REDUCING VALE'S MEMORY MANAGEMENT OVERHEAD THROUGH

STATIC ANALYSIS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Theo Watkins

June 2021

COMMITTEE MEMBERSHIP

TITLE: Reducing Vale's Memory Management Overhead Through Static Analysis

AUTHOR: Theo Watkins

DATE SUBMITTED: June 2021

COMMITTEE CHAIR: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Maria Pantoja, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Christopher Siu, M.S.
Lecturer of Computer Science

ABSTRACT

Reducing Vale's Memory Management Overhead Through Static Analysis

Theo Watkins

Vale is a multi-purpose programming language that focuses on guaranteeing memory safety with minimal effect on performance. To accomplish this, Vale utilizes a memory management system called Hybrid Generational Memory (HGM). HGM uses generational references to track the state of objects in memory, and static analysis to reduce memory management overhead at runtime. This thesis describes the program that performs static analysis on Vale source code during compilation, and analyzes its effect on the performance of Vale programs.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

A language's memory management system affects all aspects of the language from its type system, to its performance, to the cognitive model of programmers using the language. Languages must decide if memory will be managed automatically by the implementation, manually by the programmer, or some combination of the two. This decision depends on the language's desired features and use cases.

Vale [7] is a high-level language meant to be flexible and easy to use for many applications. Like Python, JavaScript, Swift, Golang, and many other languages that share these characteristics, Vale aims to provide fully automatic memory management. Automating memory management is an expensive task, and most languages that do so cannot compete with the performance of languages with manual memory management. Vale aims to close this performance gap with a novel approach to memory management called Hybrid Generational Memory [8] (HGM). HGM uses generational references to track objects in memory, and static analysis to improve performance.

1.1 Hybrid Generational Memory

HGM is Vale's system for memory management. This section introduces the three main parts of HGM: generational references, generation checks, and static analysis.

1.1.1 Generational References

HGM uses generational references to ensure memory safety. Generation numbers are assigned to chunks of memory; each number starts at zero and increases monotonically. When an object is created in a Vale program, it is placed in a suitable chunk of memory and any reference to the object stores a copy of the object's current generation number, called the *target generation*. In Vale, objects are freed via the `drop` function which marks the object's memory as available for reuse and increments the generation number of the object's memory. This setup slightly increases the size of objects and references in memory because they must store a generation number. The following section discusses why these generation numbers are necessary in HGM and how they are used.

1.1.2 Generation Checks

Each time a Vale program attempts to use a reference to an object, HGM will check if the target generation number of the reference matches the actual generation number of the associated memory. If the generation numbers do not match, then the object is no longer accessible because the memory may have been reused. In this situation, Vale will report an error and safely exit the program. This process of comparing generation numbers is called a generation check; these are the primary focus of Vale's static analysis because of the runtime overhead that they induce.

Generation checks are a large source of overhead in HGM for three main reasons: branching, cache misses, and code size.

1.1.2.1 Branching

Branch predictors help modern CPUs quickly execute branching instructions, but no predictor is perfect, so it can still be an expensive operation. In the event that the predictor is wrong, the CPU must rewind and replace the previously queued instructions which can take a significant amount of time. Generation checks in HGM require a branching operation because the processor must decide if the generation numbers match and execution should continue, or if the numbers do not match and the program should halt.

1.1.2.2 Cache Misses

In addition to branch prediction, CPUs use caches to improve execution times. Frequently used chunks of memory are stored in caches which can be accessed much faster than main memory. Cache misses occur when the CPU requires some memory not currently present in the caches. Cache misses require the CPU to access main memory, slowing down the program.

In HGM, generation checks can incur cache misses. Usually the generation number is loaded in to the cache with the object, so there will only be cache miss if the object being accessed is not already in the cache. However, some large objects occupy more memory than a single cache line. For these objects, it is possible that the requested memory is loaded into the cache, but the generation number is not. This could lead to an additional cache miss in order to perform a generation check.

1.1.2.3 Code Size

Generation checks in HGM inflate the code size and instruction count of a program. The additional instructions not only take time to execute, but also take up space in the CPU's instruction cache. This decreases the space available in the cache for other instructions, further slowing down the program.

1.1.3 Static Analysis

Generational references and generation checks make HGM safe, but static analysis is what makes the HGM fast. The three sources of runtime overhead outlined in 1.1.2 are the main motivation for HGM's static analysis. The Vale compiler produces an abstract syntax tree (AST) that can be modified to eliminate generation checks in situations where static analysis can guarantee that the accessed object is still alive. HGM's algorithm for analyzing and modifying this AST is the focus of this thesis and is described in detail in Chapter 4.

The remainder of this thesis is structured as follows: Chapter 2 evaluates some popular memory management techniques in order to contextualize HGM. Chapter 3 discusses Rust and Lobster, two languages with goals similar to Vale's that inspired some of the techniques used in HGM. Chapter 5 describes how HGM's static analysis is tested to verify accuracy, and presents the effects of static analysis on a benchmark program written in Vale. Chapter 6 discusses features that could be added to the static analysis program described in Chapter 4 to increase its effectiveness, and Chapter 7 concludes.

Chapter 2

BACKGROUND

This chapter introduces some common memory management techniques before presenting Vale's Hybrid Generational Memory (HGM) model. The description of HGM also provides some motivations for the static analysis discussed in Chapter 4.

2.1   Memory Management

In programming languages, memory management refers to tracking memory that is, or is no longer, in use by a program. Memory that is not reachable in the code should be reclaimed so that it can be reused. Unreachable memory can unnecessarily inflate the amount of memory a process is using and slow down the system.

In general, languages fall into two categories: those that require manual memory management by the programmer, and those that automatically handle or enforce memory management. Languages with manual memory management place a burden on the programmer and are vulnerable to difficult bugs, while those with automatic memory management inevitably incur a cost at runtime.

Vale's memory management system is the focus of this thesis. Vale is a language that aims to provide automatic memory management with minimal runtime overhead using a novel system called Hybrid Generational Memory (HGM). Section 1.1 discusses Vale's HGM model in more detail; this section provides some background on other memory management techniques.

2.1.1 Manual Memory Management Issues

The two most popular languages that require manual memory management are C and C++ [2]. For decades these have been the obvious language choices when speed is a priority. However, the lack of automatic memory management can introduce subtle bugs that are difficult to resolve.

The C programming language does not have any protections against accessing corrupted memory. This allows some harmful bugs to go unnoticed until the program halts due to an illegal memory access (a segmentation fault). These are very commonly caused by subtle bugs such as accessing memory that has been freed, writing to read-only memory, or accessing outside of array bounds.

Memory leaks are another common type of bug in C programs. These occur when memory that is no longer in use is not released for reuse. C does not automatically reclaim memory so it is the duty of the programmer to call `free()` on any previously allocated memory; neglecting to do so results in memory leaks [10]. Memory leaks lead to unreachable objects unnecessarily inflating the amount of memory in use by a process.

In addition to segmentation faults and memory leaks, dangling pointers are a common bug in C programs. Dangling pointers occur when a program attempts to use a pointer to memory that has already been freed [10]. Use of a dangling pointer leads to undefined behavior and must be avoided.

C++ is similar to C, but allows for object-oriented programming concepts like classes and inheritance. These concepts have been used to help reduce the number of memory-related bugs that are common in C programs. Popular classes from the C++ standard library, such as `unique_ptr` and `shared_ptr`, provide some forms of

automatic memory management. Only one `unique_ptr` can refer to its object; when the `unique_ptr` goes out of scope and the object is no longer accessible, its object is automatically freed. Similarly, multiple `shared_ptr`s can point to an object; when all the `shared_ptr`s are out of scope the object is automatically freed [5]. Classes like these can simplify writing programs, but they do incur some runtime cost. The code from these classes adds to the instruction count and can contain costly operations like branching. For example, the `shared_ptr` class uses reference counting (see Section 2.1.2.2) to ensure memory safety which adds runtime overhead each time a `shared_ptr` is created or destroyed.

C++ provides a middle ground between unmanaged memory and fully automatic memory management, giving the programmer freedom to make tradeoffs between speed and simplicity in their programs. This can be very useful to some programmers, but the speed of processors and the abundance of memory in modern machines have made fully automatic memory management popular in newer languages, despite the overhead at runtime. Unlike C or C++, Vale's goal is to completely automate memory management while minimizing runtime cost.

2.1.2   Automatic Memory Management

Automatic memory management is often desirable because it makes programs easier to write and debug. Two approaches to automating memory management have been dominant in programming languages over the last few decades: garbage collection, and reference counting. This section will discuss the advantages and disadvantages of each method, as well as introduce an alternate method that has gained traction recently due to its effective usage in Rust.

2.1.2.1 Garbage Collection

Garbage collection provides memory safety by periodically pausing execution of a program to free memory that is no longer in use. This approach is popular because it usually has better throughput than naive reference counting (see Section 2.1.2.2) which incurs more predictable but also more frequent overhead throughout execution. Many popular languages use garbage collection, including Java and Javascript, but the approach is not ideal for all problems. Games, for example, have trouble maintaining a consistent frame rate when there are non-deterministic pauses in execution.

Python uses reference counting, but it is supplemented with garbage collection. Garbage collection helps handle patterns like reference cycles that cause issues for reference counting. Python has a library that allows the programmer to opt out of the garbage collector when they wish to avoid non-deterministic pauses, and opt in when they wish to reduce the reference counting workload and improve runtime speed.

In addition to non-deterministic pauses, another drawback of garbage collection is that it does not immediately reclaim unreachable memory. Mark and sweep is a common method of garbage collection where objects are marked when they become unreachable, and when the program reaches a certain threshold of memory usage, the garbage collector "sweeps" through the program's memory and frees all unreachable objects. This approach allows unreachable objects to occupy memory for at least some period of time. For some programs this is fine, but sometimes it can be desirable for a program to constantly reclaim and reuse memory rather than periodically reclaiming memory in large sweeps, so that the maximum amount of memory is available to the program at all times.

There have been many modifications and improvements to naive garbage collection. Two of the more popular variations are generational garbage collection [1] and incremental garbage collection [12]. Generational garbage collectors organize allocations based on their age, prioritizing the reclamation of newer allocations. This is based on the idea that most objects are used briefly before becoming unreachable, and the oldest objects tend to remain in memory the longest. New pointers referencing old objects are much more common than old pointers referencing new objects, so this approach can reduce the number and length of garbage collection pauses. However, it is not a perfect improvement because it can allow unreachable objects (especially old ones) to occupy space in memory for longer than they might with mark and sweep garbage collection. One study found that generational garbage collection can rival the speed of manual memory management, but requires five times the amount of memory to do so [3]. When limited memory is available, generational garbage collection slows down significantly.

Incremental garbage collection is used to make garbage collection more suitable for real-time applications. Rather than long pauses to reclaim all unreachable memory, the tasks of the garbage collector are distributed between shorter pauses that have less of a noticeable effect on the program's output. While this approach is useful for real-time applications, it does not necessarily improve runtime, and the more frequent switching of contexts damages cache performance [13].

2.1.2.2 Reference Counting

The other popular approach to memory management is reference counting. At runtime, reference counted languages keep track of how many references a program has to an object in memory, automatically freeing the memory if this number drops to

zero. While this is a safe and simple approach to memory management, it incurs much overhead.

Each time a new reference is created or destroyed the language must increment/decrement the reference count for the object, adding to the instruction count of a program. Reference counting also suffers poor cache performance due to the locality of objects in memory. Garbage collection copies reachable objects to new pages in memory when collecting; this increases locality and improves cache performance. Reference counting never copies objects, so it does not reap this benefit [11]. Additionally, there is a branching operation each time a reference is destroyed to determine if the object can be freed. While branch predictors in modern processors are extremely accurate, it is still a costly operation; especially when the processor predicts incorrectly and must rewind instructions.

Despite its overhead, reference counting has been a popular memory management technique for decades. It is the only widely used memory management technique that promises automatic and immediate reclamation of unreachable memory. Because of its popularity, many variations of reference counting have emerged that attempt to reduce its overhead, and while improvements have been made, it remains the obviously slow approach to memory management. Vale's goal is to provide automatic and immediate memory reclamation at a speed that competes with popular garbage collected and unsafe languages.

2.1.2.3   Rust

Rust is a language with a unique approach to automatic memory management that does not require garbage collection or reference counting. Rather than performing additional tasks at runtime, Rust's compiler forces the programmer to use memory-

safe patterns. This approach makes Rust extremely fast because there is little runtime overhead, but it can also cause headaches for programmers that are not accustomed to the required patterns. Vale adopts some of Rust's memory management techniques to reduce runtime overhead, but eliminates some of Rust's required patterns to allow the programmer more freedom at the expense of, by default, runtime checks.

2.2 Single Ownership

This section focuses on single ownership, a concept adopted by Vale that was popularized by Rust and C++. Implementations vary by language, but the concepts presented in this section are essential to any language with single ownership. Single ownership is powerful because it can reduce a language's memory management workload.

2.2.1 Owning References

In languages that use single ownership, objects have a single owning reference at any given time. If an object's owning reference goes out of scope, the object will be freed. Single ownership helps programs run fast because using an owning reference does not require any memory management overhead at runtime. If an owning reference is being used, then it cannot be out of scope and the object it owns is guaranteed to be alive.

Another rule of single ownership is that if an owning reference is used in a function call or assigned to a new variable, then ownership of the object is transferred. This operation is called a *move*, and moving ownership invalidates the old reference [4]. The old reference cannot remain valid because maintaining multiple owning references

11

```
1   struct Spaceship {
2       fuel int;
3   }
4
5   fn printFuel(ship Spaceship) {
6       println(ship.fuel);
7   }
8
9   fn main() {
10      s = Spaceship(5); // s is an owning reference
11
12      printFuel(s);
13      // Ownership is moved to printFuel
14      // s is no longer valid
15
16      println(s.fuel); // compiler error
17  }
```

Figure 2.1: Vale program to illustrate moving ownership

would lead to bugs. If two owning references did exist, then when they go out of scope
the associated memory could be freed twice. Double freeing has undefined behavior
and should be avoided.

Figure 2.2.1 is a Vale program that shows how ownership can be moved. Line 10
creates an owning reference to the `Spaceship` object. Then on line 12, ownership
of the object is moved to the `ship` argument of the `printFuel` function. When
`printFuel` finishes executing and `ship` goes out of scope, the memory now owned by
`ship` is freed. Line 16 will therefore throw a compiler error because the old owning
reference, `s`, is no longer valid.

Ownership can also be moved via return values. A function can return an owning
reference that was created within the function or passed as an argument. This action
moves ownership of the returned object from the function into its caller's scope.

```
1  struct Spaceship {
2      fuel int;
3  }
4
5  // '&Spaceship' denotes that the argument type
6  // is a borrow reference to a 'Spaceship'
7  fn printFuel(ship &Spaceship) {
8      println(ship.fuel);
9
10     // 'ship' is a borrow reference, so nothing special
11     // happens when it goes out of scope
12 }
13
14 fn main() {
15     s = Spaceship(5); // s is an owning reference
16
17     printFuel(&s);
18     // '&s' creates a borrow reference
19     // to the object owned by 's'
20
21     println(s.fuel); // 's' is still a valid owning reference
22
23     // the object owned by 's' is freed
24     // when 's' goes out of scope
25 }
```

Figure 2.2: Vale program to illustrate borrow references

2.2.2   Borrow References

If a language only allowed owning references, memory management would be trivial, but ownership rules would severely limit the programmer. Most languages that use single ownership have some form of borrow references. Borrow references refer to an object without claiming ownership.

Figure 2.2.2 modifies Figure 2.2.1 to a Vale program that uses a borrow reference instead of moving ownership. The & annotation on line 17 creates a borrow reference to the object owned by s. The & is also added to the type annotation in the header of printFuel, denoting that the function receives a borrow reference. In this program, printFuel never receives ownership of the Spaceship. Line 21 therefore compiles and executes successfully because its owning reference is still in scope. Rust and Vale both use the & annotation to denote borrow references, while C++ uses the

`unique_ptr` and `shared_ptr` classes discussed in Section 2.1.1 to provide the same functionality.

2.3   Variability

In Vale, variability is a reference's ability to be pointed to a new object. References can either be 'varying' or 'final', and are 'final' by default. When declaring a variable, adding a `!` to the end of the variable name will generate a varying reference. The `set` keyword can then be used to point a varying reference at a different object of the same type. If a varying *owning* reference is pointed to a new object, the object it previously pointed to will be freed using the `drop` function (see Section 1.1.1) and its memory will no longer be valid [1]. Pointing a varying *borrow* reference to a new object will not drop the object previously pointed to by the reference.

Figure 2.3 shows a small Vale program that makes use of a varying owning reference. Line 6 creates `s`, a varying owning reference to a `Spaceship`, while line 7 creates `s2`, a final owning reference to a `Spaceship`. On line 9, `s` is pointed to `s2`'s referend, which is allowed because `s` is varying. Additionally, because both references own their referends, line 9 drops `s`'s old referend and moves ownership of `s2`'s referend to `s`, invalidating `s2`. Line 13 then returns a value of 20.

References in C++ and Rust cannot be varying, but similar functionality can be accomplished by wrapping a reference in a mutable struct. Then, because the reference is a member of the mutable struct, the object it refers to can change. C++ also has pointers which by default can be pointed to new memory unless they are declared constant with the `const` keyword.

---

[1] `set` expressions in Vale actually result in the old reference, but if the old reference is not captured its referend will be dropped

```
1   struct Spaceship {
2     fuel int;
3   }
4
5   fn main() int export {
6     s! = Spaceship(10); // 'variable' owning reference
7     s2 = Spaceship(20); // 'final' owning reference
8
9     set s = s2;
10    // the old referend of s ('Spaceship(10)') is dropped
11    // ownership of s2's referend ('Spaceship(20)') is moved to s
12
13    ret s.fuel; // 20
14  }
```

Figure 2.3: Vale program to illustrate variability

Variability is not the same as mutability. A mutable reference allows for an object's data to be mutated, but does not change the location of the data in memory. A varying reference allows the reference to point to an entirely different location in memory. This distinction is important when considering memory management techniques. Mutating the data in an object referred to by a mutable reference does not affect the safety of accessing that object's data via a different reference. The referend will be at the same memory location and the memory will have the same structure even when values change. However, when a varying owning reference is pointed to a new object, references to the old object are no longer valid. Additionally, pointing a varying borrow reference to a new object can change the scope of its referend.

Chapter 3

RELATED WORKS

This chapter provides some detail on two languages that use memory management techniques similar to Vale's Hybrid Generational Memory (HGM). The first is Rust, which was briefly discussed in Section 2.1.2.3, and the other is Lobster, a modern language for gaming and graphics that uses some static analysis techniques similar to HGM's.

3.1 Rust

Section 2.2 described some features of single ownership that apply to both Rust and Vale. This section will discuss Rust's additional single ownership rules that Vale avoids.

Rust guarantees memory safety by placing restrictions on references to an object. One of these restrictions is that in Rust programs there can only be one mutable reference to an object per scope. Figure 3.1 shows an invalid pattern in the body of a Rust program. Mutable references are created with the `mut` keyword and mutable borrow references can only be created from mutable owning references. When `r1` is initialized, `s` can no longer mutate its object. When `r2` is initialized the compiler will throw an error because this creates a second mutable reference to the same data.

Additionally, Rust does not allow read-only references if there is a read/write (mutable) reference. Figure 3.1 is another invalid pattern in Rust programs. `r1` and `r2`

```rust
let mut s = String::from("hello"); // mutable owning reference

let r1 = &mut s;
// 'r1' is a mutable borrow reference
// 's' can no longer access the string

let r2 = &mut s;
// 'r2' attempts to be the second mutable borrow reference
//   compiler error

println!("{}, {}", r1, r2);
```

Figure 3.1: Rust program to illustrate the restrictions on mutable references [4]

```rust
let mut s = String::from("hello"); // owning reference

let r1 = &s; // read only, no problem
let r2 = &s; // read only, no problem
let r3 = &mut s; // read/write, compiler error
// cannot modify data while other references have access to it

println!("{}, {}, and {}", r1, r2, r3);
```

Figure 3.2: Rust program to illustrate the restrictions on combinations of muatble and immutable references [4]

are successfully initialized because Rust allows multiple read-only borrow references. However, r3 cannot be initialized because it is a read-write reference.

In summary, Rust allows *either* one mutable reference *or* many immutable references per scope. These rules are enforced at compile time by Rust's 'borrow checker'.

A large reason these rules are necessary is because of Rust's enums. In Rust, an object can contain an instance of an enum as one of its members, and that member's value can be changed via a mutable reference to the struct. The possible values of enums are often different types, meaning they occupy a different amounts of memory and have different shapes. Additionally, if a struct member contains an enum instance, the memory associated with that enum instance is stored directly inside the struct's memory. Therefore, when the type of the enum instance is changed via a mutable reference, the shape of the struct's memory is changed. This means that other refer-

17

ences to this struct now contain pointers to invalid memory (memory that is not of the shape expected by the pointers). Rust prevents this with the 'borrow checker' which does not allow any other references to point to an object while there is a mutable reference pointing to it.

The 'borrow checker' combined with the basic rules of single ownership outlined in Section 2.2 prevent memory leaks and segmentation faults in Rust. However, there is one more memory-related bug that the Rust compiler must handle, dangling pointers. The Rust compiler prevents dangling pointers by ensuring that no borrow reference escapes the scope of its data. Usually this occurs when a borrow reference to some data is returned from a function, and the owning reference to the data goes out of scope upon this return. The data is freed when the owning reference goes out of scope, and the returned borrow reference now points to invalid memory. Rust detects and reports these occurrences at compile time.

These additional restrictions eliminate the need for runtime memory management, making Rust fast and safe, but limiting programmers. Most modern languages allow mutability at any time by performing runtime checks for memory safety. Programmers that are accustomed to this freedom have historically struggled with Rust's 'borrow checker' [4].

The patterns illustrated by Figures 3.1 and 3.1 are both valid in Vale. Vale does not yet support `enum`s, so the shape of an object is guaranteed to be constant even when there are mutable references to it. Because of this, owning references in Vale are mutable by default, and Vale allows multiple mutable borrow references to the same object. This makes it more difficult to detect some unsafe patterns like dangling pointers, but Vale uses runtime generation checks to prevent these. This adds runtime overhead to Vale that Rust does not have, but some of this overhead can be mitigated with the static analysis techniques described in Chapters 4 and 6.

18

3.2 Lobster

The Lobster programming language is a high-level language for programming games and graphics. Memory management in Lobster uses a combination of single ownership and reference counting. Similar to Vale, Lobster reduces runtime overhead by performing static analysis on a program's abstract syntax tree (AST) at compile time. Unlike Vale, Lobster's static analysis is used to eliminate runtime reference count operations (as opposed to Vale's runtime generation checks described in Section 1.1.2). Lobster's static analysis has had great success, eliminating up to 95% of runtime reference count operations for various benchmark programs [6].

Lobster's static analysis is interwoven with its type checking system. It picks an owner for every heap allocation and attempts to make every other reference to that object a borrow reference. The initial owning reference and subsequent borrow references do not affect the reference count of an object.

Every AST node has a predefined ownership type that it expects of its children and an ownership type that it passes to its parent. These pre-defined ownership types are determined by the capabilities of the type of AST node. Some nodes require ownership of their children to be memory safe, but others do not. This section will only look at two types of AST nodes necessary for a simple example: assignments and constructors, and assume that all other nodes have properly safe pre-defined ownership types. Assignment nodes want to own their children, and want their parent to borrow from them; constructors do not have children (unlike Vale's AST structure), and want to be owned by their parent.

When the static analysis algorithm parses a node, it checks that the node's expected ownership type matches the parent ownership type of its child. In most cases the types

match and no reference counts need to be updated. For example, in the line `let a = [1, 2, 3]` the list constructor node wants to be owned, and the node assigning it to `a` wants to own it, so no reference counts are required.

Sometimes though, a parent node wants to own, and the child node wants to be borrowed. Consider the line of Lobster code: `let a = b`. Assuming that `a` and `b` have already been assigned objects to own, then `a` wants to own `b`'s object , but `b` wants its parent to borrow its object (because `b` already owns it). In this situation, Lobster lets `a` and `b` both own the object, but increases the reference count on the object.

It is also possible that a parent node wants to borrow, but the child node wants to be owned. Lobster's `print` function for example only wants to borrow its argument. If a constructor is passed to `print`, an anonymous owning reference will be generated to own the constructed object, and free it at the end of the current scope.

Lobster's effective use of static analysis to eliminate reference count operations inspired Vale to use a similar approach for generation checks. While Lobster and Vale use static analysis and single ownership in different ways, their goal is the same. Both languages aim to speed up their memory management systems by eliminating runtime checks through static analysis.

Chapter 4

HYBRID GENERATIONAL MEMORY'S STATIC ANALYSIS

Vale's compiler is divided into three stages: Valestrom, Catalyst, and Midas. Valestrom parses the source code and produces a JSON-formatted abstract syntax tree (AST). Catalyst, the focus of this chapter, analyzes this AST and adds information to eliminate generation checks where possible. This updated AST is then passed to Midas, the final stage of the compiler. Catalyst is written in Java using the JSON Simple library to parse and update the AST. This chapter describes in detail the patterns that allow Catalyst to identify generation checks that can be eliminated by editing the AST.

4.1 Catalyst's Goals

Catalyst aims to eliminate as many generation checks as possible for mutable objects. Arrays and structs are mutable in Vale, while integers, floats, booleans, and strings are all immutable. References to these immutable types are called 'shared' references in Vale, and are managed with reference counting. Structs and arrays can also be declared as immutable and their references will then also be treated as 'shared' references.

Catalyst focuses on mutable structs. Mutable arrays are not yet handled, but Section 6.1 proposes a method that could allow them to be optimized in a similar fashion to mutable structs. This chapter discusses when Catalyst can and cannot identify generation checks that can be eliminated for mutable structs.

4.2 Parsing the AST

Vale's AST for a program is comprised entirely of expressions, of which there are 37 different types. Similar to a Vale program itself, the AST contains two distinct sections: struct definitions and function definitions[1] (see Figure 4.2). Section 4.3 describes the information that Catalyst extracts from the struct definitions, while Sections 4.4 - 4.8 describe how the function definitions are parsed and optimized.

Figure 4.2 shows a simplified version of Vale's AST, containing some of the fields relevant to Catalyst. A few field names are also altered for clarity. The `block` field of a function definition describes the code run by the function. Catalyst uses information extracted from other functions and from the struct definitions to identify generation checks that can be eliminated in these blocks.

In order to eliminate generation checks at runtime, Catalyst will change the `knownLive` value of certain AST nodes. Nodes that dereference pointers contain the `KnownLive` field; a boolean specifying whether the referend is known to be alive (and that the runtime generation check can be skipped). Prior to Catalyst's modifications, all `KnownLive` fields in the AST are false.

4.3 Struct Definitions

Catalyst's first task is go through the list of struct definitions and add each struct to a hashmap mapping the name of a struct to information on its members. Figure 4.3 shows pseudocode for the definition of this hashmap and the `Member` class used within it. The `Member` class stores the name, variability (see Section 2.3), and ownership (see

---

[1]Vale does not yet support globals. Vale does support interfaces, but these are currently ignored by Catalyst.

```
1  {
2     "type": "program",
3     "structs": [
4       {"type": "struct",
5       "name": ...,
6       "mutability": ...,
7       "members": [...]},
8       ...],
9     "functions": [
10      {"type": "function",
11      "name": ...,
12      "arguments": ...,
13      "return": ...,
14      "block": ...},
15      ...]
16  }
```

Figure 4.1: Vale's AST Structure

```
1  structInfo = HashMap<String, Member[]>();
2
3  class Member {
4      String Name;
5      String Variability;
6      String Ownership;
7      Optional<String> StructName;
8  }
```

Figure 4.2: `structInfo` map in Catalyst

Section 2.2) of each member, as well as an optional string that will only contain a struct name if the member is a struct itself. This hashmap contains all the struct information that Catalyst will need for the duration of its execution.

4.4 Analyzing a Single Scope

When an object is created, its allocated memory is guaranteed to be safe until its owning reference is destroyed via a call to drop (e.g., `drop(s)`). This call is usually implicit when the function with the owning reference returns, but it can also be called explicitly by the programmer. Dropping an owning reference frees the memory of its referend.

```
1  struct Spaceship {
2    fuel int;
3  }
4
5  fn main() int export {
6    s = Spaceship(10); // owning reference
7    b = &s; // borrow reference
8
9    ret b.fuel; // generation check on b's referend (the Spaceship)
10   // s goes out of scope here,
11   // so there's an implicit drop(s),
12   // freeing the Spaceship.
13 }
```

Figure 4.3: Vale program requiring a generation check

```
1  class FunctionMaps {
2      Objects = HashMap<Long, LivenessInfo>();
3      Variables = HashMap<Long, ReferenceInfo>();
4      Return = ReturnInfo();
5
6      // Methods for maintaining the maps and return information
7  }
8
9  class ReferenceInfo {
10     Long Object;
11     String Name;
12     String Variability;
13     String Ownership;
14 }
15
16 class LivenessInfo {
17     Boolean Liveness;
18     StructMember[] Members;
19     Optional<Long> Parent;
20 }
```

Figure 4.4: `FunctionMaps` **class in Catalyst**

Catalyst leverages this feature of single ownership to eliminate generation checks for references to objects whose owning reference is still guaranteed to be in scope. In Figure 4.4, s is an owning reference to a Spaceship object and b is initialized as a borrow reference to the same object. Because b is not an owning reference, the return expression on line 9 normally incurs a generation check. However, it is obvious that the object referenced by b will be alive, because the owning reference has not been dropped or moved to a different scope.

24

Catalyst maintains two separate hashmaps for each function in the AST, one for mapping objects to their liveness information, and one for mapping references to their objects. Figure 4.4 contains pseudocode for the `FunctionMaps` class that maintains these maps as well as the `LivenessInfo` and `ReferenceInfo` classes used within the maps. A new `FunctionMaps` instance is created for each function in the AST. This section focuses on the `Liveness` field of the `LivenessInfo` class, and the `Object` and `Name` fields of the `ReferenceInfo` class. This information is sufficient for eliminating generation checks for objects with immutable members whose owning reference is in scope (like the `Spaceship` in Figure 4.4).

When Catalyst encounters the AST nodes describing line 6 in Figure 4.4, it first parses the call to the `Spaceship()` constructor. This call returns an owning reference to a `Spaceship` (see Section 4.6 for more information on function calls), generating an entry in `main`'s `Objects` map with a key of 0. Catalyst assigns unique keys to objects using a counter that starts at 0. The object has just been created, so Catalyst sets the `Liveness` value of the object's `LivenessInfo` class to 'true', indicating that it is known to be alive.

Once the object is created, Catalyst will analyze the AST node describing the variable assignment. This will generate an entry in `main`'s `Variables` map, pointing the variable `s` to the previously created object. Unlike the `Objects` map, keys for the `Variables` map are provided by the AST, which contains a unique number identifier for each local variable in a scope. This value is available in any AST node that dereferences a variable.

Line 7 creates a borrow reference to `s`, generating a new entry in the `Variables` map that also points to the `Spaceship`. Figure 4.4 shows a simplified illustration of `main`'s `FunctionMaps` instance after parsing line 7 of the example in Figure 4.4.
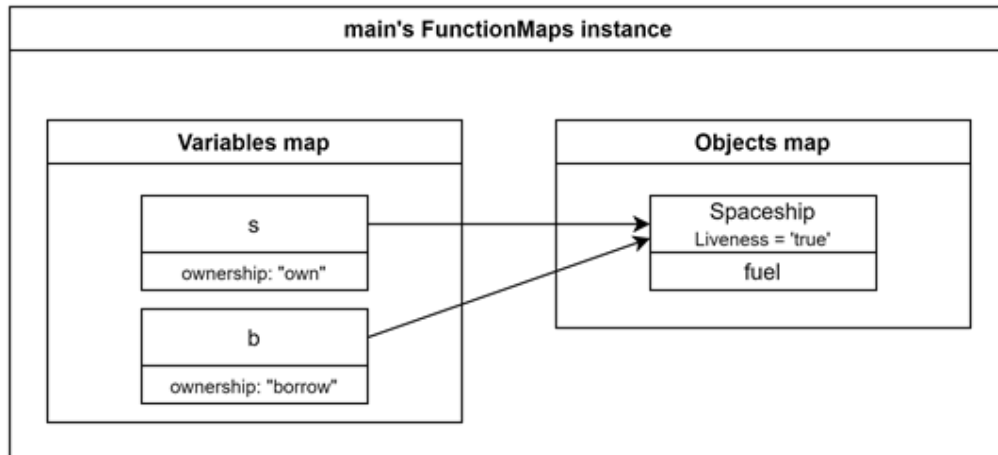
Figure 4.5: State of `main`'s `FunctionMaps` immediately prior to the return expression in Figure 4.4

Variable `b` is dereferenced on line 9. When Catalyst encounters a dereference, it looks up the reference in the `Variables` map and uses the `ReferenceInfo` associated with the variable to look up its referend in the `Objects` map. If the `Liveness` field of the `LivenessInfo` instance associated with the referend is set to 'true', then the `KnownLive` field in the AST is also set to 'true', and the generation check will not be executed at runtime. In Figure 4.4, variable `b` points to object 0 whose `Liveness` value is 'true', so the generation check on line 9 can be skipped at runtime.

4.5 Members

Catalyst also maintains a hierarchy of objects via the `Members` and `Parent` attributes of the `LivenessInfo` class. If an object has members that are not shared references, these too must populate the `Objects` map.

Figure 4.5 shows a Vale program that generates two objects in `main`'s `Objects` map. Lines 10 and 11 create an `Engine` and `Spaceship` object respectively. The first argument to the `Spaceship` constructor is a borrow reference to the `Engine` object

```
1  struct Engine {
2     fuel int;
3  }
4  struct Spaceship {
5     engine &Engine;
6     numWings int;
7  }
8
9  fn main() export {
10    engine = Engine(10)
11    ship = Spaceship(&engine, 4);
12    borrowShip = &ship;
13    println(borrowShip.engine.fuel);
14 }
```

Figure 4.6: Vale program to illustrate Catalyst's member hierarchy

```
1  class StructMember {
2      Optional<Long> Id;
3      String Ownership;
4      String Variability;
5  }
```

Figure 4.7: StructMember class in Catalyst

created on the previous line. When Catalyst analyzes the AST node for this borrow reference, it will extract the key (in the Objects map) of the Engine object pointed to by the reference. When Catalyst finishes analyzing the node associated with the Spaceship constructor and adds the Spaceship to the Objects map, it will also add two StructMember entries to the Members array in the LivenessInfo instance associated with the object.

Figure 4.5 shows pseudocode for the StructMember class. The Id field of the first StructMember in the Spaceship from Figure 4.5 will be set to the key of the Engine in the Objects map (0 because it was the first object created in the function). The Id field of the second StructMember will be empty because the member is an immutable int, and immutable objects do not generate entries in the Objects map.
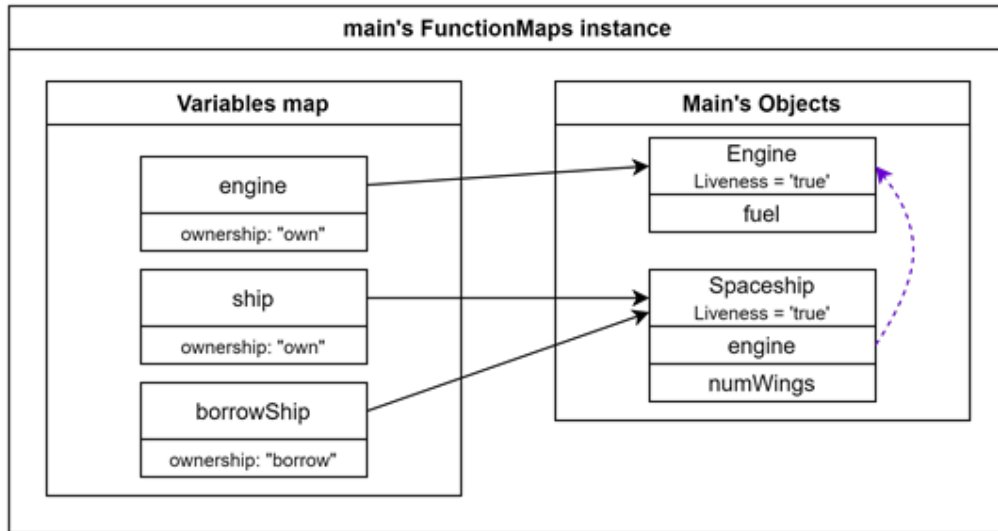
Figure 4.8: State of `main`'s `FunctionMaps` **immediately prior to the print expression in Figure 4.5**

Figure 4.5 shows the state of `main`'s `FunctionMaps` after the `borrowShip` variable is created on line 12 of Figure 4.5. Note the purple dashed arrow indicating that the `Engine` object is a member of the `Spaceship` object.

The `Parent` field of the `LivenessInfo` instance (see Figure 4.4) associated with the `Engine` will remain empty because the `Spaceship` has a borrow reference to the `Engine`. Only objects owned by another object have a `Parent` (there is one exception to this rule discussed in Section 4.6.3). If the `Spaceship` constructor's first argument were an owning reference (i.e., `engine` instead of `&engine`), then the `Engine` object's `Parent` would be set to the `Spaceship`'s key in the `Objects` map.

Normally Figure 4.5 would require two generation checks, both on line 13. The first checks the generation of the `Spaceship` and can be eliminated for the same reasons we were able to eliminate the generation check in Figure 4.4. The second checks the generation of the `Engine` and requires that the `Engine` be accessed through the `Members` field of the `Spaceship`'s `LivenessInfo` instance. The AST will provide the

28

index of the member being accessed, and Catalyst will use the `Id` of the `StructMember` instance (if it is not empty) at the appropriate index to look up the member's entry in the `Objects` map. Catalyst then checks the `Liveness` field of this object (the `Engine`) and if it is 'true' changes the `KnownLive` field of the current AST node to match. The `Engine`'s `Liveness` value will be 'true' in this example because the owning reference has not been dropped or handed to a new scope.

4.6   Analyzing Multiple Scopes

Function calls introduce new complexity to Catalyst's tasks. They allow owning references to be moved out of their original scope, and therefore Catalyst must store more information about the program to optimize function calls. The most important information that Catalyst needs to know about a function call is the relationship between the arguments to the function and the returned value. The struct information (see Section 4.3) and object hierarchy (see Section 4.5) pair with function-specific information to allow Catalyst to make some inferences about objects that are guaranteed to be alive through function calls. This section discusses the information that Catalyst maintains on each function and how it is used to identify more generation checks that can be eliminated.

4.6.1   Global Function Information

So far Catalyst has only utilized information on the local scope to eliminate generation checks, but maintaining information about all scopes in the program is beneficial for handling function calls. Similar to the `structInfo` map from Figure 4.3, Catalyst maintains a map of function names to information about a function's objects and return value. Figure 4.6.1 shows the pseudocode initialization of this map. Each time

```
1  functionInfo = HashMap<String, FunctionMaps>();
```

Figure 4.9: `functionInfo` **map in Catalyst**

Catalyst finishes parsing a function, the entire `FunctionMaps` class (see Figure 4.4) associated with the function is added to the map. Catalyst begins by parsing the function definitions in the order they appear in the AST, but if it encounters a call to a function that does not yet have an entry in the `functionInfo` map, Catalyst will parse the entire definition of the callee and add its information to the map before proceeding with the call.

To handle recursive calls, Catalyst adds a function's name as a key to the `function-Info` map as soon as it begins parsing the function. The value of the `functionInfo` entry will be `null` until the entire function definition is parsed. If a call to the function occurs before the entire definition is parsed, this indicates some form of recursion. Because Catalyst has not finished parsing the callee's definition, Catalyst cannot infer anything about the object returned from the call (as it does in Section 4.6.4). If the callee returns a mutable object, Catalyst will recognize that the callee's `functionInfo` is not yet available and will create a dummy return object. If the call returns an owning reference, this dummy object can have a `Liveness` value of 'true', but if the call returns a borrow reference, the object will have a `Liveness` value of 'false'. This is Catalyst's default approach to function calls, and it is used when patterns arise that do not allow for the optimizations described in Section 4.6.4.

4.6.2   Return Information

Section 4.4 discussed the `Objects` and `Variables` attributes of the `FunctionMaps` class; now we will discuss the `Returns` attribute. As shown in Figure 4.4, the `Returns`

```
1  class ReturnInfo {
2      Optional<String> StructName;
3      String Ownership;
4      PathToArg ReturnArg;
5      HashMap<Long, MembersToArgs> MemberMap;
6  }
```

Figure 4.10: `ReturnInfo` **class in Catalyst**

attribute is an instance of the `ReturnInfo` class. Figure 4.6.2 contains a pseudocode definition of this class.

The `StructName` attribute of the `ReturnInfo` class contains a value if the function returns a mutable struct, and is empty otherwise. The `Ownership` attribute contains the ownership type of the returned value. The `PathToArg` and `MemberMap` attributes contain information relating the returned value and its members to the function's arguments; Section 4.6.4 describes these attributes in more detail.

It is easiest to understand Catalyst's design choices for function calls with an example. Figure 4.6.2 contains a Vale program that uses some of the function call patterns optimized by Catalyst; this program is referenced frequently for the remainder of the chapter.

4.6.3   Arguments

The first thing Catalyst does when it begins parsing a function, is add the function's arguments to its `Variables` map (see Figure 4.4). To differentiate the arguments from other local variables, they are added to the map in the order they appear in the argument list with keys that begin at -1 and decrease monotonically (the unique identifiers in the AST for each local are always positive, so using negative keys for arguments will not interfere with the keys for locals). There are no entries yet in the `Objects` map, so Catalyst will add dummy objects to the map based on the

31

```
1   struct Engine{
2      size string;
3   }
4
5   struct Spaceship{
6      wingSpan int;
7      engineA &Engine;
8      engineB &Engine;
9   }
10
11  struct Fleet{
12     ship1 Spaceship;
13     ship2 Spaceship;
14     ship3 Spaceship;
15  }
16
17  fn duplicateShip3(f &Fleet) Spaceship {
18     newShip = Spaceship(f.ship3.wingSpan, f.ship3.engineA, f.ship3.
          engineB);
19     ret newShip;
20  }
21
22  fn main() int export {
23     e = Engine("large");
24     e2 = Engine("medium");
25     s1 = Spaceship(300, &e, &e2);
26     s2 = Spaceship(200, &e, &e2);
27     s3 = Spaceship(100, &e2, &e2);
28     f = Fleet(s1, s2, s3);
29
30     dupShip = duplicateShip3(&f);
31
32     println(dupShip.engineA.size);
33  }
```

Figure 4.11: Vale program to illustrate Catalyst's design choices for function calls

argument type. If the argument is an owning reference, then ownership of the object is being moved to the function and the generated entry in the `Objects` map can have a `Liveness` value of 'true'. However, if the argument is a borrow reference then the function cannot infer anything about the scope of the object's owning reference and the generated entry in the `Objects` map will have a `Liveness` value of 'false'. This inability to guarantee that objects passed to functions via borrow references are alive, limits Catalysts optimization abilities, but could potentially be avoided using methods described in Chapter 6.

Populating a function's `Objects` map for arguments is the exception to the parent rule mentioned in Section 4.5. This rule states that only objects owned by other objects have a `Parent` entry in their `LivenessInfo` instance. The reason for this is that a borrow reference can be a member of multiple objects, but an owning reference can only be a member of one object at a time. Therefore when a borrow reference to an object is made a member of another object, the `Parent` value of its referend is not touched.

Objects associated with arguments are the exception to this rule because all references within an argument's object generate a new entry in the `Objects` map. The function can only access these objects through the parent argument object, and therefore can only generate borrow references to them. Borrow references will never allow ownership to be moved, so the parent of these objects can never change in the function's scope. Therefore, it is safe to give all objects generated as members of an argument `Parent` values, regardless of whether the parent object owns the member. This parent information is crucial for constructing returned objects in the caller's scope as described in Section 4.6.4.3.

Figure 4.6.3 illustrates the state of the `Objects` map for `duplicateShip3` from Figure 4.6.2 directly before the return expression. At the start of the function, two dummy

Key	Liveness	Members	Parent
0	'false'	[empty]	2
1	'false'	[empty]	2
2	'false'	[empty, 0, 1]	9
3	'false'	[empty]	5
4	'false'	[empty]	5
5	'false'	[empty, 3, 4]	9
6	'false'	[empty]	8
7	'false'	[empty]	8
8	'false'	[empty, 6, 7]	9
9	'false'	[2, 5, 8]	empty
10	'true'	[empty, 6, 7]	empty

Argument 0 ⟶ (points to key 9)

Figure 4.12: State of the `Objects` map for the `duplicateShip3` function in Figure 4.6.2 immediately prior to the return expression

`Engine` objects (objects 0 and 1) are generated as the members of the argument's (the `Fleet`'s) first `Spaceship` member. Once this `Spaceship`'s members are eagerly generated, Catalyst will add the first `Spaceship` object (object 2) to the `Objects` map. Catalyst then repeats this process for the other two `Spaceship`s in the `Fleet`, generating objects 3-8. Now there are entries in the `Objects` map for all of the argument's members, and an object can be generated as the argument's referend (object 9). Lastly, line 18 of Figure 4.6.2 generates a `Spaceship` object with a `Liveness` value of 'true' corresponding to the `newShip` local generated within `duplicateShip3`'s scope.

The `Variables` map for `duplicateShip3` at this point (directly before the return expression) will resemble Figure 4.6.3. There is a single entry with a negative key referring to the `Fleet` passed as an argument, and an entry with a non-negative key

Key	Object	Ownership	Variability
-1	9	'Borrow'	'Final'
0	10	'Own'	'Final'

Figure 4.13: State of the `Variables` map for the `duplicateShip3` function in Figure 4.6.2 immediately prior to the return expression

referring to the local labeled `newShip` [2]. Objects 0-9 in Figure 4.6.3 can be accessed through the variable with a key of -1 corresponding to `duplicateShip3`'s first (and only) argument because of the member hierarchy in the `Objects` table (see Section 4.5).

4.6.4   Relating Returned Objects to Argument Objects

The argument variables and objects described in Section 4.6.3 paired with the object hierarchy described in Section 4.5 allow Catalyst to determine if an object in the return value of a function was passed as an argument or descendant of an argument (i.e., member of an argument, member of a member of an argument, etc.). Catalyst uses the `ReturnInfo` class from Figure 4.6.2 to store this information for each function.

4.6.4.1   The `ReturnArg` Attribute

The `ReturnArg` attribute of the `ReturnInfo` class is an instance of the `PathToArg` class. This class contains an argument index of the argument that the returned object is descended from (if it is a descendant of an argument) and a list of member

---

[2]The actual key for the local in Figure 4.6.3 is extracted from the AST and may not equal 0, but will be a non-negative integer

Key	Members	MembersToArgs
-1	[empty, 6, 7]	[{argIdx: empty, path: []}, {argIdx: 0, path: [2, 1]}, {argIdx:0, path[2, 2]}]
6	[empty]	[{argIdx: 0, path: []}]
7	[empty]	[{argIdx: 0, path: []}]
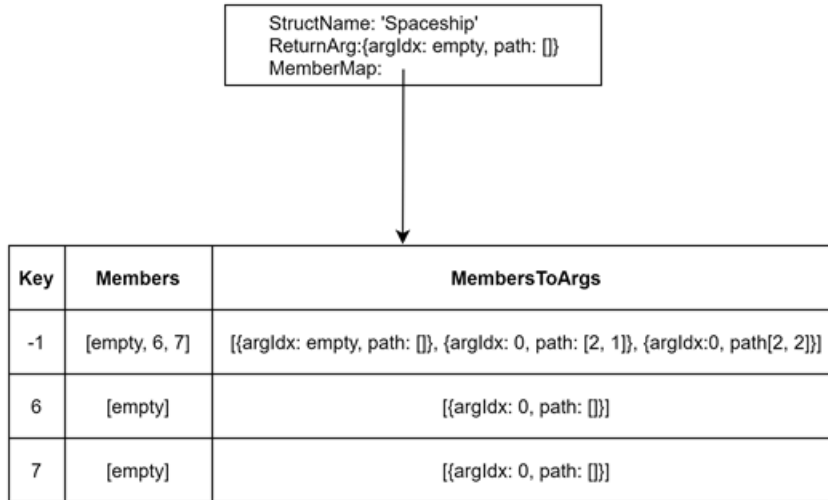
Figure 4.14: Diagram of the `ReturnInfo` instance associated with the `duplicateShip3` function from the program in Figure 4.6.2

indexes that indicate how to access the object from the argument's object (empty if the returned object is not a descendant of an argument).

When Catalyst parses the return expression of the `duplicateShip3` function from Figure 4.6.2, it will populate the function's `ReturnInfo` instance (see Figure 4.6.2) as shown in Figure 4.6.4.1.

The `ReturnArg` attribute in Figure 4.6.4.1 indicates that the object returned by `duplicateShip3` is not a descendant of an argument. However, the returned object does contain members that are descended from arguments, and this must also be indicated in the function's `ReturnInfo`. For this member information, Catalyst uses the `MemberMap` attribute of the `ReturnInfo` class.

4.6.4.2   The `MemberMap` Attribute

The `MemberMap` attribute of Catalyst's `ReturnInfo` class describes the relationship between the members of a function's returned value and the function's arguments.

36

```
1  class MembersToArgs {
2      StructMember[] Members;
3      PathToArg[] MembersAsArgs;
4  }
```

Figure 4.15: `MembersToArgs` class in Catalyst

It maps members of the returned object to `PathToArg` instances that indicate how to access the member objects from the function's arguments. Figure 4.6.4.2 shows pseudocode for the `MembersToArgs` class used in this map.

Figure 4.6.4.1 displays the `MemberMap` for the `duplicateShip3` function in the example from Figure 4.6.2. To populate this map, Catalyst first adds an entry with a key of -1, corresponding to the returned object itself. Catalyst looks up the returned object in the function's `Objects` map (using the `ReturnArg` attribute if necessary) and extracts the `Members` array associated with the object. This tells Catalyst how many members the returned object has, and where they exist in the function's `Objects` map (if they are mutable objects).

The returned object for `duplicateShip3` is object 10 in Figure 4.6.3 which contains two members with values in the `Objects` map; objects 6 and 7. Catalyst follows the `Parent` value (if it exists) for each of these entries and checks to see if any arguments point to this object. Object 8 is the parent of both object 6 and 7, but the only argument to `duplicateShip3` points to object 9, so Catalyst continues up the object hierarchy in search of an argument object. The parent of object 8 is object 9, which is referred to by the variable with key -1 (see Figure 4.6.3). This indicates to Catalyst that the argument at index 0 points to object 9, a parent of objects 6 and 7. Therefore, the `argIdx` attributes of the `PathToArg` instances for each member is set to 0.

Each time Catalyst moves up the object hierarchy in search of an argument object, it also tracks the index of the member in the parent object. This allows Catalyst to

produce the `path` attribute for each member. Objects 6 and 7 both descend from the argument object's (object 9's) member at index 2, so this is the first value in both paths (corresponds to object 8 in Figure 4.6.3). The second value in each path differs, however, because object 6 appears at index 1 in object 8's `Members` while object 7 appears at index 2.

Catalyst then recursively repeats this process for each descendant of the returned object with an entry in the `Objects` map, reusing their object keys as unique keys into the `MemberMap`. In the example from Figure 4.6.2, both members of `duplicateShip3`'s returned object are `Engine` instances. They have only one immutable member and no members that appear in the `Objects` map. Therefore Catalyst will give them each an entry in the `MemberMap` with an empty `PathToArg` instance and halt recursion. This gives us the final state of `duplicateShip3`'s `MemberMap` as displayed in Figure 4.6.4.1.

4.6.4.3   Returned Objects in the Caller's Scope

When a function call returns a reference to an object, the caller must have an entry in its `Objects` map for the reference to point to. The naive approach is to generate objects with `Liveness` values of 'true' for all owning references, and objects with `Liveness` values of 'false' for all borrow references (as mentioned in Section 4.6.1), but Catalyst can do better than that. Catalyst uses the `ReturnInfo` struct associated with the callee to construct an object that reuses the caller's object data for any references in the return value that refer to objects owned by the caller and passed as arguments. This allows some borrow references in return values to reference objects with `Liveness` values of 'true' in the caller's scope.

38

Key	Liveness	Members	Parent
0	'true'	[empty]	empty
1	'true'	[empty]	empty
2	'true'	[empty, 0, 1]	5
3	'true'	[empty, 0, 1]	5
4	'true'	[empty, 1, 1]	5
5	'true'	[2, 3, 4]	empty
6	'true'	[empty, 1, 1]	empty

Argument 0 ⟶ (points to row 5)

Figure 4.16: State of the `Objects` map associated with the `main` function in Figure 4.6.2 after the call to `duplicateShip3`

In the example from Figure 4.6.2, Catalyst analyzes the call to `duplicateShip3` by first looking up the function's `ReturnInfo` instance (Figure 4.6.4.1) in the `functionInfo` map (see Figure 4.6.1). The AST tells Catalyst that `duplicateShip3` returns an owning reference to a `Spaceship` object, and the `ReturnArg` attribute of the `ReturnInfo` instance for `duplicateShip3` tells Catalyst that the object was not passed as an argument. Therefore, a new `Spaceship` object is generated in the `Objects` map of `main` with a `Liveness` value of 'true'.

This new object is displayed in Figure 4.6.4.3 as object 6. Now we must examine how the `Members` attribute of this object was filled. Catalyst begins by searching the `MemberMap` associated with `duplicateShip3` (see Figure 4.6.4.1) starting with the value at -1. The first member at this value is empty, and so it will be in the caller's newly generated `Spaceship` object. The next two members contain values so Catalyst looks to the `MembersToArgs` attribute to determine if the `Objects` map for `main` already contains entries for these members. The `PathToArg` instances tell Catalyst that both of these members are descendants of `duplicateShip3`'s argument

at index 0 (object 5 in Figure 4.6.4.3). Catalyst then follows the path for each member through `main`'s `Objects` map. Index 2 of object 5's members leads to object 4, and both indexes 1 and 2 of object 4's members lead to object 1. Therefore, the second and third members of `main`'s new object point to object 1.

That is a lot of work just to reuse some of the caller's object information, but it eliminates generation checks that would otherwise be executed. If Catalyst took the naive approach to function calls, the dereference on line 32 of Figure 4.6.2 would require a generation check. `engineA` is a borrow reference, so it would point to a new object with a 'false' `Liveness` value. However, because Catalyst relates returned objects to arguments, it can conclude that `engineA` in `duplicateShip3`'s return value refers to an object that was passed as an argument to `duplicateShip3`, and originated in `main`'s scope. This object has a key of 1 and a `Liveness` value of 'true' as seen in Figure 4.6.4.3, so the generation check on line 32 is successfully eliminated.

4.7   Changing `Liveness` Values

The schema described in 4.6 allows Catalyst to maintain the caller's `Liveness` values for some objects even after references to these objects have been passed to function calls. However, the `Liveness` values of the caller's objects will not always be 'true'. There are two types of references that, when passed as arguments to a function call, cause their referend's `Liveness` value to become 'false': owning references that are not returned, and varying references.

4.7.1 Owning References

If an owning reference is passed as an argument to a function, ownership is moved to the callee's scope. Catalyst will use the callee's `ReturnInfo` instance to determine if the argument appears anywhere in the return value. If the argument's object *is* present in the return value Catalyst can determine that ownership of the object is moved back into the caller's scope after the call. This allows Catalyst to maintain a `Liveness` value of 'true' for the argument's referend in the caller's scope.

However, if the entire object owned by an argument does not appear in the return value, Catalyst will change the object's `Liveness` value to 'false' in the caller's scope. Catalyst will also recursively set all `Liveness` values of all the object's owned members to 'false'. When the callee receives ownership of an object and does not hand it back to the caller, it is likely that the callee will drop the object, freeing its memory for reuse. Subsequent use of borrow references (the owning references are no longer valid to the caller) to this object or any of its owned objects in the caller's scope should require generation checks.

Owning references within borrowed arguments do not change their referend's `Liveness` value. The callee can only access these objects via an argument, which will generate a borrow reference. An object cannot be dropped via a borrow reference, so these references cannot affect the `Liveness` of their referend.

4.7.2 Varying References

If a varying owning reference is passed to a function call as an argument, Catalyst will always change the `Liveness` value of its referend to 'false'. Even if the reference is present in the function's return value, the callee could have pointed the reference to

a new object, freeing and invalidating the memory of the old object. Subsequent use of references to the old object in the caller's scope should require generation checks.

As with final owning references, if a varying owning reference is nested in an argument, its referend's `Liveness` value will not change. The callee will only be able to access the referend via borrow references which can be pointed to new objects, but will not destroy the old object.

If a varying borrow reference is present *anywhere* in an argument, then a new object with a 'false' `Liveness` value will be created in the caller's scope for the reference to point to. The reference does not own the object it originally points to, so this object cannot be dropped by the callee, and should still be guaranteed to be alive in the caller. However, the callee could point the varying borrow reference to a new object from the callee's scope, so the reference gets a new dummy object as a referend in the caller's scope that is not guaranteed to be alive.

4.8 Conditionals

Conditionals like if-statements and while-loops limit Catalyst's optimization abilities. In most situations it is impossible to determine from the AST how a conditional will execute at runtime. In Vale's AST, conditional expressions determine which of one or more blocks will be executed at runtime. Catalyst is cautious when it comes to conditionals, and parses all blocks that could possibly be executed. If any block drops an object or contains a pattern from 4.7 that eliminates the guarantee that an object is alive, this change will be reflected in the state of Catalyst. If none of the blocks eliminate the guarantee that an object is alive, then the object's state in Catalyst will not be affected by the conditional.

Vale also errs on the side of caution when return expressions are nested in conditionals. If a function has more than one return expression, Catalyst will not attempt to relate any of the returned objects to arguments. Catalyst cannot determine which return expression will be executed, and the origin of the objects in each may differ. When a call to a function like this returns, Catalyst will create all new objects in the caller's scope for the references in the return value to point to (the naive approach mentioned in 4.6.4.3). Section 6.4 describes how Catalyst could potentially infer more about conditionals.

Chapter 5

VALIDATION AND RESULTS

There are three ways that Catalyst can potentially error: it can produce false negatives, produce false positives, or crash completely. A false negative occurs if Catalyst does *not* change a `knownLive` value in the abstract syntax tree (AST) to 'true' in a situation that matches the supported patterns. A false positive occurs if Catalyst *does* change a `knownLive` value in the AST to 'true' when it should not have done so. Crashes halt the Vale compiler completely.

This chapter discusses some of the tests that Catalyst uses to prevent these errors, then presents some results from a benchmark program.

5.1 Crashes

Catalyst's first line of defense against errors is Java `assert` statements. These statements halt the program if certain conditions are not met during execution; they are useful for debugging and verifying that smaller components of the program function as expected. Most `assert` statements in Catalyst appear in the methods of the `FunctionMaps` class from Figure 4.4, and verify that the `Objects`, `Variables`, and `ReturnInfo` attributes are populated as expected.

One example of an `assert` statement in Catalyst verifies that the `Objects` map is eagerly populated. Each time a new entry is added to the `Objects` map that contains members which are also mutable objects (meaning they too have entries in the `Objects` map), the `assert` statement will be triggered. It verifies that the

44

AST expression for each mutable member returns a valid key into the `Objects` map. If a member expression returns an empty `Optional<Long>` or a value that is not a valid key into the `Objects` map, Catalyst will halt. It is useful to halt here for debugging purposes. Without this assertion, the bug may not cause issues until an AST node attempts to access the member that lacks an entry in the `Objects` map. Additionally, if the Vale program never accessed the member, the bug could go completely unnoticed.

Other `assert` statements verify that all `Variables` point to valid entries in the `Objects` map, that indexes in `PathToArg` instances are valid in the caller and callee's `Objects` maps, that all objects derived from `PathToArg` instances match the type of object defined by a function's return value, and many other small but crucial features of Catalyst.

No `assert` statements are triggered for any of Catlayst's test cases or benchmark programs. While this does not guarantee that Catalyst will **never** crash under any circumstances, it is a good sign.

5.2 False Negatives

To protect against false negatives, Catalyst uses integration tests. Catalyst has a test suite of small Vale programs and a script that generates an AST for each program. Additionally Catalyst has a `TestCatalyst` class that will execute Catalyst on each AST and produce a modified AST. `TestCatalyst` then traverses these modified ASTs in search of specific nodes that should have predictable `knownLive` values.

```
1  struct Spaceship {
2    fuel int;
3  }
4
5  fn main() int export {
6    s = Spaceship(10); // owning reference
7    b = &s; // borrow reference
8
9    ret b.fuel; // generation check on b's referend (the Spaceship)
10 }
```

Figure 5.1: Catalyst test program with a single scope

5.2.1   Programs With a Single Scope

Figure 5.2.1 (duplicated from Figure 4.4) is is the first of Catalyst's test programs. For the reasons given in Section 4.4, Catalyst should successfully eliminate the need for a generation check on line 9 of this program. `TestCatalyst` verifies that the node in the modified AST (produced by Catalyst) that dereferences `b` and accesses the `fuel` member has a `knownLive` value of 'true'.

Figure 5.2.1 (duplicated from Figure 4.5) is another test case used to verify that the object hierarchy in Catalyst is working properly. `TestCatalyst` must verify two nodes of the modified AST for this program, both caused by line 13. The first node dereferences the `borrowShip` reference and will have a `knownLive` value of 'true' because it follows a similar pattern as the program in Figure 5.2.1. The second node dereferences the `Spaceship` object's borrow reference corresponding to the `engine` member. This should also have a `knownLive` value of 'true' because the owning reference to the `Engine` object has not been dropped or moved out of scope. This test helps ensure that the object hierarchy is properly set up. Catalyst also has further test cases with deeper nesting of objects to reinforce the correctness of the object hierarchy.

```
1   struct Engine {
2      fuel int;
3   }
4   struct Spaceship {
5      engine &Engine;
6      numWings int;
7   }
8
9   fn main() export {
10     engine = Engine(10)
11     ship = Spaceship(&engine, 4);
12     borrowShip = &ship;
13     println(borrowShip.engine.fuel);
14  }
```

Figure 5.2: Catalyst test program to validate the member hierarchy

Additionally, Catalyst tests several variations of the programs in Figures 5.2.1 and 5.2.1 that place the dereferences within loops and conditional expressions. These ensure that Catalyst performs as expected when dereferences are nested in different types of AST nodes.

5.2.2   Programs With Multiple Scopes

Verifying that Catalyst correctly performs all the tasks described in Section 4.6 involves more complex testing programs. Figure 5.2.2 verifies two features of Catalyst mentioned in 4.6.

Firstly the program verifies that borrow references passed to function calls do not affect the state of their referend. On line 18, `getEngine` is passed a borrow reference to the `Spaceship` object created on line 15. On line 20, this same borrow reference is dereferenced. `TestCatalyst` verifies that in the modified AST, the node dereferencing `borrowS` to access `engine` has a `knownLive` value of 'true'.

The second feature that Figure 5.2.2 helps verify is that returned borrow references have the proper referend. The `getEngine` function returns a borrow reference on

47

```
1   struct Engine{
2      fuel int;
3   }
4
5   struct Spaceship{
6      engine Engine;
7   }
8
9   fn getEngine(s &Spaceship) &Engine {
10      ret s.engine;
11   }
12
13   fn main() int export {
14      e = Engine(10);
15      s = Spaceship(e);
16      borrowS = &s;
17
18      e2 = getEngine(borrowS);
19
20      e3 = borrowS.engine;
21      ret e2.fuel;
22   }
```

Figure 5.3: Catalyst test program with multiple scopes

line 10 (remember that in Vale, accessing an owning reference that is a member of
an object automatically generates a borrow reference). When `getEngine` is called
on line 18, the caller (`main`) must point the returned borrow reference to an object
in its scope. If the referend was created within the *callee's* scope, Catalyst would
generate a new object in the *caller's* scope with a `Liveness` value of 'false', because
the caller cannot know anything about the lifetime of an object created in another
scope (unless it is handed ownership). However, due to the methods described in 4.6,
Catalyst can determine that the object pointed to by the returned borrow reference,
was passed as the first member of `getEngine`'s first argument. Therefore, when
`getEngine` returns, Catalyst will create a new entry in the caller's `Variables` map,
pointing to the `Engine` object from the caller's scope that was passed as the first
member of the first argument. This object has a `Liveness` value of 'true' because its
owning reference (`e`) has not been dropped or moved out of scope. Line 21 of Figure
5.2.2 dereferences the returned borrow reference (`e2`). `TestCatalyst` verifies that the
AST node correlated to this dereference has a `knownLive` value of 'true'.

48

Figure 4.6.2 is another example of a Catalyst test case that further reinforces Catalyst's correct handling of function calls. For the reasons outlined in 4.6, Catalyst should successfully eliminate the need for the generation check in this program.

5.3 False Positives

False positives in Catalyst are more concerning than false negatives because they compromise the memory safety of Hybrid Generational Memory (HGM). If Catalyst causes a false *negative*, the only consequence is an unnecessary generation check at runtime. However, if Catalyst causes a false *positive*, the program may access corrupted memory at runtime causing undefined behavior. Catalyst uses integration tests as well as optional runtime checks to prevent false positives.

5.3.1 Integration Tests

Section 4.7 discusses patterns that cause Catalyst to no longer guarantee that an object is alive. Catalyst has integration tests for these patterns, and `TestCatalyst` double checks that `knownLive` values are 'false' in the AST when a generation check should be required.

Figure 5.3.1 is an example of an integration test to help verify that Catalyst's state is properly updated after a varying reference is passed to a function call. The `Spaceship` struct in this example contains a varying borrow reference, `activeEngine`, that initially points to the `Engine` owned by `left`. This `Engine` has a `Liveness` value of 'true' because its owning reference is still in scope. The call to `getActiveEngine` should not change the `Liveness` value of this object, but should change the object that `activeEngine` points to. Catalyst cannot determine what object `activeEngine`

```
1   struct Engine {
2     fuel int;
3   }
4   struct Spaceship {
5     leftEngine Engine;
6     rightEngine Engine;
7     activeEngine! &Engine;
8   }
9   fn getActiveEngine(ship &Spaceship) &Engine {
10    return ship.activeEngine;
11  }
12  fn main() export {
13    left = Engine(10);
14    right = Engine(20);
15    leftBorrow = &left;
16    s = Spaceship(left, right, leftBorrow);
17
18    active = getActiveEngine(&s);
19
20    println(leftBorrow.fuel);
21    println(s.activeEngine.fuel);
22  }
```

Figure 5.4: Catalyst test program to verify that varying references are properly handled through function calls

will point to after the call to `getActiveEngine` because it may have been re-pointed to an object that originated outside the scope of `main`. Therefore, as described in Section 4.7.2, Catalyst creates a new dummy object in `main`'s scope with a 'false' `Liveness` value. Figure 5.3.1 illustrates this change in the state of `main`'s objects. The solid blue components represent the state before the call to `getActiveEngine`, and the dashed green components represent updates to the state caused by the function call.

There are two potential generation checks in Figure 5.3.1's program, one caused by line 20, and one caused by line 21. `TestCatalyst` verifies that the modified AST for this program contains the correct `knownLive` values for both. The AST node that dereferences `leftBorrow` on line 20 should have a `knownLive` value of 'true'. This verifies that passing varying borrow references to calls does not affect the `Liveness` of their referend. The AST node that dereferences `activeEngine` on line 21 should have a `knownLive` value of 'false'. As shown in Figure 5.3.1, `activeEngine` should
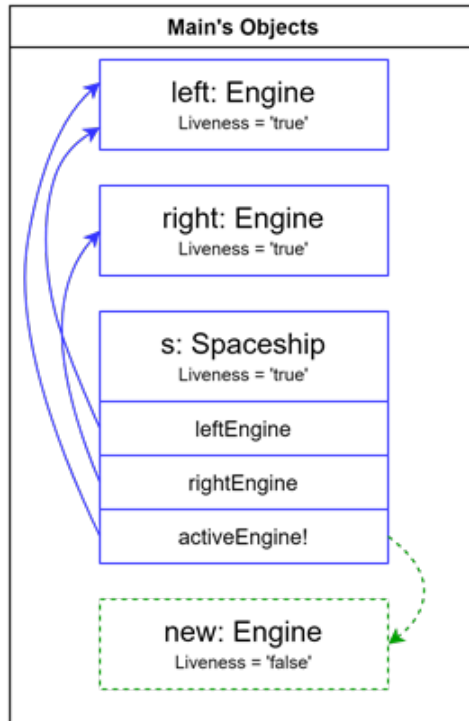
Figure 5.5: Illustration of `main`'s objects for the program in Figure 5.3.1

point to a dummy object after the call to `getActiveEngine`, and this object should have a 'false' `Liveness` value.

This is just one of many integration tests to prevent false positives. Others test different patterns mentioned in Section 4.7 like objects that are no longer guaranteed to be alive because their owning reference is passed to a call and not returned, or objects that are no longer guaranteed to be alive because their *varying* owning reference is passed to a function call. In these tests, `TestCatalyst` verifies that any attempt to dereference a borrow reference to the object after the function call requires a generation check.

5.3.2 Runtime Checks

To be extra safe when it comes to false positives, Vale has a feature that verifies any generation check that Catalyst says to skip. Compiling a Vale program with the `--override-known-live-true` flag verifies all of Catalyst's changes to the AST at runtime. With this flag, if Catalyst has changed a `knownLive` field in the AST to 'true', HGM will still perform the generation check at runtime. If the `knownLive` field in the AST is 'true', but the generation check fails (target generation of reference does not match generation number in memory), then the program will halt and report that Catalyst has created a false positive.

For all of Catalyst's integration tests and benchmark programs, no false positives are detected by this feature.

5.4 Results

At the time of this thesis Vale is in its development stages so there are few large programs written in the language. Catalyst is therefore only tested on one benchmark program, a small game written in Vale. Without Catalyst, the program required 426,256,014 generation checks at runtime. Catalyst was able to change 11 `knownLive` values in the program's AST, which amounted to 69,238 (about 0.016%) generation checks eliminated at runtime.

These results are not ideal, but also not entirely unexpected. Catalyst still has some large weaknesses in its static analysis. It currently covers some very specific patterns that are conducive to static analysis, but not common in all programs. Additionally, like most game code, the benchmark program uses arrays heavily, which are not yet

supported by Catalyst. The features described in Section 6.1 could have a significant impact on Catalyst's ability to optimize this benchmark program.

Catalyst does work for all of its test cases which are valid and potentially useful patterns, so other types of large programs might use these patterns more frequently and benefit more from Catalyst's static analysis. Additionally, in its current state, Catalyst is an excellent stepping stone for some potentially more lucrative static analysis techniques described in Chapter 6.

Chapter 6

FUTURE WORK

There is plenty more information in a Vale program's abstract syntax tree (AST) that Catalyst could potentially leverage to identify more unnecessary generation checks. This chapter discusses some of the weaknesses in Catalyst's static analysis and potential solutions.

6.1 Arrays

At the time of this thesis, Vale has not finalized the semantics of arrays. The structure of AST nodes for generating and maintaining arrays is still changing, so Catalyst currently disregards them completely and focuses on user-defined structs. However, when the AST structure is finalized, there is a simple solution that Catalyst could implement to eliminate some generation checks for arrays. There are two types of arrays in Catalyst: fixed size arrays, and unknown size arrays.

6.1.1 Fixed Size Arrays

Fixed size arrays will always have the same shape and size in memory, so it is tempting to treat the array as a struct where each member is an entry into the array. However, arrays are often dynamically accessed which means that, unlike accessing the member of a struct, the exact object being accessed is not known at compile time. For example, if an array named `arr` is accessed at index `i` (e.g., `arr[i]`), then even if the size of

`arr` is known, Catalyst would be unable to determine which object in the array is being accessed because `i` is a variable.

A better approach is to treat arrays as structs with a *single* member. Then any nodes that access an object in the array would evaluate to the member object. As long as the array does not contain varying owning references (see Section 2.3), it can then be read from or written to with no effect on the member's `Liveness` value in Catalyst. However, if a fixed size array of varying owning references is written to, then its member's `Liveness` value will be set to 'false'. The array could point one of its references to a new object, invalidating the memory of the old object. Use of borrow references to the old object should then require a generation check.

6.1.2 Unknown Size Arrays

Arrays with an unknown size at compile time can be handled almost the same as fixed size arrays except when they are re-sized. This operation drops the old array and creates a new one of the desired size. When this happens, Catalyst must change the `Liveness` value of the old array to false and create a new array object with a `Liveness` value of 'true'.

6.2 Arguments that are Always Alive

One of Catalyst's biggest weaknesses is that it does not eliminate generation checks (in the callee's scope) for any objects passed to functions via a borrow reference. This weakness could be mitigated with a feature to detect borrow reference arguments that evaluate to objects known to be alive in all instances. In other words, if a function takes a borrow reference as an argument and every call to the function in the program

hands it a borrow reference pointing to an object with a `Liveness` value of 'true', then the argument's object in the callee's scope can also have a `Liveness` value of 'true'.

To implement this feature, Catalyst would need to perform another pass of the AST. Currently Catalyst implements all of its optimizations in a single pass, which limits the information it knows when it begins analyzing a function. Catalyst analyzes most functions as soon as they are called for the first time (see Section 4.6.1), so it is impossible to determine anything about the subsequent calls to the function. However, with two passes, Catalyst could maintain a boolean for all borrow reference arguments that is 'true' until the argument evaluates to an object with a 'false' `Liveness` value in a call. Then in the second pass, if an argument's boolean is still 'true', the function can be re-analyzed and the object associated with the argument (in the callee's scope) will have a `Liveness` value of 'true'.

It is not uncommon for functions to be called just a few times with arguments known to be alive. Adding a feature to detect these arguments could eliminate many generation checks within functions of this nature.

6.3   Arguments and Scope Tethering

Another feature that could help Catalyst guarantee argument objects are alive is scope tethering, the practice of tying an object's lifetime to a scope that does not own the object. This feature cannot be implemented by Catalyst alone, it requires some setup from other parts of Hybrid Generational Memory (HGM).

6.3.1   Scope Tethering in HGM

HGM was designed with scope tethering in mind. In addition to a generation number (see Section 1.1.1), each object stores an extra bit that contains a 1 if the object is scope tethered, and a 0 otherwise. All local variables pointing to mutable objects also have a scope tether bit. When an object is initialized, its scope tether bit contains a 0, and when a variable is initialized, its scope tether bit equals that of its referend.

A function can then request (through the AST) that an argument be tethered to its scope. If a function call made this request for an argument, HGM would set the value of the argument object's scope tether bit to 1, and store the old value in the reference's scope tether bit. Then, when the function returns, HGM will revert the object's scope tether value, again storing the old value with the reference.

If `drop` is called on an object whose scope tether bit is set to 1, the object will not be freed until its scope tether bit reverts to 0. The scope tether is like a reference count, but it keeps track of scopes that have access to an object rather than references. Scope tethering is also more efficient than reference counting because it maintains a boolean which requires fewer instructions per update (load/store as opposed to load/add/store), and because it requires fewer updates (creating/destroying scopes is less common than creating/destroying references). Additionally, scope tethering could allow Catalyst to eliminate more generation checks, speeding up programs.

6.3.2   Scope Tethering in Catalyst

In the AST, borrowed arguments have a boolean `keepAlive` field. This field is 'true' if the function requests that the argument be alive, and 'false' otherwise. It is not always necessary to request that an argument stay alive, if the function does not

dereference the argument for example, then there is no reason to scope tether it. Catalyst could detect an argument like this and change its `keepAlive` value to 'false'.

Borrowed arguments that point to an object in the caller's scope with a `Liveness` value of 'true', and have a `keepAlive` value of 'true', will be scope tethered at runtime. This means that the referend of the object will be alive for the duration of the function, and Catalyst can point the argument to an entry in the `Objects` map with a `Liveness` value of 'true'. Scope tethering, along with the feature described in Section 6.2, could eliminate many generation checks caused by Catalyst's current inability to guarantee that the borrowed arguments are alive.

6.4   Return Expressions in Conditional Blocks

Another weakness of Catalyst is its handling of conditional expressions. For most conditionals, Catalyst does all that it can, parsing each block that could potentially execute, and if any block creates or destroys an object, reflecting this change in the state of Catalyst. However, Catalyst could do more for return expressions within conditional blocks.

Currently, if a function has multiple return expressions, Catalyst aborts any attempt to relate returned objects to function arguments (as described in Section 4.6.4), but this may not be entirely necessary. If all possible return expressions return the same object passed as an argument, then it is safe to say that all calls to the function will return the object in the caller's scope corresponding to the argument. Catalyst could then use this information to populate the function's `ReturnInfo` as described in Section 4.6.4.

It is likely though that different return expressions in a function will return different objects. If only one of the expressions returns an object passed as an argument, then Catalyst cannot conclude that the returned object is related to that argument. The outcome of the conditional is unknown, so Catalyst must still use its naive approach to return values (see Section 4.6.4.3).

6.5 Recursive Calls

Recursion is another pattern that causes Catalyst to abort attempts to relate returned objects to arguments. As described in Section 4.6.1, Catalyst currently uses the naive approach to return values (see Section 4.6.4.3) for recursive calls, but this could change in the future. Catalyst could delay its attempts to relate returned objects to arguments for recursive calls. This would be easiest with a second pass of the AST. Catalyst could use the naive approach on the first pass, and save the recursive function's `functionInfo` entry. Then, it can use the function's `ReturnInfo` in the second pass, potentially eliminating more generation checks in both the recursive function and its caller using the methods described in Section 4.6.4.

Chapter 7

CONCLUSION

The goal of this thesis was to improve the runtime of Vale programs by adding a static analysis stage to Vale's compiler. Vale uses a unique memory management model called Hybrid Generational Memory (HGM). HGM uses a combination of single ownership and generational references to ensure memory safety in Vale programs. Like most frameworks for automatic memory management, HGM incurs significant overhead at runtime. Vale uses a static analysis program called Catalyst to reduce this runtime overhead. Catalyst accomplishes this by marking in the abstract syntax tree (AST) where unnecessary generation checks can be skipped.

Currently Catalyst eliminates generation checks for objects whose owning reference is in scope. Catalyst can also track references through function calls to maintain information on their owning references.

Catalyst proved effective for its test suite of small Vale programs, and was able to eliminate some generation checks in a larger benchmark program. In its current state, Catalyst was unable to remove enough generation checks in the benchmark program to have a significant effect on the program's runtime, but this could change with future improvements to Catalyst. As discussed in Chapter 6, there are some powerful static analysis features that could easily be built on top of Catalyst's current framework.

BIBLIOGRAPHY

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and experience*, 19(2):171–183, 1989.

[2] S. Cass. The top programming languages: Our latest rankings put python on top-again-[careers]. *IEEE Spectrum*, 57(8):22–22, 2020.

[3] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 313–326, 2005.

[4] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[5] M. Olsson. Smart pointers. In *C++ 20 Quick Syntax Reference*, pages 169–172. Springer, 2020.

[6] W. v. Oortmerssen. Memory management in lobster.

[7] E. Ovadia. The vale programming language. https://vale.dev/, 2020.

[8] E. Ovadia and T. Watkins. Vale's hybrid-generational memory. https://vale.dev/blog/hybrid-generational-memory, Jan 2021.

[9] Y. G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189, 1991.

[10] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[11] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking off the gloves with reference counting immix. *ACM SIGPLAN Notices*, 48(10):93–110, 2013.

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

[13] B. G. Zorn. Comparative performance evaluation of garbage collection algorithms. Technical report, California Univ. Berkeley Computer Science Div., 1989.