Adding Support for Reference Counting in the D Programming Language

Răzvan Niţu, Eduard Stăniloiu, Răzvan Deaconescu and Răzvan Rughiniş
Faculty of Automatic Control and Computer Science,University Politehnica, Bucharest, Romania

Keywords: Dlang, Reference Counting, Memory Safety, Functional Style, Language Design, Purity, Garbage Collection.

Abstract: As more and more software products are developed daily, the security risks imposed by the growing code bases increase. To help mitigate the risk, memory safe systems programming languages, such as D and Rust, are increasingly adopted by developers. The D programming language uses, by default, a garbage collector for memory management. If the performance of a program is bottle-necked by it, or a system is resource constrained, as is the case for the ever-growing Internet of Things devices, the user has the option opt out and employ a custom allocation strategy. However, in this situation, the programmer needs to manually manage memory - a complex and error-prone task. An alternative is represented by a middle ground solution in the form of automatic reference counting. This strategy offers simplicity and performance for a small cost in expressiveness. However, due to the transitive nature of type qualifiers in D and purity-based optimizations, it is impossible to implement a library solution. In this paper, we present the problems introduced by transitive type qualifiers to reference counting and we propose the addition of a new storage class for members of aggregate declarations that breaks the transitivity of type qualifiers. We present our design and show that it can be used to implement a generic automatic reference counting mechanism without disabling purity based optimizations.

1 INTRODUCTION

As more and more software products are developed daily, the security risks imposed by the growing code bases increase. In 2019, Microsoft reported that the cause for 70% of security bugs were memory related(Cimpanu, 2019). The costs incurred due to security flaws and their exploitation are in the billions (Bannister, 2021), with the IBM System Science Institute stating that patching costs 100 times more than the development costs (Dawson et al., 2010).

To help mitigate the risk, memory safe systems programming languages, such as D and Rust, are increasingly adopted by developers. A significant area where memory safe languages are desirable is represented by the Internet of Thnigs (IoT). IoT devices have become a popular target for attackers to compromise and use as an IoT Botnet army (Dange and Chatterjee, 2020) that is used to carry attacks against businesses, governments and even entire countries (McMillen, 2021) (Whittaker, 2016). In order to satisfy the needs of IoT devices, the programming languages used must also produce fast programs and be resource considerate.

D is a modern, systems-level programming language that aims to provide both high performance and memory safety in a simple, intuitive and expressive manner. Although it is an imperative language, D provides functional style concepts such as pure functions and transitive type qualifiers (Ullrich and de Moura, 2019). In addition, it is able to inter-operate with C and C++ code out of the box, thus providing a simple migration path for legacy code.

D provides a garbage collector (GC) (Lee, 1991) for built-in features that use heap memory, such as dynamic arrays and classes, but also supports manual memory management via raw pointers and *malloc/free*. Therefore, for situations where the garbage collector is unsuitable due to resource scarcity (small memory, small number of computation units or both) or real-time constraints, users have the possibility to implement a custom allocation strategy. Note that in this scenario ease of use is sacrificed for performance, since the user needs to manually manage memory. This has proven to be a complex, time consuming and error prone endeavour (Knuth, 1997).

A third option is represented by automatic reference counting (ARC) in the form of a library solu-

Niţu, R., Staniloiu, E., Deaconescu, R. and Rughiniş, R.
Adding Support for Reference Counting in the D Programming Language.
DOI: 10.5220/0011290000003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 299-306
ISBN: 978-989-758-588-3; ISSN: 2184-2833

```
struct RC(T)
{
    T* data;
    size_t count = 0;

    // constructors
    this(T data);,
    this(T data) immutable;

    // methods of RC
}

void main()
{
    // a.count is mutable
    auto a = RC!(int)(2);
    // b.count is immutable
    auto b = immutable RC!(int)(2);
}
```

Listing 1: Typical reference counting implementation where a pointer to allocated data is stored alongside the its reference count.

tion (McBeth, 1963). ARC is lightweight in terms of resource utilization, since it typically stores an extra counter field for each allocated instance. In terms of computation, the added overhead consists of simple addition or subtraction operations. In addition, the usage of ARC is almost transparent to the user: an object needs to be declared as being reference counted and everything will be taken care of behind the scenes. Providing support for such an option is important because it offers maximum flexibility in terms of allocation strategies: for the majority of cases, the GC should be sufficient; for constrained scenarios where the GC cannot be supported, ARC is to be used; for extreme situations, where not even ARC is sufficient to attain the performance guarantees, manual memory management should be employed.

Since D also supports functional style type qualifiers, such as *immutable* and *const*, implementing a general solution for ARC is not possible. Listing 1 provides a typical library implementation of a reference counted object using the D language. The *RC* struct is templetized with the type *T* that is reference counted. It stores a pointer to a heap allocated instance of type *T*, next to the reference count. Normally, *RC* also defines functions to handle the reference and to forward the *T* specific operations to the underlying type - for brevity, we have not included these functions. When instantiating a mutable version of *RC* we can successfully track the reference. However, once we construct an *immutable* instance, due to the functional transitivity of type qualifiers, both the *data* and *count* fields will be *immutable*. This makes it impossible to ever update the reference count, and thus to implement generic ARC.

In this paper, we propose the addition of a new storage class for variable declarations, *__metadata*, that may be used to tag aggregate declaration member fields. We describe the primary language changes and design decisions that were taken to integrate *__metadata* into the language. Further, we analyze the implications of our decisions and show that with our design it is possible to implement ARC in the functional contexts of the D programming language.

The remainder of this paper is organized as follows: Section 2 presents the background and motivation for this work, Section 3 presents the design of *__metadata* and Section 4 presents our evaluation. We conclude with Section 5.

2 BACKGROUND AND MOTIVATION

In this section we will discuss the benefits and downsides of garbage collection and automatic reference counting. We then provide insight on how type qualifiers, purity and memory safety are implemented in D. Throughout this section we also motivate the need for `__metadata`.

2.1 GC vs. ARC

Garbage collection is a well studied concept (Wilson, 1992) (Jones et al., 2016) (Bacon et al., 2004). Typically, garbage collection is implemented as a run-time library. Whenever an object is created, the *new* expression is rewritten to a call to the allocation method of the garbage collector. As such all of the objects are registered on the GC heap. In various moments the GC performs an analysis to decide which objects are still living and which may be freed.

Multiple types of allocation and reachability strategies have been developed: mark-sweep (McCarthy, 1960) (Hayes, 1991), mark-compact (Clark and Green, 1977) (Clark, 1979) (Cohen and Nicolau, 1983), copying collection (Cheney, 1970) (Fenichel and Yochelson, 1969) (Larson, 1977) and non-copying implicit (Baker, 1992). Although a large range of techniques and implementations exist, most require large amounts of memory to store information about the memory blocks and are non-deterministic with regards to when the memory is freed (Wentworth, 1990).

Automatic reference counting, on the other hand is very light-weight in terms of resource utilization because it simply requires an extra field to store the number of references to a specific object (Deutsch and Bobrow, 1976) (Goldberg, 1991). However, it suffers from an effectiveness problem: no circular references are supported (Bobrow, 1980). The reference count is typically modified whenever an object is copy constructed, assigned or destroyed. Therefore, a library solution needs to overload these operators to make sure that the object is properly reference counted. Reference counting is transparent for the user of the data structure.

Garbage collection has been shown to have the same performance as manual memory management and reference counting in the majority of cases. However, it comes with a memory consumption cost and may do a collection cycle and hence stop-the-world in critical moments (Romanazzi, 2018). To alleviate this problem, ARC may be used as a lighter alternative at the cost of not supporting circular references.

D implements a stop-the-world mark and sweep garbage collector. The fact that the GC may, at some point, stop the program execution to do a collection cycle is a deal breaker for real time applications. Using an ARC based management technique is desirable in such situations, however, it is not possible to implement a library solution due to D's transitive type qualifiers.

2.2 Type Qualifiers

D implements 4 type qualifiers: *const*, *immutable*, *shared* and *inout*. They all apply transitively, meaning that both the reference and all the fields accessible (directly or indirectly) through it are considered qualified.

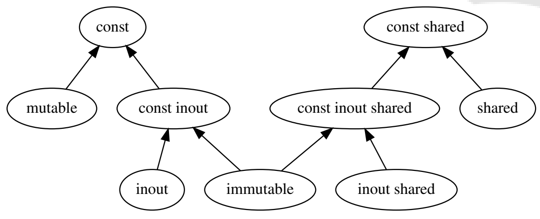

Figure 1: Qualifier conversions in D (dlang.org, 2012).

const objects may not be modified through the current reference. Figure 1 depicts the implicit conversion between qualifiers. It can be observed that mutable is implicitly convertible to *const*. This implies that a mutable reference may modify an object that is also reachable through a *const* reference.

immutable objects can never be modified, once constructed. As a consequence, such instances may be placed in read only memory. No other qualifier may be implicitly converted to *immutable*.

The *inout* keyword forms a wildcard that can be replaced with any of *mutable*, *const* and *immutable*. Functions that differ only in how their parameters are qualified can be merged into a single function with *inout* parameters. The compiler will then substitute it with the appropriate qualifiers, depending on the mutability of the arguments.

shared is used for variables that are needed in multithreaded environments. As opposed to C, D implicitly considers global variables as being thread-local. To create a variable shared among threads, the *shared* type qualifier is used in the variable declaration. In addition, the compiler will error out on any *shared* object that is being accessed without an appropriate synchronisation mechanism. *immutable* objects are implicitly *shared* and do not require any synchronisation.

When implementing reference counting, the transitivity of type qualifiers makes it impossible to update the reference count, as depicted in Listing 1.

2.3 Purity

In functional languages, a function is considered pure if it does not have any side effects. As a consequence, a pure function will yield the same result for the same set of arguments. Purity enables various compiler optimizations such as common sub-expression elimination, elision of function calls and statement reordering (Barnett et al., 2004).

D provides the ability to write functionally pure code by using the *pure* keyword. Once a function is annotated with *pure*, the compiler will check that it does not access any global mutable data. Such functions are not allowed to call other functions that are not marked as pure.

Although pure functions may not access any global mutable data, they can still alter the global state through parameters that contain indirections. Since requiring pure functions to not modify parameters would have been very restrictive, D implements two types of purity: weak and strong. A weakly pure function is capable of modifying global state through its parameters, whereas a strongly pure function is not. A strongly pure function may call a weakly pure function. From a user's perspective, there is no difference between the two: both are annotated with the same keyword. However, from the compiler's perspective, strongly pure functions are subject to purity optimizations whereas weakly pure functions are not. The compiler is able to distinguish between the two types of pure functions just by analysing the signature. If the types of the parameters of a pure function do not contain any mutable indirections, then the function is strongly pure.

As mentioned in the previous section, when performing reference counting for a non-mutable object a mechanism to modify the counter is needed. *__metadata* is used exactly for that, however, this might affect the purity level of certain functions. Consider the following function signature: `int func(ref immutable S x) pure`. Prior to the addition of *__metadata*, `func` would be considered strongly pure, because there is no possibility of modifying any global data. However, with the addition of it, `S` might contain a *__metadata* field. As a consequence, the compiler has no mean of deciding whether `func` is strongly or weakly pure without inspecting the body of the function and observing that no *__metadata* field is modified. The safer alternative in this situation is to consider `func` weakly pure, however, that would exempt it from purity based optimizations.

In this paper, we provide a solution that allows the use of *__metadata* without losing the possibility of applying purity based optimizations.

2.4 Memory Safety

D offers the possibility to mechanically check for memory safety issues at compile time. Functions that are annotated with the @*safe* keyword will be analyzed and if any unsafe operations - pointer arithmetic or taking the address of a local variable - are performed, it will issue a compile time error. Functions are considered by default to be unsafe.

There are situations where unsafe operations do not pose any threats. For such scenarios, D provides the @*trusted* keyword. @*trusted* is used to annotate functions that have been manually analyzed by the user and have been proven to be safe despite doing unsafe operations. This design makes it easier to scan for issues when doing code review, since only @*trusted* functions need to be analyzed.

Modifying a *__metadata* field cannot be @*safe* by default because, in essence, a non-mutable object is being modified. As a consequence, the user must be cautious when using *__metadata* and annotate such functions with @*trusted* so that the reference counted object is usable in @*safe* code. In short, it is the burden of the library writer to ensure that the data structure is correctly reference counted.

3 DESIGN

__metadata leverages the existing concept of *mutable* in C++ (cppreference, 2012). However, it extends it so that it supports transitive type qualifiers such as *immutable* and *const*. In addition, we define the semantics in the presence of purity based optimizations.

We have chosen the name*__metadata* to emphasize the fact that such fields are not conceptually part of the object, rather they store data that is needed to manage it. Therefore, such fields should not be subject to the same constraints as the rest of the instance. We wanted to avoid using a generic name, such as *mutable*, because we want to discourage the use of this feature outside of the realm of reference counting.

3.1 Semantics

The *__metadata* storage class modifies propagation of type qualifiers on fields. *immutable* and *const* will be transitive with the exception of *__metadata* fields. Mutating such fields has defined behavior, however, they must be *private* - only the object that defines the *__metadata* field is allowed to modify it.

Listing 2 presents a partial implementation of reference counting of an *const* object using *__metadata*. The assignment operator is omitted because you cannot technically assign a *const* field. Also, any method that does not involve the reference count such as operator overloads and underlying forwarding functions have been omitted. The referenced counted struct is templetized by a generic type *T*. The constructor allocates memory and initializes the reference count. The copy constructor initializes a new copy and increments the reference count. The *incRef* and *decRef* methods modify the *__metadata* field that would otherwise have been *immutable*. The destructor decrements the reference and frees the object if the count reaches 0.

The following operations are not allowed:

- to define a *const* or *immutable* *__metadata* field. The sole purpose of __metadata is to allow non-mutable fields to be altered.
- to have a non-private *__metadata* field. The object that encapsulates metadata should be the one that manages it. Any external access is forbidden. Ideally, such a field should be modified only indirectly by calling the public copy constructor, assignment operator or destructor, which in turn would call the private methods that update the reference count.
- to define a *__metadata* field that is not a member of an aggregate declaration. There is no point in having globals/locals be used in conjunction with *__metadata* since there is no object that could impose qualifiers transitively.
- to use *__metadata* in @*safe* code. This feature is designed as an enabler for library owners to sup-

port @*safe* and *immutable* reference counted objects. It is by no means designed for general use.

3.2 Effect on Type Qualifiers

Because *immutable* is implicitly *shared*, *__metadata* fields of *immutable* instances are regarded as shared in order to be thread-safe. Therefore, any uses a *__metadata* fields that come from *immutable* objects should be properly synchronized if used in a multi-threaded environment.

const references to objects may come from mutable, *const* and *immutable* objects, therefore the type system cannot infer whether the *__metadata* field inside the object should be *shared* or not. As a consequence, *__metadata* fields of *const* objects will be

```
const struct RefCount(T)
{
    T* payload;
    private __metadata size_t* count;

    // constructor
    this(int size)
    {
        payload = malloc(size * T.sizeof);
        count = malloc(int.sizeof);
        this.count = 1;
    }

    // copy constructor
    this(const ref RefCount src)
    {
        this.payload = src.payload;
        this.count = src.count;
        incRef();
    }

    // increment reference
    private void incRef()
    {
        *count++;
    }

    // decrement reference
    private void decRef()
    {
        *count--;
    }

    // destructor
    ~this()
    {
        decRef()
        if(*count == 0)
            free(p);
    }
}
```

Listing 2: *__metadata* semantics.

```
shared int* q;
int* r;

struct RefCounted
{
    private __metadata int* p;
    bool isImmutable;

    this(shared int *p) immutable
    {
        this.p = p;
        isImmutable = true;
    }

    void incRef() const
    {
        // typeof(p) => int*
        if (isImmutable)
            q = cast(shared int*) p;
        else
            r = p;
    }
}

void main()
{
    shared int* p;
    immutable A ia = immutable A(p);
    const A ca;
    ia.fun();
    ca.fun();
}
```

Listing 3: *__metadata* fields from *const* objects are treated as unshared mutable. The user needs to handle this case appropriately.

type checked as being mutable, leaving upon the user to reason upon the code to make the appropriate casts. Listing 3 provides an example how this situation should be handled. Whenever an object is created, the underlying type qualifier should be saved -in this situation, the *isImmutable* variable is used. Depending on the value of this variable, the user may cast it to the appropriate type qualifier.

inout references to objects may come from mutable, *const* and *immutable* objects. This similarity to *const* objects makes *inout* object instances with *__metadata* fields to be treated exactly the same as *const* ones.

3.3 Interaction with *pure*

The addition of *__metadata* should not alter the properties of strongly pure functions. To address this issue we propose a set of rules and transformations that preserve the same semantics of strongly pure functions.

Next, we list the properties of strongly pure functions and discuss whether each is affected and what

are the design decisions that preserve the semantics.
A strongly pure function that returns a type without indirections will return the same result when applied to the same *immutable* arguments, regardless of how many intermediate actions happen. Listing 4 presents a code sample that illustrates this property: *foo* may not access any global data and *arg* will never be modified.

```
// foo -> strongly pure
auto a = foo(arg);
auto b = foo(arg);
```

.

Listing 4: Srongly pure function result memoization. The result of the first call to *foo* may be memoized and reused at the second call.

Therefore, a potential optimization is to cache the result of the first call and reuse it for subsequent calls. Essentially, the declaration of *b* is rewritten to *auto b = a*, where *a* is a cached version of the initial result. With the addition of *__metadata*, this property no longer holds, because *arg* could contain a *__metadata* field. If any actions are performed between the two declarations that modify the *__metadata* field of *arg*, then the optimization should not be applied, otherwise the reference count will be invalidated. To address this issue, we never cache the value of *arg* and when we rewrite to *auto b = a*, we consider the latest version of *a*. This rewrite is correct since *arg* cannot be modified, except for the *__metadata*. For *foo* the existence of any *__metadata* field is transparent and therefore should not affect its execution.
The set of references returned from strongly pure functions can be safely converted to immutable or shared. A reference can be returned from a strongly pure function if it was created during its execution or if it was passed from one of its parameters. Since a strongly pure function may not receive parameters that have non-*immutable* indirections, it means that the returned reference is either *immutable*, if created from one of the parameters, or mutable but it was constructed inside the strongly pure function. Since the value could not escape the scope of the strongly pure function, it means that the reference returned is unique and therefore may be converted to *immutable*. This aspect does not suffer any modifications with the addition of *__metadata*.
A strongly pure function whose result is not used may be safely elided. Calling a function without side effects and ignoring its result has the same consequences as if the function would not be called at all. With the addition of *__metadata*, it is possible to have side-effects when calling strongly pure functions.

Consider Listing 5. *foo* is a strongly pure function that returns *void*. There is no point in executing it since it should not produce any effects. However, *S* has a *__metadata* field which is modified. In this context, we cannot rely on the value of the *__metadata field*. Unfortunately, there is no mechanism to prevent this situation, however, we note that *__metadata* is designed with the sole purpose to allow for the implementation of non-mutable automatic reference counted objects. As such, there is no reason for an external function to manually update or read the reference count. The feature is designed to be used by library writers to provide a mechanism to manage memory for their objects. Users of said objects should not have the ability to modify the internal state that tracks the allocation. Therefore, in this situation, we rely on users to design their data structures such that the methods that modify the reference count are only usable from within the object.

```
struct S
{
    private int __metadata x;

    void incX () immutable
    {
        ++x;
    }

    int getX() immutable
    {
        return x;
    }
}

void foo(immutable ref S s) pure
{
    s.incX();
}

void main()
{
    immutable S s;
    // optimized away
    foo(s);
    // cannot rely on this
    assert(s.getX() == 1);
}
```

Listing 5: Wrongfully optimizing away a pure function.

For example, defining a reference counted object as in Listing 2 makes it impossible to manually update a *__metadata* field. In that scenario, the reference count may be modified only indirectly by assigning, copy constructing or destroying a reference counted instance. As a consequence, strongly pure functions that receive such an object may modify the reference count inside their body, but they cannot escape the new created objects. As a result, once the function execution is over, the reference count will be unmod-

ified, preserving the *immutability* of the object. In these conditions, the elision of the function may take place.
A strongly pure function invocation can always exchange order with an adjacent function invocation, provided that data transitively reachable from the arguments of the functions do not overlap and the second function does not take as argument the result of the first function. Since a strongly pure function does not have any side effect, it can be safely swapped with a different strongly pure function. Actually, in such a scenario, the two could be run in parallel. In the case of a strongly pure function that is adjacent to an impure function, we know that the impure function cannot modify the argument to the strongly pure one.

4 EVALUATION

To evaluate our approach, we have implemented *__metadata* in a fork of the official D frontend. Our objective was to validate the correctness of the design and implementation and to quantify the gains of using reference counted objects instead of the garbage collector.

To validate the correctness of our method we have updated the D standard library implementation of the *RefCounted* object to use *__metadata*. Prior to our work, this library solution worked correctly only for mutable objects and could not be used in generic code that is heavily based on templates and leverages the benefits of type qualifiers. After the addition of *__metadata* we have created a singly linked list implementation that can accommodate *immutable* and *const RefCounted* objects. We ran these versions using a program that copy constructed instances of the singly-linked list and compared the results with the mutable version. We have tested that the reference count is correctly updated, according to the mutable version, at each step.

Next, we wanted to measure what are the benefits of using the *RefCounted* implementation with regards to performance. To that end, we have implemented two versions of the following data structures: a singly-linked list, a red-black tree, a binary heap and an array. The first version of these data structures uses the garbage collector as the underlying allocator, whereas the second version uses instances of the *RefCounted* implementation. Next, we created a program that inserts 40MB of data in each data structure and measured the overall execution time.

The results can be observed in Figure 2. All four data structures have a significant performance improvement, roughly 2x, when using reference counted objects as an alternative to the GC. This constitutes a hint that the garbage collector implementation in D has room for improvement. Until the improvements are implemented, ARC could be employed in performance-critical applications.

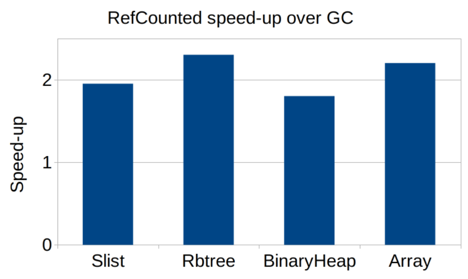

Figure 2: Speed-up when using reference counted objects instead of GC managed instances.

5 CONCLUSIONS

Garbage collection is very attractive due to its transparency to the user. However, in some scenarios the GC might have an impact on performance and the use of resources. Automatic reference counting is, in this situation, a viable alternative, provided that circular references are not required.

To enable the implementation of automatic reference counting as a library solution in the D programming language, we have designed, implemented and tested a new feature, *__metadata*. This storage class attribute is used to break the transitivity of type qualifiers on specific aggregate declaration fields. We have shown that *__metadata* can be integrated in the language without impacting the application of purity based optimizations, thus enabling the implementation of fast, immutable and safe reference counted data structures.

We believe that integrating high-level concepts such as purity into low level languages represents a good opportunity to write safer programs. By proving that it is possible to implement a memory allocation strategy that is fast, safe and pure at the same time, we hope that this work paves the way to adding more functional style concepts into low level languages

REFERENCES

Bacon, D. F., Cheng, P., and Rajan, V. (2004). A unified theory of garbage collection. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 50–68.

Baker, H. G. (1992). The treadmill: Real-time garbage collection without motion sickness. *ACM Sigplan Notices*, 27(3):66–70.

Bannister, A. (2021). Substandard software costs us economy $2tn through security flaws, legacy systems, abandoned projects. *URL: https://portswigger.net/daily-swig/substandard-software-costs-us-economy-2tn-through-security-flaws-legacy-systems-abandoned-projects*.

Barnett, M., Naumann, D. A., Schulte, W., and Sun, Q. (2004). 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*. Citeseer.

Bobrow, D. G. (1980). Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):269–273.

Cheney, C. J. (1970). A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678.

Cimpanu, C. (2019). Microsoft: 70 percent of all security bugs are memory safety issues. *URL: https://www. zdnet. com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues*.

Clark, D. W. (1979). Measurements of dynamic list structure use in lisp. *IEEE Transactions on Software Engineering*, (1):51–59.

Clark, D. W. and Green, C. C. (1977). An empirical study of list structure in lisp. *Communications of the ACM*, 20(2):78–87.

Cohen, J. and Nicolau, A. (1983). Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):532–553.

cppreference (2012). C++ mutable. https://en.cppreference.com/w/cpp/language/cv.

Dange, S. and Chatterjee, M. (2020). Iot botnet: the largest threat to the iot network. In *Data Communication and Networks*, pages 137–157. Springer.

Dawson, M., Burrell, D. N., Rahim, E., and Brewster, S. (2010). Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3(6):49–53.

Deutsch, L. P. and Bobrow, D. G. (1976). An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526.

dlang.org (2012). Implicit qualifier conversions. dlang.org/spec/const3.html#implicit_qualifier_conversions.

Fenichel, R. R. and Yochelson, J. C. (1969). A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612.

Goldberg, B. (1991). Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 165–176.

Hayes, B. (1991). Using key object opportunism to collect old objects. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 33–46.

Jones, R., Hosking, A., and Moss, E. (2016). *The garbage collection handbook: the art of automatic memory management*. CRC Press.

Knuth, D. E. (1997). The art of computer programming. volume 1: Fundamental algorithms. volume 2: Seminumerical algorithms. *Bull. Amer. Math. Soc*.

Larson, R. G. (1977). Minimizing garbage collection as a function of region size. *SIAM Journal on Computing*, 6(4):663–668.

Lee, P. (1991). *Topics in Advanced Language Implementation*. MIT Press.

McBeth, J. H. (1963). Letters to the editor: on the reference counter method. *Communications of the ACM*, 6(9):575.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195.

McMillen, D. (2021). Internet of threats: Iot botnets drive surge in network attacks. *URL: https://securityintelligence.com/posts/internet-of-threats-iot-botnets-network-attacks/*.

Romanazzi, S. (2018). From manual memory management to garbage collection. *ResearchGate [Online]. Available from: https://doi. org/10.13140/RG*, 2(22961.28006).

Ullrich, S. and de Moura, L. (2019). Counting immutable beans: Reference counting optimized for purely functional programming. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, pages 1–12.

Wentworth, E. (1990). Pitfalls of conservative garbage collection. *Software: Practice and Experience*, 20(7):719–727.

Whittaker, Z. (2016). Mirai botnet attackers are trying to knock an entire country offline. *URL: https://www.zdnet.com/article/mirai-botnet-attack-briefly-knocked-an-entire-country-offline/*.

Wilson, P. R. (1992). Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer.