Who Owns the Contents of a Doubly-Linked List?

Dimi Racordon
LAMP, EPFL, Lausanne, Switzerland

Abstract

Despite their popularity, systems enforcing full ownership guarantees such as Rust leave many users frustrated with the inability to represent notionally self-referential data structures – e.g., doubly-linked lists – using first-class references. This frustration has motivated a number of proposals to relax on full ownership to support idioms common in languages with pervasive reference semantics. In this paper, we take a look at the way value-oriented languages address this issue and study representations of arbitrary graph-like data structures without references.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Software and its engineering → Imperative languages; Software and its engineering → Data types and structures; Software and its engineering → Runtime environments

Keywords and phrases self-referential data structures, ownership, mutable value semantics, performance

Digital Object Identifier 10.4230/OASIcs.Programming.2025.25

Supplementary Material *Software*: `https://github.com/kyouko-taiga/vimpl-2025-artifact.git`, archived at `swh:1:dir:a40a79138b886202067a19edaedcafb524016239`

Funding This project has been funded by the Swiss National Science Foundation project named "Capabilities for Typing Resources and Effect" (TMAG-2_209506/1).

1 Introduction

Ownership systems [1] typically prescribe that an object – as understood through the lens of object-oriented programming [11] – has a single *owner* enjoying particular priviledges on this object. In the most restrictive discipline, sometimes dubbed *full ownership* [2], all accesses to an object must go through its owner, effectively banning aliasing. That is, if an object a owns an object b, then a third object c may not refer directly to b.

An immediate advantage of this approach is that it fends against bugs stemming from unintended mutation through shared aliases [8]. It also upholds local reasoning, which benefits both humans and machines to form strong assumptions about a particular piece of code, even in concurrent settings [9]. Full ownership implies that data structures always form a tree since an object can only be referred to by its owner – i.e., parent. Hence, access to a particular object guarantees *exclusive* access without any synchronization mechanism. But then one may wonder whether such a discpline is at all practical in applications involving arbitrary graphs. This concern is not baseless. In Rust, for example, it is notorioulsy dfficult to design a doubly-linked list using common object-oriented idioms.

While some may lament at this situation [7], we remark that full ownership does not actually prevent the definition of notionally self-referencial data structures. In fact, it is not only possible to do so without bypassing the type system's restrictions – e.g., going unsafe in Rust [14] – but to do so in a way that simplifies the maintainance of structural invariants, matches formal descriptions, and supports efficient implementations. Hence, we argue that the percieved limitations of full ownership are due to unfamiliarity rather than lack of expressiveness. We examine two case studies to substantiate this claim:

1. the construction and update of a web graph; and
2. a doubly-linked list that leverages cache locality for efficiency.

Listing 1 Representation of a web graph with first-class references in Swift.

```
final class WebPage {
  let url: String
  var topic: String?
  var links: Set<WebPage> = []
}

let p1 = WebPage(url: "https://x.abc")
let p2 = WebPage(url: "https://y.def")
let p3 = WebPage(url: "https://z.ghi")
p1.links = [p2, p3]
```

We present these case studies in Swift, a programming language that supports both classical object-orientation and mutable value semantics (MVS) [9], a programming discipline that emphasizes value independence and bans shared mutable state. Like Rust, Swift can produce high-performance native code while remaining "safe by default", meaning that one cannot trigger undefined behavior using the safe subset of the language. The goal of our inquiry is to demonstrate that this subset is expressive enough to write practical and efficient notionally self-referential data structures.

2 Representing Arbitrary Graphs

Consider the problem of modeling the reference graph of a collection of web pages: vertices are pages and edges denote hyperlinks. Listing 1 shows a familiar way to implement such a graph in an object-oriented setting. Instances of the class `WebPage` represent a single page, identified by its URL. Each page contains a set of references to the pages that it refers to. In this particular example, `p1` contains hyperlinks to `p2` and `p3`.

This representation is not directly expressible within the constraints of full ownership because two web pages would not be able to refer to each other. Yet, this approach is appealing because it is convenient to explore the graph from a particular vertex. For example, one can implement a straightforward depth-first search to enumerate all URLs reachable from a particular page. It is also easy to add or remove an edge from the graph, provided that we hold a reference to its origin. For example, one can simply write `p2.links.insert(p1)` to state that `p2` contains a link to `p1`. Furthermore, this approach is likely familiar to anyone having experience with a mainstream programming language.

Despite this convenience, however, the lack of clear ownership makes it difficult to reason about the notional value – i.e., the meaning of a value, notwithstanding its concrete representation – of the graph in general. How does one compare two graphs for equality? How does one compute the hash value of a graph? How does one copy a graph? How does one serialize a graph to a file? Taking a step back, one may thus wonder whether there exist a principled way to approach this problem.

2.1 Whole/Part Relationships

Remark that the design of `WebPage` leaves the removal of a vertex from the graph rather challenging. To keep our representation consistent, all web pages referring to the one being removed must be updated. Unfortunately, our design does not natually give us a whole view of the graph. Since there is no obvious root nor any guarantee that the graph is connected, there is no obvious way to collect the set of vertices contained in the graph. We can fix this

Listing 2 Extension of Listing 1 to support the removal of a web page.

```
final class WebPage {
  let url: String
  var topic: String?
  var container: WebGraph?
  var links: Set<WebPage> = []
  var backlinks: Set<WebPage> = []
  func remove() {
    for p in backlinks { p.links.remove(self) }
    for p in links { p.backlinks.remove(self) }
    container!.pages[url] = nil
  }
  func addLink(to p: WebPage) {
    assert(container[p.url] != nil)
    links.insert(p)
    p.backlinks.insert(self)
  }
}
final class WebGraph {
  var pages: [String: WebPage] = [:]
  func insert(url: String) -> WebPage {
    pages[url].setIfNil(WebPage(url: url, container: self))
  }
}
```

issue by storing all vertices in one collection but this approach introduces a new problem: we must guarantee that elements inserted in the `links` of a `WebPage` are also present in this collection. Another complication is that updating the outgoing edges of all the web pages is expensive. We can fix this issue by adding backward links in each web page but again, doing so introduces new structural invariants to maintain. The resulting program is shown in Listing 2. Significant complexity has been added to maintain the graph's invariants, making the whole data structure much more difficult to modify correctly. In particular, instances of `WebPage` are now responsible for updating the values of other objects to maintain structural invariants, which arguably breaks the abstraction principle.

One may opt to redesign `remove` and `addLink` as methods of `WebGraph` to better encapsulate them. Interestingly, doing so evidences the fact that these two operations notionally act on the whole graph rather than on a single vertex – unlike, e.g., modifying the topic of a web page. It also simplifies interactions with the data structure and leaves its interface easier to discover. In Listing 2, it seems arbitrary and needlessly asymmetric that one must call a method of `WebGraph` to insert a page and then call a method of `WebPage` to remove one. Pushing this observation further, one may wonder whether the relation between webpages ought to be stored in a `WebGraph` rather than spread in instances of `WebPage`, as doing so would better encapsulate the maintenance of the graph's structural invariants. If one is willing to adopt this change, the constraints of full ownership become sensible as they acknowledge the whole/part relationship relating a web graph to the web pages that it uniquely *owns*.

Recognizing whole/part relationships is an important step in the design of a data structure as it tells us what constitutes the *notional value* of an instance and thus its meaning. In turn, this concept lets us properly characterize essential operations such as equality, copying, or hashing, and distance ourselves from the cumbersome and error-prone distinction between

referential and structural – a.k.a shallow and deep – properties. Two web graphs are equal if and only if they contain equal web pages having the same relationnships. Likewise, copying a web graph means copying its pages and the relation between those.

Notional values cannot be identified automatically in general because they depend on the abstract meaning one ascribes to a particular set of bits. In other words, only the author of a type can determine the notional values of its instances. Nonetheless, choosing representations that expose whole/part relationships helps since clearly the parts of a whole belong to the latter's value. In most systems supporting value types, including Swift, this information can be leveraged to synthesize structural operations like equlity and copying.

2.2 Peer Relationships

The widespread use of first-class references muddies the difference between whole/part and peer relationships, captured in UML as the distinction between composition and association [4], respectively. Moreover, aliasing invites a litany of issues due to unintended mutation through shared mutable state [8]. These downsides highlights the benefits of full ownership but one question remains: how is one supposed to represent peer relationships?

A possible solution is shown in Listing 3. The keyword `struct` introduces a value type in Swift and the keyword `mutating` marks methods of value types that can modify their receiver. This program satisfies the constraints of full ownership: web pages do not reference each others and they can be uniquely owned by a web graph. It goes even one step further and dispenses with first-class references altogether, using only value types. The peer relationships are expressed with two tables, keyed by the URLs of the web pages involved. These URLs act as unique identifiers, thereby filling one of the two roles first-class references have. That is sufficient to represent relationships: `links` and `backlinks` do not require knowledge about the contents of a web page; only about their existence.

The second role of first-class references, namely the ability to confer access to the referred value, is fulfilled by using an URL as a key to index the `pages` of a graph. For example, the expression `g.pages["https://x.abc"]` denotes the value of a web page assumed to be contained in `g`. One may object that this approach could suffer from the misuse of a URL, e.g., by calling `addLink` on a page contained in a different graph. Note however that such a danger was already present in Listing 2.

Requiring that accesses to the web pages go through the graph has two advantages. It allows structural invariants to be upheld directly in the graph's methods, thus improving encapsulation, and it also ensures that no operation can modify the graph's value behind one's back. This guarantee can be used to fend againt data races in a concurrent setting and to reason locally about the implementation of a data structure in general. In Listing 3, for example, we know in `remove` that if `links` is assigned at `p`, then so is `backlinks`.[1]

One may wonder whether working with this representation is at all practical. We can look at classical graph algorithms to answer affirmatively. A graph is almost ubiquitously defined formally as a pair $\langle V, E \rangle$ where V is a set of vertices and $E \subseteq V \times V$ is a relation. Algorithms on graphs are therefore typically described according to this particular presentation, which happens to match the implementation shown in Listing 3 much more closely than the one shown in Listing 2. We have implemented two classical graph algorithms to confirm this observation. Our code, which is provided as a companion material, is almost a word-for-word

[1] That is assuming the fields of `WebGraph` are not modified directly, which can be enforced with traditional access control mechanisms [5, Chapter 6].

Listing 3 Representation of a web graph with value types in Swift.

```
struct WebPage { var topic: String? }
struct WebGraph {
  var pages: [String: WebPage] = [:]
  var links: [String: Set<String>] = [:]
  var backlinks: [String: Set<String>] = [:]
  mutating func insert(url: String) -> String {
    pages[url].setIfNil(WebPage()).url
  }
  mutating func addLink(from p: String, to q: String) {
    assert(pages[p] != nil && pages[q] != nil)
    links[p, default: []].insert(q)
    backlinks[q, default: []].insert(p)
  }
  mutating func remove(_ p: String) {
    for q in links[p, default: []] { backlinks[q]!.remove(p) }
    pages[p] = nil; links[p] = nil; backlinks[p] = nil
  }
}
```

translation of textbook descriptions, suggesting that the absence of first-class references does not raise implementation challenges. We also come back to the performance of the value-based approach in Section 4.

3 Doubly-Linked Lists

While the previous section has demonstrated that the constraints of full ownership do not preclude the ability to express arbitrary graphs, there are other reasons why one may use references to implement a data structure. For example, though the notional value of a doubly-linked list is isomorphic to that of an array, the former exhibits certain characteristics that the latter does not, and vice versa. In particular, a doubly-linked list typically offers constant time insertion and removal at any position, bidirectional traversal, and stable element positions – i.e., removing or inserting an element does not invalidate references to other elements. It is reasonable to expect that a useful implementation of a doubly-linked list, whether it uses references or not, should satisfy these properties.

One way to understand references is to relate them to pointers. Holding a reference to a value of type `T` in Java is similar to holding a pointer of type `T*` in C. Just like references, pointers have two roles: they identify specific addresses and they provide access to the values stored at these addresses, through dereferencing. Since an address is essentially an offset in some large table[2], a reasonable way to emulate pointers is to use indices into an array. From there, one can implement a doubly-linked list by replacing pointers with indices.

Listing 4 illustrates. It presents selected parts of the definition of a doubly-linked list whose nodes are allocated in a dynamically sized array. Just like in a classical implementation, a node is a triple storing a value and two references, one to its predecessor and one to its successor. The twist here is that these references are represented as offsets into the internal array rather than actual references, thereby satisfying unique ownership requirements. These

[2] We ignore issues of pointer provenance [6] as they are irrelevant to our discussion.

Listing 4 Implementation of a doubly-linked list with emulated pointers in Swift.

```
struct List<T> {
  private struct Node { var pred: Int, var succ: Int, var elem: T }
  struct Address { var rawValue: Int }

  private var contents: [Node] = []
  private var head: Int = -1
  private var tail: Int = -1

  mutating func insert(_ e: T, at a: Address) -> Address {
    if a == tail { return append(e) } else {
      var s = contents.count
      swap(&s, &contents[a.rawValue].successor)
      contents.append(
        Node(value: e, predecessor: a.rawValue, successor: s))
      contents[contents[s].successor].predecessor = s
      return Address(rawValue: s)
    }
  }
}
```

shenanigans are fully encapsulated in the type abstraction, which presents an interface similar to that of a data structure that would use first-class references internally. This design satisfies the properties we have enumerated above with the exception of insertion complexity, which is in *amortized* constant time rather than $\mathcal{O}(1)$, due to the cost of reallocating the internal array. The addresses returned by `insert` are indeed stable – i.e., they are not invalidated by removes or other insertions, even if the internal array must be reallocated.

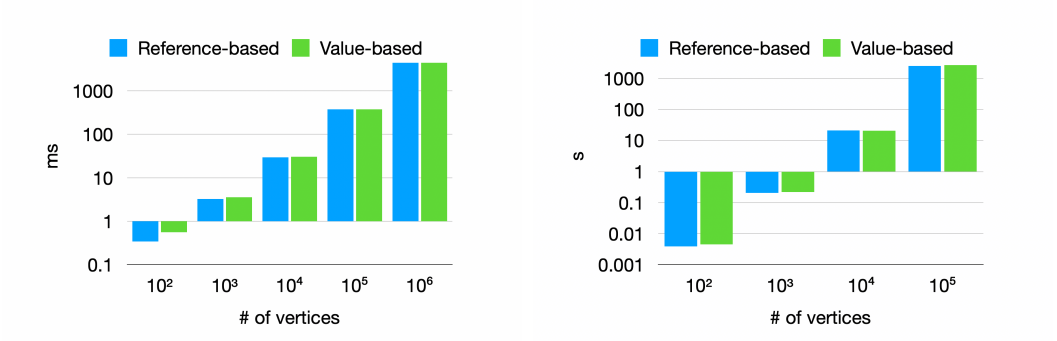
This implementation is in some ways reminiscent of region-based memory management [13]. The internal array of `List` can be understood as a separate [10] region in which new nodes are allocated by "bumping" an offset. Copying a list copies its entire region at once.[3] Likewise, destroying a list deallocates its entire region. This observation reveals that the list itself is essentially an allocator for its own nodes. A direct consequence is that one can apply well-known approaches to more cleverly (re-)use the internal array and avoid leavage after the removal of an element. For example, one could keep a free list identifying the positions of the nodes that have been removed.

Finally, remark that this implementation answers the title of the paper. *Who owns the contents of a doubly-linked list?* The list itself of course! Crucially, a node owns neither its successor nor its predecessor. Instead, the nodes forming a list are all parts of that whole while links between these nodes are peer relationships that can be represented independently, just like in the case of the web graph.

4 Performance

This section examines the performance of the schemes we have described in Sections 2 and 3, comparing them with alternative implementations based on first-class references. We present two series of benchmarks. The first reports the running time of two classical graph algorithms on inputs of increasing sizes. The second studies the efficiency of basis operations of a

[3] At least notionally. Swift actually uses copy-on-write mechanisms to mitigate the costs of copying when no mutation occurs.



(a) Tarjan's strongly connected components. **(b)** Dijsktra's shortest path.

Figure 1 Running times of two classical algorithms ran on two different representations.

doubly-linked list, namely insertion, traversal, and removal. Measurements were done on a MacBook Pro equipped with an Apple M1 chip and 16GB of memory, running macOS 15.3.1. Results are computed as an average of 10 iterations for each input size and using the same seed to generate random data.

4.1 Graph Algorithms

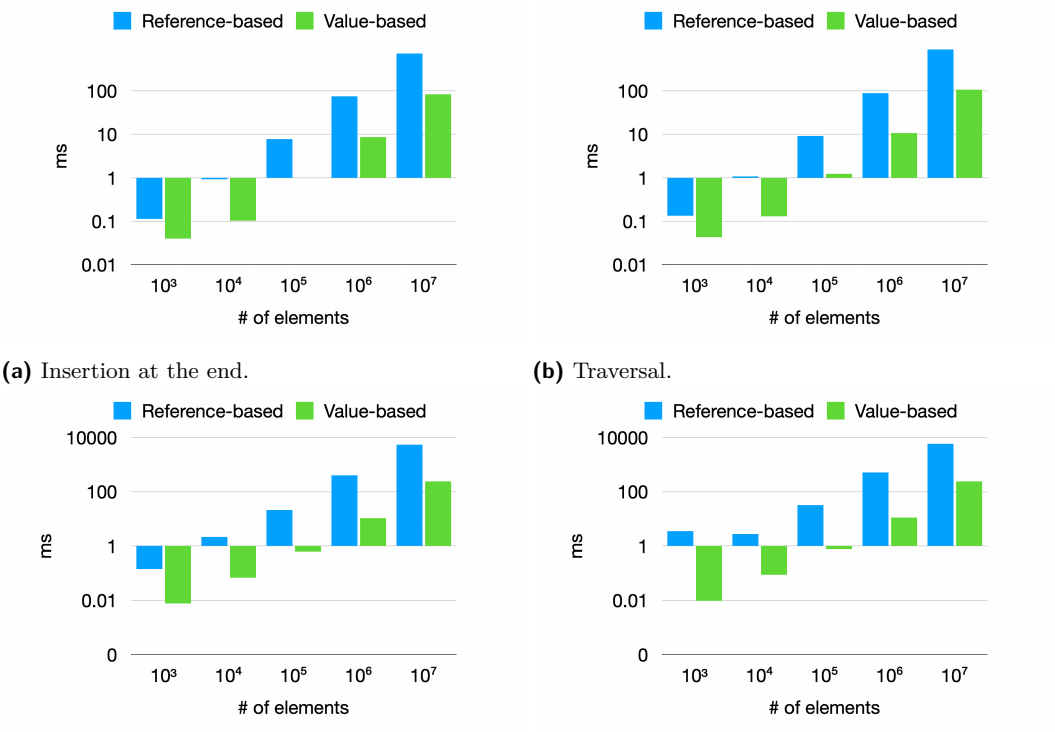
We have implemented Tarjan's strongly connected components algorithm [12] and Dijkstra's shortest path algorithm [3, Chapter 24] on the two representations of a web graph discussed in Section 2. Figure 1a reports the time taken to identify all strongly connected components in randomly generated graphs of various sizes. Figure 1b reports the time taken to compute the shortests path to all the pages available from a randomly selected origin. In both figures, running times for the reference-based and value-based implementations are shown in blue (left) and green (right), respectively. The y-axis uses a logarithmic scale.

The results show no noticeable difference between the two graph implementations, suggesting that representing peer relationship separately, without first-class references, does not negatively impact performance. Interestingly, remark that the costs to access hashtable contents do not introduce a prohibitive overhead compared to dereferencing, despite our use of strings as keys. A more careful choice of identities – e.g., using integers to avoid the cost of hashing character strings – would likely improve on performance at the cost of some convenience and ease of implementation.

4.2 Basis Operations on Doubly-Linked Lists

We have examined two implementations of doubly-linked lists. The first uses unrestricted first-class references to store forward and backward links in each node, as in traditional approaches. The second represents forward and backward links as offsets into an array, as discussed in Section 3.

Four operations were considered. Figure 2a reports the time taken to append new elements at the end of a list. Figure 2b reports the time taken to traverse a list. Figure 2c reports the time taken to remove elements at random positions until the list is empty. Figure 2d reports the time taken to traverse a list after having removed half of its contents at random. In all four cases the the lists were filled with random integers. Running times for the reference-based and value-based implementations are shown in blue (left) and green (right), respectively. The y-axis uses a logarithmic scale.



(a) Insertion at the end.

(b) Traversal.

(c) Removal at random positions.

(d) Traversal after removals at random positions.

Figure 2 Performance of basis operations on different representations of doubly-linked lists.

The results show that the value-based approach significantly outperforms its reference-based counterpart, by one order of magnitude. The biggest contributor of this difference is cache locality. In the value-based representation, the contents of the list are stored in contiguous memory locations. As a consequence, the data related to the successor of a node n is likely available in the CPU's cache after n has been accessed, especially if all elements have been appended to the end of the list, as it is the case in Figure 2b. This advantage persists even in Figure 2d, after the removals of elements create "holes" in the array.

Another contributor to the overhead is the atomic exclusivity check Swift's runtime performs before mutating an object, which protects the program from data races in concurrent settings. The cost of this check appears clearly in Figure 2c. It comes in addition to the overhead caused by the indirections necessary to access the predecessor and successor of the removed node, whose links must be updated after a removal.

Finally, Figure 2a shows that allocating individual pieces of memory for each node is considerably slower than appending new nodes at the end of a resizeable array. On large inputs, this operation effectively has a constant time complexity and, unlike individual allocations, does not involve system calls unless the array has to be resized. Note however that Swift's runtime is not particularly optimized for allocating large numbers of small objects. Hence, systems placing heavier emphasis on this aspect will likely yield better results.

Overall, these benchmarks reveal that notionally self-referential data structure can be implemented quite efficiently within the constraints of full ownership.

5 Conclusion

We have discussed approaches to represent notionally self-referential data structures in systems enforcing full owenrship, a discpline in which objects can have at most one incoming reference. We have presented a general approach that consists of identifying whole/part relationships, which naturally fit the constraints of full ownership, and representing peer relationships by substituting references for values that uniquely identify objects without conferring access to them. We have applied this approach to implement directed graphs and a doubly-linked list in Swift without using first-class references and without escaping the safe subset of the language. Finally, we have examined the performance of our encodings and concluded that dispensing with first-class references can lead to more efficient implementations.

None of the techniques that we have shocased are novel. In fact, they are well-established in value-oriented languages such as Swift, Rust, or C/C++. Surprisingly, however, they are far less known in reference-based languages, despite their interest in terms of reasoning and performance. Hence, perhaps research efforts surrounding ownership should advertise the benefits of their discipline beyond memory safety.

References

1 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer, 2013. doi:10.1007/978-3-642-36946-9_3.

2 Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003. doi:10.1007/978-3-540-45070-2_9.

3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

4 Maria C. Gomez-Fuentes, Jorge Cervantes-Ojeda, and Abel Garcia-Najera. Association and aggregation class relationships: is there a difference in terms of implementation? In *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 44–53, 2021. doi:10.1109/CONISOFT52520.2021.00018.

5 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java language specification: Java se 24 edition, 2015. Accessed: 16 April 2025. URL: https://docs.oracle.com/javase/specs/jls/se24/jls24.pdf.

6 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019. doi:10.1145/3290380.

7 James Noble, Julian Mackay, and Tobias Wrigstad. Rusty links in local chains. In Henrique Rebêlo, editor, *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2022, Berlin, Germany, 7 June 2022*, pages 1–3. ACM, 2022. doi:10.1145/3611096.3611097.

8 James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998. doi:10.1007/BFB0054091.

9 Dimi Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Implementation strategies for mutable value semantics. *J. Object Technol.*, 21(2):2:1–11, 2022. doi:10.5381/JOT.2022.21.2.A2.

10 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

11 Bjarne Stroustrup. What is "Object-Oriented Programming?". In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *ECOOP'87 European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, Proceedings*, volume 276 of *Lecture Notes in Computer Science*, pages 51–70. Springer, 1987. doi:10.1007/3-540-47891-4_6.

12 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

13 Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *High. Order Symb. Comput.*, 17(3):245–265, 2004. doi:10.1023/B:LISP.0000029446.78563.A4.

14 Rasmus Viitanen. Evaluating memory models for graph-like data structures in the Rust programming language: Performance and usability. Master's thesis, Linköping University, 2020. Accessed: 30 May 2025. URL: https://www.diva-portal.org/smash/get/diva2:1463648/FULLTEXT01.pdf.