Disjointness Analysis for Java-Like Languages

James Jenista Brian Demsky

Abstract

This paper presents a disjointness analysis for Java-like languages. Two objects are disjoint if the parts of the heap reachable from the two objects are disjoint. The analysis is based on static reachability graphs, which characterize the reachability of each object in the heap from a set of objects of interest. Reachability graphs contain nodes to represent objects and edges to represent heap references. The graphs are annotated with sets of reachability states that describe which objects can reach other objects. The analysis includes a global pruning step which analyzes the entire reachability graph to prune impossible reachability states that cannot be removed with local information alone.

We have developed an implementation of the analysis and have evaluated the implementation on several benchmarks. We examined the analysis results to verify that the analysis reported all known aliases.

1. Introduction

This paper introduces a static analysis that discovers disjointness properties for select objects in Java-like languages. Two objects are disjoint if the parts of the heap reachable from each object are disjoint. While other analyses like alias analysis (Banning 1979; Cooper and Kennedy 1989; Diwan et al. 1998; Ruf 1997), pointer analysis (Shapiro and Horwitz 1997; Landi et al. 1993; Weihl 1980; Burke et al. 1995), and shape analysis (Chase et al. 1990; Ghiya and Hendren 1996b; Sagiv et al. 2002) also extract heap reference properties from a program's source, disjointness analysis answers a different question: Given that two objects (possibly from the same allocation site) are determined to be distinct at runtime or through other means, are the parts of the heap reachable from these two objects disjoint?

Our analysis is based on *reachability graphs*, which divide the heap into disjoint regions and characterize for each region the set of heap regions with objects that can reach the given region. The analysis is interprocedural and compositional. The analysis processes a method once for a given aliasing context and uses the summarized analysis results for future calling contexts; recursive methods may require analyzing a method multiple times until a fixed-point is reached.

1.1 Uses for Disjointness

We have used disjointness analysis to automatically generate parallelized implementations of programs written using in a task-based extension to Java called Bamboo. Disjointness analysis can extract fundamental referencing properties from object-oriented programs and therefore is likely to have a wide range of applications.

Disjointness results are useful for determining whether code can be parallelized. For example, to execute two serial method calls $p(r)$ and $q(s)$ in parallel, it is necessary to determine that they do not have any conflicting data structure accesses. If the objects referenced by r and s are allocated at the same allocation site, it is difficult for many pointer analysis to determine whether they reference disjoint data structures. If disjointness analysis determines that the parameter objects referenced by r and s of the two calls are mutually disjoint, checking that $r \neq s$ suffices to ensure that the two calls cannot access the same data (by following any fields of r or s) and therefore can be safely executed in parallel.

Disjointness analysis results can also help software developers discover important data structure invariants that a program maintains. For example, developers often make changes to legacy code that cause it to modify parts of existing data structures in new ways. One potential concern is whether those parts can be shared with other data structures and therefore affect other parts of the program. Disjointness analysis extracts information that tells programmers which parts of a data structure may be shared with other data structures and could therefore help the developer determine whether such modifications are safe.

1.2 Basic Approach

The analysis represents reachability information with reachability graphs. Reachability graphs contain *label nodes* that represent program labels and *heap region nodes* that represent disjoint collections of objects. These nodes are connected with *edges* that represent heap references. We say that one object can reach a second object if there exists a path of references in the reachability graph from the first

object to the second object. A *reachability state* for an object gives the heap region nodes of the objects that can reach the given object. The reachability state contains an arity for each heap region node which constrains how many objects from that heap region can reach the given object. The analysis annotates heap region nodes with sets of reachability states that describe what objects can reach the given object. The analysis annotates edges with sets of reachability states that give the possible reachability states for the objects that can be reached from that edge. The analysis can determine that two objects are disjoint if they do not appear together in any reachability states of a reachability graph.

Our analysis is compositional — it analyzes each method once to produce a reachability graph. Future call sites to the method use the previously computed reachability graph.

The analysis can perform strong updates in certain cases. The analysis includes a global pruning step that globally analyzes the reachability graph to prune impossible reachability states that cannot be removed with just local knowledge. The global pruning step primarily serves to improve the precision of reachability information after strong updates and method calls.

1.3 Contributions

The paper makes the following contributions:

- **Disjointness Analysis:** It presents a new compositional disjointness analysis that can discover heap reachability properties for objects of interest.
- **Selective Analysis:** The analysis client can flag the set of object allocation sites for the objects whose disjointness information is of interest. The analysis only analyzes reachability information for the objects of interest.
- **Global Pruning:** It introduces a global pruning step that globally analyzes the reachability graph to remove impossible reachability states that cannot be removed with just local knowledge.
- **Experimental Results:** It presents experimental results from a prototype implementation of the analysis. The results show that the analysis successfully discovers disjointness properties for our benchmarks.

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 presents an extension to the intraprocedural analysis that reasons about reachability at the variable granularity. Section 7 presents an evaluation of the analysis on several benchmarks. Section 8 presents related work; we conclude in Section 9.

2. Example

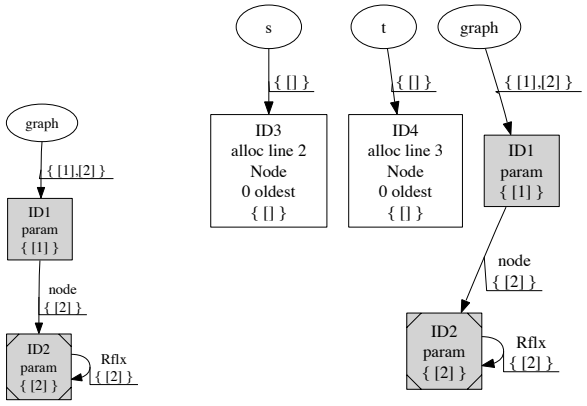
In this section we present an example to illustrate how our analysis works. Figure 1 presents an example that constructs several graphs and then runs a graph analysis that modifies information stored in the graph nodes. The method `graphLoop` populates an array with graph objects that are fully constructed by `makeGraph`. Our analysis will show that the objects reachable from a `Graph` object constructed in the loop are disjoint from objects reachable from the other `Graph` objects allocated in the same loop. This information could be used to parallelize the iterations of the second loop in conjunction with a simple dynamic check; if the iterations operate on different `Graph` objects at run-time, then our analysis results imply that the methods operate on disjoint sets of objects.

```
 1  public void makeGraph(Graph graph) {
 2    Node s = new Node();
 3    Node t = new Node();
 4    s.f = t;
 5    t.f = s;
 6    graph.node = s;
 7  }
 8
 9  public void graphLoop() {
10    Graph[] a = new Graph[nGraphs];
11    for(int i=0; i<nGraphs; i++) {
12      Graph g = new Graph();
13      makeGraph(g);
14      a[i] = g;
15    }
16    for(int i=0; i<nGraphs; i++) {
17      analyzeGraph(a[i]);
18    }
19  }
```

Figure 1. Graph Example

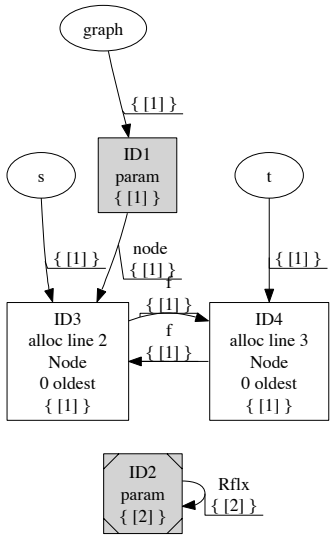
We begin with an intraprocedural analysis of the `makeGraph` method. Figure 2(a) presents the analysis results for the example at the beginning of the `makeGraph` method. The ellipse labeled `graph` represents the parameter variable `graph`. The rectangular heap region node labeled `ID1` represents the first parameter object. The shading of the node represents that the analysis must compute reachability information from this node. The set $\{[1]\}$ indicates that the objects represented by this heap region node have the *reachability state* $[1]$. The reachability state $[1]$ means that objects with this reachability state may be reachable from heap region node `ID1`. The edge from the label node `graph` to the heap region node `ID1` indicates that the label `graph` may reach an objects within the heap region `ID1`. The set $\{[1],[2]\}$ on this edge indicates that the edge models a heap reference that may reach objects in the reachability states $[1]$ and $[2]$.

The rectangular heap region node labeled `ID2` represents the part of the caller's heap reachable from the first parameter object. The chords on the corners of the node indicate that the heap region may represent multiple objects. The reflexive edge on the heap region node labeled `ID2` indicates that objects in this heap region can reference other objects in the same heap region. The set $\{[2]\}$ on this edge indicates that the edge models a heap reference that may reach objects in the reachability state $[2]$.



(a) Reachability graph after line 1

(b) Reachability graph after line 3



(c) Reachability graph after line 6

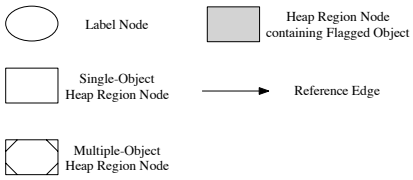


Figure 2. Intraprocedural Reachability Graphs for `makeGraph`

Figure 2(b) presents the reachability graph immediately after line 3. Two new label nodes have been created for `s` and `t` that have references to heap regions `ID3` and `ID4`, respectively. These heap regions are associated with the allocation site that allocated the objects. Heap regions `ID3` and `ID4` have no chords as they represent the most recently allocated object at the corresponding allocation sites. They are labeled with the corresponding allocation site. The analysis uses a k-limited abstraction for the allocation sites — these nodes are marked as the zeroth oldest, or newest node from the allocation sites.

As in Figure 2(a), the heap region nodes `ID1` and `ID2` are shaded. Shaded heap regions are *flagged*; heap regions are flagged only when they may contain objects in which the analysis client is interested in finding disjointness information about. The empty reachability state in the set of reachability states for heap regions `ID3` and `ID4` implies that no objects from flagged heap regions can reach heap regions `ID3` and `ID4` at this program point.

Figure 2(c) shows the reachability graph at the exit of the `makeGraph` method. Lines 4, 5, and 6 create reference edges $\langle$`ID3`, `f`, `ID4`$\rangle$, $\langle$`ID4`, `f`, `ID3`$\rangle$, and $\langle$`ID1`, `node`, `ID3`$\rangle$, respectively. Note that the set of reachability states for `ID1` remains $\{[1]\}$ and therefore the final reachability graph shows that the method does not change the reachability states of the parameter objects.

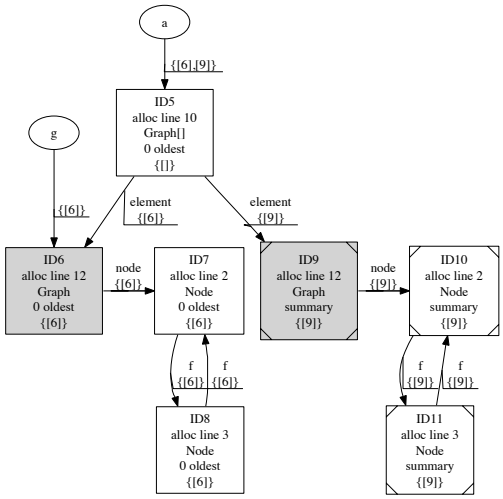


Figure 3. Reachability graph for `graphLoop` after line 15

At this point, the `makeGraph` method has been fully analyzed and its results are available for analyzing the `graphLoop` method. By inspection, it is clear that `graphLoop` populates an array with references to graphs that are disjoint. In Figure 3 the array object is represented by a single-object heap region `ID5`, and the result of assigning its elements is to create a reference edge that acts like a member field with the special label `element`. Element references are not removed by strong updates.

The analysis combines information about allocated objects that are older than the k-limit into a per allocation site, multiple-object, summary heap region node. Figure 3 shows that label node `g` always references the newest allocated `Graph` object. The objects that are allocated in line 12 are flagged so the heap region nodes for that allocation site are shaded.

Consider this reachability graph without reachability states on edges or nodes. The reference edge $\langle$`ID9`, `node`, `ID10`$\rangle$ could represent references from disjoint pairs of objects from heap regions `ID9` and `ID10` or it could represent more than one object from the heap region `ID9`

referencing the same object from ID10. Because the set of reachability states for heap region ID10 only contains a reachability state with a single *token* [9] it is clear that any object in that region is reachable only from at most one object in heap region ID9. If two different Graph objects represented by the summary node ID9 could reference the nodes ID10 and ID11, the nodes ID10 and ID11 would contain the reachability state [9*], where 9^* means that multiple objects in the summary node ID9 can reach an object represented by the heap region node. Therefore, this reachability graph implies that Graph objects from the allocation site in line 12 reference disjoint heap regions. This extra reachability information is the key difference between disjointness analysis and pointer analysis — disjointness analysis uses this reachability information to determine distinct disjointness properties for objects that are represented by the same node in the disjointness graph.

3. Analysis Representations

This section presents the representation of the input for the analysis and the representation of the reachability graph.

3.1 Program Representation

The analysis takes as input a control flow graph intermediate representation for each method; edges indicate control flow and all program statements have been decomposed into statements relevant to the analysis: copy statements, load statements, store statements, object allocation statements, and method invocation statements.

We define for a statement st in a method's control flow graph:

- •st is the program point just before st
- st• is the program point just after st
- $\mathrm{parents}(\texttt{st})$ is the set of statements that may flow to st
- $\mathrm{children}(\texttt{st})$ is the set of statements that st may flow to.

For each method m there is a single entry statement $\mathrm{entry}(m)$ and a set of return statements $\mathrm{returns}(m)$.

3.2 Reachability Graph Elements

Label nodes represent the values of program variables — there is exactly one label node $l \in L$ for each program variable. Heap region nodes represent objects in the heap. Their properties are listed below:

- Heap region nodes can bound a single object or multiple objects. Multiple-object heap region nodes have chords across each corner in visualized reachability graphs.
- Flagged heap regions contain objects that the analysis is interested in tracking the disjointness properties of or reachability from. These regions are shaded in visualized reachability graphs.
- A heap region associated with a parameter is marked with the parameter's index.
- We use a k-limited approximation for heap regions associated with an allocation site. The most recent k objects at an allocation site are assigned their own single-object heap region node, and all older object allocations are mapped to the summary node for the allocation site.

The set of all heap region nodes $n \in N$ for the method m is given by Equation 1.

$$n \in N := \texttt{Allocation sites} \times \{0, 1, \ldots, k\} \cup N_P \quad (1)$$

$$n_p \in N_P := \texttt{Parameter nodes for the method}\, m \quad (2)$$

The set of flagged heap region nodes to track reachability of is given by Equation 3.

$$n_f \in N_F := \texttt{Flagged allocation sites} \subseteq N \quad (3)$$

Reference edges $e \in E$ are of the form $\langle l, n\rangle$ or $\langle n, f, n'\rangle$. The heap region node or label node that reference edge e originates from is given by $\mathrm{src}(e)$. The heap region node e refers to is given by $\mathrm{dst}(e)$. Every reference edge between heap region nodes has an associated field $f \in F$, including element access or the special field type $\mathbb{F}$ that matches all fields, as given in Equation 4. The special field $\mathbb{F}_{\texttt{f}}$ indicates that the edge represents a possible caller context reference for field f and exempts the edge from adding new edges into a caller's reachability graph in the interprocedural mapping procedure.

The field f refers to is given by $\mathrm{field}(\langle n, f, n'\rangle)$.

$$f \in F := \texttt{Fields} \cup [] \cup \mathbb{F} \cup \mathbb{F}_{\texttt{f}} \quad (4)$$

The initial reflexive edge on a parameter heap region always has the $\mathbb{F}$ field and is specially marked Rlfx. The marking is required so that, for any particular invocation, a parameter object can be arbitrarily dereferenced to get an internal object by taking the reflexive edge. However, the Rlfx marking exempts the edge from adding new edges into a caller's reachability graph in the interprocedural mapping process.

The set of reference edges E in a reachability graph is given by Equation 5.

$$E \subseteq N \times F \times N \cup L \times N \quad (5)$$

We define five convenience functions for reachability graph elements given by Equation 10.

$$E(l) = \{n \mid \langle l, n\rangle \in E\} \quad (6)$$

$$E^\circ(l) = \{\langle l, n\rangle \mid \langle l, n\rangle \in E\} \quad (7)$$

$$E(n) = \{n' \mid \langle n, f, n'\rangle \in E\} \quad (8)$$

$$E(l, f) = \{n' \mid \langle l, n\rangle, \langle n, f, n'\rangle \in E\} \quad (9)$$

$$E^\circ(l, f) = \{\langle n, f, n'\rangle \mid \langle l, n\rangle, \langle n, f, n'\rangle \in E\} \quad (10)$$

3.3 Reachability Annotations

This section describes how the analysis extends the basic graph representation of the heap with a set of reachability

annotations. A token η_n is the symbol of a heap region node n that we are interested in the disjointness properties of. A *token tuple* $\langle \eta, \mu \rangle \in M$ is a token and arity pair where the arity value μ in this analysis is taken from the set $\{\texttt{ZERO}, \texttt{ONE}, \texttt{ZERO-OR-MORE}\}$. The arity gives the number of objects from a given heap region that can reach the relevant object. Our notation for token tuples is to write just the token η for the arity ONE or η^* for the arity ZERO-OR-MORE. Token tuples with arity ZERO are not written explicitly.

A reachability state $\phi \in \Phi$ contains exactly one token tuple for every distinct token, and when written omits token tuples with arity ZERO. For example, the reachability state $\phi_n = [\eta_{n_1}, \eta^*_{n_2}] \in \Phi_n$ at some heap region node n indicates that it is possible for at most one object in heap region n_1, zero-or-more objects from heap region n_2 and exactly zero objects from all other heap regions to reach the objects of heap region n. This reachability state implies that objects from heap regions n_1 and n_2 are not disjoint.

The function $\alpha(n) \rightarrow 2^{2^M}$ maps a heap region node n to sets of possible reachability states. The reachability of an object represented by the heap region node n is described by one of the reachability states given by the function α. Two heap regions n_1 and n_2 are definitely disjoint in a reachability graph if there is no heap region node n whose set of reachability states contains a reachability state with both η_{n_1} and η_{n_2} with non-ZERO arity. We represent the function α as a set of tuples. We define the helper function

$$\alpha(n) = \{\phi \mid \langle n, \phi \rangle \in \alpha\}. \tag{11}$$

The function $\beta(e) \rightarrow 2^{2^M}$ maps a reference edge e to sets of reachability states that are possible for objects accessible through e. We represent the function β as a set of tuples. We define the helper functions

$$\beta(l) = \{\phi \mid \langle \langle l, n \rangle, \phi \rangle \in \beta\}, \tag{12}$$

$$\beta^\circ(l) = \{\langle \langle l, n \rangle, \phi \rangle \mid \langle \langle l, n \rangle, \phi \rangle \in \beta\}, \tag{13}$$

$$\beta(e) = \{\phi \mid \langle e, \phi \rangle \in \beta\}. \tag{14}$$

The analysis maintains the invariant that for heap region node n with $\phi \in \alpha(n)$, an edge e that can reach an object o represented by n whose reachability state is represented by ϕ must have $\phi \in \beta(e)$, and $\phi \in \beta(e')$ for every edge e' along the path from e to o.

4. Intraprocedural Analysis

The intraprocedural analysis of a method m begins by initializing the reachability graph associated with $s\bullet$ for each statement s in m to the empty graph. Then $\mathrm{entry}(m)$ is scheduled for analysis.

The reachability graphs associated with statements during the intraprocedural analysis are a partial result r and may be revisited many times. The analysis uses a standard fixed-point algorithm which performs the following basic steps at each statement:

1. Create a new, empty reachability graph, r'.
2. Merge each graph in $\mathrm{parents}(s)$ into r'. This represents the reachability for $\bullet s$.
3. Use the type of s to transform r' as described below.
4. If $r' \neq r$, $r \leftarrow r'$ and schedule $\mathrm{children}(s)$ for analysis.

4.1 Method Entry

The parameter information for a method m is contained in a special statement that is always $\mathrm{entry}(m)$. We first describe how the analysis processes methods whose parameters are disjoint. We then extend the approach to handle aliases between objects reachable from different parameters. The transform for this statement creates, for each parameter p_i, a new single-object heap region node n_{p_i} that represents the parameter object and a new multiple-object heap region node n_{γ_i} that represents objects reachable from the parameter object. Then a label node p_i is added along with reference edge $\langle p_i, n_{p_i} \rangle$. A special label q_i that is out of the program scope is also added with $\langle q_i, n_{p_i} \rangle$; the purpose of q_i is described in the discussion of method calls in Section 5.1. Similarly, a special label r_i that is out of the program scope is also added with $\langle r_i, n_{\gamma_i} \rangle$. Finally, reference edges between parameter nodes as described in Section 5 are added to summarize the caller context reference edges..

Each method contains an aliasing context set Π that contains the parameter indices for any parameters that may contain aliases to or from other parameters. In this case, the analysis generates a single multiple-object heap region node n_{γ_Π} for all parameters that may be aliased. The multiple object heap nodes for each parameter in Π all refer to n_{γ_Π}. Each parameter in the aliased context set has its own single-object heap region node n_{p_i}.

The statement $\mathrm{entry}(m)$ has no parent statements and always generates the same reachability graph; therefore it is analyzed once per intraprocedural method analysis.

4.2 Copy Statement

A copy statement of the form `x = y` makes the variable `x` point to the object that `y` points to. The analysis always uses strong updates for label nodes. The analysis models the effect of this statement by discarding all the old references from label `x` and then copying all the references from label `y`. Equation 15 and Equation 16 give the transformations.

$$E' = (E - E^\circ(\texttt{x})) \cup (\{\texttt{x}\} \times E(\texttt{y})) \tag{15}$$

$$\beta' = (\beta - \beta^\circ(\texttt{x})) \cup (\{\texttt{x}\} \times \beta(\texttt{y})) \tag{16}$$

4.3 Load Statement

Load statements of the form `x = y.f` make the variable `x` point to the object that `y.f` points to. The analysis uses strong updates for the label node `x`. The reference edges from the field, including the reachability information, are copied to `x`. Note that this statement does not create any new references for reachability information to flow across.

$$E' = (E - E^{\circ}(\mathtt{x})) \cup (\{\mathtt{x}\} \times E(\mathtt{y},\mathtt{f})) \quad (17)$$

$$\beta' = (\beta - \beta^{\circ}(\mathtt{x})) \cup \bigcup_{\langle n,\mathtt{f},n'\rangle \in E^{\circ}(\mathtt{y},\mathtt{f})} \Big(\{\langle \mathtt{x}, n'\rangle\} \times \Big(\beta(\langle \mathtt{y}, n\rangle) \sqcap \beta(\langle n, \mathtt{f}, n'\rangle)\Big)\Big) \quad (18)$$

4.4 Store Statement

Store statements of the form `x.f = y` point the `f` field of the object to which `x` points at the object to which `y` points. The transform for store statements is broken into three steps:

1. Remove reference edges for strong updates.

2. Calculate reachability changes and propagate them.

3. Add reference edges to model the store operation.

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 analysis results. Weak updates are given in Equation 19.

$$E' = E \cup (E(\mathtt{x}) \times \{\mathtt{f}\} \times E(\mathtt{y})) \quad (19)$$

Strong updates are possible under either of two conditions. First, when label node `x` is the only reference to some heap region node n_u. In this case we can destroy all reference edges from n_u with field `f` because no other label nodes can reach n_u.

The second condition for strong updates is when the label node `x` references exactly one heap region node n_w and n_w is a single-object heap region. When this is true `x` definitely refers to the object in n_w and the existing edges with field `f` from n_w can be removed.

Note that when reference edges are removed by a strong update, reachability for any heap region node or reference edge in the reachability graph may change if the removed edge provided the reachability path. When a strong update occurs, a global reachability sweep is used to prune impossible reachability states following the completion of the store statement transform. The global sweep is discussed in Section 4.9.

The store statement creates reference edges between heap region nodes and may create new reachability paths. Therefore reachability must propagate in two ways when a store statement creates a new reference edge e_{new}. Heap region nodes upstream of e_{new} may now have a reachability path to heap regions downstream of e_{new} so new tokens may appear in α information downstream. Additionally, β may change for reference edges upstream of any heap region node whose reachability changes.

For each heap region node $n_{\mathtt{x}} \in E(\mathtt{x})$ and $n_{\mathtt{y}} \in E(\mathtt{y})$:

- The set of source reachability states that `x` can contribute is $R = \alpha(n_{\mathtt{x}}) \cap \beta(\langle \mathtt{x}, n_{\mathtt{x}}\rangle)$.
- The set of reachability states reachable from `y` is $O = \beta(\langle \mathtt{y}, n_{\mathtt{y}}\rangle)$.

$\cup_{\blacktriangle}$	0	1	*
0	0	1	*
1	1	*	*
*	*	*	*

Table 1. Results of taking union of two input arity values.

- Define $C_{n_{\mathtt{y}}} = \{\langle o, o \cup_{\triangle} r\rangle \mid o \in O, r \in R\}$, and $C_{n_{\mathtt{x}}} = \{\langle r, o \cup_{\triangle} r\rangle \mid o \in O, r \in R\}$, where $\cup_{\triangle}$ takes the union of two reachability states.

  Recall that each token of the token tuples in a reachability state must be unique. When two reachability states are combined, however, token tuples with matching tokens should merge arity values according to $\cup_{\blacktriangle}$ shown in Table 1.

The second step of the store statement transform is to propagate the reachability change tuple sets captured in $C_{n_{\mathtt{x}}}$ and $C_{n_{\mathtt{y}}}$. Intuitively, to update a set of reachability states the first item in a change tuple must match an existing reachability state; if it does the second item should be added to the set.

There are five phases to the propagation.

1. Calculate change function $\Lambda^{\text{node}}(n)$ for each heap region node n that is reachable from $n_{\mathtt{y}}$ using the two constraints

$$\Lambda^{\text{node}}(n_{\mathtt{y}}) \supseteq C_{n_{\mathtt{y}}}, \quad (20)$$

$$\Lambda^{\text{node}}(n') \supseteq \{\langle \phi, \phi'\rangle \mid \langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(n), \langle n, f, n'\rangle \in E, \phi \in \beta(\langle n, f, n'\rangle)\}. \quad (21)$$

The implementation uses a fixed-point strategy to compute a solution to these constraints.

2. Next, calculate the new reachability set for each heap region n

$$\alpha'(n) = \alpha(n) \cup \{\phi' \mid \phi \in \alpha(n), \langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(n)\} \cup \{\phi \mid \phi \in \alpha(n), \nexists \phi'.\langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(n)\}. \quad (22)$$

3. The analysis next computes the update for β from the changes made to α. The change function Λ^{edge} satisfies the two constraints:

$$\Lambda^{\text{edge}}(e) \supseteq \{\langle \phi, \phi'\rangle \mid \langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(dst(e)), \phi \in \alpha(dst(e)), \phi \in \beta(e)\} \quad (23)$$

$$\Lambda^{\text{edge}}(e) \supseteq \{\langle \phi, \phi'\rangle \mid \langle \phi, \phi'\rangle \in \Lambda^{\text{edge}}(e'), \phi \in \beta e, dst(e) = src(e')\} \quad (24)$$

The implementation computes a solution for Λ^{edge} with a fixed-point algorithm.

4. Similar to the previous phase, the analysis propagates $C_{n_{\mathtt{x}}}$ upstream from $n_{\mathtt{x}}$ using the change function Υ^{edge}. Υ^{edge} satisfies the two constraints:

$$\Upsilon^{\text{edge}}(e) \supseteq \{\langle\phi,\phi'\rangle \mid \langle\phi,\phi'\rangle \in C_{n_{\mathtt{x}}}, \phi \in \beta(e), dst(e) = n_{\mathtt{x}}\} \quad (25)$$

$$\Upsilon^{\text{edge}}(e) \supseteq \{\langle\phi,\phi'\rangle \mid \langle\phi,\phi'\rangle \in \Upsilon^{\text{edge}}(e'), \phi \in \beta(e), dst(e) = src(e')\} \quad (26)$$

5. Finally, the analysis calculates the new reachability set for edge e using $\Lambda^{\text{edge}}(e)$ and $\Upsilon^{\text{edge}}(e)$

$$\begin{aligned}\beta'(e) \;=\; & \beta(e) \cup \{\phi' \mid \langle\phi,\phi'\rangle \in \Lambda^{\text{edge}}(e), \phi \in \beta(e)\} \\ & \cup \{\phi' \mid \langle\phi,\phi'\rangle \in \Upsilon^{\text{edge}}(e), \phi \in \beta(e)\}. \quad (27)\end{aligned}$$

After the propagation is completed, the analysis adds the reference edges $e_{new} = \{\langle n_{\mathtt{x}}, \mathtt{f}, n_{\mathtt{y}}\rangle \mid n_{\mathtt{x}} \in E(\mathtt{x}), n_{\mathtt{y}} \in E(\mathtt{y})\}$. The reachability states for a new edge must be (1) in the set of reachability states for the edge $\langle \mathtt{y}, n_{\mathtt{y}}\rangle$ and (2) must be a superset or equal to some reachability state in $\alpha'(n_{\mathtt{x}})$ as $n_{\mathtt{x}}$ can reach the new edge.

We give the formula for β' for the new edge

$$\beta'(\langle n_{\mathtt{x}}, \mathtt{f}, n_{\mathtt{y}}\rangle) = \{\phi \in \beta'(\langle \mathtt{y}, n_{\mathtt{y}}\rangle) \mid \exists \phi' \in \alpha'(n_{\mathtt{x}}), \phi' \subseteq_{\triangle} \phi\}, \quad (28)$$

where $\phi' \subseteq_{\triangle} \phi$ if ϕ contains all tokens with a non-ZERO arity that ϕ' contains with a non-ZERO arity.

4.5 Element Load and Store Statements

Array elements are treated as a single, special field of an array object and always have weak store semantics. The analysis does not differentiate between the statements `y[1] = z` and `y[2] = z`.

4.6 Object Allocation Statement

Objects created at an allocation site are represented as single-object heap regions for the k most recently allocated objects at that allocation site. Any references to objects from the allocation site that are older than the kth object refer to a summarization node for the allocation site.

The transform for an allocation program point merges the kth single-object heap region into the site's summary node. The newest single-object heap region node is then the target of the label assignment similar to label assignments described above.

This step merges the reachability information for the kth single-object heap region n_k into the summary node n_s. Note that when η_{n_s} and η_{n_k} appear in the same token set before the aging operation there will be two η_{n_s} tokens afterward. In this case the new arity for the summary token is given by $\cup_{\blacktriangle}$. The reachability annotations enable the analysis to maintain precise reachability information in the presence of the summarization step.

4.7 Return Statement

Return statements are of the form `return x` which returns the object referenced by the label `x` to the caller. The analysis introduces a special `Return` label that is out of program scope to each reachability graph. At a method return the transform is to assign the `Return` label to the references of label `x`. We assume without loss of generality that the control flow graph has been modified to merge the control flow for all return statements. The `Return` label is discussed in the call site section when mapping callee information to the caller context.

4.8 Control Flow Join Points

To analyze a statement, the analysis first must compute 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 label nodes for r_{out} is the union of the label nodes in the input graphs r_0 and r_1.

2. The set of heap region nodes for r_{out} is the union of the heap region nodes in the input graphs. A simple union of the reachability states is taken, $\alpha(n_{\text{out}}) = \alpha(n_0) \cup \alpha(n_1)$.

3. The set of reference edges for r_{out} is the union of the reference edges of the input graphs. Recall that reference edges are unique in a reachability graph with respect to source, field, and destination. If a reference edge e_0 in r_0 and e_1 in r_1 have these attributes in common then $\beta(e_{\text{out}}) = \beta(e_0) \cup \beta(e_1)$.

4.9 Global Reachability

Strong updates for store statements may remove reference edges leaving some impossible reachability states in the reachability graph. Transformations that model method invocations (which will be given in the interprocedural analysis in Section 5.1) can also introduce impossible reachability states. These impossible reachability states potentially make the analysis results less precise. Our analysis includes a global pruning step that uses global reachability constraints to identify and prune impossible reachability states.

4.9.1 Global Reachability Constraints

Reachability information must satisfy two reachability constraints:

- **Node Reachability Constraint:** For each node n, $\forall \phi \in \alpha(n)$, $\forall \langle n', \mu\rangle \in \phi$ if $\mu \in \{\text{ONE}, \text{ZERO-OR-MORE}\}$, then there must exist a set of edges $e_1, \ldots, e_m$ such that $\phi \in \beta(e_i)$ 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, $\forall \phi \in \beta(e)$ there exists $n \in N$ and $e_1, \ldots, e_m \in E$ such that $\phi \in \alpha(n)$; $\phi \in \beta(e_i)$ 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.

4.9.2 Global Reachability Algorithm

The algorithm proceeds in two phases: the first phase enforces the node reachability constraint and the second phase enforces the edge reachability constraint.

The first phase uses the existing β information to prune impossible reachability sets to generate a consistent α' from the previous α. The algorithm iterates through each flagged node n_f. It uses a standard graph reachability algorithm to enforce the node reachability constraint. We define the function $\mathcal{B}_f : E \rightarrow 2^{2^M}$ to store reachability information from node n_f. We represent $\mathcal{B}_f$ as a set of tuples. $\mathcal{B}_f$ satisfies the constraints: $\forall e \in E(n_f), \mathcal{B}_f(e) \supseteq \beta(e)$ and $\forall e \in E, e' \in E(dst(e)), \mathcal{B}_f(e') \supseteq \beta(e') \cap \mathcal{B}_f(e)$. It uses a fixed point algorithm to propagate reachability information to solve the constraints. Finally, for each node n and each reachability state $\phi \in \alpha(n)$ the analysis shortens the reachability state ϕ to remove tokens of the form η_{n_f} or $\eta^*_{n_f}$ to generate a new reachability state ϕ' if the reachability state ϕ does not appear in $\mathcal{B}_f(e)$ of any edge e incident to n. Note the exception that this step should not shorten the reachability states of the flagged node n_f to prune the tokens η_{n_f} or $\eta^*_{n_f}$. The analysis then propagates these changes to β of the upstream edges using the same procedure described in step 3 of the propagation phase for store operation presented in Section 4.4 to generate β_r.

The second phase uses the now internally consistent α' information and the β_r information that existed before the first phase to generate a consistent β'. Conceptually, the analysis starts from every heap region node n and propagates the reachability states of $\alpha(n)$ backwards over reference edges. The analysis initializes $\beta' = \{\beta_r(e) \cap \alpha'(n) \mid \forall e \in E, n = dst(e)\}$. The analysis then propagates reachability information backwards to satisfy the constraint: $\beta'(e) \supseteq \beta_r(e) \cap \beta'(e')$ for all $e' \in E(dst(e))$. The propagation continues until a fixed-point is reached.

4.10 Static Fields

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

5. Interprocedural Analysis

The interprocedural analysis uses a standard fixed point algorithm. The analysis begins at the top level method m_{main} with the aliasing context $\Pi = \emptyset$. The analysis removes a method m and aliasing context Π from the workset for analysis. If the intraprocedural result for a method m in the aliasing context Π is different from the previously stored $r_{m,\Pi}$ then the new result replaces $r_{m,\Pi}$ and all methods that can potentially call m in context Π are added to the work set.

5.1 Analyzing Call sites

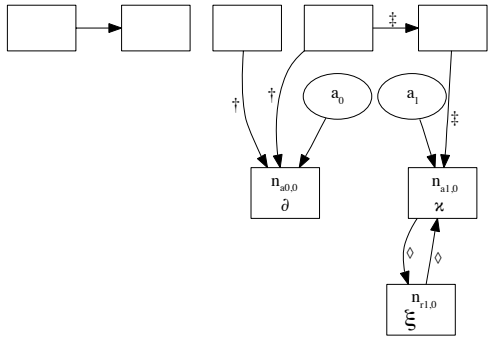
We next present how the interprocedural analysis adds support for analyzing call sites to the intraprocedural analysis. For the call site `cs`, the analysis maps the heap regions of the caller reachability graph at •`cs` onto the heap regions of the callee's current reachability graph at the call site. Then the callee graph is used to update the caller's reachability graph.

Some definitions for the concepts in call site analysis:

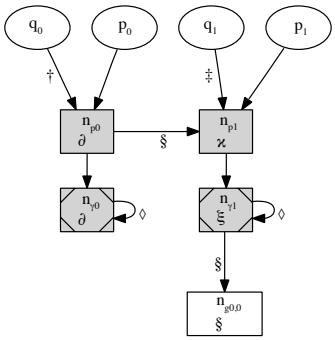
- The ith argument passed to the callee has a label node a_i in the caller reachability graph and a_i references j_i parameter object heap regions, $\{n_{ai0}, \ldots, n_{aij_i}\}$ in the caller. The heap regions $\{n_{ri0}, \ldots, n_{rik_i}\}$ are reachable from the ith parameter heap region nodes.
- The ith parameter of the callee has a label node p_i in the callee reachability graph referencing a single-object heap region n_{p_i} that models the parameter object. There is a second multiple-object heap region n_{γ_i} that models the objects reachable from the parameter object. The separate single-object heap region node enables the analysis to maintain precise information about the parameter object. The parameter part of the heap initially contains the following edges as allowed by type constraints and aliasing context: the edges $\langle n_{p_i}, \mathbb{F}_f, n_{p_j}\rangle$ for each field f of the parameter object that can point to itself or other parameter objects, the edge $\langle n_{p_i}, \mathbb{F}_f, n_{\gamma_i}\rangle$ for each field f of the parameter object, the edge $\langle n_{\gamma_i}, \mathbb{F}, n_{\gamma_i}\rangle$, and the edge $\langle n_{\gamma_i}, \mathbb{F}, n_{p_i}\rangle$. Note that n_γ may be entirely omitted if the parameter object only contains primitive fields.
- Special labels q_i and r_i model out-of-method-context references to n_{p_i} and n_{γ_i}, respectively. During analysis of the callee the reference edges $\langle q_i, n_{p_i}\rangle$ and $\langle r_i, n_{\gamma_i}\rangle$ will naturally capture changes to β useful for updating caller reachability information.
- Define $M = \{\langle n_{p_0}, n_{a00}\rangle, \langle n_{p_0}, n_{a01}\rangle, \langle n_{\gamma_0}, n_{r00}\rangle, \ldots, \langle n_{p_1}, n_{a10}\rangle, \langle n_{\gamma_1}, n_{r10}\rangle, \ldots\}$ to describe the mapping from heap region nodes in the callee graph to heap region nodes in the caller graph.
- For each allocation site $G_t = \{n_{g_{t0}}, \ldots, n_{g_{tk}}, n_{g_{ts}}\}$ of the callee the same nodes temporarily exist in the caller separately as $G'_t = \{n_{g'_{t0}}, \ldots, n_{g'_{tk}}, n_{g'_{ts}}\}$.

Let the program point for the call site `callee(a0, a1)` be `cs` and the method declaration be `void callee(p0, p1)`. Figure 4(a) presents an example caller context reachability graph for the method `caller` at the callsite `cs` and Figure 4(b) presents an example callee reachability graph for the method `callee`.

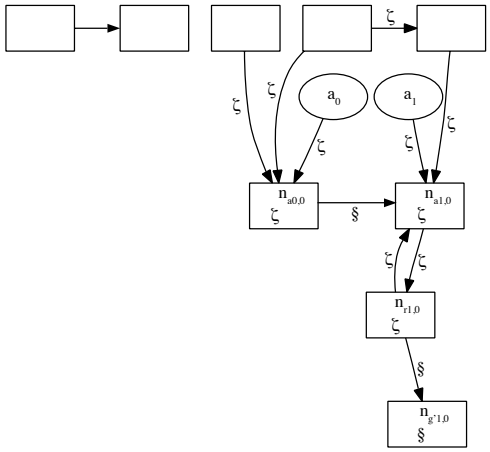
We establish a mapping between heap region nodes in the caller graph and the callee to determine how the reachability of the callee may affect the caller graph. Figure 4 shows the caller-to-callee heap region node and reference edge mapping. Figure 4(c) shows the updated caller context.



(a) Caller Context: ∂ maps into n_{p_0}, $\varkappa$ maps into n_{p_1}, ξ maps into n_{γ_1}, † maps into $\langle q_0, n_{p_0}\rangle$, ‡ maps into $\langle q_1, n_{p_1}\rangle$, $\diamond$ maps into $\langle n_{p_1}, \mathbb{F}, n_{p_1}\rangle$



(b) Callee Context: § nodes and edges generated by callee program statements.



(c) Updated Caller: § nodes and edges generated by callee program statements, ζ nodes and edges have altered reachability from call site transform.

Figure 4. Classification of nodes and edges in the caller-callee mapping are shown in (a) and (b). The effect of updating the caller context reachability graph of (a) with the callee (b) is shown in (c).

5.2 Conceptual Steps for Call Site Analysis

The following steps describe how the caller-to-callee mapping of edges and heap regions is used to fold callee effects into the caller context reachability graph.

- **Aliasing Context:** The caller computes the alias context set Π for the call site. It then checks whether the analysis has processed this call site before for the given caller aliasing context. If the analysis has already processed this call site for the same caller alias context, the analysis looks up the previous call site alias context Π_{old}. The analysis adds any parameters in Π_{old} to the new aliasing context Π to ensure termination.
- **Parameter Object Reachability:** The callee may change reachability states of the parameter objects. The reachability states in $\alpha(n_{p_i})$ summarize how the callee may change the reachability of parameter objects.
- **Other Caller Node Reachability:** The callee may change reachability states of objects reachable from the parameter objects. The reachability states in $\alpha(n_{\gamma_i})$ summarize how the callee may change the reachability of any objects that are reachable from the parameter objects.
- **Caller/Callee Edge Reachability:** The callee may change reachability states of caller reference edges that are reachable from the callee's parameters. The reachability states in β for the edges $\langle n_{p_i}, \mathbb{F}_f, n_{p_j}\rangle$, $\langle n_{p_i}, \mathbb{F}_f, n_{\gamma_i}\rangle$, $\langle n_{\gamma_i}, \mathbb{F}, n_{\gamma_i}\rangle$, and $\langle n_{\gamma_i}, \mathbb{F}, n_{p_i}\rangle$ summarize how the callee may change the reachability of the corresponding caller edges. The edge rewrite sets $J^{p\mathbb{F}p}$ and $J^{p\mathbb{F}\gamma}$ are used when the caller passes an object into the callee that is derived from the parameter class. They are used to model reachability changes to any class fields from the derived class that the callee is not aware of.
- **Upstream Caller Edge Reachability:** The callee may change the reachability states of caller reference edges that are upstream from the callee's parameters. The reachability state for the callee reference edge $\beta(\langle q_i, n_{p_i}\rangle)$ summarizes how the callee may change the reachability of edges that are upstream of the parameter object and the reachability state for the callee reference edge $\beta(\langle r_i, n_{\gamma_i}\rangle)$ summarizes how the callee may change the reachability of edges that are upstream of the objects reachable from the parameter object.
- **Callee Node Reachability:** The caller's reachability information for the reachability of the parameter objects is used to update the parameter object reachability tokens that appear in α for the nodes allocated by the callee.
- **Callee Edge Reachability:** The caller's reachability information for the reachability of the parameter objects is used to update the parameter object reachability tokens that appear in β for the edges created by the callee.
- **Summarize Allocation Site Nodes:** The graph at this point may contain allocation site nodes for the same allocation site from both the caller and callee. The analysis summarizes the oldest nodes to ensure the abstraction fits within its normal k-limit.

5.3 Helper Functions

This section defines several helper functions and operators that the call site transfer function uses.

1. **Parameter Object Reachability:** For each p_i, define the node rewrite set $H_i^p = \alpha(n_{p_i})$. The rewrite rule H_i^p captures with respect to n_{p_i}'s initial reachability state at the beginning of the callee how the callee changed the reachability of the ith parameter object heap region node.
2. **Other Caller Node Reachability:** For each γ_i, define the node rewrite set $H_i^\gamma = \alpha(n_{\gamma_i})$. The rewrite rule H_i^γ captures with respect to n_{γ_i}'s initial reachability state at the beginning of the callee how the callee changed the reachability of heap region nodes that are reachable from the ith parameter object.
3. **Caller/Callee Edge Reachability:** The reference edges $\langle n_{p_i}, \mathbb{F}_f, n_{p_j}\rangle$, $\langle n_{p_i}, \mathbb{F}_f, n_{\gamma_i}\rangle$, $\langle n_{\gamma_i}, \mathbb{F}, n_{\gamma_i}\rangle$, and $\langle n_{\gamma_i}, \mathbb{F}, n_{p_i}\rangle$ abstract the reference edges between objects reachable from the parameter p_i at the beginning of the callee. For each p_i, define the edge rewrite sets:

$$\begin{aligned} J_{ij}^{pfp} &= \beta(\langle n_{p_i}, \mathbb{F}_f, n_{p_j}\rangle) \\ J_{ij}^{p\mathbb{F}p} &= \beta(\langle n_{p_i}, \mathbb{F}, n_{p_j}\rangle) \\ J_i^{pf\gamma} &= \beta(\langle n_{p_i}, \mathbb{F}_f, n_{\gamma_i}\rangle) \\ J_i^{p\mathbb{F}\gamma} &= \beta(\langle n_{p_i}, \mathbb{F}, n_{\gamma_i}\rangle) \\ J_i^{\gamma\gamma} &= \beta(\langle n_{\gamma_i}, \mathbb{F}, n_{\gamma_i}\rangle) \\ J_i^{\gamma p} &= \beta(\langle n_{\gamma_i}, \mathbb{F}, n_{p_i}\rangle) \end{aligned}$$

   The rewrite rules J_{ij}^{pfp}, $J^{p\mathbb{F}p}$, $J_i^{pf\gamma}$, $J^{p\mathbb{F}\gamma}$, $J_i^{\gamma\gamma}$, and $J_i^{\gamma p}$ capture, with respect to the corresponding edge's initial reachability state at the beginning of the callee, how the callee changed the reachability sets of the edges between the corresponding objects. The edge rewrite sets $J^{p\mathbb{F}p}$ and $J^{p\mathbb{F}\gamma}$ model parameter object edges from derived objects that the callee is unaware of.
4. **Upstream Caller Edge Reachability:** The reference edges $\langle q_i, n_{p_i}\rangle$ and $\langle r_i, n_{\gamma_i}\rangle$ abstract the caller reference edges upstream of parameter nodes n_{p_i} and n_{γ_i}, respectively. For each p_i, define the edge rewrite sets $K_i^p = \beta(\langle q_i, n_{p_i}\rangle)$ and $K_i^\gamma = \beta(\langle r_i, n_{\gamma_i}\rangle)$. The rewrite rules K_i^p and K_i^γ capture with respect to $\langle q_i, n_{p_i}\rangle$'s and $\langle r_i, n_{\gamma_i}\rangle$'s initial reachability states at the beginning of the callee how the callee changed the reachability sets of the upstream caller edges that can reach the objects in n_{p_i} and n_{γ_i}, respectively.
5. **Reachability States For Token $\eta_{n_{p_i}}$:** For each p_i, define:

$$d_i^p = \bigcup_{\langle a_i, n_{aij}\rangle} \beta(\langle a_i, n_{aij}\rangle) \cap \alpha(n_{aij})$$

   The set of reachability states d_i^p is a simple union of all reachability states on the reference edges out of label node a_i intersection with the reachability states of the nodes they reference in the caller. Conceptually, d_i^p represents all reachability states that are present on any parameter object heap region node referenced by a_i. This set provides a conservative approximation of the caller-context reachability states in heap region n_{p_i} before the callee's execution may change its reachability.
6. **Reachability States For Token $\eta_{n_{\gamma_i}}$:** For each p_i, define:

$$d_i^\gamma = \bigcup_{\langle a_i, n_{rij}\rangle} \beta(\langle a_i, n_{rij}\rangle)$$

   The set of reachability states d_i^γ is a simple union of all reachability states on the reference edges out of label node a_i in the caller. Conceptually, d_i^γ represents all reachability states that are present on any heap region node reachable from a_i. This set provides a conservative approximation of the caller-context reachability states in heap region n_{γ_i} before the callee's execution may change its reachability.
7. **Reachability States For Token $\eta^*_{n_{\gamma_i}}$:** For each p_i, define:

   Let $d_i^\gamma = \{\phi_{i0}, \ldots, \phi_{ij}\}$

$$D_i^\gamma = \begin{cases} \Big\{ s_{i0} \cup \cdots \cup s_{ij} \mid s_{ia} \in \\ \qquad \{\emptyset, \phi_{ia}, \phi_{ia} \cup_\triangle \phi_{ia}\} \Big\} & \text{if } \mid d_i^\gamma \mid < \omega_{\max} \\ \Big\{ \bigcup_{\triangle\ [\langle \eta_{i0}, \mu_{i0}\rangle, \ldots, \langle \eta_{ij}, \mu_{ij}\rangle] \in d_i^\gamma} [\eta^*_{i0}, \ldots, \eta^*_{ij}] \Big\} & \text{otherwise} \end{cases}$$

   If a parameter token appears in a reachability state with an arity of ZERO-OR-MORE, that token represents the tokens of any number of heap regions that are reachable from the parameter in the caller. We define D_i^γ to generate all possible combinations of caller reachability sets by taking any combination of the reachability states in d_i^γ any number of times.

   Note that the calculation of D_i^γ is intractable when the set d_i^γ is large. When the size of d_i^γ is greater than a threshold $\omega_{\max}$, we approximate the calculation with one reachability state that is union of every state in d_i^γ with token arity values all ZERO-OR-MORE.

   Conceptually, D_i^γ gives the possible caller reachability states that the callee token $\eta^*_{n_{p_i}}$ represents.
8. **Mapping Operator:** The Δ operator computes caller reachability states from callee reachability states with respect to parameter p_i. Δ takes as input (1) a set of callee-context rewrite rules R and (2) a map from tokens to a set of caller-context reachability states S and produces a set of caller-context reachability states.

$$\Delta(R, S) = \bigcup_{\{\langle \eta_0, \mu_0\rangle, \ldots, \langle \eta_j, \mu_j\rangle\} \in R} \Theta(\{\langle \eta_0, \mu_0\rangle, \ldots, \langle \eta_j, \mu_j\rangle\}), \text{where}$$

$$\Theta(\{\langle \eta_0, \mu_0\rangle, \ldots, \langle \eta_j, \mu_j\rangle\}) = \{\bigcup_{a=0}^{j} \tau_a \mid \tau_0 \in \Omega(\langle \eta_0, \mu_0\rangle), \ldots, \tau_n \in \Omega(\langle \eta_j, \mu_j\rangle)\}, \text{where}$$

$$\Omega(\langle\eta,\mu\rangle) = \begin{cases} S(\eta_{n_t}) & \text{if } \langle\eta,\mu\rangle = \eta_{n_t}, S(\eta_{n_t}) \neq \emptyset \\ d_b^p & \text{if } \langle\eta,\mu\rangle = \eta_{n_{p_b}}, S(\eta_{n_{p_b}}) = \emptyset \\ d_b^\gamma & \text{if } \langle\eta,\mu\rangle = \eta_{n_{\gamma_b}}, S(\eta_{n_{\gamma_b}}) = \emptyset \\ D_b^\gamma & \text{if } \langle\eta,\mu\rangle = \eta^*_{n_{\gamma_b}} \\ \{[\langle\eta_{n_{g'_{tz}}},\mu\rangle]\} & \text{if } \langle\eta,\mu\rangle = \eta^\mu_{n_{gtz}} \\ \{[\langle\eta,\mu\rangle]\} & \text{otherwise.} \end{cases}$$

5.4 Call Site Algorithm

This section presents the call site algorithm. The algorithm performs the following steps:

1. **Parameter Object Reachability:** Rewrite α for each caller heap region node n_{aij} referenced by argument label i.

$$\begin{aligned} \delta &= \Delta\Big(H_i^p, \{\langle\eta_{n_{p_i}}, \alpha(n_{aij})\rangle\}\Big) \\ \alpha'(n_{aij}) &= \alpha'(n_{aij}) \cup \delta. \end{aligned}$$

   It is possible for a caller heap region node to be reachable from two or more argument labels, therefore the calculation for α' iteratively adds the effects from each argument index.

2. **Other Caller Node Reachability:** Rewrite α for each caller heap region node n_{rij} reachable from argument label i.

$$\begin{aligned} \delta &= \Delta\Big(H_i^\gamma, \{\langle\eta_{n_{p_i}}, \{[]\}\rangle, \langle\eta_{n_{\gamma_i}}, \alpha(n_{rij})\rangle\}\Big), \\ \alpha'(n_{rij}) &= \alpha'(n_{rij}) \cup \delta. \end{aligned}$$

   It is possible for a caller heap region node to be reachable from two or more argument labels, therefore the calculation for α' iteratively adds the effects from each argument index. Note we remove the token $\eta_{n_{p_i}}$ because the caller context reachability states for nodes in n_{rij} already accounts for those nodes being reachable from the corresponding parameter object.

3. **Caller/Callee Edge Reachability:** Rewrite β for caller reference edges reachable from argument label i. We use a_i to represent the ith parameter object and r_i to represent a non-parameter object that is reachable from a_i.
   We split the edges into 4 cases:

   (a) **Edges from a_i to a_j:** For the edge $\langle n_{a_i}, f, n_{a_j}\rangle$ we select $J = J_{ij}^{pfp}$. $S = \{\langle\eta_{n_{p_j}}, \beta(\langle n_{i_0z_0}, f, n_{i_1z_1}\rangle)\rangle\}$.

   (b) **Edges from a_i to r_j:** For the edge $\langle n_{a_i}, f, n_{r_j}\rangle$ we select $J = J_i^{pf\gamma}$. $S = \{\langle\eta_{n_{\gamma_j}}, \beta(\langle n_{i_0z_0}, f, n_{i_1z_1}\rangle)\rangle\}$.

   (c) **Edges from r_i to a_j:** For the edge $\langle n_{r_i}, f, n_{a_j}\rangle$ we select $J = J_i^{\gamma p}$. $S = \{\langle\eta_{n_{p_j}}, \beta(\langle n_{i_0z_0}, f, n_{i_1z_1}\rangle)\rangle\}$.

   (d) **Edges from r_i to r_j:** For the edge $\langle n_{r_i}, f, n_{r_j}\rangle$ we select $J = J_i^{\gamma\gamma}$. $S = \{\langle\eta_{n_{\gamma_j}}, \beta(\langle n_{i_0z_0}, f, n_{i_1z_1}\rangle)\rangle\}$.

   Note that the last three cases only need a single index because $i \neq j$ implies that both i and j are in Π and therefore share the multiple-object heap region n_{γ_Π}.

$$\begin{aligned} \delta &= \Delta\Big(J, S\Big) \\ \beta'(\langle n_{i_0z_0}, f, n_{i_1z_1}\rangle) &= \beta'(\langle n_{i_0z_0}, f, n_{i_1z_1}\rangle) \cup \delta. \end{aligned}$$

   The process for reference edges reachable from argument labels is similar to the parameter-reachable heap region nodes above. Note that an edge can fall into multiple cases and the analysis simply applies the rules from all applicable cases.

4. **Upstream Caller Edge Reachability:** This step generates β' for caller reference edges upstream from the heap region nodes reachable from parameter i. We first give the updates for edges that point directly to the parameter object. For a parameter i in the aliasing context we split K_i^p into two parts: K_i^{p1} contains the reachability states that contain either $\eta_{n_{p_i}}$ tokens or $\eta_{n_{\gamma_i}}$ tokens. K_i^{p2} contains the reachability states that contain both tokens. For a parameter i that is not in the aliasing context, we set $K_i^{p1} = K_i^p$ and $K_i^{p2} = \emptyset$.
   We differentiate between the two aliasing cases because without aliasing, the caller reachability states for the upstream edges already account for reaching objects that are reachable from the $\eta_{n_{p_i}}$ token and/or a single $\eta_{n_{\gamma_i}}$ token. If the parameter i is in the aliasing context, the callee can introduce sharing between objects reachable from other aliased parameters in the heap region n_{γ_Π} and n_{p_i} that are not accounted for by the current caller reachability states. We include K_i^{p2} to account for this case.

$$\begin{aligned} \delta_{hp} &= \Delta\Big(K_i^{p1}, \{\langle\eta_{n_{p_i}}, \beta(\langle n, f, n_{aiz}\rangle)\rangle, \\ &\qquad \langle\eta_{n_{\gamma_i}}, \beta(\langle n, f, n_{aiz}\rangle)\rangle\}\Big) \\ \delta_{hp2} &= \Delta\Big(K_i^{p2}, \{\langle\eta_{n_{p_i}}, \beta(\langle n, f, n_{aiz}\rangle)\rangle\}\Big) \\ \beta'(\langle n, f, n_{aiz}\rangle) &= \beta'(\langle n, f, n_{aiz}\rangle) \cup \delta_{hp} \cup \delta_{hp2} \\ \delta_{lp} &= \Delta\Big(K_i^{p1}, \{\langle\eta_{n_{p_i}}, \beta(\langle l, n_{aiz}\rangle)\rangle, \\ &\qquad \langle\eta_{n_{\gamma_i}}, \beta(\langle l, n_{aiz}\rangle)\rangle\}\Big) \\ \delta_{lp2} &= \Delta\Big(K_i^{p2}, \{\langle\eta_{n_{p_i}}, \beta(\langle l, n_{aiz}\rangle)\rangle\}\Big) \\ \beta'(\langle l, n_{aiz}\rangle) &= \beta'(\langle l, n_{aiz}\rangle) \cup \delta_{lp} \cup \delta_{lp2} \end{aligned}$$

   We next give the updates for edges that reference objects reachable from the parameter object.

$$\begin{aligned} \delta_{h\gamma} &= \Delta\Big(K_i^\gamma, \{\langle\eta_{n_{p_i}}, \{[]\}\rangle, \\ &\qquad \langle\eta_{n_{\gamma_i}}, \beta(\langle n, f, n_{riz}\rangle)\rangle\}\Big) \\ \beta'(\langle n, f, n_{riz}\rangle) &= \beta'(\langle n, f, n_{riz}\rangle) \cup \delta_{h\gamma} \\ \delta_{l\gamma} &= \Delta\Big(K_i^\gamma, \{\langle\eta_{n_{p_i}}, \{[]\}\rangle, \\ &\qquad \langle\eta_{n_{\gamma_i}}, \beta(\langle l, n_{riz}\rangle)\rangle\}\Big) \\ \beta'(\langle l, n_{riz}\rangle) &= \beta'(\langle l, n_{riz}\rangle) \cup \delta_{l\gamma}. \end{aligned}$$

The analysis performs these updates only on the caller reference edges that directly reference the parts of the heap reachable from parameter objects. The analysis then uses the reachability change propagation algorithm for edges described in Section 4.4 to propagate the changes through upstream caller edges. Note that an upstream an edge can fall into multiple cases and the analysis simply applies the rules for all applicable cases.

5. **Callee Nodes:** This step generates reachability information for the callee nodes. It rewrites the callee reachability sets in terms of the caller reachability tokens.

$$\begin{aligned} \delta &= \Delta(\mathcal{R}(\alpha(n_{g_{tz}})), \{\}) \\ \alpha'(n_{g'_{tz}}) &= \delta \end{aligned}$$

When bringing callee-allocated heap region node $n_{g_{tz}} \in G_t$ into the caller context note that the analysis need not convert the $\mathcal{R}(\alpha(n_{g_{tz}}))$ with respect to any particular parameter index because the set of objects represented by $n_{g_{tz}}$ in the callee were newly allocated by the callee. Therefore, an empty mapping function is supplied to Δ to prevent rewriting parameter tokens. As a result, only cases 3, 4, and 5 of Ω will be used to convert the possible tokens in $\mathcal{R}(\alpha(n_{g_{tz}}))$.

We note that the reachability state for the token $\eta_{n_{\gamma_i}}$ already accounts for reachability from the parameter token $\eta_{n_{p_i}}$. Therefore, if both tokens appear on a callee node reachability state, $\mathcal{R}$ serves to prune the token $\eta_{n_{p_i}}$ if parameter i is not in the aliasing context Π.

$$\begin{aligned} \mathcal{R}(X) &= \{\mathbb{R}(x) \mid x \in X\} \\ \mathbb{R}(x = \{t_0, \ldots, t_j\} &= \{t'_0 \cup \ldots \cup t'_j \mid t'_i = \mathbb{C}(t_i, x)\} \\ \mathbb{C}(\langle \eta, \mu \rangle, x) &= \begin{cases} \emptyset & \text{if } \langle \eta, \mu \rangle = \eta_{n_{p_b}}, b \notin \Pi, \\ & (\eta_{n_{\gamma_b}} \in x \vee \eta^*_{n_{\gamma_b}} \in x) \\ \{\langle \eta, \mu \rangle\} & \text{otherwise} \end{cases} \end{aligned}$$

6. **Callee Edges:** This step generates reachability information for the new caller edges by rewriting the callee reachability sets in terms of the caller reachability tokens.

The algorithm first calculates β' for a new edge e_{caller} from some callee edge e.

$$\begin{aligned} \delta &= \Delta(\mathcal{R}(\beta(e)), \{\}) \\ \beta'(e_{\text{caller}}) &= \delta. \end{aligned}$$

In a similar manner to callee-allocated heap region nodes described above, callee-generated reference edges must calculate caller-context reachability. The calculation is given, but the next step describes the sets of possible caller reference edges that will be created, all of which will have this β information.

The next step computes the set of possible edges created by the callee in the new reachability graph. A callee reference edge e has either parameter heap region nodes or allocated heap region nodes for the source and destination, making four classes of callee reference edges. Each callee reference edge maps into the caller context as a set of edges:

$$S_{\text{src}} = \begin{cases} \{n_{g'_{tz}}\} & \text{if } \mathrm{src}(e){=}n_{g_{tz}} \text{ and } n_{g_{tz}} \\ & \text{has a field name matching } \mathrm{field}(e), \\ \{n_{ai0}, \ldots, n_{aij}\} & \text{if } \mathrm{src}(e){=}n_{p_i} \text{ and } n_{aij} \\ & \text{has a field name matching } \mathrm{field}(e), \\ \{n_{ri0}, \ldots, n_{rij}\} & \text{if } \mathrm{src}(e){=}n_{\gamma_i} \text{ and } n_{rij} \\ & \text{has a field name matching } \mathrm{field}(e). \end{cases}$$

$$S_{\text{dst}} = \begin{cases} \{n_{g'_{tz}}\} & \text{if } \mathrm{dst}(e){=}n_{g_{tz}} \text{ and } n_{g_{tz}}\text{'s} \\ & \text{type matches } \mathrm{field}(e) \text{ or is } n_\gamma, \\ \{n_{ai0}, \ldots, n_{aij}\} & \text{if } \mathrm{dst}(e){=}n_{p_i} \text{ and } n_{aij}\text{'s} \\ & \text{type matches } \mathrm{field}(e) \text{ or is } n_\gamma, \\ \{n_{ri0}, \ldots, n_{rij}\} & \text{if } \mathrm{dst}(e){=}n_{\gamma_i} \text{ and } n_{rij}\text{'s} \\ & \text{type matches } \mathrm{field}(e) \text{ or is } n_\gamma. \end{cases}$$

$E_{\text{caller}} = \{\langle s, \mathrm{field}(e), d \rangle, s \in S_{\text{src}}, d \in S_{\text{dst}}\}$

Note that some edge $e_{\text{existing}} = e_{\text{caller}} \in E_{\text{caller}}$ may exist in the caller context already if e is between two parameter regions in the callee. If the edge does not exist in the caller then add it with $\beta(e_{\text{caller}})$. Otherwise, $\beta(e_{\text{existing}}) = \beta(e_{\text{existing}}) \cup \beta(e_{\text{caller}})$.

7. **Update α, β:** The analysis next replaces α and β with the updated versions α' and β', respectively.
8. **Return Value Assignment:** If the call site assigns the return value to a caller label then the transform discussed in Section 4.2 is used to capture the effect. The analysis identifies all heap region nodes of the callee that are referenced by the label node `Return` that is out of the program scope and then it map that set of callee heap region nodes into the caller using mappings described above. From there the copy statement transform is trivial and can be committed in the midst of this larger call site transform.
9. **Summarizing Allocation Site Nodes:** The graph at this point may contain allocation site nodes for the same allocation site from both the caller and callee. The analysis summarizes the oldest nodes to ensure the abstract fits withing its normal k-limit.

5.5 Aliasing Contexts

We next discuss how the analysis creates the initial reachability graph for a given method aliasing context Π. It creates one multiple object heap region for all the parameters in the aliasing context Π and a single parameter object heap region for each parameter. In the mapping procedure, the node n_{γ_i} for each parameter in Π refers to the same shared node n_{γ_Π}.

The analysis then proceeds to generate all references between these regions that are allowed by the types of the parameters. It generates references between a parameter label and all of the single object parameter heap regions in the aliasing context that are allowed by type constraints. It sets

α for each node equal to a reachability state that just contains that node's token. It then adds these reachability states to all edges in the graph. Finally, it performs a global sweep to clean up the reachability information.

5.6 Mapping Strong Updates

Under two conditions the analysis can map strong updates to a parameter object from a callee to a caller. First when the parameter label node `a` is the only reference to the heap region node n_u. In this case, if the callee context has removed the original special edge for a field `f`, we can destroy all references from n_u with field `f` because no other label nodes can reach n_u.

The second condition for strong updates is when the parameter label node `a` references exactly one heap region node n_w and n_w is a single-object heap region. In this case, if the callee context is missing the original special edge for a field `f`, we can destroy all references from n_u with field `f` because `a` definitely refers to the object n_u.

5.7 Termination

Termination of the disjointness analysis is straightforward. There are only two complications: strong updates and call site aliasing contexts. All of the other transfer functions in the analysis are monotonic and the reachability graphs form a lattice.

While a strong update can be initially non-monotonic if it is processed before the variable on its left hand side is defined, we note that the strong update becomes (and remains) monotonic once the program variable on the left hand side is defined.

If adding a new edge changes the aliasing context for a call site, the new callee reachability graph may be only partially analyzed and therefore can contain fewer edges than the previous callee reachability graph. We note that once the final result is computed for the callee reachability graph for the new aliasing context, it will contain at least as many edges. For a given caller aliasing context, the analysis ensures that the aliasing context for a call site monotonically increases. Therefore, at some point the aliasing context for each call site will either include all parameter indices or stop increasing. At this point the analysis becomes monotonic and therefore terminates.

5.8 Discussion

Many heap analyses that attempt to extract more precise properties than pointer analysis attempt to extract shape properties. In general, extracting shape properties has proven difficult. Our analysis is designed to carefully avoid the difficult problem of reasoning about data structures shapes and to instead extract disjointness properties for data structures.

We note that pointer analysis in some circumstances can extract disjointness information for a set of statically named data structures. Disjointness analysis was designed to maintain reachability annotations for heap nodes and therefore can reason about the mutual disjointness of an unbounded number of data structures.

We have found that many programs use object fields to temporarily store references across method calls. To maintain precise reachability information, it is critical to perform strong updates on the parameter object. We have designed the initial method context to include two nodes for each parameter: the single object parameter node that abstracts the parameter object and the multiple object parameter node abstracts the objects reachable from the parameter node.

This context also enables the analysis to maintain precision for many common paradigms. For example, disjointness analysis can precisely handle reachability information for methods that takes a linked list node as a parameter object and remove the next linked list node or splice in a new linked list node. The insight is that the analysis is able to remember that the reachability state for the multiple object parameter node already accounts for reachability from the single object parameter node.

We note the analysis loses precision if a data structure that may point to many flagged objects is passed as a parameter to a method that generates new references in this data structure. This remains a topic for further research.

6. Definite Reachability Extension

When the analysis processes code that updates summarized data structures, it can lose precision. For example, removing an object from a linked list can cause the analysis to lose information about how many instances of a summarized flagged object can reach objects in the linked list. We present a definite reachability extension that allows the analysis to reason about reachability information between variables. The idea is to keep track of what reachability is already accounted for by the reachability graph at the variable level, so that the analysis can avoid performing redundant updates that degrade the precision of the analysis results.

The definite reachability analysis computes the relation $R := L \times L \times E$ that maps pairs of variables to a set of edges. The relation R maps a pair of variables if the reachability graph already accounts for the first variable pointing to an object that is reachable from the object pointed at by the second variable. The set of edges conservatively gives the edges along the path from the object referenced by the second variable to the object referenced by the first variable.

The partial function $R_s : L \rightarrow \{\texttt{unknown}, \texttt{new}\}$ keeps track of whether the definite reachability analysis contains all reachability information for an object since the object's allocation. The relation $\mathcal{F}_u : L \times (L \cup \{\texttt{unknown}\})$ maps a label that points to an object to the set of labels for all objects that point to the first object. The relation $\mathcal{F}_d : L \times F \times L$ maps a label that points to an object and a field name to the label of the second object that the field of the first object points at.

- **Method Entry:** At the beginning of a method with the set of parameters P:

$$\begin{aligned} R' &:= \{\} && (29)\\ R'_s &:= P \times \{\texttt{unknown}\} && (30)\\ \mathcal{F}'_u &:= \{\} && (31)\\ \mathcal{F}'_d &:= \{\} && (32) \end{aligned}$$

- **Load Statement:** At the end of a load statement of the form `x = y.f`, the reachability graph definitely accounts for the object referenced by `y` and objects that can reach this object reaching the object referenced by `x`. Nothing is known about what the fields of the object pointed to by `x` reference.

$$\begin{aligned} R' &:= (R - \langle \texttt{x}, *, * \rangle - \langle *, \texttt{x}, * \rangle) \cup \\ &\quad (\{\langle \texttt{x}, \texttt{y} \rangle\} \times E^{\circ}(\texttt{y}, \texttt{f})) \cup \\ &\quad \bigcup_{\langle \texttt{y}, z, e \rangle \in R} \{\langle \texttt{x}, z \rangle\} \times (E^{\circ}(\texttt{y}, \texttt{f}) \cup \{e\}) && (33)\\ R'_s &:= (R_s - \langle \texttt{x}, * \rangle) \cup \{\langle \texttt{x}, \texttt{unknown} \rangle\} && (34)\\ \mathcal{F}'_u &:= \mathcal{F}_u - \langle \texttt{x}, * \rangle - \langle *, \texttt{x} \rangle \cup \\ &\quad \{\langle z, \texttt{unknown} \rangle \mid \langle z, \langle x \rangle \rangle \in \mathcal{F}_u\} && (35)\\ \mathcal{F}'_d &:= \mathcal{F}_d - \langle \texttt{x}, *, * \rangle - \langle *, *, \texttt{x} \rangle && (36) \end{aligned}$$

- **Copy Statement:** At the end of a copy statement of the form `x = y`, the object reference by `x` is definitely reachable from any objects that can definitely reach `y`. The analysis simply copies the information it had maintained for `y` to `x`.

$$\begin{aligned} R' &:= (R - \langle \texttt{x}, *, * \rangle - \langle *, \texttt{x} \rangle, *) \cup \\ &\quad \{\langle \texttt{x}, z, e \rangle \mid \langle \texttt{y}, z, e \rangle \in R\} \cup \\ &\quad \{\langle z, \texttt{x}, e \rangle \mid \langle z, \texttt{y}, e \rangle \in R\} && (37)\\ R'_s &:= (R_s - \langle \texttt{x}, * \rangle) \cup \\ &\quad \{\langle \texttt{x}, v \rangle \mid \langle \texttt{y}, v \rangle \in R_s\} && (38)\\ \mathcal{F}'_u &:= (\mathcal{F}_u - \langle \texttt{x}, * \rangle - \langle *, \texttt{x} \rangle) \cup \\ &\quad \{\langle \texttt{x}, v \rangle \mid \langle \texttt{y}, v \rangle \in \mathcal{F}_u\} \cup \\ &\quad \{\langle v, \texttt{x} \rangle \mid \langle v, \texttt{y} \rangle \in \mathcal{F}_u\} \cup \\ &\quad \{\langle z, \texttt{unknown} \rangle \mid \langle z, \langle x \rangle \rangle \in \mathcal{F}_u\} && (39)\\ \mathcal{F}'_d &:= (\mathcal{F}_d - \langle \texttt{x}, *, * \rangle - \langle *, *, \texttt{x} \rangle) \cup \\ &\quad \{\langle \texttt{x}, f, z \rangle \mid \langle \texttt{y}, f, z \rangle \in \mathcal{F}_d\} \cup \\ &\quad \{\langle z, f, \texttt{x} \rangle \mid \langle z, f, \texttt{y} \rangle \in \mathcal{F}_d\} && (40) \end{aligned}$$

- **Store Statements:** A store statement of the form `x.f = y` has two effects on definite reachability: 1) it may produce strong updates that remove the set of edges $E^{\texttt{remove}}$ and therefore change the definitely reachability between previous variables and 2) it makes `y` definitely reachable from `x`.

$$\begin{aligned} R' &:= (R - \{\langle w, z, e \rangle \mid \langle w, z, e \rangle \in R, \\ &\quad \forall \langle w, z, e' \rangle \in R, e' \notin E^{\texttt{remove}}\}) \cup \\ &\quad \{\langle \texttt{y}, \texttt{x}, e \rangle \mid e \in E(\texttt{x}) \times \{\texttt{f}\} \times E(\texttt{y})\} && (41)\\ R'_s &:= R_s && (42)\\ \mathcal{F}'_u &:= \mathcal{F}_u \cup \{\langle \texttt{y}, \texttt{x} \rangle\} && (43)\\ \mathcal{F}'_d &:= \mathcal{F}_d \cup \{\langle \texttt{x}, \texttt{f}, \texttt{y} \rangle\} && (44) \end{aligned}$$

- **Object Allocation Statement:** A new object statement of the form `x=new` points the variable `x` at a newly allocated object. The analysis clears definitely reachable information for `x` from R, marks `x` as a newly allocated object in R_s, marks objects that pointed to the old `x` as pointing to an unknown object in $\mathcal{F}_u$, and marks the object `x` as referencing no other objects in $\mathcal{F}_d$.

$$\begin{aligned} R' &:= R - \langle \texttt{x}, *, * \rangle - \langle *, \texttt{x}, * \rangle && (45)\\ R'_s &:= (R_s - \langle \texttt{x}, * \rangle) \cup \{\langle \texttt{x}, \texttt{new} \rangle\} && (46)\\ \mathcal{F}'_u &:= \mathcal{F}_u - \langle \texttt{x}, * \rangle - \langle *, \texttt{x} \rangle \cup \\ &\quad \{\langle z, \texttt{unknown} \rangle \mid \langle z, \langle x \rangle \rangle \in \mathcal{F}_u\} && (47)\\ \mathcal{F}'_d &:= \mathcal{F}_d - \langle \texttt{x}, *, * \rangle - \langle *, *, \texttt{x} \rangle && (48) \end{aligned}$$

- **Method Calls:** A method call of the form `x=method(...)` points `x` to the return value of the callee. The analysis clears the old information for `x`, marks the results for `x` as incomplete in D_s, and clears information about reference to `x` and references from `x` in $\mathcal{F}_u$ and $\mathcal{F}_d$, respectively.

$$\begin{aligned} R' &:= R - \langle \texttt{x}, *, * \rangle - \langle *, \texttt{x}, * \rangle && (49)\\ R'_s &:= (R_s - \langle \texttt{x}, * \rangle) \cup \{\langle \texttt{x}, \texttt{unknown} \rangle\} && (50)\\ \mathcal{F}'_u &:= \mathcal{F}_u - \langle \texttt{x}, * \rangle - \langle *, \texttt{x} \rangle \cup \\ &\quad \{\langle z, \texttt{unknown} \rangle \mid \langle z, \langle x \rangle \rangle \in \mathcal{F}_u\} && (51)\\ \mathcal{F}'_d &:= \mathcal{F}_d - \langle \texttt{x}, *, * \rangle - \langle *, *, \texttt{x} \rangle && (52) \end{aligned}$$

- **Join Points:** The analysis uses the following formula to merge definite reachability information at control flow join points.

$$\begin{aligned} R' &:= \{\langle x, y, e \rangle \mid \exists i. \langle x, y, e \rangle \in R_i, \\ &\quad \exists e_1 \in E_1, ..., e_n \in E_n. \\ &\quad \langle x, y, e_1 \rangle \in R_1, ..., \langle x, y, e_n \rangle \in R_n\} && (53)\\ R'_s &:= \{\langle x, \texttt{new} \rangle \mid \langle x, \texttt{new} \rangle \in R_{s1}, ..., \\ &\quad \langle x, \texttt{new} \rangle \in R_{sn}\} \cup \{\langle x, \texttt{unknown} \rangle \mid \\ &\quad \forall \langle x, v \rangle \in R_i, \exists i'. \langle x, \texttt{new} \rangle \notin R_{i'}\} && (54)\\ \mathcal{F}'_u &:= \bigcup_{0 \le i < n} \mathcal{F}_{u_i} && (55)\\ \mathcal{F}'_d &:= \bigcup_{0 \le i < n} \mathcal{F}_{d_i} && (56) \end{aligned}$$

We define the helper function $\mathrm{downstream}(x, y)$ to indicate whether the reachability graph already captures that the object referenced by x is reachable from the object referenced by y.

$$\mathrm{downstream}(x, y) = \exists e. \langle x, y, e \rangle \in R \tag{57}$$

We define the helper function $\mathrm{new}(x)$ to indicate whether the definite reachability analysis contains information about the object reachable from x.

$$\mathrm{new}(x) = \langle x, \texttt{new} \rangle \in R_s \tag{58}$$

We next describe how definite reachability information can be used to generate more precise updates for store operations for several scenarios:

- **Downstream Stores:** If there is a store `x.f=y` where $\mathrm{downstream}(\texttt{y}, \texttt{x})$, the reachability graph already reflects that `y` can be reached from `x`. Instead, it suffices to set $\forall n_{\texttt{x}} \in E(\texttt{x}), n_{\texttt{y}} \in E(\texttt{y}), \beta'(\langle n_{\texttt{x}}, \texttt{f}, n_{\texttt{y}} \rangle) = \beta(\langle \texttt{y}, n_{\texttt{y}} \rangle)$ and $\beta'(e) = \beta(e)$ otherwise.
- **References to New Objects:** If there is a store `x.f=y` where $\mathrm{new}(\texttt{y})$ and $\forall \langle \texttt{y}, \texttt{f}', z \rangle \in \mathcal{F}_d. \mathrm{downstream}(z, \texttt{x})$, it is not necessary to propagate changes across `y`'s $\texttt{f}'$ field nor propagate information from $\texttt{f}'$ backwards as the reachability graph already reflects that `x` can reach `y.f`$'$.
- **References from New Objects:** If there is a store `x.f=y` where $\mathrm{new}(\texttt{x})$, $\langle \texttt{x}, \texttt{unknown} \rangle \notin \mathcal{F}_u$, and $\forall \langle \texttt{x}, z \rangle \in \mathcal{F}_u, \mathrm{downstream}(\texttt{y}, z)$, the reachability graph already reflects that all objects that point to the object referenced by `x` can reach the object referenced by `y`. Instead it suffices to set $\forall n_{\texttt{x}} \in E(\texttt{x}), n_{\texttt{y}} \in E(\texttt{y}), \beta(\langle n_{\texttt{x}}, \texttt{f}, n_{\texttt{y}} \rangle = \beta(\langle \texttt{y}, n_{\texttt{y}} \rangle)$ and $\beta'(e) = \beta(e)$ otherwise.

7. Evaluation

We have implemented disjointness analysis in our compiler. We have analyzed several applications written in Bamboo (Zhou and Demsky 2009). Bamboo extends Java with a set of task extensions designed for parallel programming. Bamboo can execute tasks in parallel if it can determine that the two tasks operate on disjoint parts of the heap. Bamboo uses a similar task invocation model to the Bristlecone language (Demsky and Dash 2008) — the runtime task invokes tasks when there exists objects in the heap in the appropriate states to serve as parameter objects. Bamboo task invocation locks on the parameter objects — therefore, showing that the flagged objects reach disjoint parts of the heap is sufficient to execute tasks in parallel.

Bamboo programs are a natural choice for benchmarks as the parameter objects are typically intended to be disjoint and therefore provide a ready source of programs with flagged allocation sites. Note the Bamboo's task semantics mean that parameter objects even without explicit references are live, and therefore the first strong update condition does not apply to those heap reference nodes.

We ran the analysis on 17 benchmarks on a 2.33 Xeon with 4 GB of RAM. The benchmarks were all ported to Bamboo. The source code for the analysis and the benchmarks can be downloaded from `http://demsky.eecs.uci.edu/bamboo/`.

7.1 Benchmarks

We evaluated the analysis on the following benchmarks:

- **jHTTPp2:** jHTTPp2 is an open source HTTP Proxy server. The jHTTPp2 benchmark was taken from `http://jhttp2.sourceforge.net/`.
- **JGFMonteCarlo:** The JGFMonteCarlo benchmark is an implementation of a Monte Carlo simulation. It first computes the mean fluctuation from a series of historical data and then generates multiple sample time series with the same mean fluctuation. It was taken from the Java Grande benchmark suite (Smith et al. 2001).
- **JGFMolDyn:** The JGFMolDyn benchmark models particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. It was taken from the Java Grande benchmark suite (Smith et al. 2001).
- **JGFSeries:** The JGFSeries benchmark computes Fourier coefficients. It was taken from the Series benchmark from the Java Grande benchmark suite (Smith et al. 2001).
- **FluidAnimate:** This application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. The FluidAnimate benchmark was taken from the Parsec benchmark suite (Bienia et al. 2008).
- **Filterbank:** FilterBank is a multi-channel filter bank for performing multi-rate signal processing. FilterBank performs a down-sample, followed by an up-sample, in each channel. Finally, the benchmark combines the results for all channels. It was taken from the StreamIt benchmark suite (Gordon et al. October, 2002).
- **Multiplayer Game:** The games consists of a world with both humans and monsters. The goal of the humans is to escape through exits while the monsters try to capture the humans. The game is structured into a map component that maintains the world and AI components that use search algorithms to plan moves for both the monsters and the humans.
- **Fractal:** Fractal computes a Mandelbrot set.
- **Chat:** Chat is a multi-room chat server.
- **MapReduce:** MapReduce implements a simplified version of the MapReduce parallel programming model (Dean and Ghemawat 2004).
- **Bank:** Bank implements a simple banking application.
- **Web Portal:** The web portal collects information from various online data sources and assembles it into a web page.

Benchmark	Sharing	Time	Lines
jHTTPp2	0	2.05s	2,548
JGFMonteCarlo	0	1.02s	3,522
JGFMoldyn	2	4.38s	1,874
JGFSeries	0	0.30s	1,366
FuildAnimate	2	233s	3,544
FilterBank	0	0.40s	1,293
MultiGame	10	19.80s	2,968
Fractal	1	0.44s	1,702
Chat	3	1.93s	1,482
MapReduce	2	5.19s	2,239
Bank	0	1.72s	1,782
WebPortal	0	1.27s	1,936
PERT	0	0.91s	1,907
Spider	0	2.49s	1,565
TileSearch	0	2.35s	2,022
TicTacToe	0	0.93s	1,489
WebServer	0	2.34s	1,813

Table 2. Benchmark Results

- **PERT:** The Program Evaluation and Review Technique (PERT) estimator implements a PERT model, which is widely used in project managing.
- **Spider:** The web spider takes an initial URL as input and crawls the web starting at the initial URL.
- **TileSearch:** TileSearch solves tile puzzle problems. The problem is to discover a tile arrangement that produces a maximal score for arranging square tiles with values on each face such that adjacent faces have equal value and the contiguous arrangement scores the summation of faces left open.
- **TicTacToe:** TicTacToe is an online tic-tac-toe game server.
- **WebServer:** Webserver implements a webserver.

7.2 Disjointness Results

Table 2 presents the analysis results and execution times for our benchmark suite. The analysis identified a total of 20 possible aliases between flagged object classes over six of the benchmarks. The other eleven benchmarks were reported to have disjoint regions reachable from flagged objects. We verified that the analysis results were correct by manual inspection of the code. We observed imprecise sharing results only for the MapReduce benchmark. We discuss the benchmarks in more detail in Section 7.3.

We note that the number of aliases between flagged objects is relatively small as Bamboo applications were written to allow parallelization and therefore ensure that the parameter objects are disjoint. The alias reports for flagged objects were generated at the exit of each task invocation.

Table 2 presents the analysis times and lengths for the benchmark suite. Most of the benchmarks took only a few seconds to analyze. The benchmarks ranged from 1,293 to 3,544 lines of code.

7.3 Discussion

We next discuss the sharing that the disjointness analysis discovered.

- **JGFMolDyn:** JGFMoldyn parallelizes the computation into several pieces. The pieces share some global state they must all access, and this sharing is detected by the analysis.
- **FluidAnimate:** FluidAnimate partitions a grid of cells. Each partition keeps references to the neighboring edge cells. The analysis reports this sharing.
- **Multiplayer Game:** The analysis correctly detects that human and monster objects share a global map object in order to compute moves.
- **Fractal:** The computation is divided into sub-problems that are then gathered into a final image object where sharing is detected.
- **Chat:** Chat shares socket objects in several places. Messages contain a reference to the originating socket to avoid echoing the message back to the sender. Chat room objects also share socket objects in order to link users in the same room together.
- **MapReduce:** MapReduce necessarily shares heap references between the master and the map workers and reduce workers and this sharing is reported. However, our analysis reports false sharing between reduce workers because our imprecise modeling of arrays cannot discover that each reduce worker is given a disjoint element of the master's intermediate output array.

8. Related Work

Like disjointness analysis, both alias analysis (Banning 1979; Cooper and Kennedy 1989; Diwan et al. 1998; Ruf 1997) and pointer analysis (Shapiro and Horwitz 1997; Landi et al. 1993; Weihl 1980; Burke et al. 1995) analyze source code to discover heap referencing properties of the data structures that applications build. However, disjointness analysis extracts a different property — disjointness analysis attempts to determine whether the parts of the heap reachable from distinct objects taken from a selected set are disjoint. Our analysis can determine that distinct objects from the same static representation or name reach disjoint parts of the heap. We extract a similar properties to the conditional must not aliasing analysis by Naik and Aiken (Naik and Aiken 2007), however our analysis can maintain disjointness properties in the presence of mutation.

Connection analysis discovers which heap-directed pointers may reach a common data structure (Ghiya and Hendren 1996a). 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 are mutually disjoint.

Alias analyses vary in whether they are *flow-sensitive* (Choi et al. 1993) or *context-sensitive* (Wilson and Lam 1995; Emami et al. 1994; Steensgaard 1996). These design choices incur increased analysis complexity to gain increased precision. Our analysis is flow-sensitive and context-sensitive in an effort to produce a sound result with enough precision to maintain disjointness properties for real programs. We mitigate the algorithm's complexity by reducing the targets of reachability to only objects of interest, with the expectation that more precision will dramatically increase how effectively programs are parallelized.

Ruf (Ruf 1995) suggests that for alias analysis the benefit of context-sensitivity is rare over context-insensitive analyses. Lhotak (Lhoták and Hendren 2006) suggests that context sensitivity for pointer analysis can be helpful in some contexts. We expect that context sensitivity is more important for disjointness analysis. Without context sensitivity, simply passing two flagged parameter objects into a method that points the field of one these objects to any object would violate the disjointness property.

The literature also proposes a variety of methods for modeling structure references. Some use a k-limited approach of keeping k distinct objects in a recursive structure or from an allocation site before summarizing (Landi et al. 1993; Choi et al. 1993). Other strategies are to use symbolic access paths (Deutsch 1994) or regular expressions (Hummel et al. 1994). We create k distinct heap regions for objects generated from allocation sites and then summarize, but our reachability states maintain precision for objects within summarized heap regions. By combining the reachability information of the summarized heap region and its incoming and outgoing references we can still know the disjointness properties of different classes of objects within the summary region.

Escape analysis (Blanchet 1999; Whaley and Rinard 1999) tracks when heap elements have escaped their static scope. The computations derive different program information, but often use similar analysis techniques.

Ownership type systems have been developed to restrict aliasing of heap data structures (Aldrich et al. 2002; Boyapati et al. 2002; Clarke 2003; Clarke and Drossopoulou 2002; Clarke et al. 1998; Heine and Lam 2003). We only make similar observations when pruning method effects that are being mapped into the calling context.

Shape analysis (Chase et al. 1990; Ghiya and Hendren 1996b; Sagiv et al. 2002; McPeak and Necula 2005) discovers and verifies shape heap properties of data structures. Our analysis differs in an important way: shape analysis can verify that some object is the root of a valid tree, while our analysis can verify that trees are disjoint by inspecting their roots, but not that they are in fact trees.

9. Conclusion

If a compiler can determine that two blocks of code operate on disjoint data structures, it can safely parallelize them. Traditional pointer analysis has difficulty reasoning about disjointness properties for objects that are represented by the same node. We present a new analysis, disjointness analysis, for extracting disjointness properties from single-threaded code. The analysis uses the abstraction of reachability sets to maintain reachability information even between multiple objects from the same allocation site.

We have implemented the analysis in our compiler framework and analyzed 17 benchmark programs written in Bamboo, a set of task-based extensions to Java. The analysis identified all sharing between key data structures in our benchmark programs.

References

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1979.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, 1995.

David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990.

Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. *ACM SIGPLAN Notices*, 33

(10):48–64, 1998.

David Gerard Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2003.

David Gerard Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

K. D. Cooper and K. Kennedy. Fast interprocedual alias analysis. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Operating Systems Design and Implementation*, 2004.

Brian Demsky and Alokika Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 2008 European Conference on Object-Oriented Programming*, 2008.

Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, 1998.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

Rakesh Ghiya and Laurie J. Hendren. Connection analysis: a practical interprocedural heap analysis for C. *International Journal on Parallel Programming*, 24(6):547–578, 1996a. ISSN 0885-7458.

Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996b.

Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.

David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993.

Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the 15h International Conference on Compiler Construction*, March 2006.

S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Computer-Aided Verification*, 2005.

Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995.

Erik Ruf. Partitioning dataflow analyses using types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.

Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

L. A. Smith, J. M. Bull, and J. Obdrzalek. A parallel Java Grande benchmark suite. In *Proceedings of SC2001*, 2001.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1980.

John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995.

Jin Zhou and Brian Demsky. Bamboo: A data-centric, object-oriented approach to multi-core software. Concurrent submission to OOPSLA, available from `http://demsky.eecs.uci.edu/bamboo.pdf`, 2009.