Static Memory Management

Edward Peters

Spring 2016

1 Introduction

Computers have finite memory. Each value stored over the course of a program's lifetime takes up a greater or lesser amount of this memory. If sufficiently many values are written, the total sum of that memory will exceed the space allocated to the program. In order to avoid running out, memory must be **recycled**. Recycling memory refers to the identification of previously-used memory which contains values that will not be needed again, and thus may be made available to other values in the future.

It is not usually practical to know explicitly if a given value *will* be referenced, so instead memory management focused on whether a value *can* be referenced. A value can be referenced if it is in **reachable memory**, i.e., if it is either on the stack, or if it is pointed to by something else in living memory. (It's important to note that the existence of a reference to a memory location is not enough to call that location reachable, as **circular reference** is possible in many languages.) On the other hand, a location is un-reachable if it cannot be found by following any sequence of references from reachable memory.

The intent of this paper is to describe the broad challenges of effective memory managment and approaches to it, with a particular focus on static (rather than dynamic) techniques. Both **stateless** and **imperitive** languages require memory management, and this paper addresses both [10]. The remainder of Section 1 describes the core dangers of failed memory managment, as well as the difference between static and dynamic approaches. Section 2 covers general terminology and concepts across different areas of memory management. Section 3 goes over a small set of languages with interesting memory management options built into the syntax or compilers. Section 4 describes the qualities, techniques and challenges involved in static analysis tools that are not built in to the core language. Section 5 reviews a number of research papers on the topic. Section 6 offers a broad analysis of the problem domain, and the challenges facing software engineering.

1.1 Live Variable Analysis

The remark above is not always true; there are situations where memory analysis looks for what will be acessed, instead of just what can be accessed. In some situations, compilers will attempt to recognize whether a variable may safely be overwritten, i.e., whether it will be written to before its next write. This process is known as **live variable analysis.** This can be very important for speed optimizations, as it can free up registers or other high-speed storage for use by other operations without inserting a costly write to the larger program memory. (In general, there is a trade-off between the access speed of memory, and the amount that can be economically or technologically available. This is known as the **memory hierarchy** [12].) While live variable analysis can give significant improvement gains by making more effecient use of the memory hierarchy, it is unlikely to prevent an actual program crash, as the space for each variable will likely eventually be needed again, when it is next written to.

1.2 Dangers

If recycling is done improperly, two risks emerge. The first is called a **memory leak**. This happens when a program fails to recognize that memory has become un-reachable. In this case, the memory remains allocated after it should be freed. This can happen on a small scale without causing issues, but if it happens enough (in particular, if some repeatedly-run piece of code leaks memory with each execution), then the program may exhaust its memory resources and crash. The second danger is called a **dangling pointer**, which is when a memory location is prematurely freed, so it may still be referred to from an inappropriate context. Dangling pointers may lead to either immediate crashes or undefined, nondeterministic behavior. Almost all memory management algorithms are considered conservative, in that they tolerate memory leaks more readily than dangling pointers [12].

1.3 Static vs. Dynamic Memory Management

The task of effectively recycling program memory can be approached in one of two ways. The first is to make the freeing of memory an explicit part of the execution code, so that hopefully each allocation is paired with an explicit free directive. The second is to have another process periodically search the programs memory for space which may be released to general use.

1.4 Garbage Collection

The dynamic act of searching for un-reachable memory is referred to as **Garbage Collection**. Typically in a garbage-collected language, the program is allowed to "leak" memory relatively freely, saving the programmer and the compiler the difficulty of correctly freeing unreachable memory. When the used memory passes a certain threshold, a separate "Garbage Collection" routine is called. The basic model of garbage collection acts by following references from active memory outwards, "marking" all reachable memory as such, and then freeing everything that is left un-marked [12].

Garbage collection is highly convenient for the programmer, but carries a run-time cost not found in static techniques. This is particularly problematic for power-constrained devices, as garbage collection has to access almost all of the program's memory, which is a power-intensive process [8]. Additionally, even if the amortized cost of garbage collection may be kept low, it cannot generally be done in small parsels. In order to guarantee that a given memory location may be safely freed, it is generally necessary to pause execution and examine the entire space of reachable memory, to guarantee no paths exist. This poses a significant challenge to real-time applications [6].

While many more sophisticated algorithms exist to reduce the problems of garbage collection, they are outside the scope of this paper.

1.5 Static Memory Management

By contrast, static memory managed languages insert statements to free memory into the program code. This results in better run-time performance of the program, as the costly "search" from garbage collection is not performed. However, avoiding memory leaks and dangling pointers through static memory management is a non-trivial task. Languages such as C leave this almost entirely up to the user, which results in frequent bugs and development overhead. For example, if one programmer prematurely frees a value that is written to again in the future of the program, that can result in random changes to information needed by code written by other programmers. Since the strange behavior won't necessarily be local to the dangling pointer, it can be very difficult to find and fix it over a sufficiently large code base.

The ideal solution is a language in which the compiler is entirely responsible for recognizing when memory should be allocated or freed, and places the needed calls into the execution code without the involvement of the programmer. While this goal has not yet been realized, this paper discusses several of the efforts in that direction, both in terms of new language features and analysis tools for existing languages.

2 Concepts and Language

Before launching into programs, tools and academic papers, it is important to cover some common terminology and concepts in memory management.

2.1 Malloc and Free

The most basic form of memory management relies only on explicit, absolute calls to **malloc** and **free**. Malloc takes as argument the size of the needed region, and returns a pointer to that region. Free takes as argument a pointer, and de-allocates the referenced memory. Languages such as C require the user to use these calls explicitly, while other languages may introduce them at compile time [12].

2.2 Retain and Release

One difficulty of the malloc/free model of memory management is that multiple data structures or subroutines may reference the same region of memory, with no strong guarantee on which will live the longest. In this case, none of the functions in question can reliably issue the call to free the shared memory, as one of the other threads may yet reference it.

To handle this situation, many languages support calls to **retain** and **release** to implement **reference counting**. In effect, these are simply malloc and free with an added counter. When memory is first allocated, this counter is set to one (for whatever reference it is initially assigned to). When a new reference is created to the same memory location, it is associated with a call to retain,

which increases the counter by one. As each reference passes out of scope, a call to release should be inserted, which decreases the counter by one. If the counter ever reaches zero, then it is assumed that no references still exist to the memory in question, and it is freed [1].

In this model, memory leaks occur if an allocation or retain is not properly paired with a release, leaving the counter with a positive value after all references to it have died. A dangling pointer occurs in the opposite case, if release is somehow called multiple times for a single initialization or retain, resulting in the counter prematurely reaching zero. Like malloc and free, some languages (such as C++) require the user to explictly write out calls to retain and release, while others (such as Objective C or Swift) introduce them at the compiler level.

Note that when a structured data type's reference count reaches zero, it must call release on any member variables to which it holds a reference. If those memory locations are not referenced directly from somewhere else, they are freed as well, and the effect passes recursively downwards.

3 Language Features

Many languages provide features that implement static memory management, with a greater or lesser degree of work from the programmer. The languages discussed here are only a very small sampling of what exists, selected in order to represent interesting mechanisms and the intersection with other tools (such as null or thread safety.)

3.1 Objective C and ARC

Objective C implements Automatic Reference Counting (ARC), a memory managment scheme based around automatic insertion of retain and release calls, as describe in section 2.3 [1].

One interesting feature of this language is that the ARC too was a late arrival to objective C, and was limited by the need for backward compatibility. This has led to several requirements, such as a consistent notion of when ownership passes between caller and callee. This causes issues with some pure-C libraries, which do not respect the standard. A number of other specific requirements exist for ARC to be able to effectively track references, because the language allows flexibility that sometimes violates type saftey. For instance, pointers may be forceably cast to other types, making it difficult to predict the actual source code that may be run on the other side of a function call. Much of ARC's complexity stems from meeting these challenges.

The core mechanism of Objective C's ARC is the automated insertion of retain/release calls at particular points. Whenever a method takes a parameter or an object has a member variable set, retain is called on the referenced value. Whenever a method returns or a member variable is overwritten, the associated release call is made. Some functions will also use an "autorelease" call; this is effectively a delayed release call, used when a function expects a created

reference to be retained by the caler. If a normal release were used in this situation, the memory would be immediately freed, before the caller could claim ownership.

This system is not complete, however, as it would result in memory leaks when circular references are made (i.e., when an object eventually owns itself through some sequence of member variables.) In order to avoid this, Objective C allows for **strong pointers** and **weak pointers**. Weak pointers do not directly contribute to reference counts, and do not cause calls to retain or release. So long as no cycles exist that consist entirely of strong pointers, memory leaks will not occur. One risk is that this makes it technically possible for a weak pointer to remain in existence after memory has been freed; to handle this, any weak pointers to a freed memory location are set to null. This will still result in a program crash if the pointers are dereferenced, but prevents undefined behavior and can be handled by null-checking. C++ also includes these concepts, but unlike Objective C, does not have automatic inclusion of retain and relase calls.

3.1.1 Swift

The basic ARC system implemented in Objective C re-appears in Swift, in a simplified form. In Swift, weak pointers are impemented through an Optional type, allowing for type-checked null safety. (**Optional** is a concept from type theory that allows values which may be null in the normal execution of code to be treated as such by the type checker. Properly used, this can allow the type checker to provide a measure of safety against null-pointer exceptions.) "Unowned Pointers", however, act as automatically unwrapped weak pointers, so it is still possible to get a null reference. The intention of language is that circular references that are "needed" (i.e., an object is not properly defined without them) should be stored as unowned pointers, while "optional" member variables should use weak pointers. In this dynamic, a still-active object instance with an unowned pointer to a freed object is a logical failure anyway, so an exception is not unreasonable behavior if such is referenced [3].

3.2 Rust

One downside to reference counting is the potential vulnerability in multi-threaded applications. Consider the following interleaving, where A and B are threads and L is a memory location.

- Thread A decrements the reference counter on location L
- Thread B takes a reference to L
- Thread A checks the reference counter on L, and finds it to be zero
- Thread B increments the counter on L
- Thread A frees the memory at L

This ends with B having a dangling pointer to the already-freed L. It is possible to use a reference counter with atomic operations in order to be thread-safe, though this incurs a performance hit [12].

The designers of Rust are very focused on speed, and implemented a memory management model that is almost entirely deterministic [2]. The goal of this model is that free statements (not release statements, but actual frees) may be added at compile time, without conditional wrappers beyond whatever branching already exists in the code. This is done through **unique ownership** and **borrowing**. Consider the following code fragment:

```
let x = vec![1, 2, 3];
let x2 = x;
println!(‘‘x[0] is : {}’’, x[0]);
```

The above code allocates memory for a vector, assigns a second reference to it, and then attempts to pass it to an output function. This code looks innocent enough, but the Rust compiler will not accept it. By Rust ownership semantics, the second line passes ownership of the memory reference from x to x2; it is no longer x's to give away to the print statement. Through these rules, Rust ensures that there is only one binding to a resource at any given time.

3.2.1 Borrowing

Of course, this would be too limiting for actual development, so Rust also includes the concept of borrowing. In Rust, a reference may borrow a resource from the binding that owns it subject to one rule, checked at compile time: the scope of the borrow must not be greater than the scope of the binding. In other words, it must be safe to assume that after the binding ceases to have the reference, no other references to it exist, and it may be safely freed. Borrowing may be done with either a special assignment notation, or as part of the parameter declaration in a function header.

For difficult cases, reference counting is still available to the programmer. This is implemented by making a smart pointer that "owns" the memory, and having multiple other locations borrow from that pointer.

3.2.2 Mutable and Immutable References

Rust also uses this model of ownership and borrowing to allow safe multithreading. In Rust, borrowed references may be either mutable or immutable, and a value may not be modified through an immutable reference. The Rust compiler will allow any number of immutable borrows to the same resource to exist, but if a mutable borrow is made, no other borrows may be made until it passes out of scope. This ensures that you cannot get thread collision: if multiple threads are looking at the same resource, they must all be doing so in a read-only fashion. Most other languages do not have this sort of compiler-checked type safety; without it, either the programmer has to be careful not to allow scenarios where interleaving is possible, or must implement **mutexes** or other locks on all functions that access critical data [12].

3.2.3 Lifetimes and Stored References

The above system of ownership and borrowing works fine for simple function calls, but it may become complicated if borrowed references are stored in structs. To do this, Rust also has a concept of **lifelines**. Lifelines may be thought of as named scopes, which may be explicitly assigned by the programmer. The lifeline of a struct may not extend beyond that of the referenced data it contains.

4 Independent Analysis Tools

In many ways, a more difficult task than designing a memory-safe language is the analysis of memory safety in existing languages. The tools and methods discussed in this section don't have the luxury of defining a syntax or restricting what constitutes a legal program. Instead, they're intended to partially compile a program and inform the user of potential risks.

4.1 Qualities of Static Analysis Tools

Unlike languages, most static analysis tools use a heterogenous mix of techniques, and do not publish their individual methodologies; as such, reviewing them on an individual basis is less valuable. However, it's important to establish some properties and methods common to static analysis tools. Unlike most language features, these tools are not intended to be correct in every case. Both false potives and false ngatives exist. Additionally, analysis of a program may be very computationally intensive, to the point where performance becomes an issue even at compile time. The following qualities represent potential trade-offs between power or accuracy and performance. This trade-off arises from the complex interactions that inform program behavior; completely predicting the run-time behavior of a program from the source code would be **NP-Complete** [11]. (For a simple proof of this, consider a pogram that consists entirely of a complicated boolean expression, run on values enterred by the user; if the expression returns true, the program runs a statement to access invalid memory. Establishing if a possible execution path for this code will result in a memory hazard is the equivalent of **SAT**, a fundamental example of an NP-complete problem.)

4.1.1 Internal Program Representation

In general, tools will partially compile code in order to form an internal model, which is easier to make logical predictions across. While a number of data structures may be used, the two most important are an **abstract syntax tree** and a **function call graph**. The former is a tree-based representation of a program in which expressions or other blocks are built up of smaller sub-components, usually based off of the program's **grammar**. The latter is a **directed graph** indicating which functions call which others [5].

4.1.2 Flow Sensitivity

Flow sensitivity refers to whether or not an analysis tool takes into account the specific order of operations within a segment of code. This includes awareness that certain pointers may be dereferenced only before or after a certain point in the code, limiting their effective scope to only a subset of their actual scope. A flow-insensitive tool takes the entire scope as a whole, leading to a faster but potentially less accurate analysis [9].

```
if (true) {
    Object x = malloc(sizeof(Object))
    init(x)
    println(x.name)
    free(x)
    //statements not involving x
}
```

A flow-insensitive analysis tool would not be able to gauge if the above code was safe with regards to x, as x becomes de-allocated while it is still in scope. A flow-sensitive tool would see that while x is still lexically available after it has been freed, it is never dereferenced again and is therefore safe.

4.1.3 Path Sensitivity

Path sensitivity refers to the tools awareness of branching paths through the code's control flow (such as if statements and loops), and recognizes if certain paths are unavailable or mutually exclusive. For a simple example, take the following pseudocode:

```
if b
    free x
//statements not modifying b or referencing x
if !b
    x.foo()
```

A path-insensitive tool will report the call to x.foo() as a dangling pointer reference, as it may have been freed in the first conditional statement. A path-sensitive tool, however, could recognize that only one or the other of these statements will be run, but not both, and the code does not actually contain a memory risk.

Once again, path sensitivity comes at a performance cost [9].

4.2 Context sensitivity

Context sensitivty, also referred to as inter-procedural vs. intra-procedural analysis, refers to the tool's awareness of the larger context in which a piece of code may be executed. This may involve path-sensitivity with regard to

the values of global variables, or being able to make predictions as to the parameters passed to a given function. Even more than flow or path sensitivity, context sensitivity greatly increases the cost and complexity of analysis. This complexity can become even worse with **polymorphism**, as the possibility of **dynamic types** that differ from **static types** can make it difficult to predict what function will actually be called from a given context. In a language that is not type checked at all, if is possible for the function call graph to resemble a complete graph [9].

A brief example shows a dangling pointer reference that requires some simple context sensitivity to check. Clearly, the following code dereferences x after the associated memory has been free; however, without knowing the behavior of "bar" and "delete", this cannot be seen from analysis of any single method shown.

```
foo (x){
    //statements
    delete(x);
    //statements
    bar(x);
    //statements
}
delete(x){
    //statements
    free(x)
}
bar(x){
    println(x.name)
}
```

4.2.1 Aliasing

One difficulty faced by static analysis tools is the concept of **aliasing**. Aliasing refers to the possibility of the same memory location being referenced by differently-named variables in the code. For instance, consider the following code:

```
foo(x, y}
    println(x.name)
    free(x)
    println(y.name)
    free(y)
}
```

A naive analysis would conclude that this method is safe, so long as it is passed references to valid memory locations (and so long as those locations are not needed elsewhere, after this function is called.) However, there is another possiblity: x and y may refer to the same location. In that event, the first call

to free() will de-allocate the memory, and the second println() will reference a no-longer valid location. The possibility of aliasing substantially increases the difficulty of code analysis.

4.2.2 Guard

One popular method for formalizing the above concepts is the use of a **guard**. A guard is a boolean statement about the execution state of a program, that my be knowable at compile time. To see this, view the following code section, with the guard described to the right of each line:

```
if ( x == 3)  { //Guard : {}
    y = 7 //Guard : (x==3)
    if (x > y) { //Guard: {x==2 && y == 7}
        ... //Unreachable code
    }
}
foo() //Guard: {}
```

When the observed section opens, we have no information about the execution state of the program. After the initial if statement, we know that x must hold the value of 3, in order for us to be in that execution branch. The next line caries an assignment to a known literal value, which is added to the guard. The two of those allow the analysis tool to recognize that the inner if statement is unreachable. After the initial if statement is exited, the guard returns to empty, as there is no guarantee about the value of x or if the assignment to y was made [14].

5 Theory and Analysis of Papers

A great deal of academic research has gone into the exploration of memory management. Several important or interesting papers are described below.

5.1 Dijkstra; Stack Discipline

In the Recursive Programming section of Numerische Mathematik, Dijkstra proposed what remains as one half of modern memory management algorithms, namely, the use of a stack. In his conception, code is arranged into functional blocks, each of which may only reference memory allocated within that block or higher. This creates the commony references stack of frames, where each newly-added variable is added to the topmost frame, and the topmost frame is entirely freed when code leaves that block [7].

The notion of a stack is so much at the core of modern memory management that it is often forgotten entirely, and the problem of memory management is termed entirely in terms of the heap. However, languages that existed before the notion of stack discipline were severely limited in their structual option;

frequently recursion was limited or impossible, as there was no clear way to give the same named variable different bindings as code was re-entered.

Efficient memory managment is very easily accomplished within this structure, but it comes with several significant limitations. First and foremost, it is not possible for a "child" frame to do the work of allocating memory that could then be referenced by a parent or sibling function. This is a substantial limitation for several coding practices. For example, in this model it is not possible to have a function return a variably-sized list without the parent function knowing the size of the list and allocating space for it in advance [13].

5.2 Tofte and Talpin, Region-Based Memory Management

Tofte and Talpin authored a 1994 paper proposing a hybrid approach to memory managment. Their scheme revolves around dividing a program's memory into a stack of memory regions, each of which may be dynamically grown and then de-allocated together. An inference algorithm is given the task of determining to which region should claim each piece of allocated memory [13].

Fundamentally, Tofte and Talpin's work exists as a partial relaxation on the stack discipline describe above, with three major differences:

- Values created may be associated with a region other than the current top of the region stack
- Not all regions require defined size at the time of creation
- Allocation follows from the type system rather than the grammar.

The first two differences are particularly crucial to the expanded functionality of the system. The first allows a value (in particular, one relevant to a function closure) to refer to a higher location on the region stack. The inference algorithm places the value high enough in order to be visible from any locations that call the relevant function. The second difference is crucial for dynamically-sized types such as lists, which will be allocated entirely to the same region. The third difference allows for the system to remain safe while allowing this expanded functionality.

The specific process they describe modifies a source program by the inclusion of region annotations, of two forms:

- e_i at p_j
- letregion p_i in e_j end

An annotation of the first form is attached to each expression, indicating which memory region the value should be inserted into. The second annotation creates the region stack, with regions de-allocated when the associated "end" statement is reached. In addition to these core annotations, regions may also be passed as arguments to functions at runtime, and the body of a function may place values in regions it recieved as an argument.

It is important to note that this scheme does not catch every potential memory leak in every possible program. Instead, it is intended to be safe (i.e., guaranteed to never result in dangling pointers) while allowing only minor memory leakage, which may be handled by occasional garbage collection without the overhead of a purely garbage-collected model.

5.3 Aiken, Fahndrich, Levien: Improving region-based management of Higher-Order Languages

The authors of this paper make a single substantial divergence from the methods of the previously discussed paper. Aiken and co. lift the restriction that region allocation and de-allocation follow a stack format at all. (While Tofte and Talpin's model allowed the attachment of variables to arbitrary regions, regions would still be introduced and eliminated in a stack-based manner following a first-in first-out methodology [4].)

The technique discussed in this paper involves taking the regional allocations defined by Tofte & Talpin's method and adding explicit calls to allocation and free, which may be placed at any points in the code. (This allows regions to overlap in an arbitrary manner.) The annotations added by Aiken & co. are:

- free_before(*expr*)
- free_after(*expr*)
- alloc_before(*expr*)
- alloc_after(*expr*)

Trivially, this model can matches the performance of Tofte & Talpin's work by attaching an "alloc_before" and "free_after" to the expression block for which each region was allocated in T&T's annotations. This describes the most conservative **completion**. In the language of this paper, a completion refers to any assignment of the above annotations such that there is exactly one allocation and one free per each region along each possible execution path, and such that the memory in each region is only accessed while that region has been allocated. The optimization task of Aiken and co.'s work is to find such a completion that places the allocation as late and the frees as early as possible, thus improving overall memory performance. This optimization is handled by generating constraints based on the first and last usages of an instance of program memory, and then solving that set of constraints to find valid completions.

5.4 Context- and Path-sensitive Memory Leak Detection by Yichen Xie and Alex Aiken

This paper presents an analytical method for the automatic detection of memory leaks across a base of source code. While the algorithm used is sensitive to context, path and flow, their testing still showed it to be tractable even across

large projects (50 minutes to alanlyze 5,000,000 lines of code in the Linux kernel.) This method is named SATURN [14].

5.4.1 Path Sensitivity

Path sensitivity within a function is handled through the use of guards as described above. Each non-branching sequence of steps forms an edge, while statments that have multiple possible predecessors or successors are represented by nodes. Guards are calculated at every step along the network, with branches handled by adding information to the guard based on the condition necessary to follow each path, and merges handled as the disjunction of the guards of the incoming paths. (A certain amount of analysis and simpification must be done at each step, in order to prevent loops from causing un-bounded growth of the guard.) All functions are asumed to have a single exit block; multiple return statements may be handled by inserting a single exit block linked to by all return statements. SATURN analyizes a function in topological order from this graph, and checks for possible leaks in the exit block.

5.4.2 Context Sensitivity

Context sensitivity is handled by the creation of summary representations of functions. The summary of a function must track two crucial pieces of information: whether the function returns newly allocated memory, and any locations where it places parameter data. The first may cause a leak in a caller of the function, as the newly allocated memory must be freed. On the other hand, if parameters are stored at an external location, that may mean that something that appears locally as a leak is not an actual leak in the global context:

```
foo(){
    x = malloc(...)
    store(x)
}
```

The above code will appear as a memory leak to intra-procedural analysis. However, if the summary of store() recognizes that it keeps the passed parameter at another location, SATTURN will not necessarily find a leak.

In analysis of a function, called methods are replaced by short code blocks that have identical behavior with regard to the summary. This relative independence of function analysis is an important part of SATURN's performance. Functions must be analyzed in order by their dependent funtions, but otherwise may be handled in an isolated manner, leading to easy parallelization.

6 Conclusion

Memory managment remains a difficult task in the world of software engineering. While garbage collection is a sufficient solution for many applications, more

performance-bound tasks remain reliant on static memory management. Static analysis tools can substantially ease this task on older languages such as C or C++, but the time complexity inherent in the problems guarantees these will not provide complete solutions. Greater cause for optimism may be found in newer languages, which have the potential to enforce memory managment practices in their syntax and type systems.

7 References

References

[1] Objective-c automatic reference counting (arc). "http://clang.llvm.org/docs/AutomaticReferenceCounting.html#general".

[2] Rust: Ownership. "https://doc.rust-lang.org/book/ownership.html".

[3] The swift programming language. "https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html".

[4] Alexander Aiken, Manuel Fähndrich, and Raph Levien. *Better static memory management: Improving region-based analysis of higher-order languages*, volume 30. ACM, 1995.

[5] Jeffrey Ullman Alfred Aho, Ravi Sethi. *Compilers: Principles, Techniques, and Tools*. Compilers: Principles, Techniques, and Tools, second edition, 2006.

[6] Andrew Borg, Andy Wellings, Christopher Gill, and Ron K Cytron. Real-time memory management: Life and times. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 11–pp. IEEE, 2006.

[7] Edsger W Dijkstra. Recursive programming. *Numerische Mathematik*, 2(1):312–318, 1960.

[8] Amer Diwan, Han Lee, Dirk Grunwald, and Keith Farkas. Energy consumption and garbage collection in low-powered computing. 2002.

[9] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[10] Mark A. Sheldon Franklyn Turbak, David Gifford. *Design Concepts in Programming Languages*. MIT Press, 2008.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[12] David R. O'Hallaron Randal E. Bryant. *Computer Systems: A Programmer's Perspective*. Pearson, second edition, 2010.

[13] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[14] Yichen Xie and Alex Aiken. Context-and path-sensitive memory leak detection. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 115–125. ACM, 2005.