Reference Counting Deeply Immutable Data Structures with Cycles: an Intellectual Abstract

Matthew J. Parkinson
Microsoft Azure Research
Cambridge, United Kingdom
mattpark@microsoft.com

Sylvan Clebsch
Microsoft Azure Research
Austin, USA
sylvan.clebsch@microsoft.com

Tobias Wrigstad
Uppsala University
Uppsala, Sweden
tobias.wrigstad@it.uu.se

Abstract

Immutable data structures are a powerful tool for building concurrent programs. They allow the sharing of data without the need for locks or other synchronisation mechanisms. This makes it much easier to reason about the correctness of the program.

In this paper, we focus on what we call *deep immutability from freeze*, that is, objects are initially mutable, and then can be frozen, and from that point on the object and everything it refers to (transitively) can no longer be mutated. A key challenge with this form of immutability is "how to manage the memory of cyclic data structures?" The standard approach is to use a garbage collector (GC), or a back-up cycle detector. These approaches sacrifice the promptness of memory reclamation, and the determinism of memory usage.

In this paper, we argue that memory underlying an immutable data structure can be efficiently managed using reference counting even in the presence of cycles, based on the observation that the cycles are themselves immutable. Our approach takes a classic algorithm for calculating strongly connected components (SCCs) and managing equivalence classes with union-find (UF), and combines them so that the liveness of each SCC can be tracked efficiently using only a single reference counter. The key observation is that since the graph is unchanging, we can calculate the SCCs once, in time that is almost linear in the size of the graph, and then use the result to reference count at the level of the SCCs. This gives precise reachability information, and does not require any backup mechanism to detect or handle cycles.

CCS Concepts: **• Software and its engineering → Garbage collection**;

Keywords: Reference Counting, Immutable Data Structures, Strongly Connected Components, Union-Find

ACM Reference Format:
Matthew J. Parkinson, Sylvan Clebsch, and Tobias Wrigstad. 2024. Reference Counting Deeply Immutable Data Structures with Cycles: an Intellectual Abstract. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3652024.3665507

1 Introduction

Immutable data structures are a powerful tool for building concurrent programs. The core of most functional languages is immutable data structures [19], although recent years have seen increased interest in safe in-place updates of unique or linear data in functional languages like Koka [23].

Immutable data structures allow the sharing of data without the need for locks or other synchronisation mechanisms. This makes it much easier to reason about the correctness of the program.

There are many forms of immutability in the literature. We can categorise these many forms with two axes: temporal, and spatial. The temporal axis is "when does immutability become enforced?" The two common cases are: *immutable from construction*, when allocating an object we must know the value of the fields, and they can never be changed from that initial value; and *immutable from freeze*, an object is initially mutable, but can be "frozen", and from that point on is immutable. The spatial axis is "how deep does immutability go?" The two common cases are: *shallow*, immutable objects can reference mutable objects; and *deep*, immutable objects can only reference immutable objects.

In this paper, we focus on what we call *deep immutability from freeze*. This is the form of immutability that is found in the Pony and System C♯ programming languages [7, 8, 11]. Neither of these languages exploit the immutability to manage the memory of the objects. They use a classic tracing GC to discover when the objects become unreachable. In these languages immutable state is used to share data between actors/threads meaning the objects are likely long-lived, which means the GC will scan that numerous times.

An alternative approach to tracing GC is to use reference counting. Reference counting has the advantage of promptness of memory reclamation, and the determinism of memory usage. However, as the objects become immutable after the freeze operation it is possible for the frozen objects to form cycles. Cycles are challenging for reference counting as they can lead to memory leaks [18]. Back up cycle detection can be used, but would lose the advantage of promptness of memory reclamation, and would result in the object graph being scanned multiple times.

In this paper, we argue that memory underlying an immutable data structure can be efficiently managed using reference counting even in the presence of cycles. Our approach takes a classic algorithm for calculating strongly connected components (SCCs) and managing equivalence classes with union-find (UF), and combines them so that the liveness of each SCC can be tracked efficiently using only a single reference counter. SCCs are used to factor the graph into an equivalence relation, and a directed acyclic graph. The key observation is that since the graph is unchanging, we can calculate the SCCs once, in time that is almost linear in the size of the graph, and then use the result to reference count at the level of the SCCs, which are acyclic. This gives precise reachability information, and does not require any backup mechanism to detect or handle cycles.

The paper is structured as follows: In section 2 we give a brief overview of the required background. In section 3 we present the algorithm. In section 4 we give a proof of correctness. In section 5 we give a complexity analysis. In section 6 we evaluate the performance of the algorithm and discuss its potential applications. In section 7 we discuss related work. In section 8 we conclude.

2 Background

Our algorithm depends on two classic algorithms: union-find and strongly connected components. We give a brief overview of the required background in this section.

2.1 Union-Find

Union-find is a well-known algorithm [26] for maintaining equivalence classes. The algorithm supports two operations, `find` and `union`. The `find` operation takes an element and returns the *representative* (see below) of the equivalence class. The `union` operation takes two elements and merges the equivalence classes that the elements are in.

The algorithm maintains a forest of inverted trees, where each tree represents an equivalence class. The root of the tree is called the *representative* of the equivalence class. With this representation, the `find` operation is simply a traversal up the tree to the root, and the `union` operation is a merge of the two trees by making one of the roots a child of the other root.

The core challenge with union-find is to minimise the distance `find` must travel from any element to its representative. There are two key optimisations in the literature to achieve almost linear time in the number of operations.

The first optimisation is to make the inverted tree balanced. The optimisation involves tracking the maximum inverted tree depth, called the *rank*, and always merge the tree with

```
def find(r):
  match r.status
  | REP(ptr) =>
    let result = find(ptr);
    r.status = REP(result);
    result
  | _ => r

def rank(r):
  match r.status
  | RANK(N) => N

def union(r1, r2):
  r1 = find(r1); r2 = find(r2);
  if (r1 == r2) return false;
  if (rank(r1) > rank(r2))
    r1,r2 = r2,r1; // swap r1 and r2
  else if (rank(r1) == rank(r2))
    r1.status = RANK(rank(r1) + 1);
  r2.status = REP(r1);
  true
```

Figure 1. Pseudo-code for union-find.

the smaller rank into the tree with the larger rank. If the two trees have the same rank, then the rank of the merged tree is considered one larger. By doing this, the maximum path in the inverted tree is at most $\log_2 n$ where n is the number of elements in the tree. The second optimisation is to compress the path from an element to its representative. With many union operations the path from any particular element to the root can become long. Path compression is the optimisation of making the path from any element to the root shorter. This can be done by either making the parent of every element on the path the root, or pointing each element on the path to its grandparent. When combined with the earlier balancing, both approaches have been shown to achieve the same complexity [27].

We can implement union-find using a single, `status`, field in each element, that contains either:

`RANK(N)` the object is a representative, and has a maximum tree depth of `N`; or

`REP(ptr)` the object is part of an SCC, and its representative is reachable from `ptr`.

We give a pseudo-code implementation of the two operations in Figure 1. The implementation performs both path compression and balancing. Note that we use the compress to the root in the `find` operation.

2.2 Strongly Connected Components (SCCs)

Let us define a graph, G, as a pair of a set of nodes, N, and a set of edges, E, where each edge is a pair of nodes. We define

SCCs as the symmetric core of reachability:

$$\mathrm{SCC}_{(N,E)} \quad = \quad E^* \cap (E^*)^{-1}$$

where we use R^* to denote the reflexive and transitive closure of R, and R^{-1} to denote the inverse of the relation R.

Informally, this can be explained as two nodes are in an SCC iff there is a path in both directions between them. Note that SCC is an equivalence relation, and that the equivalence classes are the strongly connected components of the graph. We refer to the edges between SCCs as external. We can form a quotient of a graph using the SCC equivalence relation, and that quotient is a directed acyclic graph (DAG). First, we define a mapping from an element to the set of equivalent elements as follows:

$$n/R = \{n' \mid n\ R\ n'\}$$

We write N/R where we lift the mapping to sets of elements. We can quotient the edges, E, of a graph by an equivalence relation, R, as follows:

$$E/R = \{n_1/R, n_2/R \mid n_1\ (E \setminus R)\ n_2)\}$$

Theorem 2.1. *The graph* $((N/SCC_{(N,E)}), (E/SCC_{(N,E)}))$ *is acyclic.*

3 Algorithm

The core observation of our memory management algorithm is the following:

> By factoring a graph into strongly connected components, we can use reference counting at the level of SCCs to manage its memory.

Strongly connected components can be used to factor an arbitrary graph into an equivalence relation, and a directed acyclic graph. This means that reference counting at the level of SCCs does not have to handle cycles, they are handled by the SCC algorithm. As we are focussing on immutable state, we never have to recalculate the SCCs of the graph after the initial freeze operation.

This section splits the implementation into three core pieces: `freeze` which calculates the strongly connected components of a graph and the reference counts, `dispose` which deallocates the memory of an SCC, and removes references to all the components it can reach, and basic reference counting operations (`acquire` and `release`).

3.1 Freeze

The `freeze` function is responsible for calculating both the SCCs of the graph, and each SCC's external reference count. The function uses union-find to represent the SCCs, and the reference count is stored in the SCC's representative. The freeze operation performs a linear number of `find` operations on the graph, and is hence almost linear in the size of the graph.

Our algorithm is based on the Purdom's path-based algorithm [22] for calculating the SCCs of a graph. Path-based algorithms keep track of the tentative SCCs on the path from the root of the traversal to the current node, and when they detect a back edge they merge all the SCCs on the path from the current node to the target of the back edge. To track the SCCs on the path from the root of the traversal to the current node, we use the union-find datastructure. This differs from the classic presentation but provides the union-find structure we need for reference counting at the level of SCCs.

To implement the algorithm, we extend the representation from the union-find algorithm earlier with two new states, `UNMARKED` and `RC(`N`)`. Thus, it can contain one of four states:

`UNMARKED` the object has not been visited by the algorithm yet;

`RANK(`N`)` the object is on the pending stack, and is part of an SCC with rank N;

`REP(`ptr`)` the object is part of an SCC, and its representative is reachable from ptr; or

`RC(`N`)` the object is the representative of an SCC with a current external reference count of N.

We give the freeze algorithm in pseudo-code in Figure 2. We use **`def`** to define functions; **`atomic`** to indicate that the function is atomic; and **`match`** to perform a pattern match on the status field (similar to Rust or OCaml). The pattern match on lines 29 to 31 will return true if it has the status of `RC` with the value 1. If the status is `RC` with a value greater than 1, then the status is decremented by 1, and the function returns false.

The algorithm takes a root object, and calculates the SCCs of the graph reachable from the root, and their incoming external reference count. We present the algorithm as a recursive function, `freeze_inner`, that takes a node, and modifies the captured `pending` stack. The recursive function performs a depth-first search on the graph, and pushes nodes onto the `pending` stack as they are first visited (line 40). Importantly, the algorithm performs operations both on the pre-order and post-order of the depth-first search. The post-order step (lines 43 to 45) is responsible for processing a complete SCC, and giving it its initial (external) reference count. The pre-order steps are responsible for adding unexplored nodes to the `pending` stack (line 40), finding additional edges into completed SCCs and increasing their reference count (lines 49 to 50), and merging SCCs when a back edge is detected (lines 46 to 48).

We illustrate the execution of the algorithm in fig. 3. The algorithm starts at Node A, which has two outgoing edges (shown in sub-diagram 1). It takes the first edge to Node B and adds it to the pending stack (line 40) (shown in subdiagram 2). As B has no outgoing edges, then we move to the post-order step (line 43). This observes that the current node is the top of the pending stack, and hence it is a completed SCC, and gives it a reference count of 1 and removes it from the pending stack (shown in sub-diagram 3).

```
22 def incref(r):
23   atomic
24     match r.status
25     | RC(n) => r.status = RC(n+1)
26
27 def decref(r):
28   atomic
29     match r.status
30     | RC(1) => true
31     | RC(n) => r.status = RC(n-1); false
32
33 def freeze(r):
34   pending = []
35
36   def freeze_inner(x):
37     match find(x).status
38     | UNMARKED =>
39       x.status = RANK(1);
40       pending.push(x);
41       for each f in x
42         freeze_inner(x.f);
43       if (pending.peek() == x)
44         pending.pop();
45         find(x).status = RC(1);
46     | RANK(N) =>
47       while (union(x, pending.peek()))
48         pending.pop();
49     | RC(N) =>
50       incref(find(x));
51
52   freeze_inner(r)
```

Figure 2. Pseudo-code for the freeze algorithm.

The algorithm then explores the second outgoing edge of A to Node C, and adds C to the pending stack (shown in sub-diagram 4). This has two outgoing edges, and the algorithm follows the first to Node D and also adds it to the pending stack (shown in sub-diagram 5). The algorithm then follows the first outgoing edge of D to Node B. This node is part of a completed SCC and hence the algorithm increases its reference count (shown in sub-diagram 7) using lines 49 to 50.

The algorithm then explores the second outgoing edge of D to Node C. This is an edge to an SCC on the pending stack. The algorithm detects this as a back edge, and hence merges the SCCs (shown in sub-diagram 8) using lines 46 to 48. This removes D from the pending stack. The postorder step for D, does not create a completed SCC as it is not the top of the pending stack.

The algorithm then explores the second outgoing edge of C to Node E, and adds it to the pending stack (shown in sub-diagram 9). Node E has a single outgoing edge to Node A. This is a back edge to an SCC on the pending stack. It then unions A with E (shown in sub-diagram 10), and then A with C (shown in sub-diagram 11).

The algorithm then executes the postorder step for E and C, but neither do anything as they have been removed from the pending stack. The final post-order step for A detects that it is the top of the pending stack, and hence it is a completed SCC. It gives it a reference count of 1, and removes it from the pending stack (shown in sub-diagram 12).

The end result is a graph with two SCCs, one containing just B, and the second containing all the other nodes. The reference count of the SCC containing all the other nodes is 1, and the reference count of B is 2.

3.2 Dispose

Once we have calculated the SCCs of the graph and their reference counts, we can use this to manage the memory associated with them. The second key challenge is the correct deallocation of an SCC. The key observation is that if the reference count of an SCC is 0, then it is safe to deallocate all the nodes in the SCC, and remove all the references from the SCC.

In fig. 4 we give the pseudo-code for the dispose algorithm. The traversal order is subtle as it must deallocate each SCC in one go. The algorithm uses two stacks, `dfs` and `scc`, and a `free_list`. The `dfs` stack is used to traverse the DAG of SCCs, and the `scc` stack is used to traverse the SCCs themselves. Once an object has been traversed, it is added to the `free_list`, which is processed when all the elements of the SCC have been traversed.

The algorithm uses the same `status` field from before, but with a different interpretation. It uses a `PROCESSING` state to indicate that the object has been visited by the algorithm. If an object is in a `PROCESSING` state, then it is currently in one of the three stacks: `dfs`, `scc`, or `free_list`.

When we begin processing an SCC for disposal, we always start with the representative of the SCC, and mark it as `PROCESSING` (lines 59 and 73). If the representative pointer points to an object that is marked as `PROCESSING`, then this node must also be in the same SCC, but has not been visited yet. This is detected on lines 68 to 70, and the object is added to the `scc` stack, and the algorithm continues to process the SCC.

Once the algorithm has finished processing an SCC, it can deallocate all the objects in the SCC (lines 75 to 76). It then pops the next SCC from the `dfs` stack (lines 60 to 61), and continues processing the next SCC (lines 62 to 73).

3.3 Reference Counting

In fig. 4, we provide two methods for modifying the reference count of an object: `acquire` and `release`. These objects are standard except that instead of manipulating the object, they first follow the representative pointer to the representative

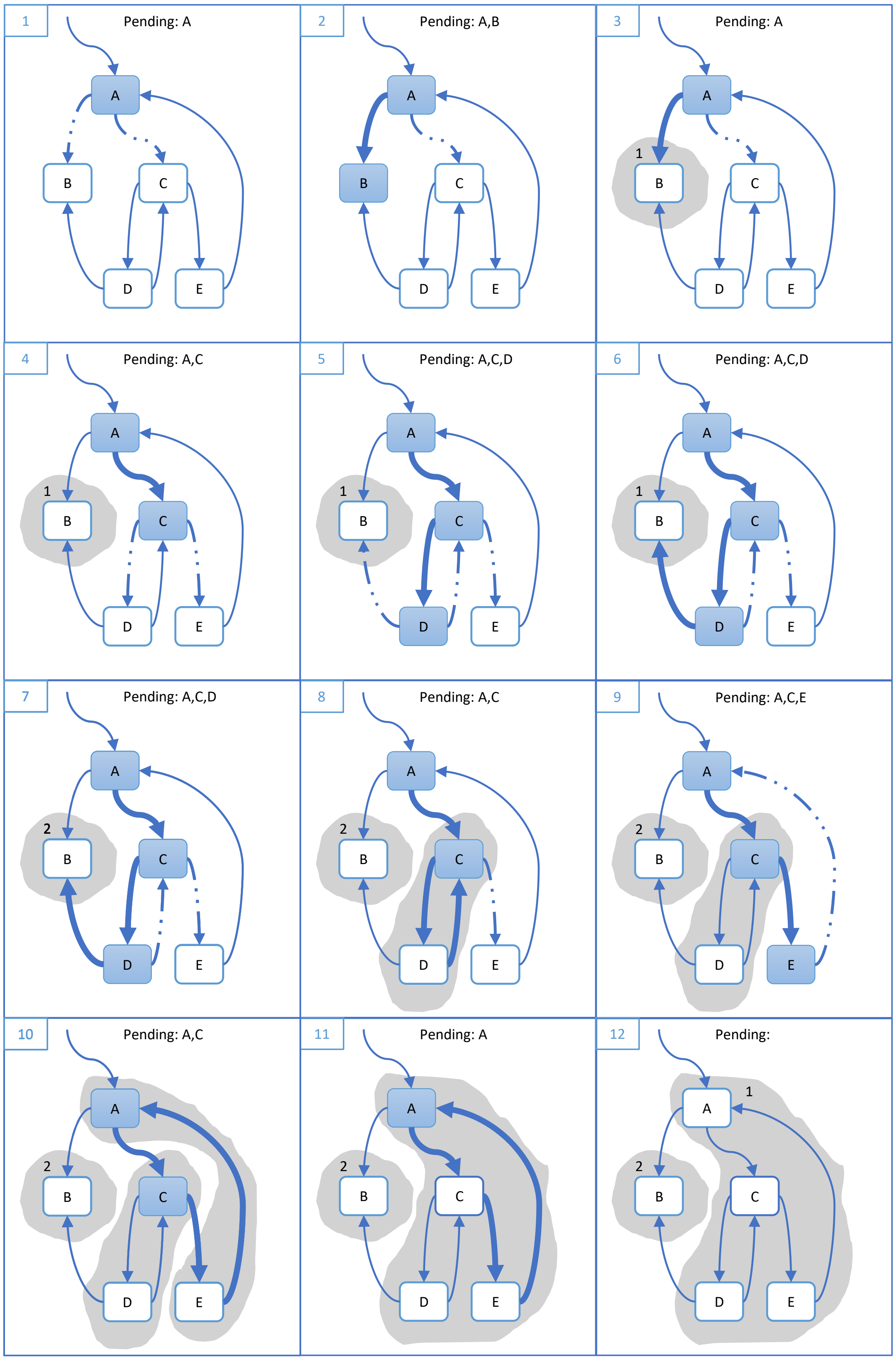

Figure 3. Calculating the strongly connected components of a graph.

```
def add_stack(stack, r):
  stack.push(r)
  r.status = PROCESSING

def dispose(r):
  dfs = []; scc = []; free_list = [];
  add_stack(dfs, find(r));
  while (!dfs.empty())
    scc.push(dfs.pop())
    while(!scc.empty())
      x = scc.pop()
      free_list.push(x)
      for each f in x
        n = find(x.f);
        match n.status
        | PROCESSING =>
          if (x.f != n)
            add_stack(scc, x.f);
        | RC(_) =>
          if (decref(n))
            add_stack(dfs, n);

    while(!free_list.empty())
      deallocate(free_list.pop())

def acquire(r):
  incref(find(r))

def release(r):
  if (decref(find(r)))
    dispose(r)
```

Figure 4. Pseudo-code for the dispose algorithm.

of the SCC, and then manipulate the reference count of the SCC.

If the reference count reaches zero, then the object is passed to `dispose` and the memory of that SCC, and potentially others that it can reach, is deallocated.

3.4 Note on the Implementation

The freeze algorithm in this section is presented as a recursive algorithm for clarity. An implementation naturally uses a work-list as object graphs can often be too large to process with a recursive algorithm. This is indeed the case for the benchmarked code (see next section) which processes both pre-order and post-order steps of a depth-first search.

4 Correctness

The core correctness of the algorithm depends on the following invariant:

> A node whose representative is marked as RC can only reach other nodes whose representative is marked as RC.

This invariant is maintained by the freeze algorithm. The post-order step (lines 43 to 45) will only be called once there are no nodes reachable that are in an `UNMARKED` state. The current node is necessarily still `PENDING`. If this node is not the top of the `pending_stack`, then it could reach something visited earlier, and the SCC may have out-going edges that have not been explored yet. If this is the case, then it remains in the pending state. Otherwise, it is the top of the stack, and we have followed every field out of this SCC, and it is safe to convert it into an `RC`, because it cannot reach anything that is `UNMARKED`.

Based on this invariant, we can see that if one starts with all nodes `UNMARKED`, then the algorithm will only create DAGs of SCCs. This is because we create `RC`s one at a time, and thus cannot create a cycle of `RC`s.

We can also see that this invariant ensures that upon termination there are no reachable nodes marked as `UNMARKED` or `PENDING`. The last step of the algorithm is guaranteed to turn the original root into an `RC`, and thus everything reachable must be an `RC`.

5 Complexity

The freeze algorithm is almost linear in the size of the graph with its complexity being:

$$O(|E| \cdot \alpha(|N|))$$

where $\alpha(x)$ is the inverse-Ackermann function. As the algorithm uses union-find to represent the SCCs, the complexity of the freeze algorithm takes on the inverse-Ackermann cost for find.

To prove this complexity bound, we observe two things:

- Each edge is followed at most once.
- Each node is added at most once to the pending stack.

Both of these properties are true because we only add edges when a node has not previously been visited (`UNMARKED`), and we only add nodes to pending on the first visit.

The most subtle part of the complexity argument is dealing with the loop on line 46. This loop removes one element from the `pending` stack for each iteration. We know that `x` has the same representative as a single element in the pending stack. This means that the loop is guaranteed to terminate once it reaches that element. Moreover, as each iteration of the loop body removes an element from the pending stack, the loop can only iterate at most $|N|$ times across the entire run of the algorithm. Thus, this loop can contribute at most

$$O(|N| \cdot \alpha(|N|))$$

to the complexity of the algorithm, as the `union` on each iteration performs two `find` operations.

The core algorithm visits each edge once (lines 41 to 42). Each visit will involve two to three calls to `find` in addition to the `find` calls in the iterations of the nested loop. Hence, the complexity of the algorithm is:

$$O((|N| + |E|) \cdot \alpha(|N|))$$

As the algorithm is only concerned with reachable elements of the graph, we can assume $|N|$ is less than $|E|$. We can therefore simplify to

$$O(|E| \cdot \alpha(|N|))$$

The dispose algorithm is also almost linear in the size of the graph with its complexity also being:

$$O(|E| \cdot \alpha(|N|))$$

The argument for this is simpler than the freeze algorithm. The dispose algorithm visits each edge once, and each visit involves a single call to `find`.

6 Evaluation

We split the evaluation of the algorithm into two parts. A quantitative evaluation of the performance of the algorithm relative to other operations, and a qualitative evaluation of the potential applications of the algorithm.

6.1 Performance

This section provides a small quantitative evaluation of the algorithm for a range of data structures to illustrate the performance characteristics of the algorithm. Our aim is to provide the reader with an idea of the cost relative to other operations such as construction of the data structure. We have not attempted to compare the algorithm to other memory management algorithms, which we believe is beyond the scope of an intellectual abstract.

We have implemented this algorithm as part of the Project Verona runtime.[1] The micro-benchmark directly calls the operations on the runtime using C++ and is available on GitHub.

To evaluate the performance of our algorithm, we implemented a few common data-structures that represent different types of graphs. We chose various graphs with different types of cycles, and chains. All the graphs were implemented using binary nodes, and using linked lists to form larger logical nodes where required. Each node contained the `status` field from the previous section, a descriptor field containing how to trace the object, and two pointer fields that were used to form the graph.

We chose the following data structures as they represent a wide range of graphs:

Linked List a linked list with a single forward pointer.

Doubly-Linked List a doubly-linked list with forward and backward pointers.

Balanced Binary Tree a binary tree where the right subtree has at most one more node than the left subtree.

Balanced Binary Tree with Leaf to Root cycle same as the balanced binary tree, but instead of using a null pointer for the empty tree, a pointer to the root is used.

Tree with parent pointers a balanced binary tree with a pointer to the parent. Two nodes are used to encode a node with three pointers.

Balanced 4-tree a balanced tree where each node has up to four children. To represent the nodes of the tree a singly-linked list of node objects is used.

We then measured the time taken to allocate the graph (Alloc), freeze it (Freeze), and then dispose of it (Dispose). Additionally, our runtime contains a simple mark and sweep collector that we ran before freezing the graph (Trace). We use `snmalloc` [14] to reduce the noise associated with allocating nodes compared to the system allocator. Each benchmark was run with a variety of sizes of the graph,

$$\bigcup_{1 \leq i \leq 22} (\{4, 5, 6, 7\} * 2^i)$$

We repeated the run $max(100000/\texttt{graph\_size}, 50)$ times to reduce the noise in the benchmark. We have an additional warm-up run that is not reported to remove any set up costs. The presented results are normalised to the time taken per binary node to make comparison between the different data structures easier.

The results are presented in fig. 5. The shaded area for each line represents the 95 percentile interval. The graph shows that the Freeze operation is generally about twice the cost of the Trace operation. The worst case is the Linked List, where it is $\tilde{2}$.5x the cost. This implies that this approach could lead to a net win if the frozen graph is going to be traced more than a couple of times.

The graph shows that the freeze operation is cheaper than allocation for small sizes for all the acyclic benchmarks. For the linked list, the freeze operation becomes more expensive than allocations between 10^5 and 10^6 nodes. This is due to the state required for both the depth-first search, and the pending stack. In the case of the linked list example, these are both linear in the size of the graph. We believe this means the cache will be under more pressure, and the cost of the freeze operation will increase. In the balanced binary tree example, the freeze operation stays cheaper than allocation as these other data structures are only logarithmic in the size of the graph.

For the cyclic examples ("Doubly Linked List", "Balanced Binary Trees with Leaf to root cycle", and "Tree with Parent Pointers"), freeze generally costs a bit more than allocation, but stays under double the cost. These larger costs only occur at the largest sizes again due to additional cache pressure. The traversal order of the graph can have a dramatic effect on the cost of the freeze operation. If you compare the "Doubly Linked List" and "Doubly Linked List (Reverse field order)"

[1] https://github.com/Microsoft/verona-rt

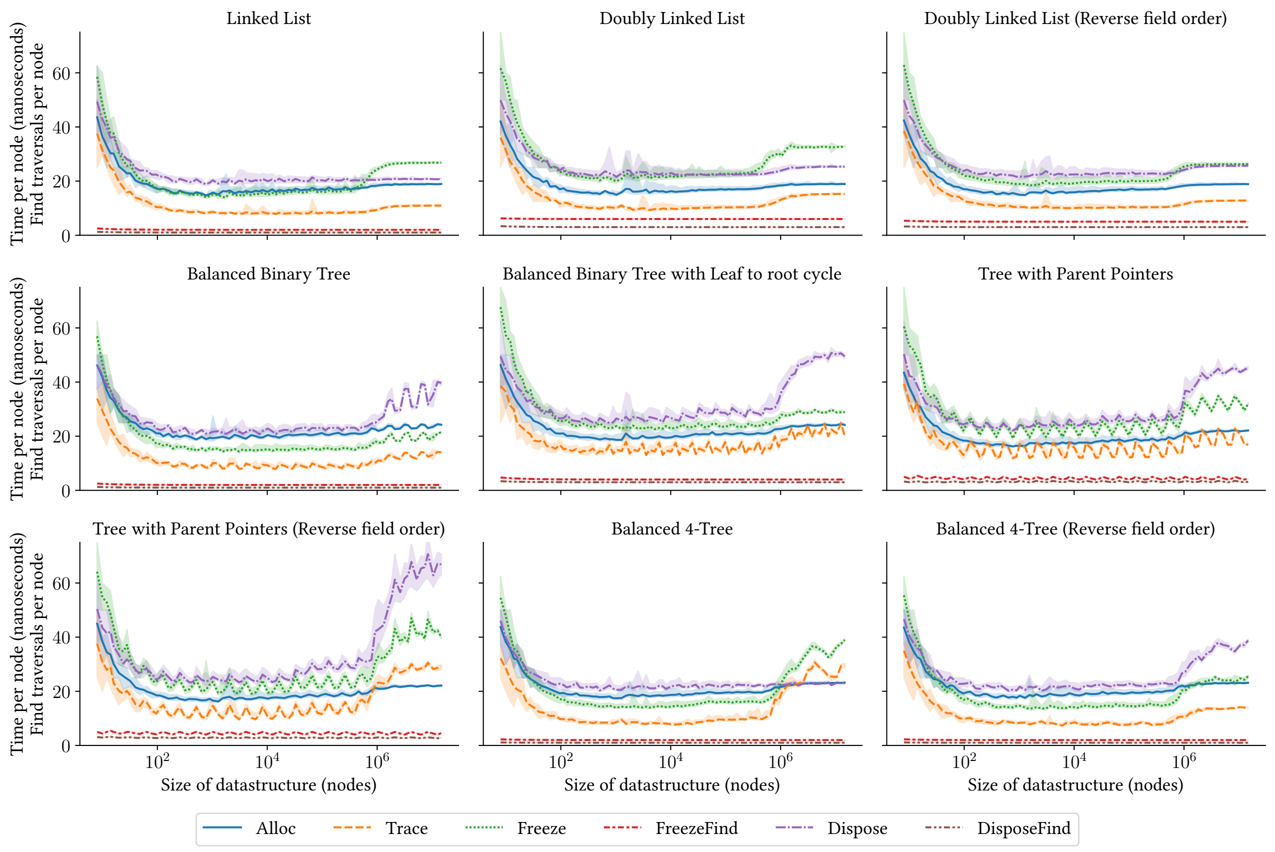

Figure 5. Latency of various stages of processing different data structures.

benchmarks, the latter performs better. This is due to the cycles being discovered instantly with the "Reverse field order", which means the pending stack is only ever a single element. Whereas the "Doubly Linked List" has a pending stack that is linear in the size of the graph. Moreover, in the reverse order the previous pointer is followed, and then the next pointer. This means the two accesses are very close in time, and likely to remain in cache. This is not the case for the "Doubly Linked List" benchmark as the two accesses are far apart in time.

The graphs also contain the time taken to run the dispose operation. This is generally slightly higher than the cost of allocation in the acyclic cases, and in the cyclic cases as the graph grows the dispose operation becomes the most costly. The example here has SCCs that cover the entire graph, and hence the dispose operation must first traverse the entire graph to calculate the SCC, and only then can it begin to deallocate it.

Overall, the results again show that in the majority of scenarios the cost of dispose is close to the cost of constructing the graph.

The algorithm's complexity is almost linear, but empirically it seems the costs increase as the size of the data structure grows. For each of the data structures, we also plot the total length of find traversals divided by the number of nodes, "FreezeFind" and "DisposeFind". These operations can be seen to not be growing as the data structure grows. This shows empirically that the algorithm's find traversal lengths are linear in the size of the graph. The lack of linearity in the reported time is due to cache effects, which become more pronounced as the size of the graph grows.

6.2 Applicabilty

We are currently building a new language runtime that takes advantage of our algorithm. We believe the algorithm is applicable to other existing languages.

Our algorithm requires a language design for deeply immutable state with a freeze operation. There are a couple of notable languages that have this feature: System C♯ [11] and Pony [7, 8]. Both of these languages have exactly the right type of immutable state for our algorithm. System C♯ was an extension of C♯ with various capabilities to enable data-race free concurrency. It has a freeze operation that can be used to make regions of memory deeply immutable. The freeze operation is a purely type-level operation and has no runtime operation or representation associated with it. The language runtime was not designed to take advantage of the frozen memory, and thus used a standard tracing garbage collector to handle the memory. Pony also uses the same notion of deep immutablity with a freeze operation. It does not represent immutability at the runtime level and thus uses a tracing garbage collector to manage the lifetime of the immutable state.

Both of these language designs could benefit from our algorithm. If the freeze operation was turned into a runtime operation, then the language runtime could use our algorithm to manage the memory of the immutable state. This would be a substantial change to either implementation. It is not possible to make this change to System C♯ as it is not publicly available. Making this change to Pony would be a substantial amount of work, and would likely require a complete redesign of the runtime's representation of memory management.

We believe, in these languages, any long-lived immutable state would benefit from our algorithm. As the immutable state in these languages is primarily used to share data between threads/actors, the freeze operation is likely to be used on long-lived data. If the data survives more than a couple of rounds of GC, then the freeze operation will be cheaper than the cost of the GC tracing.

Access to the immutable objects would require reference counting. However, as the state is deeply immutable holding reference counts on dominating references would be sufficient to ensure the memory is not collected. This would drastically reduce any intermediate reference count operations, but would require compiler support.

There is also potential applications in more mainstream languages. In an analysis of Java object connectivity, Hirzel et al. study the object graphs of 22 different benchmarks and find that between 0.4 % and 78.8 % of all objects (14.4 % on average) are part of "non-trivial SCC's", which are defined as SCC's with more than two objects [13]. Analysing the DaCapo benchmark suite, Brandauer and Wrigstad [3] found that 54.9 % of all objects were shallow-immutable, 47.9 % were deeply immutable, and 72.6 % of all classes only create stationary objects (each field is written once, but not necessarily during object creation). Revisiting and combining these two studies would be an interesting direction for future work that could demonstrate the potential application of our algorithm to Java.

7 Related Work

7.1 Freezing Objects

The world of programming languages contains many examples of creating mutable data structures and then "freezing" them with various amount of language support. For example, Ruby comes with a built-in `freeze` method that makes an object unmodifiable. The immutability is notably *shallow*, meaning it only freezes the receiver object, not its transitive closure. This is arguably a reasonable design in an untyped language, where it can be extremely difficult to reason about the propagating effects of a deep freeze operation. JavaScript supports a similar, also shallow, `freeze` method.

An early application of capturing transition from mutable to immutable in static types is found in Parameterized race-free Java [2] (PRFJ). PRFJ permitted the creation of immutable objects by destructively reading a variable of unique type and storing the result in a variable with read-only type. Freezing in PRFJ affects all objects that are dominated (aka owned [6]) by the unique object. PRFJ's adoption of ownership types [6] permits the distinction between paths internal to a structure (leading to transitively owned objects) and paths to external objects (not owned). Thus, it is close in spirit to our freeze operation, but permits outgoing references from an immutable aggregate to mutable objects. Given that types can be used to tell references to outside of the structure from references to inside of the structure, our algorithm could be successfully applied to PRFJ's freeze-equivalent, as well as to several other works based on ownership types that apply similar tricks [5, 21, 28]. Type systems that permit flexible object initalisation [10, 12] also permit staged construction of immutable objects where a typestate change is the static equivalent of our freeze operation, like in PRFJ. PRFJ did not leverage immutability for memory management.

7.2 Clusters and Cycles

A close work in spirit is Cutler and Morris' Clustered Collection [9], which was implemented as a modification of the Racket collector. The goal of clustered collection is to reduce pause times and repeated scanning of object structures by identifying clusters which are kept alive by a specific dominating object. Liveness of the dominator thus governs liveness of its dominated structure. Note that clusters are not SCC's – the root of a tree is a dominator for all subtrees. Like our SCC's, clusters can have outgoing pointers, which must be kept track of as they act as roots as long as their containing cluster is still alive. Foreshadowing our work, the main challenge with clustered collection is that clusters are mutable. A write that replaces a reference to an object in the same cluster forces the cluster to be traced fully during the next GC cycle for completeness.

The problems of cycles in a reference counted systems leading to memory leaks has been long known [18]. There are two common approaches to handling cycles. The first is to add a second class of reference, a weak reference, that does not prevent the object from being collected [4]. If any cycle contains at least one weak reference, then the cycle can be collected. The second approach is to introduce a backup mechanism that can detect cycles and collect them. Martínez *et al.* [17] present a backup tracing mechanism for reference counted systems to find cycles. Lins [15, 16] provides numerous optimisations to reduce the cost and frequency of the back-up tracing. Python uses a different algorithm to detect and collect cycles in its reference counted system [24], which can cope with some untraceable objects. The first concurrent cycle detector was presented by Bacon *et al.* [1]. All these algorithms are for detecting cycles in reference counts of mutable object graphs. They take the classic lazy approach of GC of finding the cycles when collection occurs, rather than calculating the cycles of the graph up front. By restricting our attention to deeply immutable state, we can calculate the cycles of the graph up front, and thus avoid the need to recalculate the cycles as the program evolves.

7.3 Immutability in Functional Languages

Functional languages such as Haskell and ML have a lot of immutable state, but it is not deeply immutable. ML allows for reference cells that can be updated, which would alter the topology of the graph, and thus means the SCCs would no longer be up-to-date. Haskell uses lazy evaluation, and thus the connectivity of the object graph can change as graph is evaluated. It may be possible to create a strict and freeze operation that would allow the algorithm to be used in Haskell on fully evaluated object graphs.

There are *pure* functional languages that are built around deeply immutable state. It would be interesting to apply our algorithm to these languages like Idris or Elm. Currently, Idris compiles to Racket, and Elm compiles to JavaScript, and thus depend on a classic tracing garbage collector.

7.4 SCC's in GC

As part of work on automatic parallelisation on the JVM, Österlund and Löwe [20] developed a modified version of Tarjan's algorithm [25] for finding SCC's that only consist of immutable objects, piggybacking on tracing GC. The modified Tarjan's algorithm includes a purity analysis that detect objects with only immutable fields. Thus, the SCC's detected are just like ours: SCC's of deeply immutable objects. The realisation driving this work is that method calls on receivers in different such SCC's are guaranteed to not mutate anything and therefore cannot interfere. This means the method calls are safe to execute in parallel from the perspective of data races. In contrast, the work in this paper builds SCC's at freeze-time knowing that the graph consists of only immutable objects. Österlund's and Löwe's work also does not use SCC's to collect garbage, only to drive automatic parallelisation of the program.

8 Conclusion

We have presented a new memory management algorithm for deeply immutable state. The algorithm uses union-find to calculate the strongly connected components of the graph, and then uses reference counting at the level of the SCCs to manage the memory. We have shown that the algorithm is almost linear in the size of the graph, and empirically shown that the cost of the freeze operation is close to the cost of constructing the graph.

We believe the algorithm is applicable to languages that support deeply immutable state, and are currently building a new language runtime to take advantage of our algorithm.

Acknowledgements

We would like to thank Wes Filardo and the anonymous reviewers for their input on the paper. This research was supported by the Swedish Research Council through the projects Accelerating Managed Languages (2020-05346), Pervasive Memory Safety through Ownership Types (2023-05526), and by the Swedish Foundation for Strategic Research through the project Deploying Memory Management Research in the Mainstream (SM19-0059).

References

[1] David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, Berlin, Heidelberg, 207–235. https://doi.org/10.1007/3-540-45337-7_12

[2] Chandrasekhar Boyapati and Martin Rinard. 2001. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/504282.504287

[3] Stephan Brandauer and Tobias Wrigstad. 2017. Mining for Safety using Interactive Trace Analysis. In *Pre-proceedings of the Fifteenth International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*. https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-334817

[4] D. R. Brownbridge. 1985. Cyclic Reference Counting for Combinator Machines. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture* (Nancy, France). Springer-Verlag, Berlin, Heidelberg, 273–288. https://doi.org/10.1007/3-540-15975-4_42

[5] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5356)*, G. Ramalingam (Ed.). Springer, 139–154. https://doi.org/10.1007/978-3-540-89330-1_11

[6] David G. Clarke. 2003. *Object ownership and containment.* Ph. D. Dissertation. https://doi.org/10.26190/unsworks/8187

[7] Sylvan Clebsch. 2017. *'Pony': co-designing a type system and a runtime.* Ph. D. Dissertation. Imperial College London, UK. https://doi.org/10.25560/65656

[8] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816

[9] Cody Cutler and Robert Morris. 2015. Reducing pause times with clustered collection. In *Proceedings of the 2015 International Symposium on Memory Management* (Portland, OR, USA) *(ISMM '15)*. Association

for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/2754169.2754184

[10] Manuel Fahndrich and Songtao Xia. 2007. Establishing object invariants with delayed types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/1297027.1297052

[11] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. *SIGPLAN Not.* 47, 10 (oct 2012), 21–40. https://doi.org/10.1145/2398857.2384619

[12] Christian Haack and Erik Poll. 2009. Type-Based Object Immutability with Flexible Initialization. In *ECOOP 2009 – Object-Oriented Programming*, Sophia Drossopoulou (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 520–545.

[13] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. 2002. Understanding the connectivity of heap objects. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) *(ISMM '02)*. Association for Computing Machinery, New York, NY, USA, 36–49. https://doi.org/10.1145/512429.512435

[14] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management, ISMM 2019, Phoenix, AZ, USA, June 23-23, 2019*, Jeremy Singer and Harry Xu (Eds.). ACM, 122–135. https://doi.org/10.1145/3315573.3329980

[15] Rafael D. Lins. 1992. Cyclic reference counting with lazy mark-scan. *Inf. Process. Lett.* 44, 4 (dec 1992), 215–220. https://doi.org/10.1016/0020-0190(92)90088-D

[16] Rafael Dueire Lins. 2002. An efficient algorithm for cyclic reference counting. *Inf. Process. Lett.* 83, 3 (aug 2002), 145–150. https://doi.org/10.1016/S0020-0190(01)00328-3

[17] A. D. Martínez, R. Wachenchauzer, and R. D. Lins. 1990. Cyclic reference counting with local mark-scan. *Inf. Process. Lett.* 34, 1 (feb 1990), 31–35. https://doi.org/10.1016/0020-0190(90)90226-N

[18] J. Harold McBeth. 1963. Letters to the editor: on the reference counter method. *Commun. ACM* 6, 9 (sep 1963), 575. https://doi.org/10.1145/367593.367649

[19] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in scala.* Artima Inc.

[20] Erik Österlund and Welf Löwe. 2012. Analysis of pure methods using garbage collection. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (Beijing, China) *(MSPC '12)*. Association for Computing Machinery, New York, NY, USA, 48–57. https://doi.org/10.1145/2247684.2247694

[21] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. 2008. Ownership, Uniqueness, and Immutability. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings (Lecture Notes in Business Information Processing, Vol. 11)*, Richard F. Paige and Bertrand Meyer (Eds.). Springer, 178–197. https://doi.org/10.1007/978-3-540-69824-1_11

[22] Paul Walton Purdom. 1970. A transitive closure algorithm. *BIT Numerical Mathematics* 10 (1970), 76–94. https://doi.org/10.1007/BF01940892

[23] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 96–111. https://doi.org/10.1145/3453483.3454032

[24] Neil Schemenauer. 2000. Garbage Collection for Python. http://www.arctrix.com/nas/python/gc/.

[25] Robert E. Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

[26] Robert E. Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (apr 1975), 215–225. https://doi.org/10.1145/321879.321884

[27] Robert E. Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (mar 1984), 245–281. https://doi.org/10.1145/62.2160

[28] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and immutability in generic Java. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 598–617. https://doi.org/10.1145/1869459.1869509

Received 2024-03-22; accepted 2024-05-10