Disjointness Analysis for Java-Like Languages

James C. Jenista
University of California, Irvine
jjenista@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

February 2009

ISR Technical Report # UCI-ISR-09-1

Disjointness Analysis for Java-Like Languages

James C. Jenista, Brian Demsky
University of California, Irvine

Institute for Software Research
UCI-ISR-09-1

February 2009

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. Our evaluation shows that the analysis reported all known aliases and no false aliases for our benchmark suite.

Disjointness Analysis for Java-Like Languages

James C. Jenista and Brian Demsky

University of California, Irvine
Institute for Software Research

UCI-ISR-09-1
February 2009

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. Our evaluation shows that the analysis reported all known aliases and no false aliases for our benchmark suite.

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 [2, 11, 14, 24], pointer analysis [26, 20, 29, 5], and shape analysis [6, 16, 25] 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 analyzes a given 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.

The analysis results are useful for determining whether code can be parallelized. For example, to execute two serial method calls in parallel, it is necessary to determine that they do not have any conflicting data structure accesses. If our analysis determines that all of the parameter objects of the two calls are mutually disjoint, checking that the same object does not serve as a parameter for both method calls suffices to ensure that the two calls can be safely executed in parallel.

1.1 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.

1.2 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 evaluation of the analysis on several benchmarks. Section 7 presents related work; we conclude in Section 8.

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 first loop are disjoint from objects reachable from other `Graph` objects allocated in that 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.

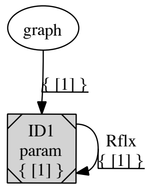
```
 1  public void makeGraph(Graph graph) {
 2    Node s = new Node();
 3    Node t = new Node();
 4    s.f = t;
 5    t.f = s;
 6    graph.node = s;
 7  }
 8
 9  public void graphLoop() {
10    Graph[] a = new Graph[nGraphs];
11    for(int i=0; i<nGraphs; i++) {
12      Graph g = new Graph();
13  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

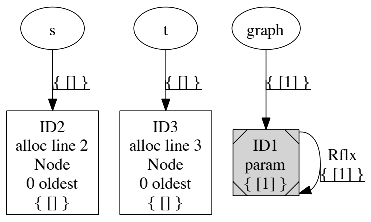
We begin with an intraprocedural analysis of the `makeGraph` method. Figure 2(a) presents the analysis results for the example at the beginning of the `makeGraph` method. The ellipse labeled `graph` represents the parameter variable `graph`. The rectangular heap region node labeled `ID1` represents the 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 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 are reachable from heap region node `ID1`. The reflexive edge on the heap region node labeled `ID1` indicates that objects in this heap region can reference other objects in the same heap region. The set $\{[1]\}$ on this edge indicates that the edge models a heap reference that can reach objects in the reachability state $[1]$. Similarly, the edge from the label node `graph` to the heap region node `ID1` indicates that the label `graph` may point to an object in the heap region `ID1`.

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 `ID2` and `ID3`, respectively. These heap regions are associated with the allocation site that allocated the objects. Heap regions `ID2` and `ID3` 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.

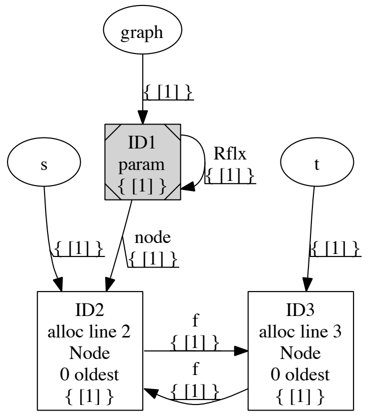
As in Figure 2(a), the heap region node `ID1` is shaded. Shaded heap regions are *flagged*; heap regions are flagged only when they contain objects in which the analysis client is interested in finding



(a) Reachability graph after line 1



(b) Reachability graph after line 3



(c) Reachability graph after line 6

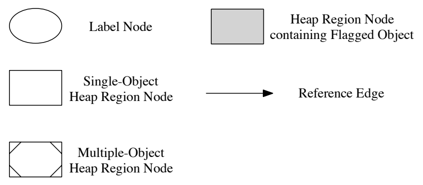

Label Node
Heap Region Node containing Flagged Object
Single-Object Heap Region Node
Reference Edge
Multiple-Object Heap Region Node

Figure 2. Intraprocedural Reachability Graphs

disjointness information about. The empty reachability state in the set of reachability states for heap regions `ID2` and `ID3` implies that no objects from flagged heap regions can reach heap regions `ID2` and `ID3` 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 $\langle \texttt{ID2}, \texttt{f}, \texttt{ID3} \rangle$, $\langle \texttt{ID3}, \texttt{f}, \texttt{ID2} \rangle$, and $\langle \texttt{ID1}, \texttt{node}, \texttt{ID2} \rangle$, 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.

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 `ID4`, 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.

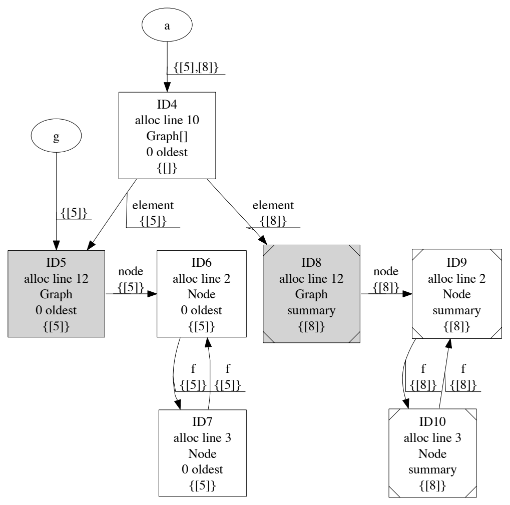


Figure 3. Reachability graph after line 15

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 $\langle$`ID8`, `node`, `ID9`$\rangle$ could represent references from disjoint pairs of objects from heap regions `ID8` and `ID9` or it could represent more than one object from the heap region `ID8` referencing the same object from `ID9`. By maintaining a set of reachability states for heap region `ID9` it is clear that any object in that region is reachable only from exactly one object in heap region `ID8`. Therefore, this reachability graph implies that `Graph` objects from the allocation site in line 12 reference disjoint heap regions.

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`
- $\mathrm{parents}($`st`$)$ is the set of statements that may flow to `st`
- $\mathrm{children}($`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.

$$n \in N := \texttt{Allocation sites} \times \{0, 1, \ldots, k\} \cup N_P \tag{1}$$
$$n_p \in N_P := \texttt{Parameter nodes for the method}\, m \tag{2}$$

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

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

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

$$f \in F := \texttt{Fields} \cup [] \cup \mathbb{F} \tag{4}$$

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.

$$E \subseteq N \times F \times N \cup L \times N \tag{5}$$

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

$$\begin{aligned}
E(l) &= \{n \mid \langle l, n\rangle \in E\} && (6)\\
E^{\circ}(l) &= \{\langle l, n\rangle \mid \langle l, n\rangle \in E\} && (7)\\
E(n) &= \{n' \mid \langle n, f, n'\rangle \in E\} && (8)\\
E(l, f) &= \{n' \mid \langle l, n\rangle, \langle n, f, n'\rangle \in E\} && (9)\\
E^{\circ}(l, f) &= \{\langle n, f, n'\rangle \mid \langle l, n\rangle, \langle n, f, n'\rangle \in E\} && (10)
\end{aligned}$$

3.3 Reachability Annotations

This sections 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 $\{$`ZERO`, `ONE`, `ZERO-OR-MORE`, `ONE-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^* and η^+ for the arities `ZERO-OR-MORE` and `ONE-OR-MORE`, respectively. 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 exactly one object in heap region n_1, one-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

$$\begin{aligned} \beta(l) &= \{\phi \mid \langle\langle l, n\rangle, \phi\rangle \in \beta\}, & (12)\\ \beta^\circ(l) &= \{\langle\langle l, n\rangle, \phi\rangle \mid \langle\langle l, n\rangle, \phi\rangle \in \beta\}, & (13)\\ \beta(e) &= \{\phi \mid \langle e, \phi\rangle \in \beta\}. & (14) \end{aligned}$$

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

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 $\text{entry}(m)$ 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 $\text{parents}(s)$ 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 $\text{children}(s)$ for analysis.

4.1 Method Entry

The parameter information for a method m is contained in a special statement that is always $\text{entry}(m)$. We first describe how the analysis analyzes methods whose parameters are disjoint. We then extend the approach to handle aliases between parameter objects. The transform for this statements creates, for each parameter p_i, a new multiple-object heap region node n_{p_i}. 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. Finally, a reference edge $\langle n_{p_i}, \mathbb{F}, n_{p_i}\rangle$ marked as reflexive is added. This reference edge models an arbitrary internal structure for the heap objects reachable from label p_i.

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_{p_Π} for all parameters that may be aliased. The label nodes for each parameter in Π all initially point to n_{p_Π}. The node n_{p_Π} contains the special label q_Π and the reflexive edge as described above.

The statement $\text{entry}(m)$ 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.

$$\begin{aligned} E' &= (E - E^\circ(\mathtt{x})) \cup (\{\mathtt{x}\} \times E(\mathtt{y})) & (15)\\ \beta' &= (\beta - \beta^\circ(\mathtt{x})) \cup (\{\mathtt{x}\} \times \beta(\mathtt{y})) & (16) \end{aligned}$$

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.

$$\begin{aligned} E' &= (E - E^\circ(\mathtt{x})) \cup (\{\mathtt{x}\} \times E(\mathtt{y}, \mathtt{f})) & (17)\\ \beta' &= (\beta - \beta^\circ(\mathtt{x})) \cup \\ & \bigcup_{\langle n, \mathtt{f}, n'\rangle \in E^\circ(\mathtt{y}, \mathtt{f})} \Big(\{\langle \mathtt{x}, n'\rangle\} \times \Big(\beta(\langle \mathtt{y}, n\rangle) \sqcap \beta(\langle n, \mathtt{f}, n'\rangle)\Big)\Big) & (18) \end{aligned}$$

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.

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

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 with whose reachability changes.

$\cup_\blacktriangle$	0	1	+	*
0	0	1	+	*
1	1	+	+	+
+	+	+	+	+
*	*	+	+	*

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

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 $O = \beta(\langle \mathtt{y}, n_{\mathtt{y}} \rangle)$.
- 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

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

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

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

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

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

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

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

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:

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

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

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

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

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

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

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 $\mu \in \{\texttt{ONE}, \texttt{ONE-OR-MORE}\}$, 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 $\mathcal{B}$ as a set of tuples. $\mathcal{B}$ satisfies the constraints: $\forall e \in E(n_f), \mathcal{B}(e) \supseteq \beta(e)$ and $\forall e \in E, e' \in E(dst(e))\mathcal{B}(e') \supseteq \beta(e') \cap \mathcal{B}(e)$. It uses fixed point algorithm to propagate reachability information to solve the constraints. Finally, for each node n the analysis prunes from each α all reachability sets that contain either η_{n_f} or $\eta^+_{n_f}$ but do not appear in the $\mathcal{B}(e)$ of any edge e incident to n. Note the exception that this step should not prune the reachability state $[n_f]$ from $\alpha(n_f)$ of the flagged node n_f. The analysis next computes reachability from the next flagged node.

The second phase uses the now internally consistent α' information and the β 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 $\beta' = \{\beta(e) \cap \alpha'(n) \mid \forall e \in E, n = dst(e)\}$. The analysis then propagates reachability information backwards to satisfy the constraint: $\beta'(e) \supseteq \beta(e) \cap \beta'(e')$ 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 the 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 back into 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. Conceptually 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. We execute this update conservatively by restricting precision such that results apply for all invocations possible at `cs`.

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 heap regions, $\{n_{i0}, \ldots, n_{ij_i}\}$ in the caller.
- The ith parameter of the callee has a label node p_i in the callee reachability graph referencing a multi-object heap region n_{p_i}.
- A special, label q_i out of program context references n_{p_i}. During analysis of the callee the reference edge $\langle q_i, n_{p_i} \rangle$ will naturally capture changes to β useful for fixing callers.
- Define $M = \{\langle n_{p_0}, n_{00} \rangle, \langle n_{p_0}, n_{01} \rangle, \ldots, \langle n_{p_1}, n_{10} \rangle\}$ to describe the mapping between heap region nodes in the callee graph and 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 $G'_t = \{n_{g'_{t0}}, \ldots, n_{g'_{tk}}, n_{g'_{ts}}\}$.

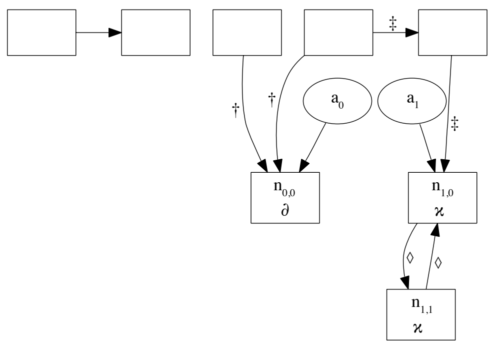
Let the program point for the call site `callee(a0, a1)` be `cs` and the method declaration be `void callee(p0, p1)`. Figure 4(a) presents an example caller context reachability graph for the method `caller` at the callsite `cs` and Figure 4(b) presents an example callee reachability graph for the method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

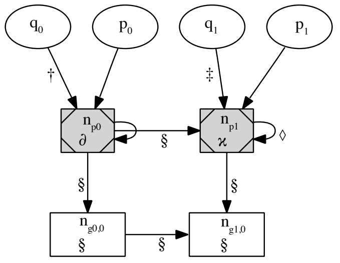
5.2 Conceptual Steps for Call Site Analysis

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

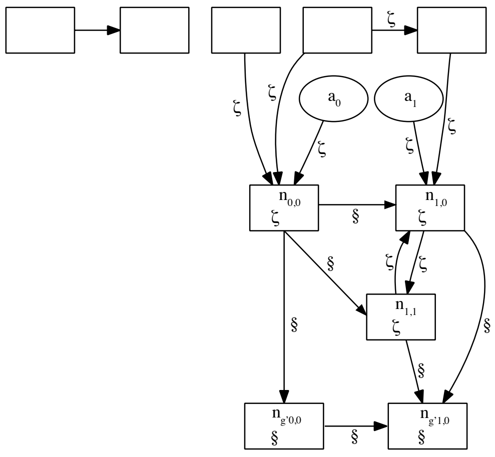
- **Aliasing Context:** The caller computes the alias context set Π for the call site. It then checks whether the analysis has processed this call site before for the given caller aliasing context. If the analysis has already processed this call site for the same caller alias context, the analysis looks 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.
- **Caller Node Reachability:** The callee may change reachability states of objects reachable from its parameters. Conceptually, the reachability states in $\alpha(n_{p_i})$ summarize how the callee may change the reachability of parameter objects.
- **Caller/Callee Edge Reachability:** The callee may change reachability states of caller reference edges that are reachable from the callee's parameters. Conceptually, the reachability state for the reflexive edge $\beta(\langle n_{p_i}, \mathbb{F}, n_{p_i} \rangle)$ summarizes how the callee may change the reachability of these edges.
- **Upstream Caller Edge Reachability:** The callee may change the reachability states of caller reference edges that are upstream



(a) Caller Context: ∂ maps into n_{p_0}, $\varkappa$ maps into n_{p_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).

from the callee's parameters. Conceptually, 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 upstream edges.

- **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. **Caller Node Reachability:** For each p_i, define the node rewrite set $H_i = \alpha(n_{p_i})$. Conceptually, the rewrite rule H_i captures with respect to n_{p_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.
2. **Caller/Callee Edge Reachability:** The reference edge $\langle n_{p_i}, \mathbb{F}, n_{p_i}\rangle$ abstracts all of 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 set $J_i = \beta(\langle n_{p_i}, \mathbb{F}, n_{p_i}\rangle)$. Conceptually, the rewrite rule J_i captures with respect to $\langle n_{p_i}, \mathbb{F}, n_{p_i}\rangle$'s initial reachability state at the beginning of the callee how the callee changed the reachability sets of the edges between objects in n_{p_i}.
3. **Upstream Caller Edge Reachability:** The reference edge $\langle q_i, n_{p_i}\rangle$ abstracts the caller reference edges upstream of parameter n_{p_i}. For each p_i, define the edge rewrite set $K_i = \beta(\langle q_i, n_{p_i}\rangle)$. Conceptually, the rewrite rule K_i captures with respect to $\langle q_i, n_{p_i}\rangle$'s initial reachability state 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}.
4. **Reachability States For Token $\eta_{n_{p_i}}$:** For each p_i, define:
$$d_i \;=\; \bigcup_{\langle a_i, n_{ij}\rangle} \beta(\langle a_i, n_{ij}\rangle)$$
The set of reachability states d_i is a simple union of all reachability states on the reference edges out of label node a_i in the caller. Conceptually, d_i 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_{p_i} before the callee's execution may change its reachability.
5. **Reachability States For $\eta^*_{n_{p_i}}$, $\eta^+_{n_{p_i}}$:** For each p_i, define:
Let $d_i \;=\; \{\phi_{i0}, \ldots, \phi_{ij}\}$
$$D_i = \begin{cases} \Big\{ s_{i0} \cup \cdots \cup s_{ij} \mid s_{ia} \in \\ \qquad\qquad \{\emptyset, \phi_{ia}, \phi_{ia} \cup_{\triangle} \phi_{ia}\} \Big\} & \text{if } \mid d_i \mid < \omega_{\max} \\ \Big\{ \bigcup_{\triangle [\langle \eta_{i0}, \mu_{i0}\rangle, \ldots, \langle \eta_{ij}, \mu_{ij}\rangle] \in d_i} [\eta^*_{i0}, \ldots, \eta^*_{ij}] \Big\} & \text{otherwise} \end{cases}$$
If a parameter token appears in a reachability state with an arity of `ZERO-OR-MORE` or `ONE-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 to generate all possible combinations of caller reachability sets by taking any combination of the reachability states in d_i any number of times. Note that the calculation of D_i is intractable when the set d_i is large. When the size of d_i is greater than a threshold $\omega_{\max}$, we approximate the calculation with one reachability state that is union of every state in d_i with token arity values all `ZERO-OR-MORE`.
Conceptually, D_i gives the possible reachability states that the callee tokens $\eta^*_{n_{p_i}}$ or $\eta^+_{n_{p_i}}$ represent.
6. **Mapping Operator:** The Δ operator computes caller reachability states from callee reachability states with respect to parameter p_i. Δ takes as input the parameter index i, a set of callee-context rewrite rules R, and a set of caller-context reachability states S and produces a set of caller-context reachability states.

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

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

$$\Omega(\langle\eta, \mu\rangle) = \begin{cases} S & \text{if } \langle\eta, \mu\rangle = \eta_{n_{p_i}} \\ D_i & \text{if } \langle\eta, \mu\rangle = \eta^*_{n_{p_i}} \text{ or } \eta^+_{n_{p_i}} \\ d_j & \text{if } \langle\eta, \mu\rangle = \eta_{n_{p_b}}, b \neq i \\ D_j & \text{if } \langle\eta, \mu\rangle = \eta^*_{n_{p_b}} \text{ or } \eta^+_{n_{p_b}}, b \neq i \\ \{[\langle\eta_{n_{g'_{tz}}}, \mu\rangle]\} & \text{if } \langle\eta, \mu\rangle = \eta^\mu_{n_{g_{tz}}} \\ \{[\langle\eta, \mu\rangle]\} & \text{otherwise.} \end{cases}$$

To illustrate Δ take $i = 0$, $R = \{[\eta_{n_{p_0}}, \eta^+_{n_{p_1}}, \eta^*_{n_{g_{3,1}}}]\}$, $S = \{[\eta_{n_4}], [\eta_{n_4}, \eta_{n_5}]\}$, and $D_1 = \{[\eta_{n_6}], [\eta_{n_7}]\}$.
The expansion of each callee-context token tuple in a callee-context reachability state results in a set of possible sets of caller-context token tuples. In this example

$$\Omega(\eta_{n_{p_0}}) = S = \{[\eta_{n_4}], [\eta_{n_4}, \eta_{n_5}]\}$$
$$\Omega(\eta^+_{n_{p_1}}) = D_1 = \{[\eta_{n_6}], [\eta_{n_7}]\}$$
$$\Omega(\eta^*_{n_{g_{3,1}}}) = \{[\eta^+_{n_{g'_{3,1}}}]\}$$

and then we have

$$\Theta(\{[\eta_{p_0}, \eta^+_{p_1}, \eta^*_{n_{g_{3,1}}}]\}) = \left\{ \begin{array}{c} [\eta_{n_4}, \eta_{n_6}, \eta^+_{n_{g'_{3,1}}}], \\ [\eta_{n_4}, \eta_{n_5}, \eta_{n_6}, \eta^+_{n_{g'_{3,1}}}], \\ [\eta_{n_4}, \eta_{n_7}, \eta^+_{n_{g'_{3,1}}}], \\ [\eta_{n_4}, \eta_{n_5}, \eta_{n_7}, \eta^+_{n_{g'_{3,1}}}] \end{array} \right\}.$$

5.4 Call Site Algorithm

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

1. **Caller Node Reachability:** Rewrite α for each caller heap region node n_{ij} reachable from argument label i. Initialize $\alpha'(n_{ij}) = \emptyset$.

$$\begin{aligned} \delta &= \Delta\Big(i, H, \alpha(n_{ij})\Big) \\ \alpha'(n_{ij}) &= \alpha'(n_{ij}) \cup \delta. \end{aligned}$$

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. **Caller/Callee Edge Reachability:** Rewrite β for caller reference edges reachable from argument label i. Initialize $\beta' = \emptyset$.

$$\begin{aligned} \delta &= \Delta\Big(i, J, \beta(\langle n_{i_0 z_0}, f, n_{i_1 z_1}\rangle)\Big) \\ \beta'(\langle n_{i_0 z_0}, f, n_{i_1 z_1}\rangle) &= \beta'(\langle n_{i_0 z_0}, f, n_{i_1 z_1}\rangle) \cup \delta. \end{aligned}$$

The process for reference edges reachable from argument labels is similar to the parameter-reachable heap region nodes above.

3. **Upstream Caller Edge Reachability:** This step generates β' for caller reference edges upstream from the heap region nodes reachable from parameter i. Initialize $\beta' = \emptyset$.

$$\begin{aligned} \delta &= \Delta\Big(i, K, \beta(\langle n, f, n_{iz}\rangle)\Big) \\ \beta'(\langle n, f, n_{iz}\rangle) &= \beta'(\langle n, f, n_{iz}\rangle) \cup \delta \\ \delta &= \Delta\Big(i, K, \beta(\langle l, n_{iz}\rangle)\Big) \\ \beta'(\langle l, n_{iz}\rangle) &= \beta'(\langle l, n_{iz}\rangle) \cup \delta. \end{aligned}$$

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.

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

$$\begin{aligned} \delta &= \Delta(-1, \alpha(n_{g_{tz}}), \emptyset) \\ \alpha'(n_{g'_{tz}}) &= \delta. \end{aligned}$$

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 $\alpha(n_{g_{tz}})$ 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, the parameter index -1 and the $\emptyset$ caller-context 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 $\alpha(n_{g_{tz}})$.

5. **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.

$$\begin{aligned} \delta &= \Delta(-1, \beta(e), \emptyset) \\ \beta'(e_{\text{caller}}) &= \delta. \end{aligned}$$

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:

$$S_{\text{src}} = \begin{cases} \{n_{g'_{tz}}\} & \text{if } \mathrm{src}(e){=}n_{g_{tz}} \text{ and } n_{g_{tz}} \text{ has a field name matching } \mathrm{field}(e), \\ \{n_{i0}, \ldots, n_{ij}\} & \text{if } \mathrm{src}(e){=}n_{p_i} \text{ and } n_{ij} \text{ has a field name matching } \mathrm{field}(e). \end{cases}$$

$$S_{\text{dst}} = \begin{cases} \{n_{g'_{tz}}\} & \text{if } \mathrm{dst}(e){=}n_{g_{tz}} \text{ and } n_{g_{tz}}\text{'s type matches } \mathrm{field}(e) \text{ or is multi-obj}, \\ \{n_{i0}, \ldots, n_{ij}\} & \text{if } \mathrm{dst}(e){=}n_{p_i} \text{ and } n_{ij}\text{'s type matches } \mathrm{field}(e) \text{ or is a multi-obj}. \end{cases}$$

$E_{\text{caller}} = \{\langle s, d\rangle, s \in S_{\text{src}}, d \in S_{\text{dst}}\}$
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, $\beta(e_{\text{existing}}) = \beta(e_{\text{existing}}) \cup \beta(e_{\text{caller}})$.

6. **Update** α, β**:** The analysis next replaces α and β with the updated versions α' and β', respectively.

7. **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.
8. **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 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 remain) 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.

6. Evaluation

We have implemented the disjointness analysis in our compiler. We have analyzed several applications written in the Bamboo language. Bamboo extends a Java-like core language 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. We flagged all objects that can serve as parameters to the Bamboo tasks. Bamboo uses a similar task invocation model to the Bristlecone language [12] — the runtime task invokes tasks when there exists objects in the heap in the appropriate states to serve as parameter objects. These semantics mean that flag objects even without explicit references are live, and therefore the second strong update condition does not apply to them.

Bamboo task invocation locks on the parameter objects — therefore, showing that the flagged objects can reach disjoint parts of the heap is sufficient to execute tasks in parallel. We ran the analysis on 17 benchmarks using the Bamboo extensions to Java on a 2.4 GHz Core 2 Duo with 1 GB of RAM. The source code for the analysis and the benchmarks can be downloaded from the web. The jHTTPp2 benchmark was ported from `http://jhttp2.sourceforge.net/`. The JGFSeries, JGFMoldyn, and JGFMonteCarlo benchmarks were ported from the Java Grande benchmark suite [27]. The FilterBank benchmark was ported from the StreamIt benchmark suite [17].

6.1 Disjointness Results

Table 2 presents the analysis results and execution times for our benchmark suite. The analysis identified a total of 11 possible aliases between flagged object classes over five of the benchmarks.

Benchmark	Sharing	Time
Chat	3	2.065s
OnlineMultiGame	4	54.986s
MapReduceA	1	8.792s
MapReduceB	2	10.867s
JGFMoldyn	1	14.257s
BankApp	0	1.222s
WebConglomerator	0	1.297s
jHTTPp2	0	3.114s
PERT	0	0.797s
FilterBank	0	0.188s
JGFMonteCarlo	0	1.961s
JGFSeries	0	0.162s
SpiderA	0	2.676s
SpiderB	0	4.042s
TileSearch	0	5.632s
TicTacToe	0	0.952s
WebServerA	0	2.939s
WebServerB	0	4.843s

Table 2. Benchmark Results

The other thirteen 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. While it is possible for the analysis to report false aliases, none appeared in the analysis results for our benchmarks.

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. Bamboo provides language constructs that facilitate maintaining disjointness. The alias reports for flagged objects were generated at the exit of each task invocation.

6.2 Performance

Table 2 presents the analysis times for the benchmark suite. The benchmarks contain several hundred to a few thousand lines of application code excluding the class libraries. The largest benchmark, JGFMonteCarlo, contains 2,418 lines. Our current implementation makes no effort to optimize the evaluation order of statements or methods — it often must reanalyze methods that are not recursive simply due to non-optimal evaluation ordering.

7. Related Work

Like disjointness analysis, both alias analysis [2, 11, 14, 24] and pointer analysis [26, 20, 29, 5] 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 [22], however our analysis can maintain disjointness properties in the presence of mutation.

Alias analyses vary in whether they are *flow-sensitive* [7] or *context-sensitive* [31, 15, 28]. 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 [23] suggests that for alias analysis the benefit of context-sensitivity is rare over context-insensitive analyses. 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 [20, 7]. Other strategies are to use symbolic access paths [13] or regular expressions [19]. 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 [3, 30] 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 [1, 4, 8, 9, 10, 18]. We only make similar observations when pruning method effects that are being mapped into the calling context.

Shape analysis [6, 16, 25, 21] 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.

8. Conclusion

If a compiler can determine that two blocks of code operate on disjoint data structures, it can safely parallelize them. 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.

We have implemented the analysis in our compiler framework and analyzed several benchmark programs written in Bamboo, a set of task-based extension to a Java-like core language. For our benchmark programs, the analysis precisely identified sharing between the key data structures.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[2] J. 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.

[3] B. 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.

[4] C. Boyapati, R. Lee, and M. 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.

[5] M. G. Burke, P. R. Carini, J.-D. Choi, and M. 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.

[6] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990.

[7] J.-D. Choi, M. Burke, and P. 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.

[8] D. G. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2003.

[9] D. G. Clarke and S. 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.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. *ACM SIGPLAN Notices*, 33(10):48–64, 1998.

[11] 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.

[12] B. Demsky and A. Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 2008 European Conference o n Object-Oriented Programming*, 2008.

[13] A. 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.

[14] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998.

[15] M. Emami, R. Ghiya, and L. 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.

[16] R. Ghiya and L. 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*, 1996.

[17] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. 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.

[18] D. L. Heine and M. 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.

[19] J. Hummel, L. J. Hendren, and A. 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.

[20] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993.

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

[22] M. Naik and A. 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.

[23] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995.

[24] E. Ruf. Partitioning dataflow analyses using types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 2002.

[26] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, 1997.

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

[28] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996.

[29] W. 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.

[30] J. Whaley and M. 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.

[31] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*.