GoA: Actors with Locally Managed Memory for Go

by

Daniel Caccamo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Reasoning about concurrent programs and the way they manage memory can be difficult. Single-process programs can allocate memory without concern regarding data races or memory corruption, but multi-threaded programs must have a system in place to ensure safe memory allocation. Typically, threads and processes use a system of locks or mutexes that are explicitly managed by the user. These locks allow safe access to shared data. Languages that use Actors as a concurrency construct attempt to solve the problem without explicit locks. Actors use a system of message passing to ensure data is being shared correctly among processes. However, this message passing system requires all data to be shared by copying the data, not by reference. If references to data are to be shared safely, another safety mechanism is needed. This thesis discusses a way to bring the actor paradigm to an existing highly concurrent language, Go. The project, named GoA, is an actor-based library for Go. GoA provides actor-local memory management, and a custom system to ensure data is shared safely among actors.

GoA comes with a custom memory-management library that replaces all of Go's existing allocation, garbage collection, and message passing techniques. These new methods derive from the open source language Pony, a language with a memory management system called ORCA. Inspiration is drawn from ORCA to integrate similar techniques into GoA. The custom memory manager aims to alleviate the overhead of Go's global garbage collection and allow actors to manage themselves so they do not slow down or interrupt other working actors.

A memory safety system is also introduced that provides a way for all memory usage across all actors to remain safe from races and corruption. Using ideas from Pony, a capability system that annotates allocated objects with specific rules that apply when sharing data is constructed. This system allows for local objects and sharable objects. Local objects are not allowed to leave the scope of the owning actor. Shareable objects must be annotated with one of the following three capabilities: mutable, immutable, or opaque. Mutable data is free to be manipulated, immutable data can only be read, and opaque data can neither be read nor overwritten. Each of these capabilities serve their specific purposes and when declared on an object and used incorrectly, a runtime error is thrown to the user. A runtime checker system is used to check if every read and write on a variable is safe, and will handle any resulting errors

Experiments and results indicate the local memory manager successfully speeds up the overall performance of the language. The basic speed benchmarks indicate the new library is slower than Go at allocating small objects, but significantly faster for allocating

large objects. The N-Body garbage creation simulation showcases how GoA and its locally managed memory allows important actors to work without interference from other actors with large allocation needs, speeding up the effectiveness of the system.

Acknowledgements

I would like to thank my supervisor, Dr. Dietl, for his constant guidance and advice throughout this project. Without his support, this project may not have met conclusion. Thanks to Dr. Buhr for helping and contributing to the benchmarks, specifically the producer-consumer examples.

I would like to extend gratitude to my family for always supporting me in everything I do. A big thank you to my Fiance, Adrienne Smith, for always being there when I needed it most and encouraging me to get through the hard times.

Financial support and original inception for this project provided by Huawei.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Concurrency has always been a highly debated subject in programming. Over the years, there have been many different approaches taken to realize this concept. Some of the more primitive forms include spawning threads [12] or creating processes [27] to handle specific chunks of an algorithm (divide and conquer). These methods can be complicated and may require additional constructs to maintain the integrity of the program. These constructs may include locks/mutexes [18], semaphores [17], or monitors [21]. The Actor Model [10] uses "actors" as the primitive of concurrent computation, which can alleviate some of the user overhead by allowing the developer to maintain an organized class-like structure over the concurrent program. This model is utilized by several languages such as Erlang [3], Pony [5], and uC++ [11]. Java and Scala can also use the actor model with the Akka toolkit [1]. This project mainly draws inspiration from Pony, an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language [5] and Go, an open-source programming language that makes it easy to build simple, reliable, and efficient software [4]. Pony is a new language designed to bring memory and type safety into the actor model of concurrency. Google's language Go uses lightweight threads, called goroutines, as a model for concurrency.

While both goroutines and actors provide some form of concurrency, they do have differences. Structurally, the actor-model encapsulates fields and methods like a class. This approach means concurrent methods are actor-specific and must be declared that way. Only designated concurrent methods may be executed concurrently. An actor's fields have state, which is managed by whichever memory model is equipped with the language. When sharing information, Pony uses a system of per-actor queues called "mailboxes". When sending information to an actor, it is stored in these queues until it is processed by the receiving actor.

Go doesn't have to worry about the same type of state as Pony, as its concurrency model doesn't revolve around actors. However, the state of any objects being used concurrently is still of concern. Go allows any declared function to be executed concurrently. At any point, if the user requires concurrency, a routine just has to be executed with the keyword *go*. Go's built-in method for communication is through channels. A channel is like a mailbox; a channel is N to N, while a mailbox is N to 1. A channel is essentially a pipe that connects goroutines. If a goroutine has a reference to an existing channel, it can send and receive information through it.

Ideas from Pony are used to introduce GoA, which aims to bring the actor model to Go, with per-actor memory management, and safety guarantees for objects shared among actors. Per-actor memory provides greater concurrency efficiency as actors are never interrupted by a global garbage collector or a garbage collection sequence by another actor. When using the custom memory manager, actors are able to work free of complicated locking mechanisms. The memory manager developed for GoA takes inspiration from Pony's runtime system, ORCA (Ownership and Reference Counting based Garbage Collection in the Actor World). ORCA also integrates the use of Pony's deny-capabilities [15] which allow for type safety and memory guarantees. GoA provides its memory safety features through a similar capability system as Pony. The capabilities in GoA are enforced through a runtime mechanism that prevents unsafe sharing and mutating of objects. Each object allocation using the local memory manager gives the user the option of providing a capability statically as a variable annotation.

1.1 Motivation & Approach

The main motivation for this thesis is to improve the concurrency features that the Go language offers by adding support for actors with locally managed memory. Initial NBody benchmarks with a rudimentary implementation of the local memory manager already showed improvements to the concurrent execution speed. The design lacked proper allocation, message passing, and garbage collection, but it was enough to prove that locally managed actors improve performance over the stop-the-world garbage collector that comes with the native Go runtime. With these promising results, the implementation of the preliminary local memory manager is extended and completed to provide a way to develop cleaner programs that have fewer garbage collector interruptions, leading to faster performance.

When dealing with concurrency, memory safety becomes a concern. Many concurrent languages now have constructs in place to protect the user from making mistakes when

accessing shared data. Languages such as Pony use a capability system to annotate objects with specific rules that dictate how they can be shared among actors. GoA uses a similar system to ensure memory safety. Unlike Pony, GoA is not integrated into the Go compiler. This limitation means the capabilities are enforced at runtime instead of compile time.

The Go language is the basis for this project. The Proto-Actor framework [6] is utilized to help integrate actors into the system. Proto-Actor is an open-source framework that provides all the constructs necessary to use actors, and it is discussed in more detail in Section 2.6. There are a only few small modifications necessary to make the custom local memory library work with the Proto-Actor framework.

The local memory management system is primarily based off of another open-source language Pony. They provide similar ideas with their language that are a promising foundation. The capability system is a product of Pony as well. A subset of their system serves as the baseline for GoA. A tool is developed to facilitate the use of the capability system. This tool is necessary as integrating the system directly into the Go compiler proved to be too large of a scope for this project.

1.2 Contributions

This thesis makes the following contributions:

1. GoA, a completely remodeled memory management system for Go and a built-in object annotation system designed to ensure memory safety.
   (a) This new memory manager has been built to work in conjunction with actors. It provides entirely local memory management per actor. It comes complete with its own memory allocation mechanisms that runs independently from Go's built in system. GoA also has an actor-local garbage collector, that cleans freed memory and ensures any objects passing among actors are not collected prematurely. This is done through object reference counting.
   (b) An object annotation system to give GoA enhanced memory safety. This system is designed to ensure all allocated memory is safely shared or sent among actors without any fear of data races or data corruption. The annotations, called "capabilities", are provided by the user and enforced at runtime. These capability features are built into GoA, but are not handled without extra processing through the use of the AST Translate tool.

2. An AST Translate Tool to modify the given program to provide necessary features. This tool serves two purposes. The main purpose is to create functions for every unique object that allow the runtime to traverse into its fields when sending/receiving/marking/sweeping an object. The other is to inject the capability safety calls into the developed program whenever an object is read or written to. The latter portion of the tool is intended to be used only as a way to test the memory safety of a program before deployment, and the former should always be used to ensure peak performance.

GoA (Contribution 1) is conceptualized through the Pony programming language. As mentioned, the design of the memory management and capability system are based on the Pony language. The challenges addressed in this thesis arise from the engineering aspects involved in integrating these existing concepts into Go. GoA is engineered through extensive implementation and testing of these new concepts in a different platform. The largest challenges are a product of the fact that GoA cannot be integrated directly into the Go language. GoA acts as an external library, and the capability system is engineered to work at runtime with the AST Translate Tool (Contribution 2).

Figure 1.1 shows how the entire system interacts. A developer can normally construct a program using Go with the Proto-Actor extension. If the developer desires to utilize GoA, they need to link the localmm library. This step includes completing three tasks. First, adding the localmm extension to every actor in the program. Second, changing all object allocations within the actors to local by using the localmm library's *Alloc* function. Lastly, they must analyze their shared objects to assign each the correct capability. After the library has been successfully linked into the program, the AST Translate tool must be run. If preparing for release, the AST Translate Tool can be used to create special functions that aid the performance of tracing through objects when garbage collecting or sending/receiving. If wanting to evaluate the memory safety of the desired program, the AST Translate Tool should be used to inject the runtime capability checks.

1.3 Organization

The remainder of this thesis is organized as follows: Chapter 2 details some important background information as well as related works. This chapter covers the actor model, capabilities and ownership systems, the Pony, Rust, and Go Languages, existing data race detection techniques, and the open-source library proto-actor that is extremely helpful for this project. Chapter 3 introduces the local memory manager and its inner workings such

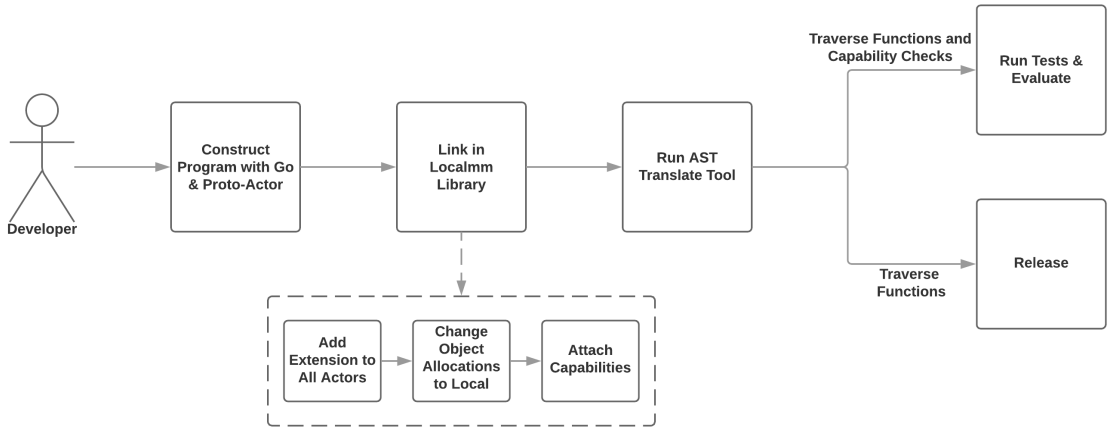


Figure 1.1: GoA System Diagram

as the memory design, memory allocation technique, garbage collection, message passing, reference counting, and how objects are traversed when marking, sweeping, sending and receiving. Chapter 4 discusses how some of the capabilities are implemented to allow for safe message passing among actors. The implemented capabilities, and how they are integrated into the system are the focus of this section. Chapter 6 shows the results of the GoA benchmarks. Specifically, the NBody test to demonstrate the advantages of the local garbage collector, and the Producer-Consumer benchmark to show how the runtime system compares to other concurrent languages. Chapter 7 covers potential future work for GoA including an enhanced capability system and potential compiler integration. The conclusions of this thesis are summarized in Chapter 8.

Chapter 2

Background and Related Works

2.1 Actor Model

The Actor Model [10] provides a developer with an easier way to design and develop concurrent and distributed systems. It revolves around the fact that "everything is an actor", similar to some object-oriented languages that abide by "everything is an object". This means even the entry point to a program is an Actor. Actor programs are built as class-like structures that contain special methods that can be run concurrently with other actors, but sequentially within an actor. These special methods are typically designated "behaviours" and can be declared alongside other normal non-concurrent methods.

Actors are designed to send and receive "messages" throughout their lifetime. A message is essentially a behaviour that is called upon an actor. In response to a message an actor receives it can do four things [28]:

- act on the received message
- send messages to other actors
- create new actors
- designate a new behaviour for the next message it receives

These messages are the primary method of communication among actors. This system is essential as it allows users to avoid having to deal with locks and threads. Messages are delivered into an actor's "mailbox", which is processed in a first-in first-out order. Since

```
class HelloActor extends Actor {
    def receive = {
        case "hello" =>
            println("Hello World!")
    }
}

object Main extends App {

    // create actor system
    val system = ActorSystem("HelloSystem")

    // create and start actor
    val helloActor = system.actorOf(Props[HelloActor],name = "helloactor")

    // send the actor a message
    helloActor ! "hello"
}
```

Figure 2.1: Scala with Akka Toolkit Actor Creation

an actor can change its own state, they are dealt with sequentially to preserve integrity. The actor handling multiple messages at the same time can cause potential data races. For this reason, messages are one of the defining characteristics of the actor model.

Some languages now use actors as their main form of implementing concurrency. Erlang was the first to use actors, and languages such as Scala and Java can use actors with the Akka toolkit. The new language Pony uses actors as well, and is a main inspiration for the local memory design. Proto-Actor, as mentioned in Section 2.6 is a framework available to bring the actor model into a few already existing non-actor languages. An example actor program written in Scala Akka can be seen in Figure 2.1. The example shows how an actor is structured like a class and defines a *receive* method to parse and act on any message it receives.

2.2 Go

Go [4] is an open-source programming language that makes it easy to build simple, reliable, and efficient software. It was first conceived by Google in 2007 and announced in 2009. It and has been continually updated with new features since. They set out to create a new language that improved on criticisms from other languages while still retaining their helpful features [25]. The language bis statically typed like C++ or Java, efficient, productive, easy to use, and supports networking and multiprocessing. Go comes with a simple syntactical style, even when dealing with its native concurrency features. Go uses goroutines and channels to facilitate concurrency (Section 2.2.1). However, when dealing with concurrency Go has no built-in notion of safe or verifiable concurrency, unlike Pony or Rust, and instead relies on concurrency control constructs such as locks.

2.2.1 Concurrency Model

Go starts a user-kernel thread in a routine for concurrency, providing a lightweight process. These processes are called goroutines. Any declared function can be run as a goroutine by simply affixing the keyword *go* before the call, which can be seen in Figure 2.2. Function *helloWorld* is called in two ways, synchronously and asynchronously. The latter is called by using the *go* keyword to spawn the function into a goroutine. Goroutines communicate through the use of channels. Objects can be sent through channels from one goroutine to another. By default, the channel buffer length is zero which can handle a single object. Any buffer length can be specified during channel creation. If an object is sent to a full buffer it blocks until a value is received. An extension of the previous code snippet with channels can be seen in Figure 2.3. Here, the *helloWorld* function is called asynchronously, and waits for communication through a channel. The main thread send a "hello world!" message through the channel and once received by the goroutine, it is printed to the console. Multiple goroutines can send and receive on the same channel. The actor model can be simulated in Go by assigning each goroutine a specific channel. This allows communication to a specific "actor" by sending message to its unique channel.

2.2.2 Memory Model

The key features of Go's memory model, as relating to this thesis, are the garbage collection techniques and the concurrent memory handling. Go has a global stop-the-world, concurrent, tri-color, mark-sweep collector. This approach means that when memory usage

```go
func helloWorld() {
    fmt.println("Hello World!")
}

func main() {
    // synchronous call
    helloWorld()
    // asynchronous call with goroutine
    go helloWorld()
}
```

Figure 2.2: Goroutine Execution

```go
var helloWorldChan = make(chan string)

func helloWorld() {
    msg := <- helloWorldChan
    fmt.Println(msg)
}

func main() {
    go helloWorld()
    helloWorldChan <- "hello world!"
}
```

Figure 2.3: Goroutine Execution with Channel Communication

across the entire system reaches a certain point, the garbage collector stops all running processes. This stop-the-world technique introduces overhead as some concurrent low memory processes may not actually need to be swept for memory.

Go does not have any built-in notion of data protection, like Pony with capabilities, and Rust with ownership. Instead, Go relies on the user to provide synchronization among goroutines. This includes all the usual tools such as atomic operations, locks, mutexes and semaphores.

2.3 Pony

Pony [5] is an open-source, object-oriented, actor-model, capabilities-secure [15], high-performance programming language. Pony provides many different guarantees and features such as [9]:

- Type Safe
- Memory Safe
- Exception Safe
- Data-Race Free
- Deadlock Free
- Native Code
- Compatible with C

These safety guarantees come through the extensively designed memory model and the deny capabilities they have implemented.

2.3.1 Concurrency Model

Pony uses the actor model to bring concurrency to their language. Pony actors are similar to a class but they can have concurrent methods declared as behaviours. These actors are extremely lightweight as they are executed by a thread pool. While actors provide an easy way to implement concurrency, the actors themselves operate sequentially. This means

```
actor newActor
  let name: String
  let env: Env

  new create(_name: String, _env: Env) =>
    name = _name
    env = _env

  be concurrent() =>
    env.print.out("hello" + name)

actor Main
  new create(env: Env) =>
    let newActor = newActor("actor")
    newActor.concurrent()
```

Figure 2.4: Pony Actor Creation

while actors can work concurrently to other actors, any methods called on an actor are run sequentially. Figure 2.4 shows how an actor is defined and created, as well as how its behaviours are run. If a method is defined with the keyword *be*, any call to it is queued in the actor's mailbox and run concurrently. This predefined concurrency is different than Go, where any method can be run concurrently.

2.3.2 Memory Model

Pony boasts a great deal of safety guarantees. These guarantees are achieved through their memory management and capability system ORCA [16, 13]. Actors are responsible for managing their own memory, which alleviates the overhead of stop-the-world garbage collection. When memory is shared to other actors, Pony's capability system is in charge of ensuring all data is handled responsibly and without corruption. This system is enforced through the use of causal message delivery. This means that the cause of a message is always enqueued in an actors mailbox before its effect [14]. This is an integral part of Pony's memory system, although it has been proposed that it can still function as intended without it [26]. The causal message handling requirement allows all reference counting to maintain integrity by handling them sequentially in the correct order. When an actor

begins garbage collection, it marks all objects the actor has a reference to and sweep all remaining objects. When sweeping, they also have access to a list of shared objects and their reference counts, so they know not to collect those objects. This is why keeping reference counts up-to-date is of the utmost importance.

The garbage collection obviously has to occur during runtime, but the capability system that enforces memory safety runs at compile time. Pony has a system in place where you add extra annotations to any declared object, which determines how it can be shared throughout the system. These capabilities include annotations such as:

- Val (immutable)
- Iso (isolated)
- Tag (opaque)
- Trn (transition)
- Box (wrapper for transitional objects)
- Ref (reference)

Val represents immutable data. It cannot be modified after it has been created, but can be read by any actor. Iso represents isolated data. Isolated means the object is safe to read and write to as it is guaranteed to be the only reference to the object. Tag is for objects that do not need to be modified or read, meaning the object can only be used for calling functions or sending messages. Trn is for when an actor needs to modify an object but also share read-only references to other actors. If presented with the previous scenario, when an actor shares a Trn object, the receiving actor sees a Box object. Box is essentially a read-only object where the original owner maintains the ability to modify the value. Finally, Ref objects are mutable objects that are treated as "normal" data. They can be written to or read from at any time but cannot be shared. Val, Iso, Tag, and the Trn/Box combo are all safe to share among actors. This is because the receiving actor has either an immutable object reference, an isolated reference (sending actor loses ownership), or a completely opaque object that can only receive messages. This ensures the data can be read/written to without worry of corruption. If an object is assigned Ref, the compile-time system is able to warn the user that the data is not safe to share. With these checks and the reference counting memory management system in place, Pony can guarantee that all memory is safely handled.

2.4 Rust

Rust[7] is another open-source programming language that offers its users absolute thread safety when developing concurrent programs. It is in development mainly by Mozilla, who began the project looking for a more secure programming language than what was typically available. It is a systems programming language that "runs blazingly fast, prevents segfaults, and guarantees thread safety" [23]. Rust comes equipped with many features such as [24]

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

2.4.1 Concurrency Model

Rust is very much designed to simplify concurrency and make its use more comfortable and safe to use. It uses a more primitive approach to concurrency than Pony. While Pony uses high level abstraction with actors, Rust sticks with threads. The creation and use of these threads can be seen in Figure 2.5. Rust attempts to mitigate the issues of using threads through the use of its unique memory system discussed in Section 2.4.2. It allows communication among the threads through the use of message passing. Much like Go, channels are used for sharing data among threads. Channels provide a safe and effective way of communication among concurrent processes. These channels also make sure to enforce certain rules the compiler has for sharing data. If certain data has not been declared sharable, the compiler is able to catch it and throw a static error during compilation.

```rust
fn main() {
    thread::spawn(|| {
            println!("hello from the spawned thread!");
    });

    println!("hello from the main thread!");
}
```

Figure 2.5: Rust Thread Spawning

```rust
fn main() {
    // share immutable reference
    shareRef(&obj);

    // share mutable reference
    shareRef(&mut obj);

    // share by value
    shareVal(obj);
}
```

Figure 2.6: Rust Object Sharing

2.4.2 Memory Model

While Rust provides similar memory safety guarantees as Pony, they work them into the language in a completely different way. Pony works through the use of their capability system, and Rust uses what they call "ownership"[22]. Every variable is scoped at compile time. Scoping means the lifetime of the variable is determined through its use, and when the variable falls out of use, the compiler knows the memory can be reclaimed. This technique is how Rust handles all memory within a program. It does not have any garbage collection mechanism like Pony or Go. When sharing objects between two threads, references can be used through the use of the & symbol. Figure 2.6 shows how objects can be shared three ways: by immutable reference, mutable reference, and by value. The Rust compiler is able to enforce the immutable and mutable rules at compile time.

2.5 Data Race Detection Techniques

Most of the related technologies discussed in this thesis for memory protection are built into the languages this thesis covers. The goal for GoA is to include a built in system as well, but the first step towards that is to include a runtime capability checker system which is able to determine if memory is handled safely. While GoA is designed to be easily integrated into the compiler eventually, there are other existing solutions for data race detection.

Go has a race detection tool available as an external resource [2]. It is primarily based on the Thread Sanitizer Algorithm they developed [8]. The tool is integrated with the go tool chain. To run the command all the user needs to do is pass a command-line flag while building or running:

```
go run -race raceTest.go
```

The tool works by reading the compiled code for all memory accesses and while the program is being executed, looking for unsynchronized accesses to shared variables. When an error is detected, a warning is printed to the user. The design of the tool allows for detection only when the executing code actually triggers a data race. This design is different than the proposed system, as GoA wants to catch all potential data races, regardless if they are executed upon or not.

2.6 Proto-Actor

Proto-Actor [6] is an open source cross-platform actor framework for .NET, Go, Java, and Kotlin. In Go, it builds off of the existing concurrency system, goroutines. After establishing the actor's properties such as mailbox, process ID, and local context, it spawns a goroutine that encapsulates the actor's main loop and yields the scheduler when not working. This loop endlessly iterates over its own mailbox looking for new messages. If a message is found, it executes the message, if not it yields the scheduler briefly for other actors and repeat the loop. The loop stops only when an actor stop message has been received, or the system itself is explicitly stopped.

As proto-actor is an external package, the usage within Go must use its existing constructs. There are no new keywords added so the actors and methods are built from structs. An actor declaration looks like the code in Figure 2.7.

```
type Actor struct {
        //fields
}
```

Figure 2.7: Proto-Actor Actor Definition

```
type actorMethod struct {
        //args
}
```

Figure 2.8: Proto-Actor Actor Method Definition

Any fields required for an actor can be declared as normal within the struct. Unlike other actor languages where behaviours and methods can be declared within a class-like format, they must also be declared as structs. An example method is shown in Figure 2.8:

Once again, all arguments that belong to the method are declared as the struct's fields. When a method is called on an actor, the struct is encapsulated as a message and sent to the actor's mailbox. For a struct to be an actor, it must implement the Receive function. The Receive function takes an actor context argument that contains the message to be executed. A switch statement is used to determine which message the actor should act on. The Receive function definition is shown in Figure 2.9.

The body of the behaviour is declared within the case statement. Only the concurrent methods need to be declared in the receive function. Normal synchronous functions can be declared with the usual Go syntax. Executing these behaviours first requires a few calls to the proto actor library to create the actor correctly. The creation of a proto-actor is

```
func (a *Actor) Receive(context actor.Context){
        switch msgType := context.Message().(type)
        case *actorMethod:
            //method body
}
```

Figure 2.9: Proto-Actor Actor Receive Function

```
func main() {
        props := actor.FromProducer(func() actor.Actor {
                return &helloActor{}
        })
        pid := actor.Spawn(props)
        pid.Tell(&actorMethod{/*args*/})
}
```

Figure 2.10: Proto-Actor Actor Creation

shown in Figure 2.10.

First, the actor props should be made. Props represents the configuration of how an actor should be created. Typically, this involves passing a creation function with the struct you wish to define as an actor. Next, the props are passed to the spawn function which handles the actor creation, and initiates the goroutine that iterates over the mailbox; effectively starting the actor. This function returns a unique id that allows behaviours to be run. With the id, the *Tell* function sends a message to the actor. The argument within the tell function represents the behaviour to be executed. After the actor has been started, the system can continue with other operations while it works in the background. The user should hold on to the PID to stop the actor before the program terminates. If the main program needs to wait for an actor to finish, the actor should be started with a "future". A future is a structure that waits for a return message from an actor method. Setting up an actor remains the same, but the Tell() method is replaced with a RequestFuture() method as shown in Figure 2.11.

```go
func main() {
        props := actor.FromProducer(func() actor.Actor {
                return &helloActor{}
        })
        pid := actor.Spawn(props)
        // Start method, with timeout
        f := pid.RequestFuture(&actorMethod{/*args*/}, timeout)

        // DO STUFF

        // Wait for method result
        result := f.Result()
}
```

Figure 2.11: Proto-Actor Futures

Chapter 3

Local Memory Manager

3.1 Introduction

The purpose for this memory manager is to allow all actors to manage their own memory operations, including allocations and garbage collection. That way, dealing with memory does not interfere with any other actor operations. This interference is especially prominent with regards to the garbage collector. Currently, Go has a global memory manager, which does not pair well with the actor paradigm. The Go garbage collector stops execution and pauses all actors. For applications with heavy memory usage, these pauses are an issue. With an actor-local memory system, the actor can garbage collect itself without interrupting other processes running in the program. However, when an actor is garbage collecting, it is not receiving any messages. In a heavily interconnected actor system, this can introduce some delay in actor response time.

Sections 3.1.1 and 3.1.2 discuss the changes made to Go and Proto-Actor to integrate the system. Section 3.2 covers the design of memory layout, specifically the data structures and how they interact. Section 3.3 explains how memory is allocated and freed. The garbage collection algorithms, mark and sweep, are discussed in detail in Section 3.4. In Section 3.5, the message passing and reference counting techniques are explained as well as how they integrate with the Proto-Actor changes discussed in Section 3.1.1. Section 3.6 details the methods used to dynamically traverse into objects where a *Traverse* function is not declared.

```
type NewActor struct {
        ctx localmm.Context
}
```

Figure 3.1: Localmm Actor Definition

3.1.1 Go Integration

The rudimentary memory management system was built to be contained in a single Go package. The package is defined as "localmm" (local memory manager). It has been designed to completely take over all memory operations, but can only be used when integrated with actors. This requirement is because the techniques used for garbage collection depend on message notifications from the actor's mailboxes after finishing work. For the manager to be integrated into an actor, the package must be included in each definition through a field representing the localmm Context class. Figure 3.1 is a modified version of the actor definition struct.

The context field can be in any order with other actor fields. With this field included within an actor, the user is only required to handle the allocations with a call to the library. The context object is equipped with an *Alloc* function to get memory from the system. The function accepts an instantiation of whichever object the user would like to allocate. The limitations of building a library as apposed to language support requires that the *Alloc* function can only return a pointer to a memory location. The user must then cast the function to the object passed into the call.

```
base := (*obj)(a.ctx.Alloc(obj{}))
```

In the example shown, *base* is allocated locally with type "obj". The allocation algorithm is further discussed in Section 3.3. All other memory management, including garbage collection and deallocations (Section 3.4), message passing (Section 3.5), and subsequent reference counting (Section 3.5), are all handled behind the scenes. The following sections explain these important processes that are involved in this memory manager as well as how they are implemented.

3.1.2 Proto-Actor Integration

The Proto-Actor framework is an important resource for the creation of GoA. However, there are some changes that must be made to fully integrate the local memory system. These changes mean that a new Proto-Actor library is provided with GoA, and the standard version will not work. There are two major changes that need to be dealt with. The first change deals with how an actor knows when to start garbage collection. In Pony, after an actor has processed a message in its entirety, it checks if the memory usage has exceeded a set threshold, and garbage collects accordingly. GoA has been designed to emulate the same methodology. This technique requires knowledge of message delivery, reception, and when it has finished processing. Proto-Actor allows users to define custom mailboxes when creating an actor. Custom mailboxes are required to meet a certain interface that defines individual actions to execute when dealing with messages. GoA uses this construct to auto-inject a custom mailbox that provides methods detailing when to start the garbage collection process. After a message has been delivered, it enters the custom messageProcessed function that is able to execute the localmm garbage collector.

The second change modifies the way messages are handled when they are sent and received. When the custom mailbox is being added, as discussed above, there needs to be extra steps added to the mailbox processing chain. Currently, messages are immediately executed upon reception, and nothing specific happens when sending. When sending and receiving, GoA needs to be able to track reference counts of objects. For incoming messages, it also needs to check for an actor start message to initialize the local memory manager. Proto-Actor allows for custom code to be injected into this processing chain through the use of "middleware". GoA creates custom middleware to add the necessary processing to both outgoing messages and incoming messages. The inbound and outbound middleware algorithms are shown in Algorithms 1 and 2. The exact details of the counting algorithm is covered in Section 3.5.

Most of these changes are handled automatically in the local memory library. The user is only required to add a single line of code to their actor creation. After creating the actor props, a new method added to the props structure, *WithExtension*, must be called with the *LocalHeapExtProducer* function from the localmm context object passed as an argument. This extension tells the spawn function to insert the custom mailbox discussed above. The code snippet in Figure 3.2 shows the required additional statement.

Algorithm 1: Inbound Middleware for Custom Mailbox

```
1  Procedure SetInboundMiddleware(actorCtx, localCtx)
2      switch actorCtx.Message() do
3          case actor.Start do
4              localCtx.Init()
5          end
6          case RcDec do
7              obj ←localCtx.getObj(msg)
8              obj.referenceCount ←obj.referenceCount - 1
9              if obj.referenceCount = 0 then
10                 delete(obj)
11             end
12         end
13         case RcInc do
14             obj ←localCtx.getObj(msg)
15             obj.referenceCount ←obj.referenceCount + 1
16         end
17         case Default do
18             receiveObjects(msg)
19         end
20     end
```

Algorithm 2: Outbound Middleware for Custom Mailbox

```
1  Procedure SetOutboundMiddleware(localCtx)
2      sendObjects(localCtx.Message())
```

```
func main() {
        helloActor := &helloActor{}
        props := actor.FromProducer(func() actor.Actor {
                return &helloActor{}
        })
        props.WithExtension(helloActor.ctx.LocalHeapExtProducer)
        pid := actor.Spawn(props)
        pid.Tell(&actorMethod{/*args*/})
}
```

Figure 3.2: Actor Creation with Local Heap Extension

3.2 Memory Design

The system has three tiers of memory management as shown in Figure 3.3. The first and largest is the Go managed memory. This memory is accessed only to allocate the next tier of memory, which is called a mempool. The mempools hold another memory construct, chunks. Chunks are where the actual objects are stored. The chunks are further divided up, depending on the size of the object being allocated. The mempools and chunks are allocated in a linked-list style pointing to the next available free structure. This design allows for quick recycling and reuse. The mempools and chunks are the main building blocks of the memory system. Sections 3.2.1, and 3.2.2 dive further into how each of these are defined. Section 3.2.3 covers how they come together with the rest of the package to create GoA's memory system. This section includes a more detailed memory diagram that shows how each memory structure integrates together.

3.2.1 Mempool

Mempools are the largest data structure used in the memory manager. They hold a large chunk of memory that the manager can divide into chunks as objects are allocated. The mempool structure is shown in Figure 3.4. When memory is grabbed from the Go global memory, the mempool mem pointer is assigned to point to the given memory location. Size stores the length of the memory, and start stores the beginning memory address. UsedCount is a simple counter to help garbage collection know when a mempool is still in use. As mentioned previously, mempool are allocated in a linked-list like structure. This

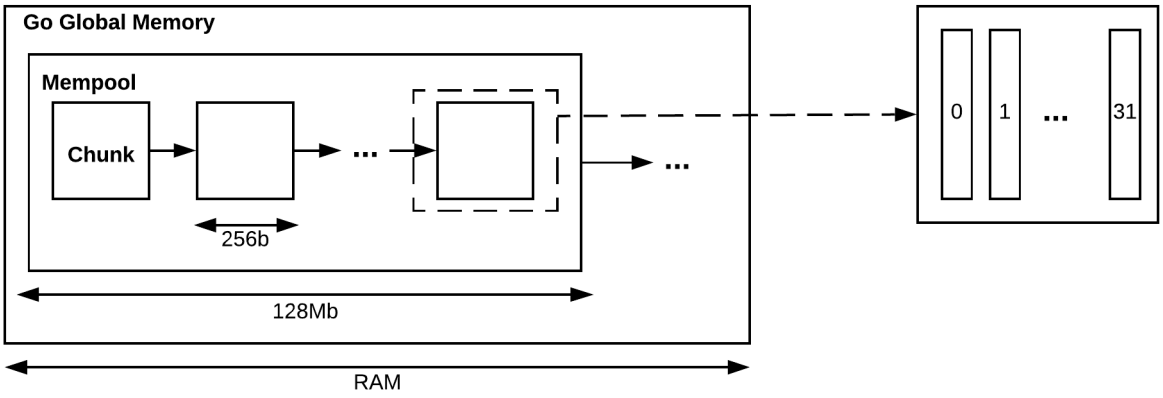


Figure 3.3: GoA Memory Layout

```
type mempool struct {
        mem         Pointer
        size        uintptr
        usedCount   uintptr
        start       uintptr
        next        *mempool
}
```

Figure 3.4: Mempool data structure

is realized through the next pointer. Next is only utilized if a mempool becomes full, and cannot be garbage collected yet. A new mempool is allocated and the old mempool is assigned to the next pointer until the garbage collector can deal with it.

Creating a mempool is a simple process. The first step the memory manger takes when allocating a variable is securing a portion of Go memory. This memory is taken in the form of a byte array and is cast to type mempool. The mempool structure itself is allocated within this memory. This mempool memory layout means a portion of the memory allocated can not be used for chunks as it is reserved for the mempool structure. GoA is responsible for protecting this information against being overwritten. The mempool information is kept within the first 256 bits of the mempool, and the remaining memory is calculated to reflect this. Chunks are allocated from the end of the mempool first, taking

```
type chunk struct {
        ownerPID   *actor.PID
        ownerHeap  *Heap
        mem        Pointer
        size       uintptr
        slots      uint32
        next       *chunk
}
```

Figure 3.5: Chunk Data Structure

the last 256 bits. As the allocated memory approaches the beginning of the mempool, the remaining free memory is tracked and when it hits zero, it is marked as full and a new mempool is created, repeating the process. As mentioned earlier, the old mempool is assigned to the next pointer of the new mempool until its held chunks have been garbage collected.

3.2.2 Chunk

Chunks are the smaller blocks of memory that fill mempools. Every object allocated through the local memory manger occupies some memory within a chunk. Large objects have their own chunk and small objects share chunks. The chunk structure is shown in Figure 3.5. Chunks have a reference to its owning actor, as well as the heap it belongs to. Heaps are an actor-local data structure responsible for holding chunks. They are discussed in more detail in Section 3.2.3. There is a mem pointer indicating the memory available and the slots field is a 32 bit bitmap that helps determine where objects can be allocated in the chunk. The next field performs the same function as in the mempool structure. As chunks are filled, they are placed in the next pointer until they can be reused or garbage collected.

How to allocate a new chunk is shown in Figure 3. First, GoA tries to obtain storage from the current free mempool. If the mempool is available, a chunk can be chosen by taking the next available memory location. The mempool size is then decreased by the size of a chunk and the used count is incremented. If the mempool is now completely full, it is moved into the full mempool linked-list in the global pool data structure, shown in Figure 3.9. If there is no existing mempool, a new one is created and the last slot is taken

```
var sizeclassEmpty = [5]uint32{
        0xFFFFFFFF,
        0x55555555,
        0x11111111,
        0x01010101,
        0x00010001,
}
```

Figure 3.6: Sizeclass Definition

for the new chunk.

Once the chunk has been created it needs to be partitioned according to the size of the incoming object. For small allocations, each chunk can hold a specific amount of objects depending on the new object's sizeclass. With a chunk size of 256 bits (32 bytes), there are 5 different sizeclasses. The purpose is to have chunks specifically designed for small objects of similar size. The size classes are defined for 8, 16, 32, 64, and 128 bit objects. Anything larger has its own chunk. The smallest possible slot in a chunk is 8 bits. The sizeclass is used in conjunction with the slots bitmap. The code in Figure 3.6 is how the sizeclasses are defined and Figure 3.7 shows how each chunk divides up memory according to the sizeclass. The 1s in the slot bitmap denote a free space for an object. In the first sizeclass of 8, every slot can be filled because 256 bits can fit 32 1 byte objects. In the second sizeclass of 16, every second slot can be filled. This pattern continues for the other sizes up to 128, which can only fit two objects in the 256 bit chunk. Equipped with these bitmaps and knowledge of which sizeclass, a chunk can determine how many free slots are left and where to put the object. It searches slots from the right and if a free spot is found, that index is used to determine where in the memory the object should be placed. The index should be multiplied by the minimum object size, and then added to the start memory address of the chunk. Figure 3.8 shows what allocating a 1 byte object might look like, if the chunk has some objects already allocated. Allocated slots appear in red, and *HEAP_MINBITS* represents the minimum size of object allocated to a slot, which is 8 bits.

Algorithm 3: Allocating a New Chunk

```
1  Procedure newChunk(globalPool)
2      m ←Pointer
3      currentPool ←globalPool.free
4      if currentPool ≠ NULL then
5          remainingMem ←currentPool.size - CHUNKSIZE
6          currentPool.usedCount ←currentPool.usedCount + 1
7          m ←currentPool.mem + remainingMem
8          if currentPool is full then
9              gp.free ←currentPool.next
10             currentPool.next ←globalPool.full
11             globalPool.full ←currentPool
12         end
13         else
14             currentPool.size ←remainingMem
15         end
16     end
17     else
18         newPool ←newMempool()
19         m ←newPool.mem + newPool.size
20     end
21     newChunk ←Init
22     newChunk.mem ←m
23     return newChunk
```

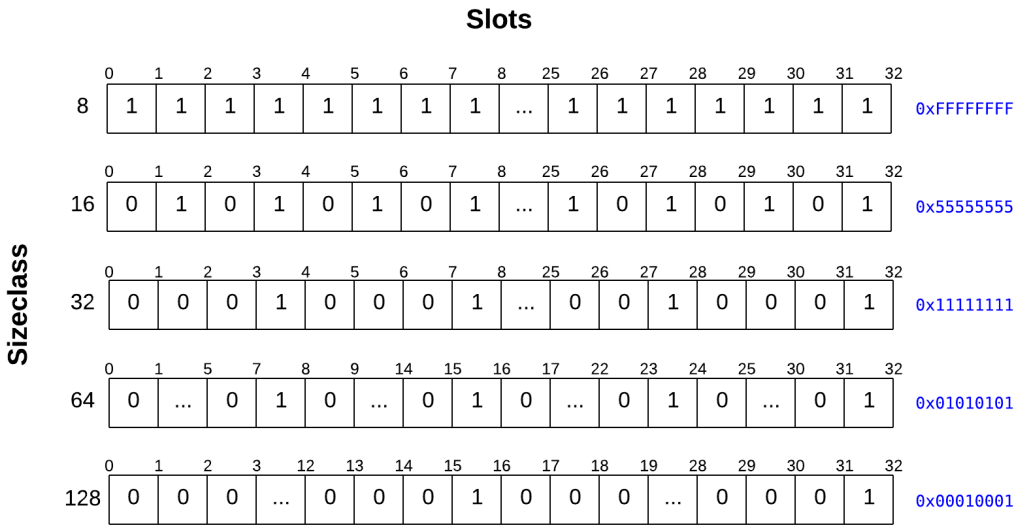


Figure 3.7: Sizeclass Layout

3.2.3 Integration

Now that the core memory structure has been covered, how it all fits together to build GoA can be discussed. The overarching data structure that contains the memory is the Global Pool shown in Figure 3.9. The global pool is truly global so it requires a mutex for access. It is only used when allocating a new mempool, or creating/destroying chunks. Since chunks recycle their own memory, the global pool does not have to be constantly accessed when allocating new objects. The global pool contains two linked-lists of mempools. The free list contains the current mempool, connected to any other free mempools that have been recycled. The full list contains all the exhausted mempools. These are kept in the list until all memory within a mempool has been garbage collected and can be removed. When a free mempool fills, it is immediately moved into the full list and upon a new chunk allocation, a new free mempool is grabbed from Go using the process described above in Section 3.2.1. The global pool also has a reference to a list of all chunks active in the system, called a pagemap. The pagemap structure is shown in Figure 3.10. The pagemap is a mapping that links the chunk structure definition with its location in memory. The global pool is responsible for maintaining the global memory and ensuring only active chunks are contained within the pagemap.

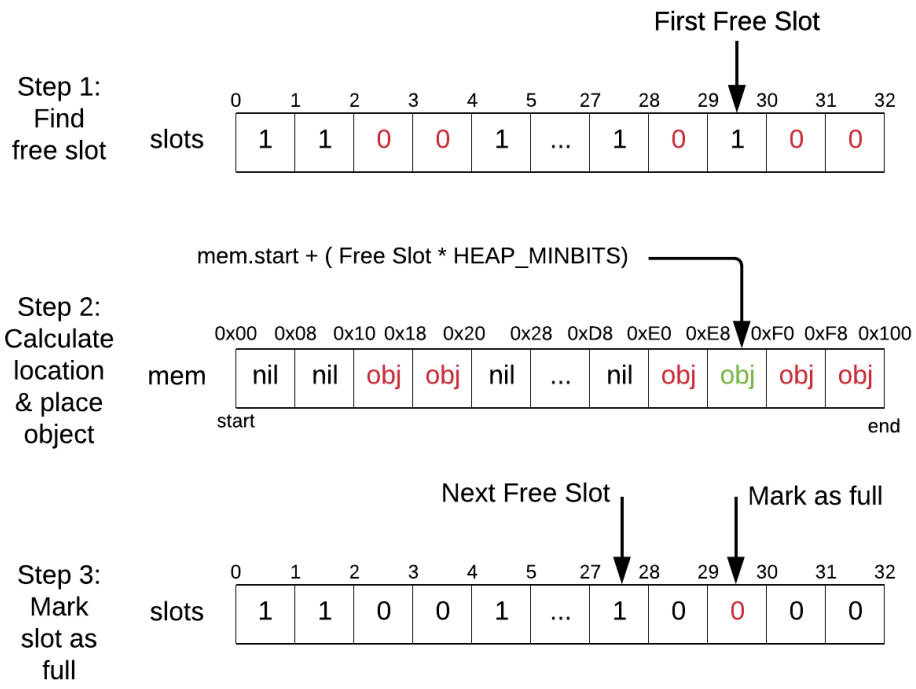


Figure 3.8: Placing a 1 byte Object in Memory

```
type GlobalPool struct {
        current  *mempool
        full     *mempool
        mtx      *sync.Mutex
        pagemap  *pageMap
}
```

Figure 3.9: Global Pool

```
type pagemap struct {
        items map[uintptr]*chunk
        sync.RWMutex
}
```

Figure 3.10: Pagemap

```
type Heap struct {
        global     *GlobalPool
        ctx        *Context
        smallFree  [sizeclasses]*chunk
        smallFull  [sizeclasses]*chunk
        large      *chunk
        used       uintptr
        nextGC     uintptr
}
```

Figure 3.11: Heap

Another important data structure is the Heap (Figure 3.11). Heaps are structures that hold the actor local memory. This means heaps have a 1-1 relationship with an actor. The memory comes in the form of the chunks described in Section 3.2.2. Much like the global pool, each heap contains a linked-list of free, and full chunks. They are slightly different as there needs to be a linked-list for every different sizeclass, so the lists are contained within a fixed size array. Large objects are held in another separate linked-list called *large*. As they are allocated into their own chunks, they are placed at the head of the large chunk linked-list. The heap also has a link to the global pool, a total used memory counter, and a threshold value denoting when garbage collection should occur. There is a context field which is essentially the information that comes from the actor itself, as well as important garbage collection information. The heap is responsible for handling all local memory in an actor. It handles the allocations as well as the deallocations of objects. As objects are allocated they are placed into existing or new chunks. If a new chunk is allocated, the current one must be moved to the appropriate full linked-list. The Heap also is the home to the mark and sweep algorithms that are used within the garbage collector.

The context structure is responsible for information that pertains to the actor itself.

```
type Context struct {
        pid    *actor.PID
        actor  interface{}
        heap   Heap
        gc     garbageCollector
}

type garbageCollector struct {
        mark    uint32
        local   map[uintptr]*object
        foreign map[*actor.PID]actorRef
}
```

Figure 3.12: Context and Garbage Collector

It has a reference to the proto-actor ID of the spawned actor, as well as a field to denote the actor type as defined by the user. There is also a link to the heap as it, the heap, and the actor, all maintain a 1-1 relationship. The gc field contains all information relating to the garbage collection algorithms. These include the mark field, and two maps: local and foreign. The mark field denotes which garbage collector cycle the context is on. The maps are responsible for keeping track of shared objects and their respective reference counts. The local map contains objects that the linked actor owns, and has shared with other actors. The foreign map is the opposite, it contains objects that the linked actor has received from other actors. This information is important when garbage collecting as it prevents a shared object from being collected prematurely.

A more complete and detailed memory diagram is shown in Figure 3.13. The memory layout is divided into two portions: static and dynamic. The static side is initialized when the program starts. It only consists of the global pool structure that contains two pointers to the full and current mempools. The dynamic side is formed as memory is allocated. It is built from the mempool structures, and the chunks within them. In the figure, grey symbolizes allocated memory and white represents free memory. There are two allocated mempools: one is full and the other is currently being used. A third mempool has yet to be utilized. Two actors, A1 and A2, are created and each contain their own actor-local heaps. An actor-local heap consists of linked-lists of every chunk size that an actor has allocated. Actor A1 is shown to have a linked-list of 8 bit and 128 bit chunks. Actor A2 only has a linked-list of 256 bit chunks. Similarly sized chunks are chained together if they

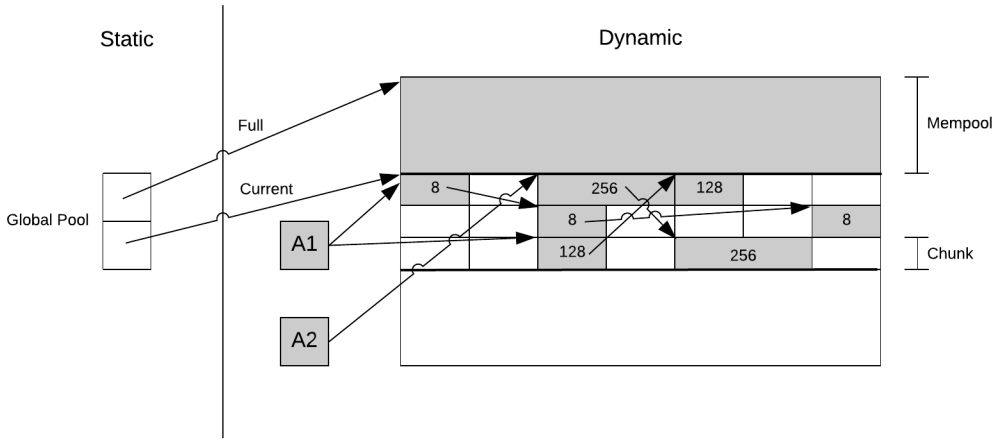


Figure 3.13: GoA Complete Memory Diagram

are owned by the same actor.

3.3 Memory Management

GoA completely redesigns the way Go allocates memory. Obviously, this system only works when allocating within an actor, and the custom Alloc function is used. If memory is allocated outside of an actor, or the *new* keyword is used, Go manages the memory normally. GoA's allocation algorithms are primarily based off of Pony's memory system. This section covers how the allocation works (3.3.1) as well as the deallocation (3.3.2).

3.3.1 Allocation

When the user creates an object in GoA, the local memory manager is responsible for finding the next, appropriately sized, free memory to fit the object in. This process happens in a number of different steps, and depends on the size of the object. The basic algorithm for allocating small objects is shown in Algorithm 4. For objects that are less than the set threshold value, they are handled with the mempool and chunks. First, the object size is found to calculate the correct size of chunk to use. This is determined through sizeclasses. Once the sizeclass is obtained the current free chunk for that sizeclass is fetched. If no chunk is found, a new chunk is allocated, and the variable is assigned to the first slot within

that chunk. If there is already an allocated chunk, the first free slot within the chunk is calculated. This process is a simple algorithm where the slots bitmap is taken, and the first trailing 0 is found indicating an empty slot. This algorithm works well, as when a slot within the chunk is garbage collected, the location can be immediately reused for the next allocation. The variable can then be assigned to the corresponding memory location. Upon filling a slot in a chunk, if the chunk is now full, it needs to be handled accordingly. If the slots are completely full, the chunk is moved into the full list, and the current chunk is cleared and ready to be assigned a new one when the next allocation call occurs.

Algorithm 4: Memory Allocation for Small Objects

```
1  Procedure allocSmall(object, heap)
2      m ←Pointer
3      sizeclass ←calculateSizeclass(object.size
4      currentChunk ←heap.smallFree[sizeclass]
5      if currentChunk ≠ NULL then
6          freeSlot ←findNextFree(currentChunk.slots)
7          m ←currentChunk.mem + freeSlot
8          if currentChunk.slots is full then
9              heap.smallFree[sizeclass] ←currentChunk.next
10             currentChunk.next ←heap.smallFull[sizeclass]
11             heap.smallFull[sizeclass] ←currentChunk
12         end
13     end
14     else
15         newChunk ←heap.newChunk(sizeclass)
16         m ←newChunk.mem
17     end
18     return m
```

For handling large objects, the algorithm is much more streamlined. The algorithm above is complicated because it needs to fit multiple small objects into the size of a single chunk, and handle when an object mid-chunk may be freed. When allocating memory for a large object, it is assigned its own chunk with as much memory as is required. The object's size is calculated and the system calls for that much memory from the mempool. The only stipulation is the mempool requires memory to be taken in chunks of uniform size. If the chunk size is 256 bits, the amount of memory allocated for this large object must conform to this structure. After requesting the object's required memory, it is padded to fit neatly

into the mempool. This padding ensures that memory allocated from the mempool is ordered correctly, and makes it more efficient for the garbage collector to sweep over this memory.

3.3.2 Deallocation

Algorithm 5: Memory Deallocation for Small Objects

```
1  Procedure destroySmall(chunk, globalPool)
2      globalPool.removeFromPagemap(chunk)
3      memAddress ←chunk.mem
4      chunk ←NULL
5      if memAddress ≥ gp.free.start & memAddress < gp.free.end then
6          globalPool.free.usedCount ←globalPool.free.usedCount - 1
7      end
8      else
9          foreach mempool ∈ globalPool.full do
10             if memAddress ≥ mempool.start & memAddress < mempool.end then
11                 mempool.usedCount ←mempool.usedCount - 1
12                 if mempool.usedCount = 0 then
13                     mempool ←NULL
14                 end
15             end
16         end
17     end
```

Deallocation is the process of removing unused memory from a chunk, back to the global pool. When a chunk has lost all references to its contained objects, it can be collected. The mechanism that determines when and how this happens is discussed in Section 3.4, but this section gives an overview of how the memory itself is reclaimed. The detailed destroy algorithm is shown in Algorithm 5. Here, there is a chunk that has been completely emptied and can be reclaimed. First, the algorithm must remove the reference of the chunk in the global pagemap. This removal ensures the other algorithms that search the pagemap do not find a deallocated chunk. Then a value referencing the chunk's location in memory is kept and the chunk is set to null. Next, the current mempool is searched. If the chunk memory falls in the range of this mempool, the amount of chunks used can be decreased by one. If the program has allocated enough memory to have a chain of mempools currently

in use, searching those may be necessary if the chunk is not found in the current mempool. Once found the amount of chunks used by that mempool can be decreased by one.

Large object deallocation, much like the allocation, is easier than that for small objects. When a large object falls out of scope and needs to be destroyed there are only three steps involved, two of which match the small object algorithm. First, the chunk needs to be removed from the pagemap. Second, the fields of the chunk are set to null. Lastly, the memory used by the object is freed from the mempool. Each available mempool is searched for the object's location, and once found the amount of chunks used by the object are freed, and recycled back into the mempool.

The deallocation algorithm is not only responsible for the chunk memory, it also determines when to recycle or destroy the global pool back to the Go memory system. When a mempool is allocated, it is assigned a value to determine how many chunks can be taken from it. Every time a chunk is being deallocated it subtracts one from the owning mempool. When this count hits zero, the mempool has been cleared and is destroyed. This destruction is accomplished by simply assigning the mempool to null.

3.4 Garbage Collection

Throughout the execution of a large program, there may be an abundant amount of objects created. Computers only have a finite amount of memory to operate with. This means that when too many objects are allocated, memory available for storage will run out. However, these created objects do not tend to stay within the scope of the program during its entire lifetime. When objects fall out of the scope of the program, they need to be collected by the garbage collector. Garbage collection allows memory to be recycled and allocated again to newer objects. While memory can still become exhausted through poor programming practices, the garbage collection mechanism helps keep memory available. There are many different techniques for garbage collection, but GoA uses a simple mark and sweep technique. Mark (Section 3.4.1) is responsible for finding every reachable object in the program and marking it as found, so it is not swept. Sweep (Section 3.4.2) iterates over every object in storage and sweeps all the objects that do not have a mark. This process is entirely contained within the actor's themselves so they do not interrupt the execution of any other actors. However, an actor doing garbage collection is not acting on messages, so may cause latency if other actors depend on its execution.

3.4.1 Mark

The mark algorithm shown in Algorithm 6 is responsible for finding every object that is within scope of an actor. Initially, it starts with the actor itself as the object to be marked. It attempts to find the object located within a localmm chunk. If not successful, that means the object is not locally allocated, and no additional processing is required for this object. However, this object may contain references to objects that have been locally allocated.

Algorithm 6: Garbage Collector Mark

```
1  Procedure Mark(context, obj)
2      chunk ←context.getChunkFromPagemap(obj)
3      if chunk = NULL then
4          Traverse(obj)
5          return false
6      end
7      marked ←boolean
8      if chunk.owner ≠ context.actor then
9          marked ←context.markRemote(chunk, obj)
10     end
11     else
12         marked ←chunk.slots&slot = 0
13         chunk.slots& = ¬slot
14     end
15     if !marked then
16         Traverse(obj)
17     end
18     return marked
```

This means the process must repeat for each of the object's fields. If the object has been locally allocated, the actor executing the garbage collection is checked if it is the owner of the object. If that is the case, line 12 checks if the object has been previously marked, and sets the *marked* variable accordingly. Line 13 marks the object using the slot location in the chunk. If the found object is remote, the object reference is marked. These are evaluated during the sweep phase on whether or not to notify the owning actor that a reference to their object has been dropped. This notification system ensures that a remote object that a foreign actor has a reference to is not collected by the actor that owns it.

The final step is to repeat this process for the object's fields, and mark those as well. This step is not necessary if the object is found to be previously marked.

3.4.2 Sweep

The sweep phase of the garbage collector is initiated immediately after the marking has finished. The sweep algorithm shown in Algorithm 7 works in four different parts. First, small unmarked objects are searched. Secondly, it sweeps all large objects. Next, the *heap.used* value is updated to the un-swept memory, and the next garbage collection factor is set. Lastly, any remote references that have been swept notify the owning actor. A variable, *used*, is kept to track the amount of memory maintained by the chunks.

The first step is checking for empty chunks and destroying them. The algorithm must loop over each of the sizeclasses and work on the respective chunks separately. For each of the chunks in the current sizeclass, the action taken is dependent on the *slots* value. *slots* is a bitmap that represents how many objects are allocated within a chunk. If the *slots* value is zero, then no objects have been marked and the chunk is completely full. This means the used value can be increased by a full chunk size. If the *slots* value is completely full, the chunk may be destroyed as per Algorithm 5. Finally, if the chunk neither full or empty, how much of the used chunk memory is determined and added to the used value.

Sweeping large objects is simple. Since there is a single chunk holding a single large object, all that is required is to determine whether the *slots* value is zero or not. If the value is zero, it has not been marked and therefore can be kept and the used value can be updated. If the chunk has not been marked its destruction sequence can be initiated as discussed in Section 3.3.2.

Next the *heap.used* value is updated to what was calculated and the next garbage collection factor is increased. This update is a simple calculation of the current used value multiplied by the preset garbage collection factor. The factor grows with every garbage collection to decrease the frequency of garbage collections in heavy allocated programs.

Finally, the actor checks its foreign objects. If an actor received an object from another actor, and no longer has a reference to it, it may need to be garbage collected. However, it can only be swept by the owning actor, and cannot do so until all foreign references to it have been destroyed. Starting at line 20 the sweep algorithm begins iterating over all of the actor references it sees. Each of these actor references contain specific objects that they own, that they have references to. If the references were not marked during the mark phase at line 9, they are deleted from the references. This deletion involves removing it

from the actor map and sending a message to the owning actor telling it a reference to its object has been dropped. These special messages are handled in Section 3.5.

Algorithm 7: Garbage Collector Sweep

```
1  Procedure Sweep(context, heap)
2      used ←0
3      foreach sizeclass ∈sizeclasses do
4          chunks ←heap.getChunks(sizeclass)
5          foreach chunk ∈chunks do
6              if chunk.slots = 0 then
7                  used ←used + CHUNKSIZE
8              end
9              else if chunk.slots = sizeclassEmpty[sizeclass] then
10                 chunk.destroy()
11             end
12             else
13                 used ←used + chunk.used
14             end
15         end
16     end
17     used ←used + sweepLarge()
18     heap.used ←used
19     h.nextGC ←nextGcFactor
20     foreach actorRef ∈context.gc.foreign do
21         foreach object ∈actorRef.objects do
22             if isNotMarked(object) then
23                 notifyOwner(object)
24             end
25         end
26     end
```

3.5 Message Passing & Reference Counting

When developing concurrent programs, there often needs to be a way for the concurrent processes to communicate with each other. There are many different approaches to con-

current communication, but the best solution for providing communication among actors is message passing. Each actor comes equipped with a mailbox that holds all incoming messages. A message is any function call on an actor, along with all of the objects that make up the arguments. They are accepted by the actor and executed on sequentially as they arrive. Actors may work concurrently with each other, but a single actor only executes one message at a time. This process is handled through the proto-actor framework discussed in Section 2.6.

To track where each object has traveled, the local memory manager needs to make some adjustments. The purpose of tracking is to determine when and if it can be garbage collected, and is accomplished through reference counting. When a message is sent and received by an actor, the custom middleware (Section 3.1.2) is used to inject code that can handle the reference counting. There are four types of messages that can be processed with this middleware as seen in Algorithm 1:

- Actor Start
- Reference Count Decrease (RcDec)
- Reference Count Increase (RcInc)
- Regular Message

For sending objects, only regular messages are processed. Sending consists of all actor to actor message calls. The outbound middleware takes each of these messages and processes each of the arguments within. This process is discussed in Section 3.5.1. When receiving messages, it must be parsed into one of the four different types of message. When receiving an actor start message, the actor context and local memory are initialized by creating its first mempool. RcDec and RcInc messages handle the reference count of an object. When an actor has a reference to a foreign actor's object, there needs to be a way to communicate when the object is being shared again or deleted. If this actor sends a foreign object, an RcInc message is sent to the owning actor and its reference count is increased by one. If the object is being swept and garbage collected, an RcDec message is sent and the object's reference count is decreased by one. If the reference count hits zero, it is deleted. Any other message is processed by the receive algorithm shown in Section 3.5.2.

3.5.1 Sending

When a message containing objects is shared among actors, it must undergo processing to determine how the objects should be counted. The algorithm is shown in Algorithm 8. First, it must find the object's chunk in the global pagemap.

Algorithm 8: Sending Object to Another Actor

```
1  Procedure SendObject(obj, context)
2      chunk ←context.getChunkFromPageMap(obj)
3      if chunk = NULL then
4          Traverse(obj)
5          return false
6      end
7      if isNotShareable(obj) then
8          Panic()
9      end
10     if chunk.owner = context.actor then
11         orc ←context.getOrc(obj)
12         if orc = NULL then
13             context.createOrc(obj)
14         end
15         else
16             orc.rc ←orc.rc + 1
17             orc.mark ←context.mark
18         end
19     end
20     else
21         chunk.owner.SendRcInc(obj)
22     end
23     Traverse(obj)
24     return true
```

If a chunk cannot be found, the object is not locally allocated. This means it does not have a reference count. The object's fields, if any, still need to be traversed in case they were locally allocated, and this process is repeated. If this is the case the function exits as no more processing needs to occur on this non-local object. If the object is found, its assigned capability is checked. Capabilities and their purpose are covered in Section

4. Most capability checks are handled separately, but here its determined if it has been assigned a sendable/shareable capability. If it has been assigned a local capability, the safety of the program cannot be ensured, and a runtime error is thrown explicitly stating so. The algorithm then determines if the object being sent belongs to the current actor trying to send. If the actor owns the object in question its reference count data structure is obtained. If it does not yet exist, it is created, otherwise its reference count is incremented by one. If the object is foreign to the current actor, a RcInc message is sent to the original owner. This is handled as discussed above. After each of these steps have been completed, any potential fields the object may have are traversed into and the process is repeated.

3.5.2 Receiving

Every time an object is sent, it is received by another actor. The processing each object undergoes after being received is similar in structure to the send process. Each object must be processed in order to determine how to handle the reference counts for the object. The receiving algorithm is shown in Algorithm 9. Much like the send algorithm, it begins by finding the object's chunk. If the chunk is not found, the object's fields are traversed, repeating this process, and exits afterwards. If the object is local to the receiving actor, the object's reference count data structure is obtained. If it does not exist, that means the actor received this object in error, or the object was not sent correctly. Therefore, an error is thrown stating the issue. Otherwise, since the actor received its own object its reference count can be decreased by one. If the reference count for the object reaches zero, the reference count data structure is deleted. If the object being received is foreign, the actor reference count object is obtained first. If it does not yet exist, it is created. Otherwise, the actor reference count object is used to grab the object reference count object. If that is null it is created, but if it exists, the count is incremented by one. If the object is foreign, the memory used count is increased so that it might trigger garbage collection. Otherwise, an object that uses only remotely allocated objects never triggers garbage collection. Finally, the object's fields are traversed and the process is repeated if necessary.

3.6 Object Traversals

There are a few algorithms covered in this thesis that depend on a traverse function. This function is written to take any object and run a given function on each of the object's fields. This, of course, only applies to object's that are of specific types. The *Struct* type may have fields to traverse and the *Array* type has elements to traverse. This function is called

Algorithm 9: Receiving Object from Another Actor

```
1  Procedure ReceiveObject(obj, context)
2      chunk ←context.getChunkFromPageMap(obj)
3      if chunk = NULL then
4          Traverse(obj)
5          return false
6      end
7      if chunk.owner = context.actor then
8          orc ←context.getOrc(obj)
9          if orc = NULL then
10             Panic()
11         end
12         orc.rc ←orc.rc - 1
13         if orc.rc = 0 then
14             Delete(orc)
15         end
16         else
17             orc.mark ←context.mark
18         end
19     end
20     else
21         arc ←context.getArc(obj)
22         if arc = NULL then
23             context.createArc(obj)
24         end
25         orc ←arc.getOrc(obj)
26         if orc = NULL then
27             arc.createOrc(obj)
28         end
29         else
30             orc.rc ←orc.rc + 1
31             orc.mark ←context.mark
32         end
33         context.heap.used+ =chunk.size
34     end
35     Traverse(obj)
36     return true
```

regardless of the type, and simply returns if these types are not encountered. The traverse function is implemented in two different ways. One is written using Go's reflect package and the other is an explicit definition of each traverse function created through the AST Translate Tool discussed in Chapter 5. The reflection implementation is used to determine the types of objects at runtime. It is an extremely effective tool for this purpose, but can be slower than having an explicitly declared traverse function for a given struct. For this reason, the AST Tool should be used for user created objects and the reflect method should only be used for objects the user does not have access to.

Algorithm 10: Traverse Into Object Fields

```
1  Procedure Traverse(obj, function)
2      value ←reflect.Value(obj)
3      switch value.Kind do
4          case reflect.Struct do
5              foreach field ∈value.Fields do
6                  function(field)
7              end
8          end
9          case reflect.Array do
10             foreach element ∈value.Elements do
11                 function(element)
12             end
13         end
14         case reflect.Ptr do
15             function(value.Element)
16         end
17     end
```

The reflection version of the traverse algorithm can be seen in Algorithm 10. The first step is to utilize the reflect package to get the runtime type value of the object. With that information, a switch statement is used on the type with three cases to work on. If the object is a struct the function is applied to each of the struct's fields. For an array type the function is applied for all of the elements within the array. Lastly, by default if the type is a pointer, the element pointed to by the pointer is obtained and the function is applied to the object. All other types are not traversable and can be ignored.

3.7 Conclusion

This chapter establishes the foundation of the local memory management system that GoA provides. The entire memory system is redesigned to work locally within individual actors. New allocation/deallocation algorithms are introduced to maintain this actor-local principle. These algorithms paired with the garbage collection and message passing techniques come together to create a system that works without the overhead that stop-the-world garbage collection may contain. This system is improved even more with the safety guarantees that the capability system adds. These capabilities are discussed in the next section, Chapter 4.

Chapter 4

Capabilities

4.1 Introduction

The capability system introduced in this section provides a way to eliminate data races and ensure all memory shared among actors maintains its integrity. These capabilities are annotations that are attached to objects as they are declared. They follow the object when being shared and each capability comes with specific rules on how the data can be used after being shared. This system is based off the capability system Pony provides. Pony's capability system is discussed in Section 2.3. GoA comes equipped with four capabilities:

- Loc (local)
- Imm (Immutable)
- Mut (Mutable)
- Tag (Opaque)

This section details these four capabilities and how they are implemented. The design of the capabilities is explained in detail as well as how they are integrated into the system. The integration details are covered in Section 4.2. Section 4.3 covers each of the four capabilities used in GoA. It explains the importance of each, and defines the exact rules they place on different scenarios when sharing/passing data. Section 4.4 shows how the capabilities are enforced through runtime checks. The checks are ingrained into the runtime, but require the AST Translate Tool described in Chapter 5 to create the necessary calls before an object is used.

```
const (
    Mut capability = iota   // mutable
    Tag                     // cannot read or write, only send messages
    Imm                     // immutable
    Loc                     // local variable
    ToBeImm                 // intermediate cap
    ToBeTag                 // intermediate cap
)
```

Figure 4.1: Capability Enum

```
base := (*obj)(a.ctx.AllocSecure(obj{}, capability))
```

Figure 4.2: Modified Alloc Call to Include Capability

4.2 Integration

GoA does not yet have language support, which means integrating the capabilities requires some changes to the local memory manager library. At the user level, an object creation currently requires a call to the library's custom allocation call, *Alloc*. There is another call implemented to allow the user to specify a capability on an object. *AllocSecure* has an extra argument that allows the user to attach a capability. Capabilities are stored as an enum shown in Figure 4.1 and do not require any instantiation to use. The secure allocation call is shown in Figure 4.2

In this example, the allocated base object is restricted by whatever capability is passed into the function. The current supported capabilities are defined in Section 4.1. However, when using immutable capabilities such as Tag and Imm, immediately locking the object from writes does not allow the user to instantiate that object. Two extra capabilities are added as a way to work around this issue. These capabilities are dubbed *ToBe* capabilities and they are apart of the original enum. Pony has no need for *ToBe* be capabilities, as immutable objects can be initialized when they are declared. GoA is forced to grab memory first, then initialize variables.

The *ToBe* capabilities are used exclusively as intermediate representations of their respective capabilities. When an object is in a *ToBe* state, it is treated as a local variable. Changes can be made freely, and can even be passed into synchronous functions. When

```
type obj struct {
    x int
    y int
}

func (a *Actor) randomMethod() {
    // Allocate base to eventually be immutable
    base := (*obj)(a.ctx.Alloc(obj{}, ToBeImm))
    // initialize base fields
    base.x = 0
    base.y = 1

    a.ctx.LockCapability(base) // Lock base as Imm
}
```

Figure 4.3: Using a ToBe capability

attempting to pass a *ToBe* object to another actor, a runtime error is thrown as it is not yet safe. Ideally, when using a *ToBe* object, it only remains in that state for its initialization and then it is locked into its actual capability. Capability locking can be done with a call to the localmm library's *LockCapability* method. This method essentially takes an object, finds what *ToBe* capability it was assigned and determines what actual capability it requires and locks it in. Figure 4.3 shows how a user should use a *ToBe* capability.

When declaring a new object, the capability information needs to be stored with it. Upon calling *AllocSecure*, the capability is kept until the memory is being grabbed from whichever chunk is available. There is an internal field added to the chunk to store the capability attached to an object. With the current design, each allocated object already is assigned a specific location within a chunk's memory. This location is tracked with a bitmap. Conveniently, storing an object's capability can utilize the same location and bitmap strategy. An array is stored with the chunk struct to represent the capability bitmap. The length is equal to the amount of slots available in the chunk and the array's elements contain the capabilities. This new field is called *capmap*. The chunk struct in Figure 4.4 shows the added field.

Since the bit location is already calculated for the memory allocation, the same bit location can be used by the capmap to store the object's capability. This ensures no performance degradation. However, there is more space used by the structure and also

```
type chunk struct {
    ...
    capmap  [CHUNK_SIZE << HEAP_MINBITS]capability
    ...
}
```

Figure 4.4: Chunk with capability map

introduces an extra array access.

4.3 Implemented Capabilities

As mentioned in the background Section 2.3, Pony provides a multitude of different capabilities. For the purposes of this project, a more streamlined set is used. The capabilities used can be summarized in three categories: send-able, shareable, and local. Two capabilities provide the user with completely safe shareable data, Tag and Immutable. These provide a guarantee of immutability, which means the data can be shared among multiple actors without threat of data corruption. The Mutable capability guarantees only a single actor holds an object, which means it is send-able; the object ownership changes to the receiving actor. The Local capability means the data cannot be sent or shared, and should be treated like an ordinary object. The rules of the capabilities are defined through their ability to be read from, write to, or be used to invoke methods / send messages after the object has been shared. Each capability in this section breaks down these rules when applied to two scenarios:

- An actor sends an object
- An actor receives an object

4.3.1 Local

The Local capability is responsible for all objects that are allocated and are not intended to be shared. It is given the keyword *Loc*. Local is a simple capability and is the default applied to all locally allocated objects. This annotation is enforced during sending. If an

object with a local capability is sent in a message to another actor, an error is thrown. A Loc object is safe to use in any situation where the object remains with the actor that created it.

4.3.2 Tag

The Tag capability is the most restrictive of the sendable. While other capabilities provide some degree of flexibility and may allow certain reads or writes, the Tag capability disallows all. Tag objects are completely opaque, and therefore, provide the safest guarantee when sharing. The only permissible operation on a Tag capability is the use of its methods. The keyword for this capability is *Tag*. The primary purpose of this capability is to share an object that has methods that need to be used by other actors. By default, all actors are treated as Tags. Table 4.1 shows the rules the Tag capability follows. When sharing a Tag variable, it remains visible to both actors involved. When an object is sent or received, all read/write operations are forbidden. The only permissible operation is method calling. This capability must be declared with the *ToBe* feature and locked in after initialization.

	Actor Sends Object	Actor Receives Object
Object Visible	✓	✓
Read Permission	x	x
Write Permission	x	x
Call Method / Send Message	✓	✓

Table 4.1: Tag Capability Rules

4.3.3 Immutable

The Imm capability essentially makes an object immutable. This provides the user with a way to share objects with other actors without fear of the object being modified, while keeping the ability to read. This means all actors with a reference to it can read from the object without data corruption or data race concerns. The Imm capability rules in Table 4.2 displays what operations are valid. Being immutable, all reads are allowed, while no writes are permitted, regardless of having sent or received the object. Any actor with a reference can use the object to send messages. This capability utilizes the *ToBe* capability feature to allow for initialization. Once the capability is locked in, no actor, including the owner can modify it.

	Actor Sends Object	Actor Receives Object
Object Visible	✓	✓
Read Permission	✓	✓
Write Permission	x	x
Call Method / Send Message	✓	✓

Table 4.2: Imm Capability Rules

4.3.4 Mutable

The Mut capability is meant for objects that are to remain completely mutable throughout their lifetime. When an object is created with a Mut annotation, it can be treated as a normal variable when an actor has ownership of it. It is the most complicated of the four implemented. This complexity is because the object needs to maintain its mutable properties for only a single actor at a time. This requires knowledge of where this object originated, when it is passed to another actor, and who the newest current owner is. All this must be dynamically tracked, while the object is being sent and received. When an actor sends a Mut variable to another actor, the capability checker is able to determine if it is incorrectly used after rescinding ownership. If incorrectly used, a runtime error occurs. If used correctly, the variable can then be used by the new actor until it finishes with it or sends it to another actor. Table 4.3 highlights the rules a Mut capability must follow. If an actor sends an object, it loses its reference permission and is not able to perform any operation. An actor that receives a Mut object is able to perform any operation.

	Actor Sends Object	Actor Receives Object
Object Visible	x	✓
Read Permission	x	✓
Write Permission	x	✓
Call Method / Send Message	x	✓

Table 4.3: Mut Capability Rules

```
// Function before translation
func (a *Actor) exampleFunction() {
    x, y := 0, 5  //declare variables
    x = y         // write on x, read on y
}

// Result after AST translation
func (a *Actor) exampleFunction() {
    x, y := 0, 5 //declare new variables, not checked

    a.ctx.CheckWrite(x) // check write on x
    a.ctx.CheckRead(y) // check read of y
    x = y
}
```

Figure 4.5: AST Translation Example

4.4 Runtime Checks

4.4.1 Motivation

GoA's capability system works on top of the Go language using the local memory management library. This design means that any safety checks can only happen at runtime. Eventually, compiler supported checks may be implemented. Using the AST Translate tool discussed in Chapter 5, these checks are inserted before every variable read and write. There is a separate check before both to ensure that the correct rules are being asserted based on the given operation. For example, in Figure 4.5 on the last line, the variable read on y and the variable write on x have their own runtime checks. At that point the local memory management library takes over and perform the checks outlined in Algorithms 11 and 12.

4.4.2 Implementation

The algorithm for checking reads and writes can be seen in Algorithms 11 and 12. They are very similar in terms of algorithm structure. The only required information is the

object itself, and the local actor context. The checks use the object's chunk to access the capability assigned at its creation. First, the checks begin by obtaining the chunk where the object is stored. If there is no chunk found, the object is not locally allocated and can be skipped. After the chunk is found, the capability can be obtained from the capmap. For Tag and Imm, the check is simple. Tag variables can never be read from or written to and Imm cannot be written to. These can be enforced regardless of which actors are involved. For Mut variables, both the actor that currently owns the object and the actor that is attempting a read or write operation on it are obtained. If the current owner and the modifier/reader are different, an error is thrown because only the current owner may modify a Mut. Both algorithms simply return if no error is found and the execution can continue.

Algorithm 11: Runtime Check: Variable Write

```
1  Procedure CheckWrite(obj, context)
2      chunk ←context.getChunkFromPageMap(obj)
3      if chunk = NULL then
4          return
5      end
6      capability ←chunk.getCapability(obj)
7      switch capability do
8          case Mut do
9              orc ←context.getOrc(obj)
10             if orc ≠ NULL then
11                 modifyingActor ←context.currentActor
12                 currentOwner ←orc.Owner
13                 if modifyingActor ≠ currentOwner then
14                     panic("This actor cannot modify this variable")
15                 end
16             end
17         end
18         case Tag, Imm do
19             Panic("Cannot modify this variable")
20         end
21     end
```

Algorithm 12: Runtime Check: Variable Read

```
1  Procedure CheckRead(obj, context)
2      chunk ←context.getChunkFromPageMap(obj)
3      if chunk = NULL then
4          return
5      end
6      capability ←chunk.getCapability(obj)
7      switch capability do
8          case Mut do
9              orc ←context.getOrc(obj)
10             if orc ≠ NULL then
11                 readingActor ←context.currentActor
12                 currentOwner ←orc.Owner
13                 if readingActor ≠ currentOwner then
14                     Panic("This actor cannot read this variable")
15                 end
16             end
17         end
18         case Tag do
19             Panic("Cannot read this variable")
20         end
21     end
```

4.5 Conclusion

The capabilities introduced in this section provide the user with a way to secure their program from data races and data corruption. Allowing a default local capability alleviates the user from having to always state a capability for every local object, and forcing the user to use send-able capabilities when communicating among actors ensures the program remains memory-safe. The three degrees of send-able capabilities allows some flexibility when deciding which capability to use. If a user wants to keep a shared variable mutable, or if they need absolute immutability GoA's capabilities can provide that. The Mutable capability enforces there be only a single reference to that object, allowing it to change. The Immutable and Tag capabilities allow the user to share an object while maintaining a reference to it so it can continue to be read or utilized for method invocations. These capabilities are built into the runtime system of GoA and are accessed through explicit calls injected by the AST Translate Tool, which is discussed in the next chapter, Chapter 5.

Chapter 5

AST Translate Tool

5.1 Motivation

Pony has the ability to enforce their capability checks and generate object traversal functions statically at compile time. Without the ability to fully integrate these features into the Go compiler, runtime checks are needed for the capability enforcement. Object traversals can be left to the runtime implementation using reflection, as discussed in Section 3.6, or they can be explicitly declared. To ensure the capabilities work to their desired correctness, the amount of checks inserted may be substantial. Every time there is a locally managed variable used, every read and write it is involved in requires a call to a runtime check. This system would be unsustainable if the user had to manually inject these calls into their program. The same is true for object traversals. As the size of any given program grows, the more custom objects may be declared and more traversal functions may be needed. The solution to these problem comes in the form of a separate utility program which the user has to run. The program parses the AST tree from the desired input and manipulate it to insert the checks, and create custom traversal functions.

The idea is to have the AST Translate Tool translate all programs developed with GoA. With the substantial amount of runtime checks needed, it comes with significant overhead. The results can be seen in the benchmarks shown in Chapter 6. The overhead would be unsustainable for production code and therefore should not be deployed. The user should run these translated programs with comprehensive test cases to determine if what they have developed is secure and if memory is handled correctly. Once the test has been run, and the original code has been determined safe, it can be deployed without the checks.

5.2 Implementation

The AST translation tool is developed as a stand alone application in Go. The script accepts a Go file that is developed using the GoA model. The main algorithm is shown in Algorithm 13.

Algorithm 13: AST Translate Tool

input : User defined program utilizing GoA constructs
output: Modified input with calls to capability checker before every local variable read, write, and field access

```
1  checkSafety ←flag.Parse()
2  input ←userProgram
3  inputAST ←parseFile(input)
4  isLocalmm ←false
5  foreach node ∈ inputAST do
6      switch type(node) do
7          case ast.ObjectDeclaration do
8              generateTraverseFunction(node)
9          end
10         case ast.FunctionDeclaration do
11             isLocalmm ←isActorFunction(node)
12         end
13         case ast.Statement do
14             if isLocalmm & checkSafety then
15                 node.Body ←handleBody(node)
16             end
17         end
18     end
19 end
```

First, the command line flag is parsed into a boolean *checkSafety* to determine if the capability checks should be created. Then the input is parsed into an AST tree using some of Go's built-in utilities. The tool is then able to iterate over each node and check for certain node types. If the node is a object declaration, the object's traversal function is generated. The *generateTraverseFunction* algorithm is shown in Section 5.2.1. If the node is a function declaration, it is analyzed to determine if it contains local allocations. Then, if the function contains local allocations, and the *checkSafety* flag is set, the body of the

function is handled to create and inject the capability checks. These methods are discussed in Section 5.2.2.

5.2.1 Traverse Functions

The first step for creating a traverse function for an object is to create the traverse function declaration. The main algorithm can be seen in Algorithm 14. The function's receiver is set to the given object, the name of the function is "Traverse", the function type is "bool", and the body is created through another function *createTraverseBody*. The *createTraverseBody* requires the object's fields as an argument.

Algorithm 14: Generate Traverse Function Declaration

```
1 Procedure generateTraverseFunction(objDeclNode)
2     funcDecl ←Init
3     funcDecl.receiver ←objDeclNode.Name
4     funcDecl.name ←"Traverse"
5     funcDecl.type ←boolean
6     funcDecl.body ←generateTraverseBody(objDeclNode.fields)
7     return funcDecl
```

The body of the traverse function is created by analyzing the fields and determining if they need to be traversed. Only certain types of fields need to be traversed. If a field is a reference type or an array with elements that are reference types, then those fields are traversed. Algorithm 15 shows this process. If an array type is encountered, each field in the array is traversed further by creating another traverse call with the field as the argument. If the field is a reference, a single traverse call is created with that field as an argument.

5.2.2 Capability Checks

The first step to generating capability checks is to determine if a given function contains locally allocated memory. Determining local memory is accomplished by traversing through the function declaration node until it finds, or fails to find, the actor that it is declared on. If the actor contains the *localmm.Context* field, it is able to locally allocate variables using the library. This function exits and the isLocalmm variable is set to true. If the function

Algorithm 15: Generate Traverse Function Body

```
1  Procedure createTraverseBody(objFields)
2      body ←Init foreach field ∈ fields do
3          switch type(field) do
4              case *ast.ArrayType do
5                  foreach arrayField ∈field.fields do
6                      body ←body + createTraverseCall(field)
7                  end
8              end
9              case *ast.Reference do
10                 body ←body + createTraverseCall(field)
11             end
12         end
13     end
14     return body
```

fails to find the localmm context field, more processing happens on that function. This process is detailed in Algorithm 16.

Algorithm 16: Check if function has localmm package

```
1  Procedure isActorFunction(functionNode)
2      functionReceiver ←functionNode.receiver
3      if type(functionReceiver) = ActorStruct then
4          foreach field ∈ functionReceiver.fields do
5              if type(field) = localmm.Context then
6                  return true
7              end
8          end
9      end
```

For each body node which satisfies the isLocalmm property, the handleBody function is called. This function explores every child node starting from the main function body. When it encounters an assign statement, For statement, If statement, or Inc-Dec statement, it processes each node accordingly. For an assign statement, line 5 to line 12 shows how every read is checked, but a write is only checked if the variable already exists. This is necessary as a variable cannot be checked if it does not yet exist in the program. Lines

13 to 17 simply add the elements in the init, condition, and post sections to the items to be checked and generate the code accordingly. The algorithm for the remaining statement types works similarly. A skeleton outline is shown in Algorithm 17.

Creating the capability check is a simple process. The *createCapabilityCheck* method takes the variables that need to be checked and whether it is a read or write. An AST expression node is generated with the given info and the new node is added to the body, a line above the read/write operation in question. When all the nodes have been visited and all capability checks created, a modified version of the existing body node is returned and replaces the original. The end result is the original program with the checks inserted. An example transformation can be found in Figure 4.5.

5.3 Conclusion

The AST Translate Tool alleviates the burden of manually creating Traverse functions and the capability checks. The AST Translate Tool should be run for every program developed with GoA. The Traverse functions are a necessary addition to all programs as an explicit declaration provides much better performance than the reflection based version of Traverse. These functions are easily created for all user declared objects in a given program without the need for any user involvement. The capability checks should be generated in a test environment for the purposes of analyzing the memory-safety. The checks are injected before any object operation in an actor function. These checks include reads, writes, and method invocations. The next chapter, Chapter 6, demonstrations the effectiveness of these capability checks, as well as benchmarks the two Traverse function implementations (explicit and reflection), and the overhead of the capability checks.

Algorithm 17: Insert capability checks into function body

```
1  Procedure handleBody(bodyNode)
2      stmtList ←bodyNode.StmtList
3      foreach statement ∈stmtList do
4          switch type(statement) do
5              case ast.AssignStmt do
6                  items ←statement.Right
7                  stmtList.add(createCapabilityCheck(items, readFlag))
8                  if statement ≠ newAlloc then
9                      items ←statement.Left
10                     stmtList.add(createCapabilityCheck(items, writeFlag))
11                 end
12             end
13             case ast.ForStmt do
14                 stmtList.add(createCapabilityCheck(statement.Init, writeFlag))
15                 stmtList.add(createCapabilityCheck(statement.Cond, readFlag))
16                 stmtList.add(createCapabilityCheck(statement.Post, writeFlag))
17             end
18             case ast.IncDecStmt do
19                 items ←statement.Left
20                 stmtList.add(createCapabilityCheck(items, writeFlag))
21             end
22             case ast.IfStmt do
23                 items ←statement.Init, statement.Cond
24                 stmtList.add(createCapabilityCheck(items, readFlag))
25             end
26             case ast.IfElseStmt do
27                 items ←statement.Init, statement.Cond
28                 stmtList.add(createCapabilityCheck(items, readFlag))
29             end
30         end
31     end
```

Chapter 6

Experiments

A series of different experiments are performed on GoA. Each one attempts to showcase different aspects of the system and display the effectiveness of the new memory management and capability system. This section explains each of the experiments and dive into the purpose of each as well as the impact they have on the system. Section 6.1 covers basic benchmarks on the system such as memory allocations. The next section, Section 6.2, uses the N-Body simulation with background garbage creation to stress the garbage collector. Section 6.3 showcases the speed of the overall system using producer-consumer examples compared with similar languages, such as Pony and uC++. It also demonstrates the capability system and the limitations that come with it. These experiments were run in the following environment:

- Operating System: Linux
- Architecture: amd64
- RAM: 512 GB
- Cores: 64
- Clock: 2500 MHz

Each benchmark provided is written in Go. The basic Benchmarks (Section 6.1 use Go's testing and bench-marking framework with each test in its own module. The NBody experiment (Section 6.2) and the producer-consumer experiments (Section 6.3 are stand alone projects of approximately 300 and 200 LOC respectively.

6.1 Basic Benchmarks

A series of basic benchmarks are completed to display the general performance of the local memory system. There are several groups of tests that were run that showcase the allocation algorithm performance compared to the native Go algorithm. The tests are a series of allocations, performed in a loop. They are executed in a variety of different configurations, with multiple goroutines, array allocation, and goroutine array allocation. Table 6.1 shows how the local algorithm compares to Go's with small and large objects. The results show that the system runs slower for small allocations, between 26.28% for single allocations and 87.93% slower for multiple goroutine, array allocations. However, these tests are very intensive and allocate at a rate that would not normally be applicable. GoA also operate as a library, and upon integrating with the Go language, should see some speedup because it eliminates the locking around GoA's single global pool. When looking at the large object allocations, the system actually is an improvement from the native Go algorithm. There is a speedup of 212.55% for single objects.

Benchmark Name	Performance (ns/op)	% Difference
Local Allocation	39.2	-26.28
Go Allocation	28.9	
Large Local Allocation	1036	212.55
Large Go Allocation	3238	
Local Allocation - 32 Goroutine	2088	-27.3
Go Allocation - 32 Goroutine	1518	
Large Local Allocation - 32 Goroutine	77563	-26.67
Large Go Allocation - 32 Goroutine	56877	
Local Array	10681	-27.41
Go Array	7753	
Multi Local Array	121068	-87.93
Multi Go Array	14615	

Table 6.1: Allocation Benchmarks

6.2 N-Body Simulation

The N-Body benchmark is designed to show how the local memory manager alleviates garbage collection interruptions on goroutines that use little memory. The main idea is that

there are multiple goroutines (or actors depending on the setup) working simultaneously. A single actor calculates the N-Body simulation for a set amount of iterations. In the background, the other goroutines generate garbage. Garbage creation involves an endless loop that is constantly creating small (8 bit) objects and putting them into an array. The array index loops around and the objects are written over by new ones. Overwriting allows the release of all references to the object and the garbage collector can do its cleaning.

A second version of this experiment is run with the NBody actor sending the NBody system to itself to continue the calculation. This test is to benchmark potential message passing overhead that the reference counting mechanism may introduce.

6.2.1 Results

The experiments were run with the following criteria:

- 100,000,000 N-Body iterations
- Garbage inserted into 256-element array
- Results averaged among 3 executions

There are two sets of NBody experiments run. The first is with a single actor performing the NBody calculation. It is evaluated with 32 garbage creator actors and a second time with 320 garbage creator actors. The second experiment is run with an actor sending messages to itself to continue calculating the NBody system. For this experiment, the amount of messages in each run increased by a factor of 10.

The raw results of the first experiments can be seen in Table 6.2 and Table 6.3. Here, the different setups and results are shown. The experiments are run with goroutines and proto-actors with the native memory manager, and proto-actor with the local memory manager. The type of garbage collection cycles depends on the setup of the test. When running with localmm, the cycles are through the local memory system. Without localmm, the cycles are through Go's built-in runtime. On average, with 32 garbage creators, proto-actor with the local memory manager ran 9% faster than with the native memory manager. The improvement is even greater with 320 garbage actors at 78%. This improvement is substantial as the simulation experienced about 50 times as many garbage collection cycles during the runtime period. The total number of garbage collection cycles decreased in the 320 garbage actor run because the number of allocations per actor also decreased. The

NBody - 32 Garbage Actors				
Setup	Time (s)	GC Cycles	# of Allocations	Allocations/second
Go-routines	26.0	94	173,203,516	6,654,821
Proto-Actor	29.4	451	175,307,337	5,972,888
Proto-Actor w localmm	24.1	4534	101,905,729	4,223,997

Table 6.2: N-Body Simulation Results - 32 Garbage Actors

NBody - 320 Garbage Actors				
Setup	Time (s)	GC Cycles	# of Allocations	Allocations/second
Go-routines	107.4	34	661,613,844	6,160,520
Proto-Actor	129.1	376	815,764,032	6,321,675
Proto-Actor w localmm	23.6	4082	91,538,211	3,871,724

Table 6.3: N-Body Simulation Results - 320 Garage Actors

contention for the memory allocator between the 320 actors meant they were not able to create as many objects, and therefore did not need to garbage collect as frequently.

In Tables 6.4 and 6.5, the specific threads running the N-Body simulation are analyzed. For the base experiment, there is significant garbage collection pause time, garbage collector sweep time, and scheduler wait time. These interruptions led to the N-Body simulation only being executed for 96% of the total time. For the local memory manager experiment, there is a small scheduler wait time, and no garbage collector interrupts. These factors mean the N-Body simulation is able to run during 99.99% of the total time. This effect is where the significant speed up comes from. The garbage creation threads handle themselves and nothing interrupts the main thread. Figures 6.1 and 6.2 showcase these results even further. These are traces of the executions of the base case and the local memory run. In the base case trace, the garbage collector (GC) is constantly stopping the N-Body thread (G10), and the scheduling is sporadic across the processors. In the local memory trace, the N-Body thread (G25) is a single continuous execution. The garbage collection threads are also tightly arranged, while still looking a bit messy as message passing allows the scheduler to move them among processors.

The second experiment is run with different magnitudes of messages being passed throughout the course of the simulation. It is run with the base Go memory manager and the local memory manager. This test is done to demonstrate the overhead introduced by the message processing middleware. Specifically the algorithms involved in the reference counting techniques when sending and receiving objects. Figure 6.3 shows the

Goroutine	Total Time (ns)	Execution Time (ns)	Scheduler Wait Time (ns)	GC Sweep Time (ns)	GC Pause Time (ns)
10	17241555083	16555579067	685976016	5068	8623384521

Table 6.4: Base Goroutine Analysis (NBody)

Goroutine	Total Time (ns)	Execution Time (ns)	Scheduler Wait Time (ns)	GC Sweep Time (ns)	GC Pause Time (ns)
25	13324144890	13323015398	1129492	0	0

Table 6.5: Proto-Actor with localmm Goroutine Analysis (NBody)

relation between how many messages are being sent, and how much overhead they add to the NBody calculation. The message processing middleware begins to show significant overhead around millions of messages. The overhead becomes even more significant when 10s of millions of messages are sent.

The one caveat from the N-Body experiments is the amount of allocations completed. As shown in the basic benchmarks (Section 6.1) the amount of allocated objects in the given execution period is smaller. On average there is a decrease of 64% of allocations/second. While significant, the sole purpose of 320 whole actors is to simply allocate garbage, and is not reflective of most real applications.

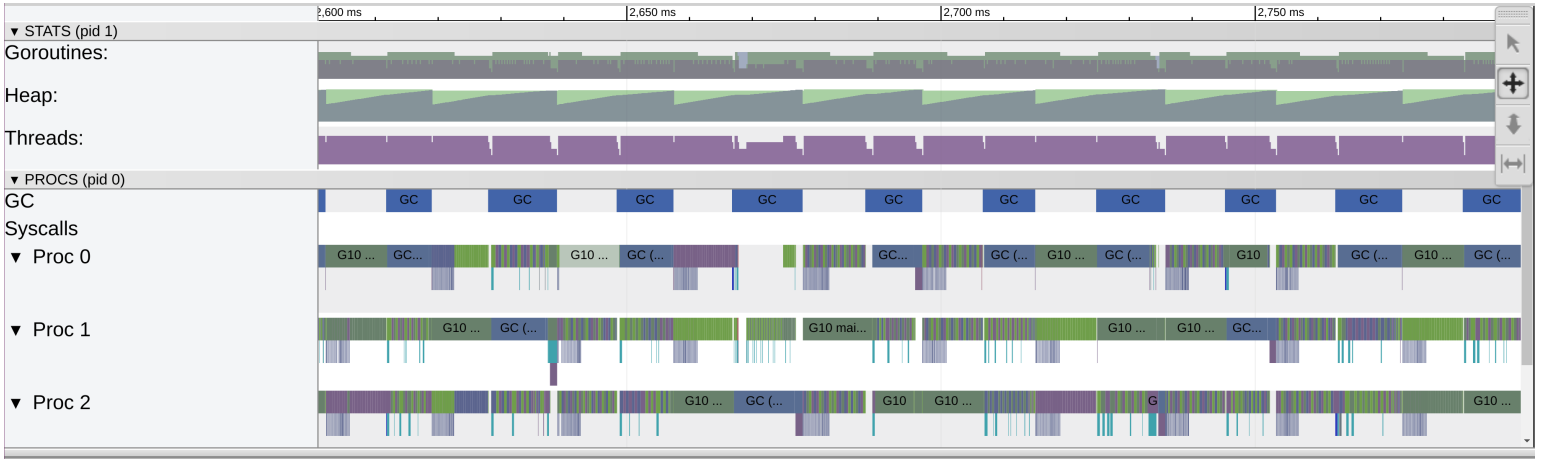


Figure 6.1: Base Goroutine Trace (NBody)

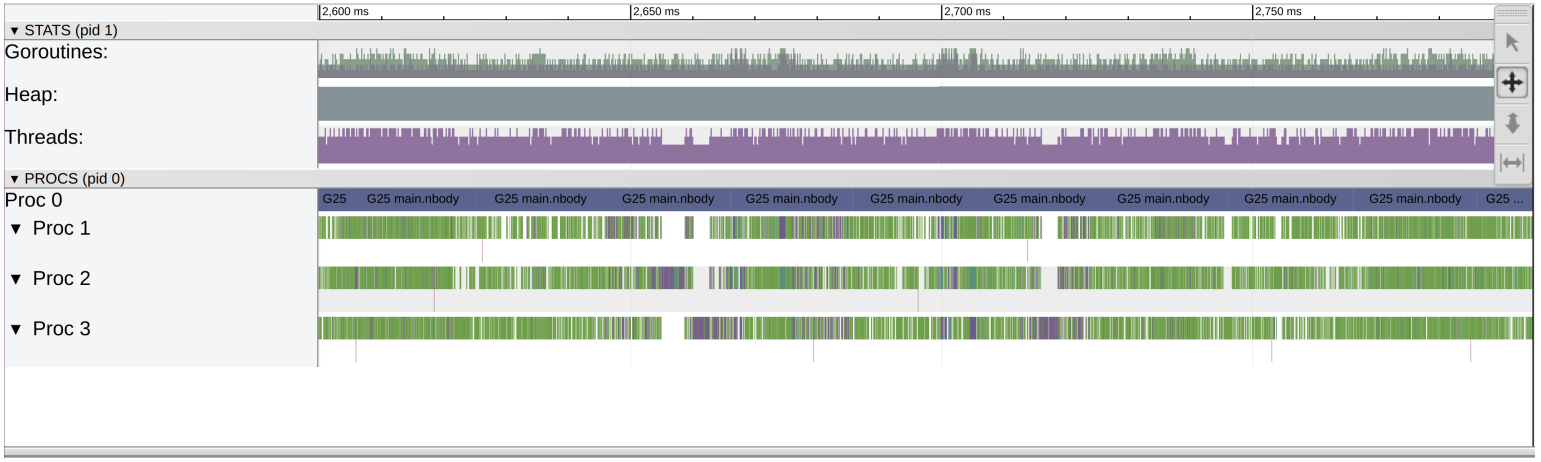


Figure 6.2: Proto-Actor with localmm Trace (NBody)

Figure 6.3: NBody Message Passing Graph

6.3 Producer-Consumer Benchmarks

The producer-consumer benchmark is designed to showcase how GoA holds up to other typical concurrency constructs across other languages. For most of the benchmarked languages, it is a simple producer-consumer implementation where the producer creates an object and queues it to a buffer, then the consumer removes this object from the buffer. The actor implementation must be a bit different as the way they implement concurrency is not as simple. The producers and consumers are actors. The actor's mailboxes are treated as the buffer. This means that the producer, and consumer communicate with messages. The benchmark is implemented so the producer first creates a message and sends directly to one of the consumers, round-robin style. This process works as the enqueue action. The consumer then accepts this message and begins the work. The consumer accepting this message from the producer acts like the dequeue. This allows us to test the entire GoA system, including allocation, message passing, reference counting, and garbage collection.

The producer-consumer setup is also used to run separate tests with the capability system and the traverse functions being the main focus. There are three runs in total. The base case is when the producer sends a message with the Traverse function (Section 5.2.1) explicitly declared, and no capability checks. This test is compared to the reflection traverse method without capabilities. The third test is with the explicit traverse functions and capability checks injected. This test allows the bench-marking of the overhead caused from the reflection traverse technique and the overhead from the runtime capability checks.

6.3.1 Benchmark Results

The producer-consumer benchmark is run with the following criteria:

- 10,000,000 work items / producer
- 320 producers
- 320 consumers
- 20 object buffer
- Results averaged among 3 executions

The results from the first experiments are in Table 6.6. The benchmark is run using goroutines with 3 different concurrency configurations: locks, channels, and monitors. Three

actor model configurations were tested: Pony, uC++, and Proto-Actor with localmm. The most contention came from the goroutines with channels. The remaining constructs performed similarly around 20-30s. The most interesting comparison is between Proto-Actor benchmarks, with the local memory manager compared to the base memory manager. The test with the local memory manager ran 44.93% slower than with the base manager. This overhead is due to the high rate of messages being processed by the middleware. As shown in the NBody message passing simulation in Figure 6.3, if the amount of messages is manageable (¡1,000,000), there is little to no overhead.

Setup	Time (s)
Go-routines w locks	27.47
Go-routines w channels	79.39
Go-routines w monitor	27.00
Pony	2.28
uC++ w actors	14.34
Proto-Actor base	28.69
Proto-Actor w localmm	41.58

Table 6.6: Producer-Consumer Simulation Results

6.3.2 Capability & Traverse Overhead Test

The producer-consumer benchmark provides a simple and easy to understand framework to test the capability system. Here, some work is added to the producer where it locally allocates an object. This object is passed to the consumer, where it undergoes all the necessary message passing processing. This step is where the Traverse function is benchmarked. It is then grabbed by the consumer, which performs a task on the received object. The capability system is benchmarked here as there needs to be a run-time check before using the received object. This experiment was tested with a logarithmic scale of items processed, 1 to 1000. The results of this experiment can be seen in Figure 6.4. The base case for this experiment is using the Traverse functions without capability checks. When using run-time reflection to traverse objects, there is an average of 21% overhead introduced. The run-time capability check begins to introduce more overhead as more items are checked by the consumer. With a single item per message the overhead is 14% and with 1000 items per message the overhead rose to 30%. This is simply due to the amount of objects being checked. The capability of every object needs to be checked to ensure memory safety, so the overhead rises as more objects are being passed and checked.

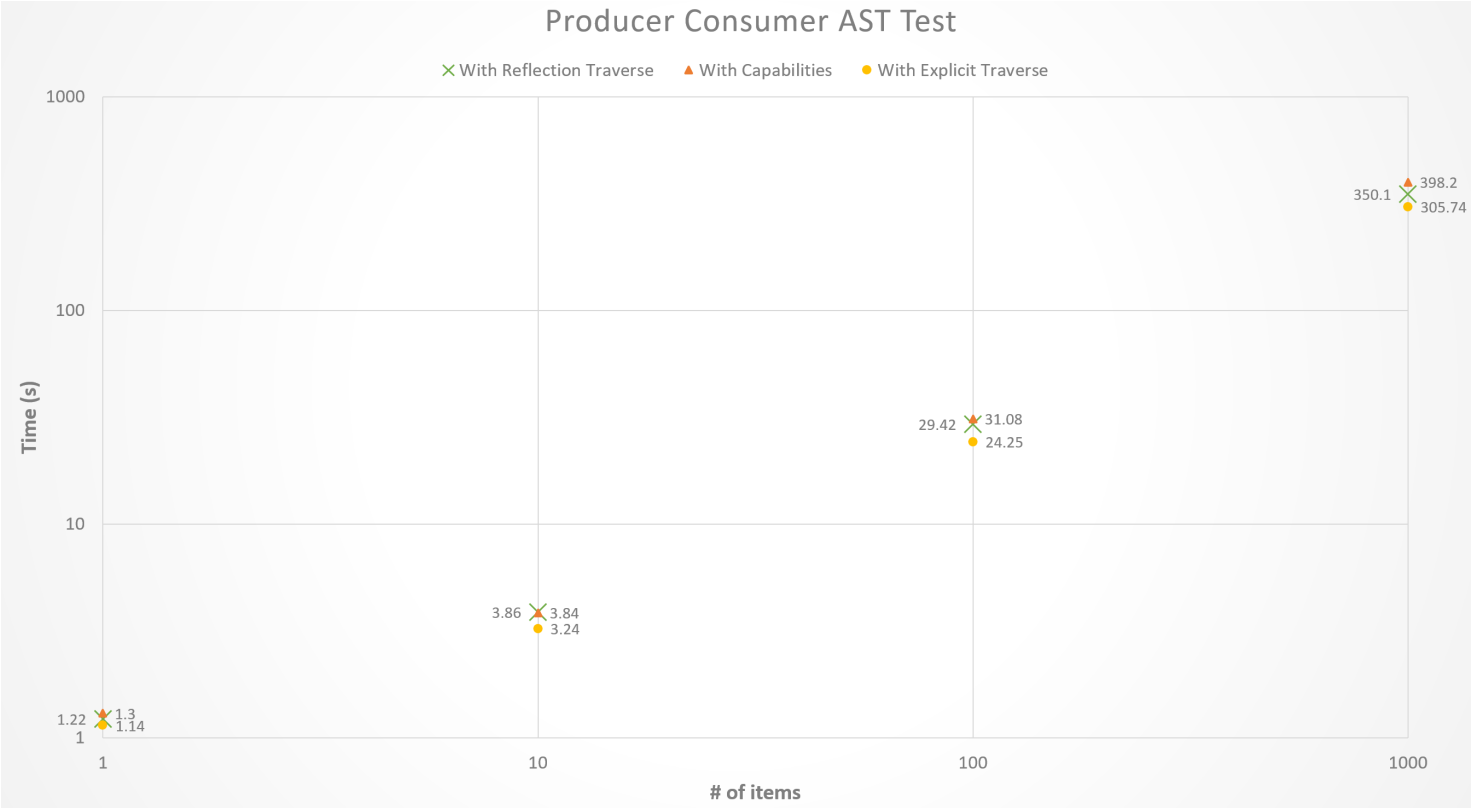


Figure 6.4: Producer-Consumer Capability/Traverse Benchmark

Chapter 7

Future Work

While GoA is able to deliver actors with locally managed memory and some memory safety runtime checks through the capabilities, it still has its limitations. There is some work envisioned for this project that simply could not be implemented in the given time frame. This section covers the potential improvements to GoA that can be implemented in the future of the project. The most important future work includes working on compiler integration and capability enhancements.

7.1 Compiler Integration

GoA is developed as an external library. This method provided a way to easily integrate the local memory system into any Go program already using Proto-Actor. It does however, come with its limitations. There are two significant improvements that would come with compiler integration of GoA. For one, it would allow the take over of the *new* keyword. Currently, allocating objects in GoA is messy. There are many disadvantages to the current process. Every allocation must be explicitly called through the actor context. Taking over the *new* keyword would enable implicit local memory allocation. This would alleviate much confusion from the user. All allocations would use the system and there would not have to be any thought into what objects should be allocated locally. This improvement would also mean the entry point of the program, the main method, would be able to be its own actor with locally allocated objects. Right now, there is no way to allow that so the only work around would be to immediately spawn a main actor and work from it. Second, it involves needing access to the actor object to reach its context, and the allocation call. This limitation means that every method needs a reference to this actor object. This can

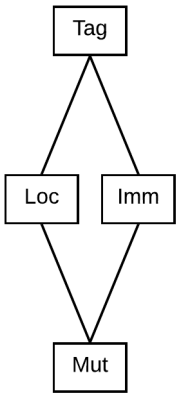
be a hindrance when calling many helper functions. As mentioned in the first point, this problem can be solved through implied scope when using the taken over *new* keyword.

Another important improvement that compiler integration would provide is the ability to statically enforce the capabilities. Pony is able to determine any potential data races at compile time because they can track capability information statically. This compile time enhancement would be a huge improvement over the runtime system as there would be no extra work involved, and would come at no extra cost to the runtime performance. Compile time enforcement also removes the need for the AST Translator Tool.

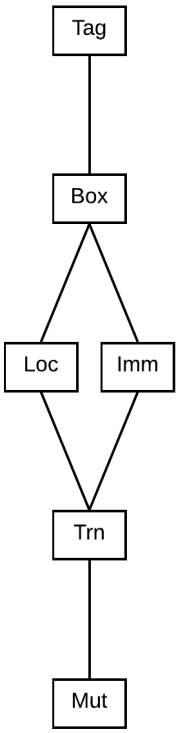
7.2 Enhanced Capability Features

GoA contains a subset of the capabilities that Pony provides with their language. This worked for GoA, as it took the necessary capabilities to provide safety checks when sharing and sending objects among actors. However, Goa does have some possible improvements that can be made by adding to what is already implemented. There are three improvements that stand out as obvious next steps for this project. One, adding the remaining capabilities that Pony offers. Currently, GoA covers objects that are local (Loc), and objects that are shareable but are immutable (Imm), isolated (Mut), or opaque (Tag). These cover sharing and local usage well enough, but there are more complex capabilities that add more complicated possibilities. Capabilities can be added that allow for giving read-only access to an object (box), while still allowing a user to modify said object or a transitional capability (trn) that can facilitate the transition from one capability to another.

Another two interesting aspects to add to the capability system are subtyping and converting. Each capability provides a different level of safety guarantees for a given object. It stands to reason that some capabilities should be able to be converted to more secure ones without sacrificing the guarantees. Figure 7.1a shows the potential subtype tree for the capabilities currently implemented and Figure 7.1b shows the potential subtype tree for the advanced capabilities planned. GoA would allow an Mut to move to Loc, or Imm and either of those to move to a Tag. This structure maintains all read/write access guarantees.



(a) Current GoA Capabilities



(b) Advanced GoA Capabilities

Figure 7.1: Potential Subtype Trees for GoA Capabilities

Chapter 8

Conclusion

This thesis presents several different additions to the Go language in the form of libraries and tools. Using the open-source proto-actor framework a new experience, GoA, is designed. GoA introduces a completely redesigned system for handling memory. Pairing with proto-actor allows the introduction of actor-local memory management. The memory manager is able to handle local allocations, reference counting for shared objects among actors, local garbage collection, and a way for the garbage collector and reference counter to implicitly traverse any object through reflection.

A way of ensuring the programs created with GoA are memory safe is also introduced. Using the capability system designed for Pony as inspiration, a subset is developed that works for GoA. Using the four capabilities, Loc, Tag, Mut, and Imm, paired with the runtime checks, provides a way to ensure memory shared among multiple actors are not subject to data races, or data corruption.

In order to reduce the overhead of manually injecting calls to the capability checker, an AST translator tool is implemented that automatically inserts these checks before every variable read and write. This tool is designed to be used before the code is sent to production. This way, the program can be checked for safety beforehand. This system allows the production code to be executed without the overhead of the checks involved while maintaining its memory safety status.

Finally, GoA is put through several benchmarks to determine its effectiveness compared to other similar languages. The benchmarks show that although the local memory management library introduced a slight overhead for small object allocations, the overall effectiveness of the system is proven. The NBody benchmark shows that the local memory system is able to increase the efficiency of low memory actors, by not interrupting

them with the garbage collection from other actors. The Producer-Consumer benchmark showcases how well GoA compares with other similar languages when running highly concurrent tasks. This test also displayed the overhead introduced from the reflection traverse technique, and the capability checks, and proved why they should only be run for testing and not production.

References

[1] Akka toolkit. https://akka.io/. Accessed: 2018-08.

[2] Data race detector. https://golang.org/doc/articles/race_detector.html. Accessed: 2018-08.

[3] Erlang. https://www.erlang.org/. Accessed: 2018-08.

[4] Golang. https://golang.org. Accessed: 2018-08.

[5] Ponylang. https://www.ponylang.org/. Accessed: 2018-08.

[6] Proto-actor framework. http://proto.actor/index.html. Accessed: 2018-08.

[7] Rust. https://www.rust-lang.org/en-US/. Accessed: 2018-08.

[8] Thread sanitizer algorithm. https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm. Accessed: 2018-08.

[9] What is pony. https://www.ponylang.io/discover/#what-is-pony. Accessed 2018-08.

[10] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[11] Peter A. Buhr. uc++ annotated reference manual. https://plg.uwaterloo.ca/~usystem/pub/uSystem/uC++.pdf.

[12] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[13] Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. Ownership and reference counting based garbage collection in the actor world. https://www.ponylang.org/media/papers/OGC.pdf.

[14] Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. https://www.ponylang.org/media/papers/opsla237-clebsch.pdf.

[15] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. https://www.ponylang.org/media/papers/fast-cheap.pdf.

[16] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. Orca: Gc and type system co-design for actor languages. https://www.ponylang.org/media/papers/orca_gc_and_type_system_co-design_for_actor_languages.pdf, 2017.

[17] Edsger W. Dijkstra. About the sequentiality of process descriptions. *E.W. Dijkstra Archive. Center for American History, University of Texas at Austin.* https://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html.

[18] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 1965. http://doi.acm.org/10.1145/365559.365617.

[19] Juliana Franco, Sylvan Clebsch, Sophia Drossopoulou, Jan Vitek, and Tobias Wrigstad. Correctness of a concurrent object collector for actor languages. http://mrg.doc.ic.ac.uk/publications/correctness-of-a-concurrent-object-collector-for-actor-languages/preprint.pdf. Accessed: 2018-08.

[20] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. http://dx.doi.org/10.1016/j.tcs.2008.09.019, 2009.

[21] Per Brinch Hansen. *Operating System Principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973.

[22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. http://doi.acm.org/10.1145/3158154, 2017.

[23] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. http://doi.acm.org/10.1145/2692956.2663188, 2014.

[24] Nicholas D. Matsakis and Aaron Turon. Rust book. https://doc.rust-lang.org/book/2018-edition/index.html. Accessed: 2018-08.

[25] Rob Pike. Another go at language design. Stanford EE Computer Systems Colloquium. Stanford University, 2010. https://www.youtube.com/watch?v=7VcArS4Wpqk.

[26] Dan Plyukhin and Gul Agha. Concurrent garbage collection in the actor model. https://dl.acm.org/citation.cfm?id=3281368.

[27] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996. Chapter 2.

[28] Wikipedia. Actors. https://en.wikipedia.org/wiki/Actor_model#Early_Actor_programming_languages. Accessed: 2018-08.