2 Escape Analysis

KESO's compiler *JINO* uses alias and escape analysis to identify objects whose lifetime is bounded by the runtime of their allocating method. The algorithm was implemented in [Lan12] and is largely based on the work of Choi et al. in 2003 [CGS+03]. The following section contains a brief description of the implementation and highlights differences. For an in-depth explanation, please refer to [Lan12] and [CGS+03]. Section 2.2 lists and explains the improvements written for this thesis.

2.1 Basics

The algorithm starts with alias analysis, which is separated into a method-local (also *intraprocedural*) analysis and a global (*interprocedural*) analysis. To compute and store alias information, a specialized data structure called connection graph (CG) is used. For each analyzed method, this graph contains representations of local variables, static class members, dynamic instance variables, array indices, and objects. Variables of non-reference type are ignored because they do not contribute to alias information.

2.1.1 Intraprocedural Analysis

In intraprocedural analysis, each method in the call graph of an application is traversed and a CG representation is being computed. It is a key contribution of Choi et al. that this representation is independent of the calling context. Since the origin of objects might not be known for some objects (e.g. if the have been passed as argument), a special type of placeholder called *phantom node* is used to represent these objects. For pointer analysis as discussed in Section 1.3, summarizing independent of the aliasing relationships in the calling context is impossible [CGS+03, p. 886]. For each allocation, assignment, field or array access, return statement, method invocation, and exception throw, the CG is modified appropriately, ensuring possible alias relations are represented accurately.

Nodes in the CG have different types: *Object nodes* are added for each encountered allocation site. Note that a single object node in the graph might represent multiple objects at runtime because an allocation might be executed multiple times (e.g. if it is inside a loop). Local variables, static class members, and member variables are represented using *local reference nodes*, *global reference nodes*, and *field reference nodes*, respectively. Array indices are treated like fields and are thus also represented by a field reference node. Each reference node can point to a series of object nodes and also to other reference nodes using *deferred edges*. Deferred edges are used to simplify updates of the CG while processing assignments. After intraprocedural analysis, these edges are removed by replacing all incoming deferred edges of a reference node with edges to its successors. Different from the work of Choi et al., reference nodes with incoming deferred but no outgoing edges are preserved without change. Section 2.2.2 gives the rationale underlying this difference. Finally, object nodes can point to field reference nodes, denoting that the pointed field exists inside the object where the edge originates.

Each node in the CG has one of three *escape states*, indicating whether a node will outlive its allocating method, or even thread. Among these states, a total order exists. *Local* is the lowest state. Nodes marked local do not escape the analyzed method. Next after local is *method*. Nodes that outlive a method by being returned or assigned to an object passed as parameter are tagged method-escaping. The highest escape state is *global* and is given to objects and references that are assigned to static thread-local class members or thrown as exception. While processing a method's instructions and building the CG, operations that cause the escape state of one of their parameters to change trigger the appropriate change in the escape states recorded in the connection graph. Allocations whose object node representation in the CG is tagged local are considered for stack allocation, aber nicht die mit überlappenden Lebens-Regionen ...

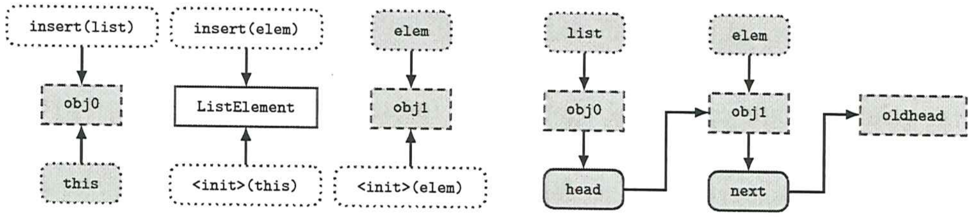
See Listing 2.1 for source code corresponding to the CGs to be explained in depth. The code example is a simple generic linked list. Using common sense we can deduce that, in the absence of a removal operation, all list elements will be reachable until the list itself has reached the end of its lifetime. Consequently, the only allocation in the given example can not be allocated on the stack, because it must outlive the method of its allocation *addElement*. Due to the structure of the example, intraprocedural analysis will not suffice to determine this. Global analysis will be necessary.

Listing 2.1: A simple generic linked list in Java

```java
public class LinkedList<T> {
  private static final class ListElement<U> {
    ListElement<U> next;
    U              elem;

    public ListElement(U elem) {
      this.elem = elem;
    }
  }

  private ListElement<T> head;

  public void addElement(T elem) {
    insert(this, new ListElement<>(elem));
  }

  private static <V> void insert(LinkedList<V> list,
                                 ListElement<V> elem) {
    elem.next = list.head;
    list.head = elem; // make elem the first entry
  }
}
```

A simple generic linked list implementation in Java. Note that this example is more complex than it would have to be, especially due to the *insert* method, for demonstration purposes.
An inner class is used to wrap the list entries with references to their successor. The *addElement* method allows insertion of new entries. Internally, *addElement* uses *insert*, which enqueues the given new element at the start of a list.

See Fig. 2.1 for the connection graphs of the methods *insert* and *addElement* given in Listing 2.1. The *addElement* method has two parameters, but only the second one is visible in the code listing. Java implicitly passes the *this* reference as first argument. These parameters are represented in Fig. 2.1a by two reference nodes with dotted borders. Since they are reachable after the method returns, they are marked as *method*-escaping, denoted by the orange fill color. Because the allocation



(a) Connection graph for *addElement*. *insert(list)* and *insert(elem)* represent the parameters passed to the *insert* method at its invocation. ⟨*init*⟩*(this)* and ⟨*init*⟩*(elem)* do the same for the implicit invocation of *ListElement*'s constructor.

(b) Connection graph for *insert*. The edge from *head* to *obj1* is the one that prevents stack allocation of the list element in *addElement*. Interprocedural analysis needs to propagate this edge into the caller context to determine this.

Figure 2.1: The connection graphs for the *addElement* and *insert* methods given in Listing 2.1 after intraprocedural analysis. Vertices with rounded corners represent *reference nodes*, where *field reference nodes* have a red ■, other reference nodes a blue ■ border. Dotted borders mark artificial reference nodes representing a method's parameter or return value. Rectangles with green ■ borders are *object nodes*. If the border is dashed, the node is a *phantom node*. The escape state of nodes is encoded in the fill color. White, orange ■, and red ■ represent *local*, *method*, and *global* respectively.

sites of the pointees of both *this* and *elem* are unknown, these objects are represented using *phantom nodes* (dashed green rectangle). Note that the escape state propagates along the edges from *this* into *obj0* and from *elem* into its pointee *obj1*. The first statement in the bytecode representation of *addElement* is the allocation of a new list element, which causes the creation of an object node (green rectangle) in the CG. The constructor of the newly created *ListElement* is called with two arguments: a reference to the object and a reference to the given parameter *elem*, represented in the connection graph by the ⟨*init*⟩*(this)* and ⟨*init*⟩*(elem)* reference nodes. The *this* parameter of the constructor invocation points to the allocated list element, denoted by a solid edge in the graph. The algorithm can deduce that ⟨*init*⟩*(elem)* points to the same object as *elem*, but the pointees of *elem* are unknown at this point. Adding a deferred edge encodes this situation. Finally, processing the invocation of *insert* creates a similar set of nodes *insert(list)* and *insert(elem)* pointing to the *this* reference and the *ListElement* object respectively.

The graph for *insert* is given in Fig. 2.1b. The two parameters are again represented by a reference node and a phantom node each. The first few statements create the *next* field node below *obj1* and add a deferred edge to *head*. This edge is not shown in the graph, because the next statement in the code changes the value of *head*. Depending on whether the analysis is flow-sensitive or not, the outcome will differ. The CG given uses the flow-sensitive variant, which causes all incoming deferred edges

Du beschreibst, wie sich fluss-sensitive Analyse in deinem Beispiel auswirkt. Was ist den konzeptionell der Unterschied und wann entscheidet man sich für die ein oder andere Variante?

of a reference node to be compressed and all outgoing edges to be removed before pointing to the new target. Because compression of deferred edges requires at least one edge outgoing from the target of the deferred edge, but *head* did not have any at this point in the analysis, a phantom node representing the previous pointees of *head* was created. It is denoted *oldhead* in Fig. 2.1b. Using flow-insensitive analysis, the deferred edge from *next* to *head* would have been preserved and later compressed into an edge from *next* to *obj1*. Finally, the edge from *head* to *obj1* is added to represent the effect of the statement in line 20 in Listing 2.1, completing intraprocedural analysis. Satz umstellen

2.1.2 Interprocedural Analysis

Looking at the summary information generated for both methods given in Listing 2.1, the object node that represents the only allocation (*LinkedList* in Fig. 2.1a) has an escape state of *local* at this point in the analysis. Recall that local nodes are considered for stack allocation, but entries of a linked list cannot be allocated on stack. The results are thus not sound and further analysis is required. Looking at the example, the edge from *head* to *obj1* in the CG of *insert* is the edge that will prevent stack allocation of the list element in *addElement*. To determine this algorithmically, the edges found while processing *insert* need to be propagated to its caller *addElement*. Once the connection from *obj0* via a new field reference node *head* to the list element is established, the escape state of the *ListElement* object node increase s to *method*, making the result sound.

A second analysis pass propagates information from the non-*local* subgraph of the CGs into the CGs of all calling methods. This *interprocedural* analysis modifies the summary information of the callers, which in turn require their callers to be updated again. To prevent unnecessary recalculation of information, this pass should use a bottom-up traversal of the call graph. In the absence of recursion, the call graph will not contain cycles, making this a simple problem. When recursion is used, identifying strongly connected components and iterating in each component until a fixpoint is reached has proved to be effective. KESO's implementation uses Tarjan's algorithm to identify strongly connected components in the call graph [Tar72].

To update the callers' connection graphs, pairs of corresponding nodes in the graph on the caller and the callee side are identified as starting points and added to a work list. Originating at these anchors, further nodes and their counterparts are found and again added to the work list. While the work list isn't empty, processing continues and builds a relation of object nodes in the callee and the caller CG called *mapsToObj*. This step is called *updateNodes*. A simplified form of the procedure is given in Algorithm 2.1. It uses a dual work list approach to avoid creating spurious phantom nodes because some of the relationships might not be known until the algorithm completes. See [Lan12, Sec. 3.2.1] for a detailed explanation of the problem that causes unneeded phantom nodes to be added and slows down the analysis. *UpdateNodes* ensures that all object nodes used in a callee are represented in its caller. It also adds field reference nodes present in the callee CG but missing from the caller's graph and marks the counterparts of globally escaping nodes in the callee's CG as globally escaping in the caller CG.

For the example given in Listing 2.1 and Fig. 2.1a, this means that the field reference nodes *head* and *next* are added below *obj0* and *ListElement* respectively. Furthermore, a phantom node is added to represent *oldhead*. Note that the *head* reference node does not yet point to the list element object node. The next step of the algorithm called *updateEdges* adds the missing connection. It takes the same parameters as *updateNodes* from Algorithm 2.1 and adds all missing edges. See Algorithm 2.2 for a pseudocode listing of the procedure. It identifies pairs of corresponding object nodes using the *mapsToObj* relation computed in *updateNodes*. For each possible pair it follows outgoing edges to any field reference nodes in the callee graph and finds the corresponding field reference node in the caller's CG. Next, the newly found correspondence pair is added to the work list and all outgoing edges to object nodes in the callee graph are added to the caller graph. Note that code to prevent endless loops in cylic data structures has been left out for simplicity, but can be easily added.

After interprocedural analysis, nodes marked *local* in the CG can be allocated on the stack. Note that KESO does not convert all allocations that fulfill this criterion into stack allocations. Instead, variable liveness information is used to compute whether multiple objects allocated at the same allocation site are needed at the same time. Objects with overlapping liveness regions are not allocated on the stack because the amount of memory used by these allocations might be unbounded, e.g. if the allocation is inside a loop. See [Lan12, Sec. 3.3] for detailed rationale and a

Algorithm 2.1: The *updateNodes* procedure [Lan12, Alg. 2]

Input : xs: method parameters, ys: invocation arguments
Result: mapsToObj relation between caller and callee nodes

```
updateNodes (xs, ys)
begin
    workList = {(x, y) | x ← xs | y ← ys };
    needsPointee = ∅;
    mapsToObj = ∅;
    while workList ≠ ∅ or needsPointee ≠ ∅ do
        while workList ≠ ∅ do
            (mParam, iArg) = pop(workList);
            // Mark mParam globally escaping if iArg is.
            updateEscapeState(mParam, iArg);
            // Find pairs of descendant object nodes
            xPointees = pointees(mParam);
            yPointees = pointees(iArg);
            if xPointees ≡ ∅ or yPointees ≠ ∅ then
                // Find pairs of field reference nodes
                foreach (xd, yd) ∈ {(x, y) | x ← xPointees, y ← yPointees} do
                    mapsToObj(xd) ∪= yd;
                    foreach calleeField ∈ fields(xd) do
                        callerField = getField(yd, calleeField);
                        workList ∪= (calleeField, callerField);
            else
                // The callee node is not represented in the caller node
                needsPointee ∪= (mParam, iArg);
        while workList ≡ ∅ and needsPointee ≠ ∅ do
            foreach (mParam, iArg) ∈ needsPointee do
                if pointees(iArg) ≠ ∅ then
                    // mParam's pointees are represented (happens with recursion)
                else
                    // Check for other representatives of pointees of mParam
                    callerObjs = {x | x ← mapsToObj(y) | y ← pointees(mParam) };
                    if callerObjs ≠ ∅ then
                        addEdges(iArg, callerObjs);
                    else
                        // Delay adding phantom nodes
                        continue;
                needsPointee \= (mParam, iArg);
                workList ∪= (mParam, iArg);
            if workList ≡ ∅ then
                // workList is still empty, no pairs found. Add a phantom node.
                (mParam, iArg) = pop(needsPointee);
                addEdges(iArg, createPhantom(mParam));
                workList ∪= (mParam, iArg);
```

Algorithm 2.2: The *updateEdges* procedure [Lan12, Alg. 3]

```
Input : xs: method parameters, ys: invocation arguments
updateEdges (xs, ys)
begin
    workList = xs;
    while workList ≠ ∅ do
        mParam = pop(workList);
        foreach (x, y) ∈ {(x, y) | y ← mapsToObj(x) | x ← pointees(mParam) } do
            foreach (i, j) ∈ {(field, getField(y, field)) | field ← fields(x) } do
                workList ∪= (i, j);
                addEdges(j, {mapsToObj(k) | k ← pointees(i) });
```

description of the implementation.

2.2 Improvements

The algorithm implemented in [Lan12], which is based on [CGS+03], was improved for this thesis in a number of ways. Among these improvements was flow-sensitivity, which was proposed by Choi et al. in 2003, but not implemented in my bachelor's thesis. The changes required to achieve flow-sensitivity are outlined in the following Section 2.2.1. Furthermore, a problem producing possibly incorrect results was discovered in the algorithm given in [CGS+03]. Section 2.2.2 gives an example and explains where the incorrect analysis results occur and how they were fixed in KESO's implementation.

Last but not least, the algorithm's runtime on large examples amounted to several minutes and was deemed unsatisfactory. Especially recursive and virtual method invocations significantly increased the size of the generated CGs, raising the runtime of interprocedural analysis. Section 2.2.3 deals with modifications implemented to reduce the runtime of the alias analysis.

2.2.1 Flow-Sensitivity

For flow-sensitive alias analysis a standard forward data flow analysis [ALSU07, Sec. 9.2] is used. The set of operations needed for data flow analysis is

$$C_o^b = f_b\left(C_i^b\right) \tag{2.1}$$

$$C_i^b = \bigwedge_{x \,\in\, \mathrm{pred}(b)} C_o^x \tag{2.2}$$

where C_i^b and C_o^b are input and output data flow information for the basic block b, f_b in Eq. (2.1) is called the *transfer function* for the basic block b, and $\wedge_{x \in \mathrm{pred}(b)}$ in Eq. (2.2) is the *meet* operation. The data flow transfer function modifies the data flow information (i.e., the connection graph) according to the statements in the basic block b. The meet operation combines the output information of all predecessors of a basic block into the input information of a given basic block b. The basics of the transfer function are explained in [Lan12, CGS+03]. In theory, all C^b are distinct. In practice, this would manifold the memory requirements for the CGs. Instead of copying the graph in each invocation of the transfer and meet operations, KESO's implementation uses a single representation. This idea is also present in [CGS+03, Sec. 3], but is not explained very well. In specific, it is not clear to the author of this thesis what Choi et al. meant when they wrote "in the flow-sensitive version, we only kill local variables" [CGS+03, p. 885].

The KESO compiler achieves flow-sensitivity while retaining a single representation of the CG by tagging all reference nodes with the basic block for which they are valid. Object nodes are not modified for flow-sensitive analysis. For each reference node used in at least one predecessor, the meet operation creates a representation of the reference in the current basic block and subsumes all outgoing edges present in the predecessors. For each assignment operation encountered by the transfer function, *ByPass(p)* is called on the reference to be written. *ByPass(p)* (as explained in [CGS+03]) redirects all incoming deferred edges of p to its successors and removes any outgoing edges (i.e., it ensures that strong updates are performed).

After implementing this improvement, a fixpoint iteration used to reduce the number of unnecessary phantom nodes in intraprocedural analysis did no longer terminate for some inputs. This happened because the iteration tracked changes to the

CG rather than comparing the graph against an older copy. Due to the use of *By-Pass(p)*, the graph was modified in every loop, but further processing returned to the previous state again. Switching to a comparison against an old copy of the CG rather than tracking of modifications fixed this particular problem. To efficiently implement comparisons against older versions of the same graph, the connection graph's nodes were extended with the ability to store a copy of a single older state of outgoing edges.

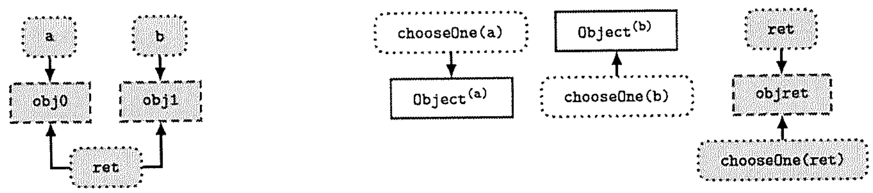
2.2.2 Fixing Incorrect Results: The Double Return Bug

KESO's implementation of escape analysis produced incorrect results given inputs similar to those generated by the idea outlined in Section 3.1. Further analysis suggests this is a conceptual flaw in the work of Choi et al. See Listing 2.2 for an example triggering this bug. The *getObject* method allocates two objects and passes them to *chooseOne*, which selects one of them at random and returns it. The return value of *chooseOne* is then returned from *getObject*. Because either of the two objects allocated in *getObject* might escape, both allocations must not use stack memory.

Listing 2.2: Example exposing the double return flaw

```
public class ChooseOne implements Runnable {
  public void run() {
    Object a = getObject();
  }

  private static Object getObject() {
    return chooseOne(new Object(), new Object()); // bug occurs here
  }

  private static Object chooseOne(Object a, Object b) {
    if (Math.random() < 0.5)
      return a;
    return b;
  }
}
```

A simplified example exposing the double return bug in KESO's escape analysis. One of the objects allocated in *getObject* is returned from its allocating method, but escape analysis did not detect this due to the use of phantom nodes to represent return values.



(a) The CG for *chooseOne*.

(b) The CG for *getObject* according to [CGS+03, Sec. 4].

Figure 2.2: The CGs for *getObject* and *chooseOne* from Listing 2.2, exhibiting the double return flaw. The object nodes in *getObject* should be pointed-to by *ret*, which would raise their escape state to *method*, but these edges are missing. Colors and shapes c.f. Fig. 2.1.

However, the CG constructed according to [CGS+03, Sec. 4] does not correctly identify the two objects as method-escaping. The connection graph for *chooseOne* is straightforward and given in Fig. 2.2a. For the CG of *getObject*, Sections 4.3 "The Connection Graph Immediately Before a Method Invocation" and 4.4 "The Connection Graph Immediately After a Method Invocation" are the relevant parts of [CGS+03]. According to the first section a new *actual reference node* is created for each argument of the invocation and an assignment $\hat{a}_i = u_i$ is processed. $\hat{a}_i$ denotes the actual reference nodes, u_i are the corresponding invocation arguments. The statement causes the creation of deferred edges from the $\hat{a}_i$'s to the u_i's, which will later be compressed.

The handling of return values is not explicitly explained in this section, but the next section mentions them as "$\hat{a}_i$'s (representing actual arguments and return value) of the caller's CG," [CGS+03, p. 891] suggesting that an *actual reference node* for the return value is added in the caller's CG. Figure 2.2b shows this connection graph: The rounded rectangles with blue dotted borders are said *actual reference nodes*. Both *chooseOne(a)* and *chooseOne(b)* initially have a single outgoing edge pointing to a local variable, which in turn points to the allocated objects. This indirection is omitted from the graph in Fig. 2.2b for simplicity. The return value of *getObject* is modelled using assignments to a special "phantom" variable called *return* (in KESO's implementation: *ret*). Since the result of the call to *chooseOne* is returned from *getObject*, a deferred edge from the phantom return variable to the *actual reference node* representing *chooseOne*'s return value is added. After completing intraprocedural escape analysis, all deferred edges are removed from the graph according to Choi et al., adding phantom nodes where necessary. This leads to the creation of the phantom node denoted *objret* in Fig. 2.2b. The following path compression removes the de-

chooseOne	getObject
obj0	$\text{Object}^{(a)}$, objret
obj1	$\text{Object}^{(b)}$, objret

(a) The *mapsToObj* relation constructed using *updateNodes* as given in [CGS+03]. Note that this is the same even after KESO's modifications.

chooseOne	getObject
a	chooseOne(a)
b	chooseOne(b)
ret	chooseOne(ret)

(b) The *mapsToRef* relation constructed using KESO's modified *updateNodes* algorithm.

Table 2.1: The *mapsToObj* and *mapsToRef* relations for the call of *chooseOne* from *getObject* as given in Listing 2.2.

ferred edges from *ret* to *chooseOne(ret)* and adds a *points-to* edge to *ret*. The *method* escape state of *chooseOne(ret)* is retained, but is not relevant for the further problem description.

The analysis ends with the *UpdateCaller* routine. It consists of *UpdateNodes* and *UpdateEdges*. The former computes equivalence pairs of object nodes in the callee's and caller's CGs (the so-called *mapsToObj* relation) and adds phantom nodes for objects that have no equivalence in the caller yet. *UpdateEdges* ensures all relevant edges present in the callee's CG are propagates into the caller's graph. The *mapsToObj* relation for the call from *getObject* to *chooseOne* is given in Table 2.1a. If *chooseOne(ret)* did not yet have any pointees at this point of the analysis, statement 7 in *UpdateNodes* as displayed in [CGS+03, Fig. 7] would have created it as a phantom node, leading to the same problem. No new edges are inserted in interprocedural analysis for this example, because no structure of the form $p \rightarrow f_p \rightarrow q$ (where p, q are object nodes and f_p is a field reference node) exists in the CG of *chooseOne*.

Note that both phantom nodes in *chooseOne*'s CG map to both their respective object node and the *objret* phantom node, but the equivalence of $Object^{(x)}\ \forall x \in \{a, b\}$ and *objret* is not represented in *getObject*'s CG, leading to incorrect escape states for the two allocated objects.

To work around this problem, KESO's alias analysis (outlined in Algorithms 2.1 and 2.2) was extended to not only track equivalences between object nodes in *mapsToObj*, but also between reference nodes in a new relation called *mapsToRef*. This data is used in a modified version of *updateEdges* to add the missing edges in the caller's CG. On each occasion of $p \rightarrow o$ in the callee's CG where p is a reference node that does not represent a parameter and o is an object node, an edge $x \rightarrow y$ is added in the caller's CG for each $x \in mapsToRef(p)$ and $y \in mapsToObj(o)$, if no such edge

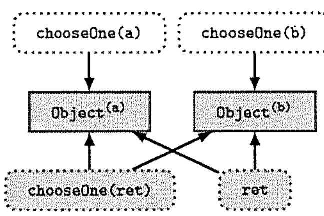


Figure 2.3: The CG for *getObject* as generated by KESO's modified algorithm to fix the double return flaw. The two object nodes are correctly marked *method*-escaping. Colors and shapes c.f. Fig. 2.1.

exists yet. Edges outgoing from parameters must be ignored in this step because Java has call-by-value semantics which means that the arguments given at a method invocation will always remain unchanged. All references reachable via other edges below the arguments can be modified, however, as can the return value.

Using this extension for the running example generates the *mapsToRef* relation as given in Table 2.1b. To avoid the superfluous phantom node *objret* that would be added because of the removal of deferred edges before interprocedural analysis, KESO's alias analysis does not attach phantom nodes to nodes that have incoming deferred but not outgoing edges. This required modifying the analysis to be able to deal with deferred edges in interprocedural analysis. After the modified interprocedural analysis finishes, the CG of *getObject* (depicted in Fig. 2.3) contains the missing edges.

2.2.3 Interprocedural Analysis Optimizations

Some of the larger applications (up to 23.8 kSLOC[1]) used in testing the KESO compiler took up to 19 minutes to compile with alias and escape analysis enabled. The compile times were dominated by the duration of alias analysis. To reduce this unacceptable overhead, a series of possible culprits were identified and modifications to the algorithm to optimize compile times were implemented. Since the vast majority of the time was spent in interprocedural analysis, all optimizations described in the following sections apply to this part of alias analysis.

[1] generated using David A. Wheeler's "SLOCCount"

2.2.3.1 No Propagation of Read Operations

Analyzing the generated CGs after interprocedural analysis revealed virtual invocations of methods that in turn call the same set of virtual methods caused the size of the graphs to increase rapidly. This situation commonly occurs in Java with simultaneous use of the *equals* method and collections (whose *equals* implementations call *equals* once for each element in the collection). Since calling *equals* usually does not change any references reachable from its parameters it does not add new aliases. Based on this observation, the intraprocedural analysis was extended to track all edges that were added to the CG due to a write operation. KESO's implementation uses a set of properties called *isWritten* and *isWriteOperand* available in each connection graph node to store this, because information cannot be easily attached to the edges themselves in KESO's adjacency list-based implementation of the CG. After intraprocedural analysis, a modified version of Tarjan's algorithm to find strongly connected components [Tar72] finds all cycle-free paths from the method's formal parameters to edges created by write operations. All edges that compose this subgraph are called *important* and marked for later use. Note that the subgraph may contain cycles because while *important* edges alone will not cause cycles, an additional edge created by a write operation might. Furthermore, intraprocedural analysis was extended to ignore all nodes and edges that have no role in a write operation and are not marked *important* (i.e., are not on a path from the method's entry points to a write operation edge).

In theory, these changes should have removed the effect of calls to *equals*, *hashCode* and similar methods completely. In practice, however, some implementations of *equals* may in fact contain write operations: For example, the *java.util.Hashtable* class from the GNU classpath project implements *equals* by comparing the entry sets of the two hash tables. This entry set is eagerly created and cached inside the hash table class. This write operation causes all edges leading up to it to be marked important. These edges are then propagated into all other invocations of *equals*, causing further edges to be considered important, nullifying the effect of the optimization for *equals*. Other implementations and functions might, however, still benefit from the improvement. If Java did have constant methods like C++ does, *equals* (and other methods that are marked constant and only have constant reference parameters) could be automatically ignored in alias analysis.

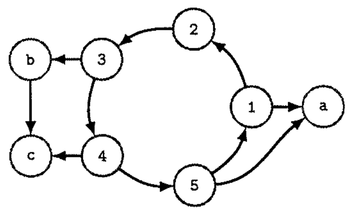


Figure 2.4: Call graph with a strongly connected component (blue ■ vertices) and a few dependent nodes (green ■ vertices). While methods inside the strongly connected component might have to be visited multiple times in interprocedural analysis, the summary information from a–c only needs to be propagated into their callers once.

2.2.3.2 No Reprocessing of Unchanged Invocations

Strongly connected components in an application's call graph (i.e., recursive methods) are handled in interprocedural analysis by iterating until a fixpoint has been reached. During this process, invocations might be reprocessed even though their callees' CGs did not change since the last iteration. This happens for all call graph edges leaving the strongly connected component. See Fig. 2.4 for a graphical representation of this situation.

KESO's implementation of interprocedural analysis only re-runs the *updateNodes* and *updateEdges* steps if a callee's CG changed since the last iteration, avoiding unnecessary overhead. Unfortunately, the savings from this optimizations are marginal.

2.2.3.3 Connection Graph Compression

While the improvements in Sections 2.2.3.1 and 2.2.3.2 reduced the compile time of large applications, the savings were still not enough to enable escape analysis by default in KESO without having a noticeable effect during development. Connection graph sizes would still surpass 10000 vertices on large inputs with these optimizations enabled, slowing down further steps of the analysis. Most of these nodes were created in interprocedural analysis as phantom nodes to represent objects allocated in callees of the current function and often had siblings that would represent the same objects. To reduce the size of the connection graphs, a graph compression transformation inspired by Steensgaard's almost linear time points-to analysis [Ste96] was implemented.

Starting at each entry point into a method's CG (i.e., every method parameter and the return value), the graph compression algorithm processes each reference node recursively but avoids loops using a color bit. For each reference node, lists of pointees segregated by escape state are collected. The separation into different escape states ensures that object nodes are only unified with nodes that have the same escape state. This avoids deterioration of the computed results up to this point. Each list that contains at least two object nodes and at least one phantom node is compressed by removing the phantom nodes. Note that any two non-phantom object nodes (i.e., any two nodes with a known allocation site) are not consolidated to preserve the one-to-one mapping between intermediate code allocation instruction and its CG representation.

Incoming edges pointing to the phantom nodes to be removed are redirected to the retained object nodes. Field reference nodes reachable from the phantom nodes are re-created below the object nodes in the compression set. Edges outgoing from the removed field reference nodes are moved to their equivalents below the retained object nodes. Since this might create new graph constellations that can be compressed, the color bit possibly marking the descendant field reference nodes as visited is reset.

Since these modifications always preserve object nodes and do not unify subgraphs with different escape states, the effect on the results is negligible. However, the compile time required for alias analysis has improved by an order of magnitude.

(ev. ~~kte~~ in diesem Kapitel Paragraph über SPCA)

wunderbar!