of-context heap node was `1`, then the analysis generates all permutations of the reachability state using the stored mapping from Section 5.2. If the arity was `MANY`, the analysis replaces the reachability tuple with a set of reachability tuples that contains one tuple for each heap node that mapped to the out-of-context summary node and that tuple has arity `MANY`.

5.6 Splice in Subgraph

This step splices the physical graphs together. The placeholder nodes are used to splice references from the caller graph to the callee graph. The placeholder edges are used to splice caller edges into the callee graph.

Finally, the reachability changes are propagated back into the out-of-context heap nodes of the caller reachable portion of the reachability graph. The analysis uses predicates to match the reachability states on the original edges from the out-of-context portion of the caller graph into G_{sub}. The analysis generates a change set for each edge that tracks the out-of-context reachability changes made by the callee. It then solves constraints of the same form as Constraints 4.15 and 4.16 to propagate these changes to upstream portions of the caller graph.

5.7 Merging Heap Nodes

At this point, the graph may have more than one single object heap node or summary heap node for a given allocation site. The algorithm next merges all but the newest single object heap node into the summary heap node. It rewrites all tokens in all reachability states to reflect this merge, and then updates the arities.

5.8 Global Pruning

Finally, the analysis calls the global pruning algorithm to remove imprecision potentially caused by our treatment of reachability from out-of-context heap nodes.

6 Evaluation

We have implemented the analysis in our compiler and analyzed eight OoOJava benchmarks [14]. In OoOJava the developer annotates code blocks as tasks that the compiler may decouple from the parent thread. OoOJava uses disjointness analysis combined with other analyses to generate a set of runtime dependence checks that parallelize the execution only when it can preserve the behavior of the original sequential code — annotations do not affect the correctness of the program. The analysis and benchmarks are available at `http://demsky.eecs.uci.edu/compiler.php`.

6.1 Benchmarks

We analyzed and parallelized the following eight benchmarks. From Java Grande [19] we ported all of the large benchmarks: RayTracer, a ray tracer; MonteCarlo, a Monte Carlo simulation; and MolDyn, a molecular dynamics simulation. We also ported Crypt, an IDEA encryption algorithm, from Java Grande. We ported KMeans, a data clustering benchmark and Labyrinth, a maze-routing benchmark, from STAMP [20]. Power is a power pricing benchmark from JOlden [21] and Tracking is a vision benchmark from the SDVBS [22]. The benchmarks range from 5,669 to 1,846 lines of code, with an average of 96.3 methods per benchmark.

Benchmark	Time (s)	Lines	Speedup
Crypt	1.2	2,035	17.7×
KMeans	3.5	3,220	13.8×
Labyrinth	84.8	4,315	11.1×
MolDyn	13.8	2,299	13.8×
MonteCarlo	4.8	5,669	18.7×
Power	6.2	1,846	20.2×
RayTracer	22.1	2,832	20.0×
Tracking	331.2	4,747	20.2×

Fig. 5. Out-of-order Java Results (24 cores)

6.2 Disjoint Reachability Analysis Results

We next examine the reachability properties that the analysis extracted for our benchmarks. We expect developers might examine the results for particular program points to learn about possible sharing. For instance, disjoint reachability analysis reported that iterations of the main loop in RayTracer reached a common scratchpad object which prevented OoOJava from parallelizing the loop. The information allowed the authors to move the scratchpad object into the loop scope to obtain a parallel implementation.

Section 4.6 states that disjoint reachability analysis does not differentiate between array indices, however, it is common programming practice to store a parallelizable workload in an array. Disjoint reachability analysis can provide sufficient precision for parallelization in such a case by examining the disjoint reachability properties of an object just after it is selected from an array, as shown in the example presented in Section 2. We ported the array-based benchmarks using this simple pattern.

Labyrinth allocates `Grid` data structures in a way that makes it difficult for many heap analyses to determine the inner loop for finding routes may be parallelized. The main complication is that an array of `Grid` data structures is allocated in a separate method and then `Grid` data structures are reused in each iteration of the inner loop for calculating a route. OoOJava flags the allocation of `Grid` root objects; disjoint reachability analysis then determines that, at the work division program point, the objects that comprise a `Grid` data structure are only reachable from exactly one `Grid` data structure root object. Determining statically that different tasks access unique `Grid` objects is likely to be challenging for any static analysis and therefore purely static approaches are likely to fail. Instead, OoOJava generates a dynamic check that each task has a unique `Grid` data structure root object that, combined with the static reachability information, guarantees that there are no conflicts on the `Grid` data structures.

Power calculates the price of power in a simulated network; each major branch of the network is modeled with a `Lateral` object that references a tree of `Branch` and `Leaf` objects. Disjoint reachability analysis discovers that all `Demand` objects which are written to during the simulation are reachable from at most one `Lateral` object. This information allows OoOJava to parallelize the task by verifying the tasks operate on different `Lateral` objects.

MolDyn allocates a collection of scratchpad data structures and gives one scratchpad data structure to each parallel task. In the main loop a second task aggregates the results by reading all scratchpad objects. OoOJava flags the scratchpad root object; disjoint reachability analysis concludes that all objects in the scratchpad data structures accessed by a parallel task are reachable from at most one scratchpad root object.

OoOJava therefore parallelized the main computation using a dynamic check on the scratchpad data structures and serializes the aggregation steps.

The benchmarks Tracking, KMeans, Crypt, and Monte use different data structures but have a common work division pattern: the main loop consists of a parallel phase followed by an aggregation phase. The results of disjoint reachability analysis for each of these benchmarks correctly informed OoOJava that the aggregation phases must be serialized because they have actual data structure conflicts.

6.3 Parallelization Speedups

We used OoOJava to analyze and parallelize eight benchmarks on a 2.27 GHz Xeon. OoOJava makes queries to disjoint reachability analysis when generating a parallel implementation. We then executed our benchmark applications on a 1.9 GHz 24-core AMD Magny-Cour Opteron with 24 cores and 16 GB of memory. Figure 5 presents the speedups and the time column shows the analysis time. The speedups are relative to the single-threaded Java versions compiled to C using the same compiler.

The significant speedups indicate that disjoint reachability analysis extracts reachability properties with sufficient precision for OoOJava to generate efficient parallel implementations. The speedups in Crypt, KMeans, Labyrinth and MolDyn are limited by significant sequential dependences. RayTracer's and MonteCarlo's speedups were impacted by task dispatch overheads. We compared our parallel implementations of KMeans and Labyrinth to the parallelized TL2 versions included in STAMP, with and without the additional overheads from array bounds checks in our compiler. With array bounds checks our versions of KMeans and Labyrinth ran 1.70× and 1.51× faster than TL2 versions, respectively, and without the checks they ran 2.62× and 2.08× faster.

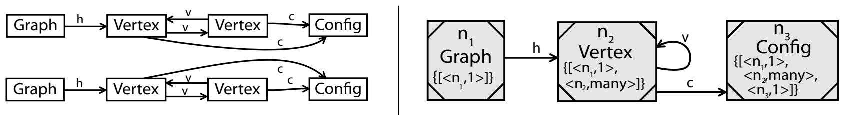
To quantify the overhead of our research compiler, we compared the generated code against the OpenJDK JVM 14.0-b16 and GCC 4.1.2. The sequential version of Crypt compiled with our compiler ran 4.6% faster than on the JVM. We also developed a C++ version compiled with GCC and found our compiler's version ran 25% slower than the C++ version. Our compiler implements array bounds checking; with array bounds checking disabled, the binary from our compiler runs only 5.4% slower than the C++ binary. We used the optimization flag `-O3` for both the C++ version and the C code generated by our compiler. This is in close agreement with more extensive experiments we performed. Those experiments measured an average overhead for our compiler with array bounds checks disabled of 4.9% relative to GCC.

7 Related Work

We discuss related work in heap analyses, logics, and type systems.

7.1 Shape Analysis

Disjoint reachability analysis discovers properties that are related to but different from those discovered by shape analysis [3–7]. Shape analysis, in general, discovers local properties of heap references and from these properties infers a rich set of derived properties including reachability and disjointness. Where shape analysis can find properties that arise from local invariants, disjoint reachability analysis can find the relative disjointness and reachability properties for any pair of objects. Disjoint reachability analysis complements shape analysis by discovering disjoint reachability properties for



(a) Concrete heap (b) Reachability graph

Fig. 6. An example concrete heap with a reachability graph

arbitrarily complex structures. Calcagno et al. present a shape analysis that focuses on discovering different heap properties [23].

We motivate our discussion of shape analysis with a concrete heap example. Figure 6(a) illustrates a simple concrete heap where a `Graph` can reach several `Vertex` objects that all point to a graph-local `Config` object. We expect that many real programs construct data structures with sharing patterns similar to this example. A possible reachability graph in Figure 6(b) contains enough information to show that `Config` and `Vertex` objects are reachable from at most one `Graph` object. Some shape analyses [3, 4] focus on local shape properties (is every node in a tree a valid tree node?) and understandably lose precision with the above example or the singleton design pattern. Singleton design patterns include references to globally shared objects. Some parallelizable phases may not even access the shared object, but the presence of a shared object will cause problems for many shape analyses. Our analysis can infer that operations on different graphs that access both `Vertex` and `Config` objects may execute in parallel. Note that this result is independent of the relative shape of Vertex objects in the heap.

Marron et al. extend the shape approach for more general heaps with edge-sharing analysis [7, 24]. Their analysis can discover that the `Vertex` objects from different `Graph` objects are disjoint. However, their edge-sharing abstraction is localized and thus cannot always resolve non-local reachability properties.

TVLA [5] is a framework for developing shape analysis. Disjointness properties can be written as instrumentation predicates in TVLA, but the system will evaluate them using the default update rule, providing acceptable results only for trivial examples. To maintain precision, update rules for the disjointness predicates must be supplied, a task that we expect is equivalent in difficulty to disjoint reachability analysis. While TVLA contains reachability predicate update rules, these cannot capture that an object is reachable from exactly one member of a summarized node. Furthermore, the TVLA framework does not scale to the size of our benchmarks.

Separation logic [12] can express that formulas hold on disjoint heap regions. Distefano [25] proposes a shape analysis for linked lists based on separation logic. Raza [26] extends separation logic with assertions to identify statements that can be parallelized. These shape analyses based on separation logic are at an early stage and cannot extract disjoint reachability properties for our examples.

7.2 Alias and Pointer Analysis

Alias analysis [8–10] and pointer analysis [1, 2], like disjoint reachability analysis, analyze source code to discover heap referencing properties. Aiken [11] is similar, but their type system names objects by allocation site and loop iteration. Unlike our analysis, their approach cannot maintain disjointness properties for mutation outside of the allocating loop. Lattner [27] employs a unification-based pointer analysis that scales well, but cannot maintain disjointness properties for data structures that are merged at a later program point. Conditional must not aliasing analysis by Naik and Chatterjee et al. describe a modular points-to analysis that does not extract disjoint reachability properties, but introduces an alternative approach to abstracting caller contexts [28].

7.3 Other Analyses and Type Systems

Sharing analysis [29] computes sharing between variables. Sharing analysis could not determine disjoint reachability properties for the example in Figure 1 of our paper as it would lose information about the relative disjointness of graphs in the array.

Connection analysis discovers which heap-directed pointers may reach a common data structure [30]. There are a finite number of pointers in a program which implies that connection analysis can only maintain a finite number of disjoint relations. For example, connection analysis cannot determine that all of the `Graph` objects in our paper's example reference mutually disjoint sets of `Vertex` objects.

Ownership type systems have been developed to restrict aliasing of heap data structures [31, 32]. Our analysis infers similar properties without requiring annotations.

Craik and Kelly use the property of ownership among objects [33] to discover disjoint ownership contexts, similar to disjoint reachability properties. They do not attempt to track side effects so their analysis, in comparison to disjoint reachability analysis, can scale better but has significantly less precision.

8 Conclusion

If a compiler can determine that code blocks perform memory accesses that do not conflict, it can safely parallelize them. Traditional pointer analyses have difficulty reasoning about reachability from objects that are abstracted by the same node. We present disjoint reachability analysis, a new analysis for extracting reachability properties from code. The analysis uses a reachability abstraction to maintain precise reachability information even for multiple objects from the same allocation site. We have implemented the analysis and analyzed eight benchmark programs. The analysis results enabled parallelization of these benchmarks that achieved significant performance improvements.

Acknowledgements

This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We thank Mark Marron and the reviewers for feedback.

References

1. Shapiro, M., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: POPL. (1997)
2. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: PLDI. (1993)
3. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI. (1990)
4. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: POPL. (1996)

5. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS (2002)
6. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: CAV. (2005)
7. Marron, M., Kapur, D., Hermenegildo, M.: Identification of logically related heap regions. In: ISMM. (2009)
8. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: PLDI. (1998)
9. Ruf, E.: Partitioning dataflow analyses using types. In: POPL. (1997)
10. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: PLDI. (1994)
11. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL. (2007)
12. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. LICS (2002)
13. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: POPL. (1993)
14. Jenista, J.C., Eom, Y., Demsky, B.: OoOJava: An out-of-order approach to parallel programming. In: HotPar. (2010)
15. Jenista, J.C., Eom, Y., Demsky, B.: OoOJava: Software out-of-order execution. In: PPoPP. (2011)
16. Zhou, J., Demsky, B.: Bamboo: A data-centric, object-oriented approach to multi-core software. In: PLDI. (June 2010)
17. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL. (2005)
18. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient context-sensitive shape analysis with graph based heap models. In: CC. (2008)
19. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel Java Grande benchmark suite. In: SC. (2001)
20. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC. (2008)
21. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in Java. In: PACT. (2001)
22. Venkata, S.K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., Taylor, M.B.: SD-VBS: The San Diego Vision Benchmark Suite. In: IISWC. (2009)
23. Calcagno, C., Distefano, D., OHearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. (2009)
24. Marron, M., Méndez-Lojo, M., Hermenegildo, M., Stefanovic, D., Kapur, D.: Sharing analysis of arrays, collections, and recursive structures. In: PASTE. (2008)
25. Distefano, D., OHearn, P.W., Yang, H.: A local shape analysis based on separation logic. LNCS (2006)
26. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: ESOP. (2009)
27. Lattner, C., Adve, V.: Automatic pool allocation: improving performance by controlling data structure layout in the heap. In: PLDI. (2005)
28. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL. (1999)
29. Méndez-Lojo, M., Hermenegildo, M.V.: Precise set sharing analysis for Java-style programs. In: VMCAI. (2008)
30. Ghiya, R., Hendren, L.J.: Connection analysis: A practical interprocedural heap analysis for C. IJPP (1996)
31. Clarke, D.G., Drossopoulou, S.: Ownership, Encapsulation and the Disjointness of Type and Effect. In: OOPSLA. (2002)

32. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: PLDI. (2003)
33. Craik, A., Kelly, W.: Using ownership to reason about inherent parallelism in object-oriented programs. In: CC. (2010)

Appendix

A Semantics for Intraprocedural Analysis

Define the concrete heap $H = \langle O, R\rangle$ as a set of objects $o \in O$ and a set of references $r \in R \subseteq O \times \{\texttt{Fields}\} \times O$. We assume a straightforward collecting semantics for the statements in the control flow graph that are relevant to our analysis. The collecting semantics would record the set of concrete heaps that a given statement operates on.

The concrete domain for the abstraction function is a set of concrete heaps $h \in \mathcal{P}(H)$. The abstract domain is defined in Section 3.2. The abstract state is given by the tuple $\langle E, \mathcal{A}^N, \mathcal{A}^E\rangle$, where E is the set of edges, $\mathcal{A}^N$ is the mapping from nodes to their sets of reachability states, and $\mathcal{A}^E$ is the mapping from edges to their sets of reachability states. We next define the lattice for the abstract domain. The bottom element has the empty set of edges E and empty reachability information for both the nodes $\mathcal{A}^N$ and the edges $\mathcal{A}^E$. The top element for the lattice has (1) all the edges in E that are allowed by type constraints between all reachability nodes, (2) each heap node n has tuples in $\mathcal{A}^N$ for the powerset of all heap nodes that are allowed by types to reach n, and (3) each edge $\langle n, f, n'\rangle \in E$ has the powerset of the maximal set of tuples in $\mathcal{A}^E$ that are allowed by type constraints.

We next define the partial order for the reachability graph lattice. Equation A.1 defines the partial order. The definition for the $\subseteq_\triangle$ relation between reachability states is given in the Update Edge Reachability step of Section 4.5.

$$\langle E, \mathcal{A}^N, \mathcal{A}^E\rangle \sqsubseteq_A \langle E', \mathcal{A}^{N'}, \mathcal{A}^{E'}\rangle \text{ iff } E \subseteq E' \wedge \langle \mathcal{A}^N, \mathcal{A}^E\rangle \sqsubseteq \langle \mathcal{A}^{N'}, \mathcal{A}^{E'}\rangle \tag{A.1}$$

$$\begin{aligned}\langle \mathcal{A}^N, \mathcal{A}^E\rangle \sqsubseteq \; & \langle \mathcal{A}^{N'}, \mathcal{A}^{E'}\rangle \text{ iff } \forall n \in N, \forall \phi \in \mathcal{A}^N(n), \exists \phi' \in \mathcal{A}^{N'}(n), \\ & \phi \subseteq_\triangle \phi' \wedge \big(\forall \langle n_1, f_1, n_2\rangle, ..., \langle n_k, f_k, n\rangle \in E, \\ & \phi \in \mathcal{A}^E(\langle n_1, f_1, n_2\rangle) \cap ... \cap \mathcal{A}^E(\langle n_k, f_k, n\rangle) \Rightarrow \\ & \phi' \in \mathcal{A}^{E'}(\langle n_1, f_2, n_2\rangle) \cap ... \cap \mathcal{A}^{E'}(\langle n_k, f_k, n\rangle)\big)\end{aligned} \tag{A.2}$$

The join operation $(\langle E_1, \mathcal{A}^N{}_1, \mathcal{A}^E{}_1\rangle \sqcup \langle E_2, \mathcal{A}^N{}_2, \mathcal{A}^E{}_2\rangle)$ on the heap reachability graph lattice simply takes the set unions of the individual components: $\langle E_1 \cup E_2, \mathcal{A}^N{}_1 \cup \mathcal{A}^N{}_2, \mathcal{A}^E{}_1 \cup \mathcal{A}^E{}_2\rangle$.

We next define several helper functions. Equation A.3 defines the meaning of the statement that object o is reachable from the object o' in the concrete heap R. We define the object abstraction function $\texttt{rgn}(o)$ to return the single object heap node for o's allocation site if o is the most recently allocated object and the allocation site's summary node otherwise. Equation A.4 returns the number of objects abstracted by heap node n_f that can reach the object o. Equation A.5 abstracts the natural numbers into one of three arities. Equation A.6 computes the abstract reachability state for object o in the concrete heap $\langle O, R\rangle$.

$$\mathtt{rch}(o', o, R) = \exists f, o_1, f_1, ..., o_l, f_l . \langle o', f, o_1\rangle, ..., \langle o_i, f_i, o_{i+1}\rangle, ..., \langle o_l, f_l, o\rangle \in R \tag{A.3}$$

$$\mathtt{count}(o, O, R, n_f) = |\ \{o' \mid \forall o' \in O.\mathtt{rgn}(o') = n_f, \mathtt{rch}(o', o, R)\}\ | \tag{A.4}$$

$$\mathtt{abst}(n) = \begin{cases} \mathtt{0} & n = 0 \\ \mathtt{1} & n = 1 \\ \mathtt{MANY} & \text{otherwise} \end{cases} \tag{A.5}$$

$$\phi(o, O, R) = \{\langle n_f, \mathtt{abst}(\mathtt{count}(o, O, R, n_f))\rangle \mid n_f \in N_F\} \tag{A.6}$$

We next define abstraction functions that return the most precise reachability graph for the set of concrete heaps $h \subseteq \mathcal{P}(H)$. We use the standard subset partial ordering relation for our concrete domain of sets of concrete heaps. Equation A.7 generates the edge abstraction, Equation A.8 generates the reachability state abstraction for each node, and Equation A.9 generates the reachability state abstraction for each edge. Note that from the form of the definition of the abstraction function, we can see that it is monotonic. We mechanically synthesize a concretization function $\gamma(\langle E, \mathcal{A}^N, \mathcal{A}^E\rangle) = \sqcup\{h \mid \alpha(h) \sqsubset \langle E, \mathcal{A}^N, \mathcal{A}^E\rangle\}$ to create a Galois connection. The pair α and γ do not form a Galois insertion as two abstract reachability graphs can have the exact same set of concretizations. The global pruning algorithm addresses the practical effects on analysis precision of this issue by converting abstract reachability graphs into equivalent graphs that contain locally more precise reachability states.

$$\alpha_E(h) = \{\langle \mathtt{rgn}(o), f, \mathtt{rgn}(o')\rangle \mid \forall \langle o, f, o'\rangle \in R, \forall \langle O, R\rangle \in h\} \tag{A.7}$$

$$\alpha_{\mathcal{A}^N}(h) = \{\langle \mathtt{rgn}(o), \phi(o, O, R)\rangle \mid \forall o \in O, \forall \langle O, R\rangle \in h\} \tag{A.8}$$

$$\alpha_{\mathcal{A}^E}(h) = \{\langle\langle \mathtt{rgn}(o'), f, \mathtt{rgn}(o'')\rangle, \phi(o, O, R)\rangle \mid \forall o \in O, \forall \langle o', f, o''\rangle \in R, \forall \langle O, R\rangle \in h . \mathtt{rch}(o'', o, R)\} \tag{A.9}$$

B Termination

Termination of the analysis is straightforward. Reachability graphs form a lattice, and for a given set of allocation sites the lattice has a finite height. All transfer functions in the analysis are monotonic except stores with strong updates and method calls. With a simple modification to enforce monotonicity the analysis will terminate.

Our approach to enforcing monotonicity is to store the latest reachability graph result for every back edge and program point after a method call. The fixed point interprocedural algorithm takes the join of its normal result with these graphs to ensure the local result becomes no smaller.

C Soundness of the Core Intraprocedural Analysis

In this section, we outline the soundness of the core intraprocedural analysis. For all soundness lemmas, we argue $(\alpha \circ f)(h) \sqsubseteq_A (f^{\#} \circ \alpha)(h)$, where f represents the concrete operation and $f^{\#}$ is the corresponding transfer function on the abstract domain, to show soundness.

Lemma 1 (Soundness of Copy Statement Transfer Function). *The transfer function for the copy statement* `x=y` *is sound with respect to the concrete copy operation.*

Proof Sketch: The soundness of the transfer function for the copy statement `x=y` is straightforward. After the execution of the copy statement on the concrete heap, the variable `x` references the object that `y` referenced before the statement. We note that applying the abstraction function after the concrete copy statement yields the exact same abstract reachability graph as applying the abstraction function followed by the transfer function for the copy statement, therefore the copy transfer function is sound.

Lemma 2 (Soundness of Load Statement Transfer Function). *The transfer function for the load statement* `x=y.f` *is sound with respect to the concrete load operation.*

Proof Sketch: The soundness of the transfer function for the load statement `x=y.f` is also relatively straightforward. After the execution of the load statement on the concrete heap, the variable `x` references the object referenced by the `f` field of the object referenced by `y`. After abstraction, the edge for `x` would reference the same objects as the `f` field of the objects referenced by `y` and have the same reachability set.

The soundness of the edge set transform follows from the definition of α_E — all objects that `y.f` could possibly reference are included in the set $E_n(\texttt{y}, \texttt{f})$. Therefore, applying the abstraction function followed by removing the previous edges for `x` and adding the set of edges $\{\texttt{x}\} \times E_n(y)$ gives an E set that contains all of the edges generated by applying the transfer function and then abstraction function.

From the definition of $\alpha_{\mathcal{A}^E}$ we can determine that for each n that could abstract the object referenced by `y` and each corresponding n' that could abstract the object referenced by `y.f`, that the reference `y.f` could only reach objects with reachability states included in the set $\mathcal{A}^E(\langle \texttt{y}, n\rangle) \cap \mathcal{A}^E(\langle n, \texttt{f}, n'\rangle)$. Note the subtle point that the correctness of the intersection operation follows from the edge reachability aspect of the abstraction function definition (and not from the lattice ordering) — there must exist a path through the `y` reference and `y.f` to any objects that can be reached by the new `x` and by the abstraction function both `y` and `y.f` will include the reachability states of those objects. Therefore, the application of the abstraction function followed by the transfer function generates a set of reachability states for edges of `y` that include all of the reachability states generated by applying the concrete load statement followed by the abstraction function.

Lemma 3 (Soundness of Allocation Statement Transfer Function). *The transfer function for the allocation statement* `x=new` *is sound with respect to the concrete allocation operation.*

Proof Sketch: The transfer function for the allocation statement is similarly straightforward. The execution of the allocation statement on the concrete heap followed by the abstraction function yields an abstract reachability graph in which the previous newest allocated object at the site is now mapped to the summary node. The allocation statement transfer function applied to the abstraction function yields the exact same reachability graph and therefore the transfer function is sound.

If the allocation site is flagged, the new heap node has a single reachability state that contains a single reachability token with its own heap node and the arity `1`. The variable edge contains the same set of reachability states. If the allocation site is not flagged, the sets of reachability states contains only the empty reachability state.

Lemma 4 (Soundness of Store Statement Transfer Function). *The transfer function for the allocation statement* `x.f=y` *is sound with respect to the concrete store operation.*

Proof Sketch: We define $o_{\mathtt{x}}$ to be the concrete object referenced by `x` and $o_{\mathtt{y}}$ to be the concrete object referenced by `y`. The store operation can only add new paths in the concrete heap that include the newly created reference $\langle o_{\mathtt{x}}, f, o_{\mathtt{y}}\rangle$. In the abstraction, $E_n(\mathtt{x})$ gives the heap nodes that abstract the objects that `x` may reference and $E_n(\mathtt{y})$ gives the heap nodes that abstract the objects that `y` may reference. The concrete operation `x.f=y` creates a reference from the `f` field of the object that `x` references to the object that `y` references. Applying the abstraction function, the creation of this new reference in all concrete heaps represented by the abstract heap adds a set of edges $E_{\text{new}} \subseteq E_n(\mathtt{x}) \times \{\mathtt{f}\} \times E_n(\mathtt{y})$ to the abstract heap. Since the application of the transfer function to the initial abstraction adds a larger set of edges, it generates an abstract edge set that is higher in the partial order and therefore our treatment of edges in the store statement is sound.

We next discuss the soundness of the transfer function with respect to the reachability states for nodes. We note that the addition of the concrete reference can only (1) introduce new reachability from objects that could reach $o_{\mathtt{x}}$ to objects that $o_{\mathtt{y}}$ can reach and (2) allow edges that could reach $o_{\mathtt{x}}$ to reach objects that $o_{\mathtt{y}}$ can reach. The set $\Psi_{\mathtt{x}}$ defined in Equation 4.10 abstracts the reachability states for the objects that can reach $o_{\mathtt{x}}$ by the abstraction function. Similarly, $\Psi_{\mathtt{y}}$ from Equation 4.11 abstracts the allocation sites for the objects that can reach the objects downstream of $o_{\mathtt{y}}$.

By the abstraction function and the partial order, if an object abstracted by a heap node $n_{\mathtt{y}} \in E_n(\mathtt{y})$ can reach an object abstracted by the heap node n' with the abstract reachability state ϕ, then there must exist a path of edges from $n_{\mathtt{y}}$ to $n' \in N$ in the abstract reachability graph in which every edge along the path has ϕ in its set of reachability states and n' has ϕ in its set of reachability states and $\phi \in \Psi_{\mathtt{y}}$. By the abstraction function, the set of reachability states $\psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}$ for $n_{\mathtt{x}}$ abstract $o_{\mathtt{x}}$'s reachability from all objects from flagged nodes. Therefore, the constraints given by Equations 4.12 and 4.13 will propagate the correct reachability change set to n' and Equation 4.14 applies these reachability changes to n'. This implies that the set of reachability states for the nodes is higher or equal in the partial order of reachability graphs to the graph generated by applying a concrete operation followed by abstraction and therefore the node reachability states are sound.

We next discuss soundness with respect to edges that are upstream of the objects downstream of $o_{\mathtt{y}}$ in the pre-transformed concrete heap. Consider an object o abstracted by the heap node n that the store operation changed its reachability state from ϕ to ϕ'. By the abstraction function and partial order function, for any reference in the concrete heap, which we abstract by e, that can reach an object abstracted by the heap node n, there must exist a path of edges from e to n in the pre-transformed heap in which ϕ is in the reachability state of each edge along the path. Therefore, Constraints 4.15 and 4.16 propagate the reachability change tuple $\langle \phi, \phi' \rangle$ to e which Equation 4.19 will then apply to e and all edges along the path from e to e'.

Finally, we discuss soundness with respect to edges upstream of $o_{\mathtt{x}}$ that the newly created edge allows to reach objects downstream of $o_{\mathtt{y}}$. Consider any upstream reference in the concrete heap, which we abstract by the edge e that can reach an object abstracted by the heap node $n_{\mathtt{x}} \in E_n(\mathtt{x})$ — any reachability state it has for the source object of the store must be abstracted by $\phi \in \Psi_{\mathtt{x}}$ in pre-transformed abstract reachability graph and there must exist a path of edges from e to $n_{\mathtt{x}}$ such that ϕ is in the reachability state of every edge along the path. Therefore, Constraints 4.17 and 4.18 propagate the new reachability change tuples $\{\langle \phi, \phi \cup \psi_{\mathtt{y}} \rangle \mid \psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}\}$ to e and Equation 4.19 will then apply the change tuple to e.

At this point, only the new edge remains. Constraint 4.20 simply copies the reachability states from the edge for $\mathtt{y}$ whose reachability must be the same. It eliminates reachability states that are smaller in the partial order than any state in the source node as they must be redundant with some larger state. The previous three paragraphs imply that the set of reachability states for edges are higher or equal in the partial order of reachability graphs to the graph generated by applying a concrete operation followed by abstraction and therefore the edge reachability states are sound.

Lemma 5 (Soundness of Global Pruning Transformation). *The global pruning transformation is sound (it generates an abstraction that abstracts the same concrete heaps).*

Proof Sketch: We begin by overviewing the soundness of the first phase of the global pruning algorithm. Consider a flagged heap node n_f and a node n that contains n_f in its reachability state ϕ with non-$\mathtt{0}$ arity. From the abstraction function and partial order, if there is no path from n_f to n with ϕ in each edge's set of reachability states, then objects in the reachability state ϕ cannot be reachable from objects abstracted by n_f. Therefore, removing n_f from the reachability set ϕ on n and adding this new reachability set to all edges that (1) have ϕ in their reachability state and (2) have a path to n in which all edges along the path have ϕ in their reachability state generates an abstract reachability graph that abstracts the same concrete heaps and therefore the first phase is sound.

We next discuss the soundness of the second phase. Consider an edge e with a reachability state ϕ. If there is no path from edge e to some node n with all edges along the path containing ϕ in their sets of reachability states and node n including ϕ in its set of reachability states, then dropping ϕ from edge e's set of reachability states yields an abstract state that abstracts the same concrete heaps because if a reference abstracted by e could actually reach an object with the reachability state ϕ then the path would exist. Therefore, the second phase is sound.

D Interprocedural Analysis

We next outline the soundness of the interprocedural analysis. There is a small issue in the interprocedural analysis with the abstraction function for single-object heap nodes. It is possible to have a callee method that only conditionally allocates an object at an allocation site that the caller has a single-object heap node for. The mapping procedure will then merge the caller's single-object heap node into the summary node even though it may abstract the most recently allocated object from the site. One can see that this does not pose a correctness issue through a simple transform of the program that adds a special instruction at each method return that allocates an unreachable object at the given allocation site if the callee did not. It is straightforward to see that such a transform preserves the semantics of the program because it does not change the reachable runtime object graph and after this transform the abstract semantics exactly match the concrete program.

We outline the soundness of the interprocedural analysis by analogy to the intraprocedural analysis with inlining. We note that the callee operates on a graph that is a superset of the callee reachable part of the heap. If we consider only those elements that are in the callee reachable part of the heap, the analysis (1) generates a reachability subgraph that is greater in the partial order than the reachability graph that the inlined version would have and (2) all of those elements get mapped to the caller's heap. We note that reachability state changes on the placeholder edges and edges from placeholder nodes summarize the reachability changes of upstream edges and are sound for the same reasons as the store transfer function.