Imperial College London
Department of Computing

Pony: Co-designing a Type System and a Runtime

Sylvan W. Clebsch

October 2017

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of Imperial College London and the Diploma of Imperial College London

Declaration

This thesis and the work presented within are my own, except where acknowledged otherwise.

Sylvan Clebsch

Abstract

We have developed a new programming language, called Pony, that allows the user to easily write fast, safe, parallel programs. The focus of this work is using a powerful and novel static type system to guarantee properties of a program in order to eliminate many runtime checks. This includes eliminating all forms of locking, as well as enabling a novel fully concurrent garbage collection protocol for both objects and actors that has no stop-the-world step, while also having no read or write barriers. Essentially, the type system is used to enable not just safer computation but faster computation.

The core type system feature that enables this is *reference capabilities*. These are a system of type annotations that guarantee data-race freedom even in the presence of asynchronous messages that can contain mutable data. Reference capabilities show that concepts such as isolation and immutability can be derived rather than being treated as fundamental.

Properties of the type system and operational semantics, such as *atomic behaviours* and data-race freedom are leveraged to allow actors themselves to be garbage collected when it can be proved they will never again have messages in their queue. Message-based Actor Collection (*MAC*) achieves this using only message passing, without the need for any other form of thread synchronisation or coordination, using a system of deferred, distributed, weighted reference counting in which reference counts do not reflect the number of reachable paths in a heap. This requires a cycle detection actor to be able to collect isolated cyclic graphs of actors. To do this with no stop-the-world step, a novel and inexpensive CNF-ACK protocol is used to determine that the cycle detector's view of the actor topology was the true actor topology when the cycle was detected.

Guaranteeing data-race freedom allows the language to use a novel fully concurrent garbage collection protocol. This protocol allows each actor to collect its own heap independently and concurrently, while still allowing objects to be sent by reference in asynchronous messages (zero-copy message passing). This is achieved using Ownership and Reference Counting for Actors (*ORCA*), a protocol derived from *MAC* that allows passive objects shared across *actor-local heaps* to be garbage collected with no stop-the-world step.

To validate these ideas, we have written a complete runtime (including a work-stealing scheduler, garbage collector, memory allocator, message queuing, asynchronous I/O, and more) and an ahead-of-time optimising compiler. The language is open source.

Acknowledgements

Although I developed the type system, operational semantics, and garbage collector protocols, and implemented the compiler and runtime, many other people have contributed to Pony. Sophia Drossopoulou, my advisor, has contributed to every part of the formalisation of Pony, especially the proofs, and has literally taught me everything I know about formal methods, as well as being a constant source of help and inspiration. This project could not possibly have happened without her. Other key contributors have been Sebastian Blessing (masters thesis on distributed Pony), Andy McNeil (implementation work on the compiler), Juliana Franco (formalisation of object garbage collection), Luke Cheeseman (value-dependent types), George Steed (formalisation of type expressions), and the SLURP group at Imperial College (invaluable feedback and discussion). Also of special note are Tobias Wrigstad, Dave Clarke, and the Upscale Project, who have bravely used the Pony runtime for their own language, Encore. Additional special thanks are due to Constantine Goulimis and Justin Fry.

More recently, the Pony open-source community has been amazing, including (in no particular order) Sean Allen, Joe Eli McIlvain, Benoit Vey, Theo Butler, Vid Jain, Chuck Blake, Perelandric, Reini Urban, Gordon Tisher, Dipin Hora, Jonas Lasauskas, John Mumm, Andy Turley, George Marrows, Darach Ennis, Tommy McGuire, Carl Quinn, Malthe Borch, Kevin Cantú, Markus Fix, Chris Double, Dan Connolly, and many others. I owe them all a beer, and I owe anyone I've failed to mention two beers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the end of the free-lunch era of ever increasing single-core performance, parallel computing has become a mainstream requirement. The multi-core era requires high-performance programs to be highly parallel: Moore's law is still holding, but the increase in transistor count is resulting in an increasing number of cores rather than faster individual cores.

To borrow the terminology of distributed computing, this shift, from *scaling-up* processor performance to *scaling-out* processors for overall system performance, has been problematic for software developers. Algorithms and patterns that were efficient on single-core or small core-count machines may become inefficient when asked to scale-out further, much as patterns that are efficient on a single node may be infeasible in a distributed context. The comparison with distributed computing is particularly apt in that cores on a single node share hardware resources, such as memory, via message passing. In effect, a multi- or many-core node can be considered as a distributed system with fast transport links.

In this context, a programming language that draws from approaches that may have been viewed as more suited for distributed systems, such as the actor-model and capabilities security, can be designed to scale efficiently as single-node core counts increase. This thesis introduces Pony, a programming language designed for this purpose. Pony is an actor-model, capabilities secure language, that allows the user to easily write fast, safe, parallel programs.

```
actor Main
  new create(env: Env) =>
    env.out.print("Hello, world.")
```

Figure 1.1: "Hello, world." implemented in Pony.

Pony uses a powerful and novel static type system to guarantee properties of a program in order to eliminate many runtime checks. This includes eliminating all forms of locking, as well as enabling a novel fully concurrent garbage collection protocol for both objects and actors that has no stop-the-world step, while also having no read or write barriers. Essentially, the type system is used to enable not just safer computation but faster computation.

1.1 Motivation

Pony is a product of my own frustration with the tools available to me. I have spent much of my professional career writing, or managing programmers that were writing, concurrent code in systems programming languages, primarily multi-threaded C/C++. Over and over again, we ran into memory errors. There were of course the usual problems with dangling pointers and memory leaks that garbage collectors are designed to solve, but the problems that consumed the most time and effort in debugging, or required more fundamental architecture changes, were problems with data-races and causality. Programmers would think about program architecture in terms of ownership and sharing between concurrent units of execution, but, without tooling support, enforcing this discipline was extremely difficult.

The motivation for using multi-threaded C/C++ was peformance, and so moving to a programming language or library that traded performance for safety was problematic. In the early 2000s, I developed a flight simulator engine written in C/C++ that used Lua as an extension language, integrating the Lua garbage collector with C++ reference counting. I built an asynchronous message queuing system for doing physics simulation that allowed the physics code to be written in Lua while the solid-body integration and graphics code was written in C++. This was useful, and allowed more productive development of physics code in a safe, garbage collected environment, but it still had problems. Message passing required copying data, the Lua code itself was not concurrent, and messages were not causally consistent. For example, a design flaw in the physics code for calculating the power output of an internal combustion engine resulted in the manifold pressure being calculated based on how the *future* exhaust velocity drove the turbocharger, which resulted in the total energy in the system increasing and the simulation blowing up. Once the flaw in the way messages were propagated was discovered, this was trivial to fix, but discovering that this was due to a causality violation took months of programmer time.

Similarly difficult problems occured with data-races in a different context. In the early 2010s, I ran European electronic trading infrastructure for a major investment bank. My team developed infrastructure code for low-latency (a few microseconds) high-throughput (hundreds of thousands of messages per second) applications. To support this, I wrote a library for writing actors in C/C++. It supported zero-copy causal messaging, had a work-stealing scheduler, and was used to develop and deploy some high-performance applications. However, programmers would sometimes send a pointer from one actor to another, convinced that they had accounted for ownership or had correctly controlled for immutability, and it would turn out that they had not and a data-race existed. Without enforcement for data-race freedom, these bugs were extremely difficult to track down. Sometimes programmers would add non-message-passing based synchronisation, such as locks or lock-free data structures, to control data-races where ownership or sharing discipline had failed. This usually resulted in subtle deadlock conditions, which often appeared only occasionally and only under high load on production systems, but even when such bugs were not encountered (which is not to say they were not present) there was a performance cost.

There are existing languages and tools that would solve some of these problems. Clearly, there are many interesting garbage collectors that can provide a memory-safe, if not data-race free, programming environment. But most come at a cost in the form of a stop-the-world phase or significant mutator performance costs. Erlang provides an actor-model, data-race free programming environment, but it is neither causally consistent, due to allowing pattern matching on message queues, nor is the sequential code performance competitive with C/C++. There are existing data-race free type systems, including type systems that provide isolation and immutability, but they are not leveraged by a strongly integrated approach to concurrency and garbage collection to improve performance.

1.2 Co-design

Designing Pony has involved a tight feedback loop between the type system and the runtime. We began with runtime requirements, such as no-stop-the-world garbage collection, an efficient work-stealing scheduler, and a lock-free runtime implementation. These requirements informed the design of the type system, necessitating a data-race free type system with strong aliasing guarantees. The existence of a data-race free type system allowed further refinement of the runtime, enabling features such as zero-copy message passing across per-actor heaps, a more efficient underlying memory allocator, and a zero-copy asynchronous I/O model.

This co-design process has continued as both the type system and the runtime have developed, and has been one of the most rewarding aspects of this work.

In the language of participatory design [64], neither the type system nor the runtime is an isolated system.

1.3 Content and Contribution

This is an overview of the content of this thesis and the contributions made.

Chapter 2 This chapter explains the design decisions made during the development of Pony. We discuss the decision to build a single model for concurrent and distributed execution based on the *introduction requirement* [3]. We examine the tools for reasoning and correctness provided by Pony, specifically the use of the novel data-race free type system described in chapter 4, logically atomic behaviours, capabilities-security, and causal messaging. We also present some of the decisions made in the interest of performance, such as modelling behaviours as methods, exceptions as partial functions, the use of a single compilation unit, and maintaining C ABI compatibility. Finally, we introduce the development philosophy that has guided us, "get-stuff-done", with apologies to Richard Gabriel.

Chapter 3 This chapter provides a syntax and operational semantics for Pony, in the form of a small-step operational semantics for both concurrent and distributed execution. A distinction is drawn in the semantics between environments where causal order is cheaply guaranteed (i.e. shared memory concurrent systems) and environments where only pairwise FIFO ordering can be cheaply guaranteed (i.e. distributed systems).

Contribution: a unified actor-model operational semantics for concurrent and distributed execution.

Chapter 4 In this chapter, we develop the concept of *reference capabilities*, a novel approach to static data-race freedom based on a matrix of *deny properties*. The concepts of *aliased and unaliased types*, *viewpoint adaptation*, *safe-to-write*, *capability recovery*, and *capability compatibility* are introduced. The type system is explained both formally and informally, including descriptions of well-formedness and consistent heap visibility, requiring an interesting treatment of *temporary* (i.e. unnamed) values, and a proof of soundness is provided.

Contribution: a novel type system for data-race freedom based on *reference capabilities*, wherein concepts such as isolation and immutability are derived rather than fundamental, the use of *aliased and unaliased types*, and the formalisation and proof of soundness.

Chapter 5 This chapter details *MAC*: message-based actor collection, a high-performance no-stop-the-world protocol for garbage collecting actors. The operational semantics from chapter 3 is extended to express a formal model of *MAC*. Completeness and robustness are discussed, as well as modifications that would allow *MAC* to collect distributed actors.

Contribution: a novel no-stop-the-world garbage collection algorithm for actors and its formalisation.

Chapter 6 In this chapter, the *MAC* algorithm from chapter 5 is used as a basis for *ORCA*: ownership and reference counting for actors, a no-stop-the-world garbage collection protocol for passive objects that allows actors to communicate using zero-copy message passing for both mutable and immutable messages. The formal model in chapter 5 is extended to cover passive object collection, and completeness, robustness, and distribution are discussed.

Contribution: a novel no-stop-the-world garbage collection algorithm for zero-copy message passing and its formalisation.

Chapter 7 This chapter concludes the thesis, and provides a brief summary of some of the opportunities for further work.

Appendix A This chapter offers a detailed description and explanation of the implementation of the runtime library. Topics covered include the memory allocator, size-classed per-actor heaps, message queues, actors, the tracing garbage collector, the cross-actor garbage collector, the actor garbage collector and cycle detector, finalisation, the work-stealing scheduler, and asynchronous I/O.

Contribution: a complete runtime implementation.

Chapter 2

Language Design Decisions

Pony is a language predicated on the idea that a sufficiently powerful static type system can be leveraged to write a faster runtime. In particular, such a type system could make guarantees that result in eliminating many dynamic checks, not just related to dynamically checking data types, but also eliminating checks that would otherwise be required for safe concurrency, garbage collection, and security.

2.1 Single Model for Concurrent and Distributed Execution

One of the driving principles behind the design of Pony has been the desire for a single model for both concurrent and distributed execution, as described in chapter 3. As a result, the communication mechanism between units of execution cannot rely on shared mutable state, since such shared memory is not available in a distributed setting. In addition, the communication mechanism cannot be synchronous, as relying on synchronous communication in a distributed context can introduce severe performance problems. These are also important considerations when ensuring that maximum use is made of the hardware available on a single node. For example, synchronous messaging can introduce deadlocks and priority inversion, and shared mutable state can cause data races and put a heavy load on cache coherency hardware.

This is a different approach than languages such as C/C++, C#, Java, Javascript, Python, and other languages that treat both concurrency and distribution as library-level rather than language-level features. Such languages are in some sense sequential-first. Pony instead takes a concurrent-first approach that is more in the style of Erlang, in which concurrency and distribution are language-level features that inform the design of the sequential portion of the language. As such, problems endemic to sequential-first languages when concurrency is introduced, such as non-composable concurrent libraries, garbage collectors that must cope with shared mutable state, deadlocking, and data-races, are avoided by design.

Like Erlang, Pony uses asynchronous message passing for communicating between units of execution. This decision, combined with a desire to provide dynamic topology and object capabilities, lead to the decision to use actors as the unit of execution. The actor model has been implemented in many ways, but at its core it has three fundamental requirements [3, 4]:

1. An actor can send messages to other actors or itself.

2. An actor can create new actors.

3. An actor can choose the way in which it will respond to the next message it receives.

These requirements are intentionally weak. For example, while the model does require that messages are eventually delivered, it does not require a delivery order. Similarly, while the third requirement allows actors to encode state, the model does not require an implementation to encode state in any particular way. For example, encoding state via recursion or via mutating object fields are both allowable strategies.

On the other hand, there are subtle implications of these three requirements. For example, since an actor can send a message to itself, and message delivery is guaranteed, messages must be buffered, as otherwise an actor would deadlock when attempting to send itself a message. Interestingly, while real world constraints limit the size of a message buffer, the model implicitly requires that buffer space is logically unbounded: any upper bound devolves to the possibility that an actor will be blocked if it tries to send a message to itself, because its message buffer is full and it cannot clear it. That is, the actor can neither enqueue the message and continue execution, nor can it dequeue messages to create space, as the actor must not interleave message handling executions.

Another subtle implication is that there can be no shared mutable state across actors. This follows from the requirement that an actor α can only send a message to another actor α' in response to some message m if:

1. α' was known to α before m was received, or

2. α' is contained in m, or

3. α created α' while handling m.

This is the *introduction requirement*, as detailed in Gul Agha's Ph.D. thesis [3]: an actor α can only send a message to another actor α' if they have been properly introduced. Shared mutable state would allow an actor to send messages without prior introduction: namely, α could read the address of α' from some shared variable that a third-party actor modified, without α either receiving α' in a message or creating α'.

The implementation of the actor model in Pony is intended to satisfy all of these aspects of the actor model, including the more subtle implications.

2.2 Tools for Correctness and Reasoning

Pony makes additional guarantees that extend the fundamental requirements of the actor model:

1. Both mutable and immutable state are supported, both within an actor and in messages between actors, as described in chapter 4.
2. Data-race freedom, including mutable state isolation, is guaranteed statically, as described in section 2.2.1 and chapter 4.
3. Message handling is *logically atomic*, as described in section 2.2.2 and chapter 4.
4. The language is *capabilities secure*, as described in section 2.2.3 and chapter 4.
5. Within a node, message delivery is *causal*, and across nodes, message delivery is pairwise FIFO ordered, and can be made optionally *causal*, as described in section 2.2.4 and chapter 3.
6. Both actors and objects are garbage collected without requiring coordination outside of the underlying message passing system, as described in section 2.2.5 and chapters 5 and 6.
7. Within a node, messages are *zero-copy*, as described in chapter 6.

Each of these guarantees is made possible due to the interaction between the type system and the runtime. In some cases this interaction is obvious, such as the semantics of the language providing no global variables, which is important for capabilities security, and in other cases it is more subtle, such as data-race freedom being important for logically atomic message handling, which is in turn important for both causal messaging and garbage collection.

It is important to note that, while Pony makes some guarantees that are useful for ensuring correctness and reasoning about programs, Pony is not a tool for formal verification, nor is the implementation of the runtime itself formally verified. It would be extremely interesting to apply techniques used in formal verification tools, such as Dafny and F*, to allow Pony to function as its own proof assistant, and to implement a formally verified runtime (perhaps using an extended Pony that provides for formal verification), but that must be left for future work.

2.2.1 Static Data-Race Freedom

If the type system can statically ensure data-race freedom, the runtime and every program written in the language need not contain dynamic locking mechanisms. This is a significant performance gain, and the impact is felt not just in computational performance, but, as we will see later, also in parallel scalability and garbage collection.

To ensure data-race free concurrent execution, it is sufficient to enforce a single principle: if a unit of execution (whether it is a thread, a process, or any other implementation) can write to a data structure, no other unit of execution can read from that data structure. This has a natural corollary: if a unit of execution can read from a data structure, no other unit of execution can write to that data structure.

Enforcing this property on individual memory locations, as opposed to complete data structures, is insufficient, as it could result in the data structure as a whole being inconsistent, even though the individual reads and writes were safe. For example, a data structure comprised of a linked list and a count of the number of nodes in the list cannot be safely updated by controlling access to individual memory locations: either the count or the list itself would have to be changed first, resulting in a period where the count does not accurately reflect the number of nodes in the list.

As a result, in order to enforce this property dynamically, access to a data structure must be controlled with some form of runtime locking mechanism. For example, the counted linked list could also include a mutex that must be acquired in order to read from or write to the data structure. However, this does not address safe access to the contents of the list nodes. If two units of execution each read the first element of the list, two threads now have access to the same contained data. In order to make the list as a whole data-race safe, every data element that could be put in the list must also be protected by a mutex. This problem is recursive: the data elements themselves may contain references to data structures that must also be lockable.

This is the fine-grained locking problem. There are many approaches to mitigating this problem, particularly in database research. The most common approach is to rely on a programer's domain knowledge to correctly guess what the minimum level of locking is to ensure correct behaviour. However, all of these approaches are domain specific, are subject to programming errors, and rely on locking mechanisms that are expensive on shared memory systems and prohibitive in distributed systems.

A type system that allows only immutable data types solves the data-race problem by disallowing all mutation. This is an approach favoured by functional programming languages. Since no unit of execution can write to the data structure, it is trivial to guarantee without dynamic checks that the program is data-race free. However, some algorithms are faster to compute or easier to implement over mutable data structures than immutable ones. As such, a type system that can express data-race free mutable data types might offer advantages over a type system that can express only immutable data types.

Purely static data-race free mutability has been expressed several ways, including via ownership types, fractional permissions, and uniqueness and immutability type systems. In Pony, the type system incorporates *deny capabilities*, expressed as *reference* capabilities. Deny capabilities are a form of uniqueness and immutability type system influenced by both *object capabilities [48]* and *deny guarantee reasoning* [21].

Deny capabilities describe what other aliases are *denied* by the existence of a reference. Furthermore, they distinguish between what is denied *locally* (i.e. aliases reachable via the same actor) and what is denied *globally* (i.e. aliases reachable via other actors). The result is a matrix of *deny properties*, with notions such as isolation, mutability, and immutability all being derived from these deny properties. What aliases to the object are allowed to do is explicit rather than implied, whereas what the reference is allowed to do is derived.

Importantly, reference capabilities obey simple type checking rules that allow for reasoning about data-race freedom to be done locally, both by the programmer and by the compiler, rather than requiring global knowledge about a program. That is, the existence of some reference capability in some lexical scope explicitly indicates that:

1. No other alias to that object can exist that violates that reference capability's deny properties.

2. No other alias to that object can exist whose deny properties would be violated by that reference capability.

Such reasoning is both local (i.e. can be done without reference to implementation details outside of the function being reasoned about) and modular (i.e. reasoning about a function once is sufficient regardless of the context in which that function is used), even in the presence of type variables, i.e. generic types.

The Pony type system accomplishes this through a combination of *viewpoint adaptation* [19], which determines a reference capability for a path based on all elements of that path, rather than simply the reference capability of the final element of the path, a separate and more liberal treatment of the types of temporary values (that is, the result of an expression which is then used in another expression, without establishing a path to the value by assigning it to a field or a stack variable), and *aliased and unaliased* types, which provide a way to refer to the alias of a reference capability (the type that results when a new path to a reference capability is created) and the unalias of a reference capability (the type that results when a path to a reference capability is removed, either through destructive read or by explicitly removing a variable in the lexical scope).

2.2.2 Logically Atomic Behaviours

The code that an actor executes upon receipt of some message m is termed a *behaviour*. In Pony, every behaviour on every actor is *logically atomic*.

When a behaviour begins execution, it has a set of object and reference capabilities defined by the actor's state and the contents of the message m that has been received. In Pony, an actor's state is represented as fields of the actor, and the contents of the message m are represented as arguments to the behaviour, in much the same way as an object-oriented method call has a receiver and a set of arguments. This initial state of capabilities is the total set of capabilities that the behaviour will have. No new capabilities that are not present when the behaviour begins executing can be acquired. Note that the behaviour may still create new actors and objects, but these new actors and objects do not represent an expansion of the set of capabilities: they must be created with the capabilities that were initially available.

As a result, a behaviour cannot witness heap mutation that the behaviour does not itself perform. This is a result of combining the reference capability type system, which statically guarantees that if an actor can read from a memory address then no other actor can write to it, with the guarantee that the initial set of capabilities available to a behaviour cannot be expanded.

It is interesting to note that, while the set of capabilities cannot be expanded, it is possible for it to be contracted. Specifically, a capability that allows mutation of an address can be *sent* to another actor, and the data-race free type system guarantees that the sending actor retains neither the ability to read from nor write to that address. As a result, it is possible for a behaviour to have the capability to read or write some memory address, to give up that capability, and for the contents of that address to be mutated by some other actor while the original behaviour is still executing. However, in this case, the behaviour will not be able to expand its set of capabilities to once again include the capability to read from that memory address. As a result, the behaviour will not be able to witness the heap mutation.

2.2.2.1 Reasoning About Atomic Behaviours

Requiring logically atomic behaviours is a powerful tool for reasoning about actors. Each behaviour can be considered as a separate sequential program, with a separate heap, that operates on some set of preexisting state (fields of the actor) and a set of input values (behaviour arguments). The resulting program will produce output in four forms:

1. Sending a finite number of messages to other actors or itself.
2. The creation of a finite number of new actors.
3. Changes to the executing actor's state that will be propagated to the next behaviour on that actor.
4. Side effects in the form of output to some device, for example a file system, network, display, etc.

Note that these possible outputs correspond to the fundamental requirements of the actor model, with the addition of device output.

Reasoning about individual behaviours as separate sequential programs removes the need to consider concurrency within a behaviour. This significantly simplifies the reasoning process. However, it does not remove the need to consider concurrency in a program as a whole. While interleaved concurrent reads and writes to memory locations no longer must be considered, message ordering remains non-deterministic where a causal relationship is not present, which can introduce program executions that exhibit race conditions at the logical level while still being data-race free.

2.2.2.2 Asynchronous Function Results

There are three primary techniques for returning asynchronous results from asynchronous functions. The first is blocking futures, where an asynchronous function returns a value that may not yet be the result, but will be populated with the result in the future. The caller can then choose to block execution at some point until the result has been populated. The second is non-blocking futures, also referred to as asynchronous await. This approach captures the stack as a continuation that will be executed when the asynchronous result is available, and allows the caller to continue handling other messages in the meantime. The third is promises, which are similar to non-blocking futures, but instead of capturing the stack as a continuation, closures are explicitly specified. Promises allow more than one receiver of an asynchronous result, and also allow exceptional behaviour to be propagated in the form of separate closures for successful and unsuccessful execution.

These approaches are closely related, and each can be used, with some effort, to implement the others. Pony uses promises rather than blocking or non-blocking futures. This decision is motivated by reasoning rather than performance: logically atomic behaviours require asynchronous results to be returned via promises rather than futures in order to avoid expanding the available set of capabilities during execution of a behaviour.

Arguably, logically atomic behaviours could be considered an implicit requirement of the actor model. Specifically, the *introduction requirement* that implies that there can be no shared mutable state could be read to also imply that a memory location that is not readable when a behaviour begins executing must not become readable during execution. This is because a newly readable memory location could contain an actor address previously unknown to the actor executing the behaviour.

On the other hand, if the event of a previously unreadable memory location becoming readable is considered to be a new message m', with the response to m being considered finished and any remaining code to execute being considered a response to m', then the introduction requirement is not violated. Such an approach treats messages as more abstract than actual messages in a mail queue: any expansion of the set of capabilities is treated as the logical arrival of a new message. This would allow, for example, blocking futures, which prevent shared mutable state but allow expanding the set of capabilities.

Awaiting an asynchronous result, i.e. using a non-blocking future, pauses a behaviour until an asynchronous result is returned, but allows the actor to continue receiving messages and executing their associated behaviours in the meantime. Effectively, the behaviour stack is captured as a continuation and resumed when the asynchronous result becomes available. This approach also results in non-atomic behaviours, as the actor may have been given new capabilities by intervening messages, resulting in the continuation being executed with a larger set of capabilities than existed when m initially arrived.

Both blocking and non-blocking futures can be encoded with logically atomic behaviours, either with or without promises, but doing so can result in code that is difficult to understand. A non-blocking future (asynchronous await) can be encoded as a closure that will be executed upon receipt of a message containing the expected result. This is continuation-passing style for asynchronous messages. Encoding a blocking future involves combining continuation-passing style with a queue to store messages that arrive before the message containing the expected result. The processing of these queued messages is delayed until after the expected result is received and the associated closure executed. In both cases, the programmer writes code to explicitly do what a compiler transformation would implicitly do.

Using promises, which are implemented in Pony as part of the standard library rather than as a language feature, preserves the introduction requirement without requiring the fulfilment of a future to be treated as a equivalent to a new message. The semantics of both blocking and non-blocking futures are still possible to achieve, but the underlying complexity of these approaches is directly exposed to the programmer.

2.2.3 Capabilities Security

A capability is an unforgeable token that both designates an object and provides access to that object. Capabilities security uses the possession of such tokens to control access to resources. Capabilities have a long history, stretching from operating system research (such as KeyKOS [31], EROS[61], Coyotos [62], and seL4 [37]) to programming languages (such as Joule [68], E [47], AmbientTalk [70], Caja [46], and BitC [63]).

Capabilities are usually compared with access control list systems, where some arbiter that holds a mapping of roles to permissions must be traversed in order to gain access to a resource. While the two approaches are both security mechanisms, capabilities security is a more interesting match for a programming language, as a language can express unforgeable tokens easily and naturally.

Pony uses the *object capability* [48] model to express capabilities security. This was an easy choice, as in the absence of hardware support for capability-based addressing, the object capability model is the most efficient way to implement capabilities security. The alternative would be using a C-list, which is a protected data structure that maps a process identifier and integer handle to a capability. In operating systems such as KeyKOS, the C-list is maintained by the kernel. The use of a C-list to express capabilities at the language level without an object model is simply too expensive; it would require a mapping of actor identifier and integer handle to capabilities, plus a mechanism for delegating capabilities to other actors. In contrast, the object capability model requires no runtime mechanism.

Adopting the object capability model had several follow-on effects. Most importantly, it required Pony to express *identity* as a first-class concept. An object's identity serves as the unforgeable token. In a memory safe language, such a mapping of identity to capability is a natural fit, as object identities cannot be manufactured or forged. The lack of mechanisms such as pointer arithmetic to manipulate identities allows those identities to be issuable only via controlled mechanisms. In Pony's case, this is via constructors.

The object capability model in Pony is designed to take advantage of type safety as well as memory safety. For example, a type with a private constructor is a capability that can only be constructed from within a certain package, providing a lexical bounds on capability creation.

2.2.4 Causal Messaging

The actor model requires that messages are delivered, but makes no requirement as to the order of delivery. Many implementations of the actor model effectively provide pairwise FIFO ordering of messages between actors, because the implementation of a message queue, even in a distributed setting, can make this relatively inexpensive to provide.

In addition, some implementations of the actor model provide *causal messaging* between actors running on the same physical hardware (node), again due to the availability of an inexpensive implementation. Here, causal messaging refers to a message order guarantee wherein an *effect* (a message) does not get delivered until after all of its *causes*, where the causes of a message are every message that the sending actor has previously sent or received. Such causality is transitive.

Pony also provides causal messaging between actors running on the same node. In addition, it was originally believed that message-based garbage collection would require causal messaging across nodes in the distributed setting. To address this, *tree-structured networks* were used to guarantee causality with no space overhead, but with $O(log\, n)$ latency overhead due to messages traversing the tree. This is a significant improvement compared to other causal messaging techniques, but still represents both a latency cost and a possible message bottleneck.

To address these problems, the garbage collection algorithm was extended to work with only pairwise FIFO ordering, which can be implemented in a distributed setting with no overhead, eliminating the additional latency and the possible bottlenecks for system-level messages.

However, distributed message causality can be useful at the application level. Using multiple tree-structured networks (overlaid on the actual underlying network), it is possible to build any number of *causality bubbles*, over which messaging, whether on a single node or across nodes, maintain causality. These causality bubbles can each have a different topology over the same nodes. As a result, the $O(log\, n)$ latency cost is still born, but bottlenecks are reduced because each bubble has a different tree-structure.

It is interesting to note that logically atomic behaviours, and thus deny capabilities for data-race freedom, play an important part in causal messaging. The ability to expand a capability set to examine some modified memory location during the execution of m, and therefore treating the modified memory location as a new message m', effectively allows an actor to reorder its message queue. Such reordering can lead to causality violations. Any data race can cause such a violation by propagating information outside of message passing system, but it is also possible to have causality violations when data-race freedom is preserved if behaviours are not logically atomic.

For example, suppose actor α_1 sends messages m_1 and m_2, in that order, to actor α_2. In this case, m_1 is a cause of m_2. Now suppose that while handling m_1, α_2 sends message m_3 to α_1 and receives the result as a blocking future, such that α_2 will treat the result generated by α_1 as a new message m'_1, handling it as a continuation of m_1 and, importantly, *before* handling m_2. When α_1 then receives m_3 and generates a result, it populates the future that α_2 has blocked on. This result is a message m_4, and its causes are m_1, m_2 (the messages α_1 has previously sent) and m_3 (the messages α_1 has previously received). Now α_2 unblocks, as the future has been fulfilled, and handles the result m_4. At this point, α_2 has handled m_1, is handling m_4 next, and will handle m_2 in the future, resulting in a causality violation: α_2 handles the effect m_4 before the cause m_2.

2.2.5 Message-based Garbage Collection

Pony's garbage collector uses the underlying message-passing architecture as its only form of concurrent coordination. This is important for performance, but it is also important for correctness and reasoning.

Most implementations of the actor model require actors to be *explicitly terminated*. That is, the programmer must manually manage actor lifetime, and send an actor that is no longer required a message requesting termination. This is effectively equivalent to manual memory management, with all of the attendant pitfalls. An actor that is prematurely collected may result in messages being ignored, or delivered to the wrong actor, or cause the program to terminate. An actor that is never collected will continue to consume memory and possibly CPU time.

Message-based garbage collection in Pony allows both actors and passive objects to be garbage collected, in both the concurrent and the distributed setting. As a result, the programmer not only need not manually manage actors, but also need not write code to defend against the possibility of a message being sent to a prematurely terminated actor. This both reduces the burden on the programmer and also allows for truly composable actor model subsystems, since the details of actor lifetime management do not have to be communicated between layers or components.

The same protocol that is used to garbage collect actors is also used to determine quiescence amongst scheduler threads on a single node and amongst nodes in a distributed environment. As a result, programs themselves also need not be explicitly terminated. Instead, a program terminates when there is no more work to be done and no more work can be created, without explicitly exiting a top level function or any other such explicit mechanism. As a result, no unit of execution is the leader, in the sense that no unit of execution can terminate and result in the termination of a program that may or may not have finished processing all pending and future work.

Message-based garbage collection relies on both the data-race free type system and causal messaging. Static data-race freedom allows coordination-free traversal of of data structures, both when sending and receiving messages and when collecting an individual actor's heap. Without static data-race freedom guarantees, the protocol would be both significantly more complicated and require runtime thread coordination, which is problematic when scaling over a large number of cores on a single node, and even more problematic in a distributed environment.

Causal messaging is used on a single node to enable the protocol to function without requiring acknowledgement messages when message-based reference counts are changed. Without causal messaging, such acknowledgement messages would effectively constitute a form of coordination, and would exhibit similar scaling problems.

2.3 Performance

In co-designing the type system and the runtime, one of the motivations has been the idea that an advanced type system can be a tool for performance as well as for correctness. Some aspects of this are already well known, such as simplifying runtime virtual dispatch, eliding error checking and handling in the presence of an invalid type assumption, or using a type system that ensures data-race freedom to write concurrent programs without locks.

There are other design decisions, however, which impact performance in a way that may not be immediately obvious.

2.3.1 Behaviours as Methods

In many actor model languages, receiving a message involves an explicit expression in the source code. In Pony, behaviours are instead written as methods on an actor type. As a result, Pony messages behave as asynchronous function calls: they are statically typed, and can interact fully with subtyping. For example, it is possible for an actor behaviour to be a subtype of a synchronous function.

This approach disallows receiving a new message while handling the current message. This is important for maintaining logically atomic behaviours, as it disallows expanding the available capability set with the arguments to a future message. In addition, pattern matching on the queue to select a specific message is also effectively disallowed, which is important for maintaining message causality.

Defining behaviours as methods has important performance implications as well. An obvious benefit is that strongly typed messages have similar performance benefits to strongly typed functions. Runtime type checking of arguments is not required, and machine words need not be boxed or otherwise distinguished at runtime from object references.

In addition, the lack of an explicit source statement to receive a message removes the need to capture a stack in order to suspend the execution of an actor while awaiting message delivery. Removing the need to capture the stack significantly simplifies the runtime. Not only does it avoid the need for user-space stack swapping or copying, but it also allows the runtime to use standard operating system stacks for scheduler threads. This allows the foreign function interface (FFI) to conform to the local platform's C ABI, allowing Pony to call code written in any language that can expose a conforming interface, without any overhead. No marshalling or other preparation is required. When link-time optimisation information is available, the Pony compiler can even inline FFI calls in Pony programs.

Disallowing pattern matching on an actor's message queue not only maintains message causality, it also allows an efficient lock-free unbounded intrusive queue to be used in the runtime. The queue can perform integral memory management without needing to garbage collect queue entries, which is important for coordination-free garbage collection.

2.3.2 Exceptions as Partial Functions

Exception handling is a complex topic, both for reasons of type safety, particularly in the presence of checked exceptions, and for performance. The approach adopted by Pony is somewhat different from existing exception handling mechanisms, in that exceptions that unwind the stack can be raised, but these exceptions do not carry information. That is, there is no type or value associated with an exception. Effectively, a function that can raise an exception represents a partial function, one for which not every value in the domain produces a value in the range.

This approach allows Pony to have fully checked exceptions without the associated problems of function type signature expansion or issues with modularity. A Pony exception still represents a non-local return, but since it carries no information, the possibility of a non-local return can be indicated with a single bit of information (i.e. annotation as a partial function) rather than requiring all possible non-local return types to be enumerated. This also addresses the modularity problem: a change in a callee's exception raising behaviour (assuming the callee was already a partial function) does not impact the caller.

At runtime, this has a noticeable performance impact. In most languages, an exception results in a two-phase stack unwinding process. The first phase the locates a landing pad (a destination for the non-local return) in the stack that is capable of handling an exception of the type being raised. The second phase iterates through the stack frames that will be discarded, running any finalisation code, for example `finally` blocks in Java or destructors for objects on the stack in C++, and then continues execution at the landing pad. Some languages, such as Java, compound the resulting performance issues by carrying a backtrace in the exception information, which must be built before unwinding since frames will be discarded during the unwinding process.

In contrast, Pony effectively has a one-phase unwind process. The stack is searched for a landing pad, without having to determine if the landing pad is capable of handling the exception, since the exception has no value and can be handled by any landing pad. Once the landing pad is found, execution immediately continues at the landing pad, without the need to walk the stack or reify a backtrace.

Including exceptions in the language at all was also a performance decision. In the absence of exceptions, methods could return union types that include error values (an approach that is also used in the Pony standard library when a reason for failure must be propagated up the stack). However, this would require machine-word return values to be boxed in order to be able to distinguish them from types representing errors, which is a significant performance penalty. In addition, the return value would have to be checked by the caller, which has a runtime cost. Exceptions allow for code that does not normally fail to elide the cost of checking the return value, improving performance when no error occurs, at the expense of an increased cost (due to stack unwinding) when an error does happen.

2.3.3 Single Compilation Unit

Many ahead-of-time compiled languages provide *separate* or even *independent* compilation. In contrast, even though the Pony type system allows type checking to be both local and modular, the compiler treats a program as a single compilation unit. Compiler passes before code generation are separate, but code generation occurs for the entire program at once.

Performance oriented compilers, such as C and C++, have been moving towards this approach with *link-time optimisation*, wherein object files include the compiler's internal program representation (such as GIMPLE or LLVM IR) and the linker can then perform inter-procedural optimisations across object files.

Pony leverages the single compilation unit approach not just for inter-procedural optimisation, but also to perform reachability analysis, which is used for dead code elimination, generic type and method instantiation, to enable *selector colouring* [73] for virtual dispatch, and to reify structural types for runtime pattern-matching on type.

Reachability analysis in Pony is performed by tracing the program from the entry point and finding all types and methods that are instantiated in the program. When a method receiver is a trait, an interface, or a union type, the method is instantiated on all subtypes of the receiving type. This process simultaneously performs dead code elimination.

Type and method reifications account for the full type of all type arguments, including reference capabilities and viewpoint adaptation, to allow the most specific possible code to be generated, including elements not present in the source, such as code generation to support garbage collection. During this process, all subtyping information required at runtime is reified, which allows both nominal and structural subtyping to be determined at compile time and stored in the type descriptors.

Precomputing subtyping information allows a selector colouring algorithm to be used to calculate method indices for virtual dispatch. This algorithm is similar to register colouring. It assigns an integer index to each method based on a matrix of types and the methods that are instantiated on those types. As a result, at runtime, virtual dispatch is always a single integer lookup in an array, without requiring thunking or other methods of selecting the correct vtable.

Since this approach precomputes both nominal and structural subtyping, there is no additional runtime cost when using structural types as compared to nominal types. Effectively, during reachability, structural types are reified as anonymous nominal types.

The downside of using a single compilation unit is that it prevents incremental compilation. This results in longer compilation times when making small edits to programs, since the entire program must be recompiled. In addition, it prevents constructing a REPL for the language, as adding code to a running program is not possible. This will be addressed in future work.

2.3.4 C ABI Compatibility

The runtime is designed so that the *foreign function interface* does not require marshalling or unmarshalling of arguments, nor any stack manipulation. The underlying scheduler threads use standard kernel threads on every platform, and do not require the stack to be copied or modified. The use of logically atomic behaviours allows this to be done efficiently, as each behaviour runs to completion on a single scheduler thread, using the stack in the usual manner.

As a result, FFI calls can be made without any cost other than the function prologue and epilogue of the target function, i.e. the same cost paid in C, C++ or Fortran. A Pony program can link to a static or dynamic library written in any language, which allows Pony programs to leverage existing software. For example, the standard library uses both OpenSSL (for SSL, TLS and cryptographic functions) and PCRE2 (for Perl compatible regular expressions).

If the target library provides link time optimisation information, a Pony program can inline functions from the library, providing a significant performance improvement.

In order to provide more complete FFI compatibility, Pony has a separate object type called a `struct` that is ABI compatible with a C `struct`. As a result, it has no object header, and so a Pony `struct` cannot be a subtype of any other type, since it would not be able to disambiguate the type at runtime. Their use is confined solely to FFI interoperability.

2.4 Code Example

Figure 2.1 contains a complete example program that is included with the compiler. It is an illustration of how to use asynchronous timers. The program first creates an actor that handles timers. Then, it sets a timer and immediately cancels it. Then, it sets a timer without cancelling it. The program produces the output in figure 2.2. This brief example illustrates a number of core concepts in Pony, including asynchronous message passing, causality, the use of reference capabilities, passing isolated mutable state, sharing globally immutable state, actor and object garbage collection, and program quiescence.

One line 1, the `time` package is imported into the current module's namespace. It is also possible to import a package using an enclosing identifier to avoid

```
use "time"

class TimerPrint is TimerNotify
  let _env: Env
  var _count: U64 = 0

  new iso create(env: Env) =>
    _env = env

  fun ref apply(timer: Timer, count: U64): Bool =>
    _count = _count + count
    _env.out.print("timer: " + _count.string())
    _count < 10

  fun ref cancel(timer: Timer box) =>
    _env.out.print("timer cancelled")

actor Main
  new create(env: Env) =>
    let timers = Timers

    let t1 = Timer(TimerPrint(env), 500000000, 500000000)
        // 500 ms
    let t1' = t1
    timers(consume t1)
    timers.cancel(t1')

    let t2 = Timer(TimerPrint(env), 500000000, 500000000)
        // 500 ms
    timers(consume t2)
```

Figure 2.1: A program illustrating the use of asynchronous timers.

```
$ ./timers.exe
timer cancelled
timer: 1
timer: 2
timer: 3
timer: 4
timer: 5
timer: 6
timer: 7
timer: 8
timer: 9
timer: 10
timer cancelled
```

Figure 2.2: The output of the program in figure 2.1.

namespace collisions. On line 3, a new type `TimerPrint` is defined, and it specifies that it implements the `TimerNotify` trait. On line 4, `TimerPrint` is given a single-assignment private field `_env` of type `Env`, which is an immutable environment that gives the program access to its command line arguments, environment variables, and the console via actors associated with `stdout`, `stderr`, and `stdin`. On line 5, a multiple-assignment field `_count` of type `U64`, an unsigned 64-bit integer, is declared, and is initialised to `0` in all constructors.

On line 6, `TimerPrint` is given a constructor named `create`, which is the default constructor used if no named constructor is specified when instantiating a type. Here, `TimerPrint.create` takes a single argument `env` of type `Env`. This constructor has the `iso` reference capability, as described in chapter 4. This means the constructed object is mutable, but isolated, i.e. externally unique. In order for this to hold, all arguments to the constructor must themselves be either `iso` or `val`, i.e. globally immutable (also described in chapter 4). In this case, `Env` is defined has having a default reference capability of `val`, so this constructor is valid.

On line 7, the single-assignment field `_env` is initialised. At this point, `_env` cannot be assigned again, and this is enforced statically. Pony constructors are required to initialise all fields of the instantiated type, and `this` in a constructor has the reference capability `tag`, as described in chapter 4, until it is fully initialised. Note that `_count` is not explicitly initialised in the body of `create`, as it is initialised where it is defined.

On line 10, `TimerPrint` is given an `apply` method, which is the default method if an object is invoked without a method being specified. This is called from the `time` package when the associated timer fires. It takes two arguments: the `Timer` that is firing, and the number of times the `Timer` has fired since the notifier was last informed. The default reference capability for `Timer` is defined in the `time` package as `ref`, i.e. mutable but not isolated, as described in chapter 4. This allows the `TimerPrint` notifier to mutate the `Timer` when a notification occurs, perhaps by changing its expiry time. In this case, the body of `TimerPrint.apply` advances the notifier's `_count`, prints to the console by sending a message to `_env.out` (an actor with access to `stdout`), and then returns `true` if the total `_count` is less than ten, or `false` otherwise. In the `time` package, a timer is cancelled if the notifier for the timer returns `false`.

Note that the message sent to `_env.out` on line 12 contains a globally immutable `String`. It would be safe for the notifier to keep a referenec to the `String` being sent, but in this case it does not.

On line 15, `TimerPrint` is given a `cancel` method, which is called from the `time` package when the associated timer is cancelled. It takes a single argument of type `Timer box`, i.e. a locally immutable `Timer`, as described in chapter 4. A `box` reference allows `TimerPrint.cancel` to read from the `Timer`, but neither mutate it nor pass it in a message. In this case, the `cancel` method simply prints a notification to the console, by again sending a message to `_env.out` containing a globally immutable `String`.

On line 18, the program's `Main` actor is defined. A Pony program begins with its `Main` actor being constructed using the `create` constructor, which is defined here on line 19. The `Main.create` constructor always takes a single argument, which is an immutable environment `Env`. This program begins by creating a `Timers` actor on line 20, which is a hierarchical timing wheel actor defined in the `time` package. The local variable `timer` is of type `Timer tag`, as an actor is viewed as an opaque type from other actors, as described in chapter 4.

On line 22, a `Timer` is created, using the default `Timer.create` constructor. Its first argument is a `TimerNotify`, created with `TimerNotify.create` and passed `env` as an argument. It is safe to share the reference to `env` as it is a `val` reference, i.e. globally immutable, and as such its use cannot result in a data-race. The resulting notifier is of type `TimerNotify iso`, that is, it is isolated. The other two arguments to the `Timer` constructor are integers, which type inference will discover are of type `U64`. All basic machine-word types, i.e. booleans, integers, and floating point number, are treated as `val`, so here the `Timer` constructor takes only `iso` and `val` arguments. As a result, the type of `t1` when it is assigned on line 22, is `Timer iso`, that is, an isolated `Timer`, with a 500 millisecond initial firing time that repeatedly fires every 500 milliseconds after that.

On line 23, an alias of `t1`, here called `t1'`, is created. However, as described in chapter 4, an alias of an `iso` reference is a `tag` reference. To maintain isolation, an alias to an isolated reference must be opaque, i.e. allow neither reading from nor writing to the reference object.

On line 24, `t1` is `consume`'d from the current lexical scope, making it unavailable in that scope. This results in an *unaliased* type, that is, a type for which one alias has been removed, again as described in chapter 4. This allows the object formerly referenced by `t1` to be safely sent in a message to the `Timers` actor that was created on line 20. Line 24 implicitly calls `Timers.apply`, which in this case is a *behaviour*, that is, an asynchronous method call, as described in chapter 3. This in turn uses the garabage collection protocol described in chapter 6 to ensure that the `Timer` can eventually be collected without requiring a stop-the-world step.

Then, on line 25, the `timer` actor is sent another message, this time `Timers.cancel`. The argument is `t1'`, which is a `tag` alias to the same `Timer` that was set on line 24. By keeping a `tag` alias to the `Timer`, the `Main` actor can refer to it by its identity when cancelling it. The `tag` reference to the `Timer` is a capability that allows cancelling the possessor to cancel it, with no risk of name collision. Because messages are causally ordered, the `Timers` actor will receive the creation of the `Timer` before the cancellation, so the cancellation cannot be lost.

On line 27, the `Main` actor creates another `Timer`, in the same manner as previously, and on line 28, that timer is sent to the `Timers` actor. At this point, the `Main` actor's constructor completes, but, unlike the `main` function in a C or C++ program, the program does not terminate. This is because the `Timers` actor still has pending work, and so the program has not yet reached quiescence, as described in section 5.4. However, the actor garbage collection protocol described in chapter 5 may at this point collect the `Main` actor, as it is not executing and has no pending work, i.e. it is *blocked* as described in section 5.2.3, and cannot receive work in the future, as no other actor has a reference to it, i.e. it is *dead* as described in section 5.2.5.

Concurrently with the execution of `Main`'s constructor, the `Timers` actor will begin handling its own messages, as shown in chapter 3. Specifically, it will set the first `Timer`, including making a system call to set up an operating system timer, then cancel the first `Timer`, and then set the second `Timer`. Interleaved with these messages will be messages from the asynchronous I/O layer in the runtime, as described in section A.13, indicating that the operating system timer has fired. When those occur, the `Timers` actor examines its hierarchical timing wheel, as implemented in the `time` package, and fires the `Timer` objects it holds as appropriate.

In this case, the `TimerPrint` notifier for the first `Timer` is called with `cancel` first, before any timers have fired. Then, the second `Timer` fires ten times, after which it returns `false` and the `Timers` actor cancels it. At this point, the `Timers` actor has no pending `Timer` objects, and so cancels its operating system timer.

This results in the `Timers` actor becoming *blocked*, as it is not executing and has no pending work. Since the only actor that may have a reference to it, the `Main` actor, is also *blocked*, and the asynchronous I/O layer will not generate new work for it, the `Timers` actor is also *dead*, and eligible for garbage collection. At this point, all actors are *dead* and the program has reached quiescence, as described in section A.12.3, and terminates.

2.5 Philosophy

In the spirit of Richard Gabriel, the Pony philosophy is neither "the-right-thing" nor "worse-is-better". It is "get-stuff-done".

- Correctness. Incorrectness is simply not allowed. *It's pointless to try to get stuff done if you can't guarantee the result is correct.*
- Performance. Runtime speed is more important than everything except correctness. If performance must be sacrificed for correctness, try to come up with a new way to do things. *The faster the program can get stuff done, the better. This is more important than anything except a correct result.*
- Simplicity. Simplicity can be sacrificed for performance. It is more important for the interface to be simple than the implementation. *The faster the programmer can get stuff done, the better. It's ok to make things a bit harder on the programmer to improve performance, but it's more important to make things easier on the programmer than it is to make things easier on the language/runtime.*
- Consistency. Consistency can be sacrificed for simplicity or performance. *Don't let excessive consistency get in the way of getting stuff done.*
- Completeness. It's nice to cover as many things as possible, but completeness can be sacrificed for anything else. *It's better to get some stuff done now than wait until everything can get done later.*

The "get-stuff-done" approach has the same attitude towards correctness and simplicity as "the-right-thing", but the same attitude towards consistency and completeness as "worse-is-better". It also adds performance as a new principle, treating it as the second most important thing, after correctness.

As with any self-professed philosophy, this one should be taken with a grain of salt. It has helped guide the development of Pony, but it is the map, not the territory.

Chapter 3

Syntax and Operational Semantics

The syntax and operational semantics of Pony is presented here as a small-step operational semantics that expresses a simplified subset of the full Pony language. Additional features in the full language can be encoded in this subset, including control structures, exception handling, algebraic data types (tuples, unions, and intersections), machine word types, singleton types, C ABI compatible structures, single-assignment fields and variables, embedded fields, nominal and structural subtyping via traits and interfaces, object literals, lambdas, partial application, pattern matching, case functions, and generic types and methods. Various forms of syntactic sugar are also used in the full language to allow for more succinct code.

The syntax and formalisation of inheritance, union types, tuples, and intersection types appears in Steed [66]. The syntax and formalisation of generics, value-dependent types, and compile-time expressions appears in Cheeseman [14]. The operational semantics presented here differs from previous work [17], particularly in the treatment of message sends and in providing a distributed semantics.

3.1 Syntax

The syntax is presented in figure 3.1. Actors have a type, in the mould of active objects, and are introduced with the keyword `actor`. These can have both synchronous methods (*functions*, introduced through the keyword `fun`) and asynchronous methods (*behaviours*, introduced through the keyword `be`) as well as named constructors (introduced through the keyword `new`).

$$
\begin{array}{rclcl}
\mathtt{P} & \in & \mathit{Program} & ::= & \overline{\mathtt{CT}}\,\overline{\mathtt{AT}} \\
\mathtt{CT} & \in & \mathit{ClassDef} & ::= & \mathtt{class}\,\mathtt{C}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}} \\
\mathtt{AT} & \in & \mathit{ActorDef} & ::= & \mathtt{actor}\,\mathtt{A}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \\
\mathtt{S} & \in & \mathit{TypeID} & ::= & \mathtt{A} \mid \mathtt{C} \\
\mathtt{T} & \in & \mathit{Type} & ::= & \mathtt{S}\,\kappa \\
\mathtt{ET} & \in & \mathit{ExtType} & ::= & \mathtt{T} \mid \mathtt{S}\,\kappa\circ \\
\mathtt{F} & \in & \mathit{Field} & ::= & \mathtt{var}\,\mathtt{f} : \mathtt{T} \\
\mathtt{K} & \in & \mathit{Ctor} & ::= & \mathtt{new}\,\mathtt{k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e} \\
\mathtt{M} & \in & \mathit{Func} & ::= & \mathtt{fun}\,\kappa\,\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{ET} \Rightarrow \mathtt{e} \\
\mathtt{B} & \in & \mathit{Behv} & ::= & \mathtt{be}\,\mathtt{b}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e} \\
\mathtt{n} & \in & \mathit{MethodID} & ::= & \mathtt{k} \mid \mathtt{m} \mid \mathtt{b} \\
\kappa & \in & \mathit{Cap} & ::= & \mathtt{iso} \mid \mathtt{trn} \mid \mathtt{ref} \mid \mathtt{val} \mid \mathtt{box} \mid \mathtt{tag} \\
\mathtt{e} & \in & \mathit{Expr} & ::= & \mathtt{this} \mid \mathtt{x} \mid \mathtt{var}\,\mathtt{x} : \mathtt{T} \mid \mathtt{x} = \mathtt{e} \mid \mathtt{e};\mathtt{e} \mid \mathtt{e.f} \\
 & & & \mid & \mathtt{e.f} = \mathtt{e} \mid \mathtt{consume}\,\mathtt{x} \mid \mathtt{recover}\,\mathtt{e} \\
 & & & \mid & \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e.b}(\overline{\mathtt{e}}) \mid \mathtt{S.k}(\overline{\mathtt{e}}) \\
\mathtt{E}[\cdot] & \in & \mathit{ExprHole} & ::= & \mathtt{x} = \mathtt{E}[\cdot] \mid \mathtt{E}[\cdot];\mathtt{e} \mid (\mathtt{E}[\cdot]) \mid \mathtt{E}[\cdot]\mathtt{.f} \\
 & & & \mid & \mathtt{e.f} = \mathtt{E}[\cdot] \mid \mathtt{E}[\cdot]\mathtt{.f} = \mathtt{z} \mid \mathtt{E}[\cdot]\mathtt{.n}(\overline{\mathtt{z}}) \\
 & & & \mid & \mathtt{e.n}(\overline{\mathtt{z}}, \mathtt{E}[\cdot], \overline{\mathtt{e}}) \mid \mathtt{recover}\,\mathtt{E}[\cdot]
\end{array}
$$

Figure 3.1: Syntax

$$
\begin{array}{rcl@{\qquad}rcl}
\mathtt{C} & \in & \mathit{ClassID} & \mathtt{k} & \in & \mathit{CtorID} \\
\mathtt{A} & \in & \mathit{ActorID} & \mathtt{m} & \in & \mathit{FuncID} \\
\mathtt{f} & \in & \mathit{FieldID} & \mathtt{b} & \in & \mathit{BehvID} \\
\mathtt{this}, \mathtt{x} & \in & \mathit{SourceID} & \mathtt{n} & \in & \mathit{CtorID} \cup \mathit{BehvID} \\
\mathtt{t} & \in & \mathit{TempID} & \mathtt{y}, \mathtt{z} & \in & \mathit{LocalID}
\end{array}
$$

Figure 3.2: Identifiers

$$
\begin{array}{lllll}
\chi & \in & \mathit{Heap} & = & \mathit{Addr} \rightarrow (\mathit{Actor} \vee \mathit{Object}) \\
\sigma & \in & \mathit{Stack} & = & \mathit{ActorAddr} \cdot \overline{\mathit{Frame}} \\
\varphi & \in & \mathit{Frame} & = & \mathit{MethodID} \times (\mathit{LocalID} \rightarrow \mathit{Value}) \\
 & & & \times & \mathit{ExprHole} \\
 & & \mathit{LocalID} & = & \mathit{SourceID} \cup \mathit{TempID} \\
v & \in & \mathit{Value} & = & \mathit{Addr} \cup \{\mathit{null}\} \\
\iota & \in & \mathit{Addr} & = & \mathit{ActorAddr} \cup \mathit{ObjectAddr} \\
\alpha & \in & \mathit{ActorAddr} & & \\
\omega & \in & \mathit{ObjectAddr} & & \\
 & & \mathit{Actor} & = & \underline{\mathit{ActorID}} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \\
 & & & \times & \overline{\mathit{Message}} \times \mathit{Stack} \times \mathit{Expr} \\
 & & & \times & (\mathit{ActorID} \rightarrow \overline{\mathit{Message}}) \\
 & & \mathit{Object} & = & \mathit{ClassID} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \\
\mu & \in & \mathit{Message} & = & \mathit{MethodID} \times \overline{\mathit{Value}}
\end{array}
$$

Figure 3.3: Runtime entities

Passive objects (introduced through the keyword `class`) have only synchronous methods (functions) and constructors. The term *method* is used to refer to constructors, functions, and behaviours.

The novel element of the syntax is the inclusion of *reference capability annotations* κ on types and functions, where:

$\kappa \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{ref}, \mathtt{val}, \mathtt{box}, \mathtt{tag}\}$

These reference capabilities are the foundation of the Pony type system.

Types consist of a class or actor identifier `S` followed by a reference capability κ. In addition, extended types `ET` can be *unaliased*, $\circ$. An *unaliased type* is created with constructors and destructive reads. This is described in detail in chapter 4.

The over-bar notation indicates a sequence of elements such as $\overline{\mathtt{F}}$, with the convention that the n^{th} element is referred to as $\mathtt{F_n}$. Similarly, $\overline{\mathtt{x}} : \overline{\mathtt{T}}$ indicates a pairwise sequence of identifiers and types. To reduce notation, a *fixed* program `P` is assumed.

3.2 Operational Semantics

The operational semantics has the shape $\chi \rightarrow \chi'$, where χ, χ' are heaps mapping object addresses ω to their class identifier and their fields, and actor addresses α to their actor identifier, their fields, their message queue, their stack, and the next expression to execute. These runtime entities are defined in figure 3.3. Some shorthand notation is used for clarity, as defined in figure 3.6.

Note that *null* is used to indicate an undefined field or variable in the operational semantics. However, the full language does not permit the use of

$$\frac{\chi, \sigma \cdot \varphi, \texttt{e} \leadsto \chi', \sigma \cdot \varphi', \texttt{e}'}{\chi, \sigma \cdot \varphi, \texttt{E}[\texttt{e}] \leadsto \chi', \sigma \cdot \varphi', \texttt{E}[\texttt{e}']}\ \textsc{ExprHole}$$

$$\frac{\texttt{t} \notin \varphi \quad \iota = \varphi(\texttt{z}) \quad \varphi' = \varphi[\texttt{t} \mapsto \chi(\iota, \texttt{f})]}{\chi, \sigma \cdot \varphi, \texttt{z.f} \leadsto \chi, \sigma \cdot \varphi', \texttt{t}}\ \textsc{Fld}$$

$$\frac{\texttt{x} \notin \varphi \quad \varphi' = \varphi[\texttt{x} \mapsto \mathit{null}]}{\chi, \sigma \cdot \varphi, \texttt{var}\,\texttt{x} : \texttt{ET} \leadsto \chi, \sigma \cdot \varphi', \mathit{null}}\ \textsc{DeclLocal}$$

$$\frac{}{\chi, \sigma, \texttt{z}; \texttt{e} \leadsto \chi, \sigma, \texttt{e}}\ \textsc{Seq}$$

$$\frac{\texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{x} \mapsto \varphi(\texttt{z}), \texttt{t} \mapsto \varphi(\texttt{x})]}{\chi, \sigma \cdot \varphi, \texttt{x} = \texttt{z} \leadsto \chi, \sigma \cdot \varphi', \texttt{t}}\ \textsc{AsnLocal}$$

$$\frac{\begin{array}{c}\texttt{t} \notin \varphi \quad \iota = \varphi(\texttt{z}) \quad \varphi' = \varphi[\texttt{t} \mapsto \chi(\iota, \texttt{f})] \\ \chi' = \chi[\varphi(\texttt{z}), \texttt{f} \mapsto \varphi(\texttt{y})]\end{array}}{\chi, \sigma \cdot \varphi, \texttt{z.f} = \texttt{y} \leadsto \chi', \sigma \cdot \varphi', \texttt{t}}\ \textsc{AsnFld}$$

$$\frac{\begin{array}{c}\iota = \varphi(\texttt{z}) \quad \mathcal{M}(\chi(\iota) \downarrow_1, \texttt{m}) = (\_, \overline{\texttt{x}} : \_, \texttt{e}, \_) \\ \varphi' = (\texttt{m}, [\texttt{this} \mapsto \iota, \overline{\texttt{x}} \mapsto \varphi(\overline{\texttt{y}})], \texttt{E}[\cdot])\end{array}}{\chi, \sigma \cdot \varphi, \texttt{E}[\texttt{z.m}(\overline{\texttt{y}})] \leadsto \chi, \sigma \cdot \varphi \cdot \varphi', \texttt{e}}\ \textsc{Sync}$$

$$\frac{\begin{array}{c}\texttt{t} \notin \varphi \quad \iota = \varphi'(\texttt{z}) \\ \varphi' \downarrow_3 = \texttt{E}[\cdot] \quad \varphi'' = \varphi[\texttt{t} \mapsto \iota]\end{array}}{\chi, \sigma \cdot \varphi \cdot \varphi', \texttt{z} \leadsto \chi, \sigma \cdot \varphi'', \texttt{E}[\texttt{t}]}\ \textsc{Return}$$

$$\frac{\begin{array}{c}\alpha' = \varphi(\texttt{z}) \\ \chi' = \chi[\alpha' + +(\texttt{b}, \varphi(\overline{\texttt{y}}))]\end{array}}{\chi, \alpha \cdot \overline{\varphi} \cdot \varphi, \texttt{z.b}(\overline{\texttt{y}}) \leadsto \chi', \alpha \cdot \overline{\varphi} \cdot \varphi, \texttt{z}}\ \textsc{Async}$$

$$\frac{\begin{array}{c}\texttt{A} = \chi(\alpha) \downarrow_1 \quad (\texttt{n}, \overline{v}) \cdot \overline{\mu} = \chi(\alpha) \downarrow_3 \\ \mathcal{M}(\texttt{A}, \texttt{n}) = (\_, \overline{\texttt{x}} : \_, \texttt{e}, \_) \\ \varphi = (\texttt{n}, [\texttt{this} \mapsto \alpha, \overline{\texttt{x}} \mapsto \overline{v}], \cdot)\end{array}}{\chi, \alpha, \varepsilon \leadsto \chi[\alpha \mapsto \overline{\mu}], \alpha \cdot \varphi, \texttt{e}}\ \textsc{Behave}$$

$$\frac{\begin{array}{c}(\omega, \chi') = \mathit{New}(\chi, \texttt{C}) \\ \mathcal{M}(\texttt{C}, \texttt{k}) = (\_, \overline{\texttt{x}} : \_, \texttt{e}, \_) \\ \varphi' = (\texttt{k}, [\texttt{this} \mapsto \omega, \overline{\texttt{x}} \mapsto \varphi(\overline{\texttt{y}})], \texttt{E}[\cdot])\end{array}}{\chi, \sigma \cdot \varphi, \texttt{E}[\texttt{C.k}(\overline{\texttt{y}})] \leadsto \chi', \sigma \cdot \varphi \cdot \varphi', \texttt{e}}\ \textsc{Ctor}$$

$$\frac{\begin{array}{c}(\alpha, \chi') = \mathit{New}(\chi, \texttt{A}, \texttt{k}, \varphi(\overline{\texttt{y}})) \\ \texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{t} \mapsto \alpha]\end{array}}{\chi, \sigma \cdot \varphi, \texttt{A.k}(\overline{\texttt{y}}) \leadsto \chi', \sigma \cdot \varphi', \texttt{t}}\ \textsc{Ator}$$

$$\frac{\texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{t} \mapsto \varphi(\texttt{x})] \backslash \texttt{x}}{\chi, \sigma \cdot \varphi, \texttt{consume}\,\texttt{x} \leadsto \chi, \sigma \cdot \varphi', \texttt{t}}\ \textsc{Consume}$$

$$\frac{\texttt{t} \notin \varphi \quad \varphi' = \varphi[\texttt{t} \mapsto \varphi(\texttt{z})]}{\chi, \sigma, \texttt{recover}\,\texttt{z} \leadsto \chi, \sigma, \texttt{t}}\ \textsc{Recover}$$

Figure 3.4: Local execution

$$\frac{\chi, \chi(\alpha) \downarrow_4, \chi(\alpha) \downarrow_5 \leadsto \chi', \sigma, \texttt{e}}{\chi \to \chi'[\alpha \mapsto (\sigma, \texttt{e})]}\ \textsc{Global}$$

Figure 3.5: Global execution

- $\varphi(\texttt{x}) = \varphi\downarrow_2 (\texttt{x})\downarrow_1$
- $\varphi[\texttt{x} \mapsto v] = (\varphi\downarrow_1, \varphi\downarrow_2 [\texttt{x} \mapsto v], \varphi\downarrow_3)$
- $\varphi\backslash\texttt{x} = (\varphi\downarrow_1, [\texttt{z} \mapsto v \,|\, \varphi(\texttt{z}) = v \wedge \texttt{z} \neq \texttt{x}], \varphi\downarrow_3)$
- $\chi(\iota, \texttt{f}) = \chi(\iota)\downarrow_2 (\texttt{f})$
- $\chi[\omega, \texttt{f} \mapsto v] = \chi[\omega \mapsto (\chi(\omega)\downarrow_1, \chi(\omega)\downarrow_2 [\texttt{f} \mapsto v]]$
- $\chi[\alpha, \texttt{f} \mapsto v] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1, \chi(\alpha)\downarrow_2 [\texttt{f} \mapsto v], \chi(\alpha)\downarrow_3 ... \chi(\alpha)\downarrow_5)]$
- $\chi[\alpha \mapsto (\sigma, \texttt{e})] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1 ... \chi(\alpha)\downarrow_3, \sigma, \texttt{e})]$
- $\chi[\alpha + +\mu] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1, \chi(\alpha)\downarrow_2, \chi(\alpha)\downarrow_3 \cdot\mu, \chi(\alpha)\downarrow_4, \chi(\alpha)\downarrow_5)]$
- $\chi[\alpha \mapsto \overline{\mu}] = \chi[\alpha \mapsto (\chi(\alpha)\downarrow_1, \chi(\alpha)\downarrow_2, \overline{\mu}, \chi(\alpha)\downarrow_4, \chi(\alpha)\downarrow_5)]$
- $\mathit{New}(\chi, \texttt{C}) = (\omega, \chi')\ \mathit{where}\, \omega \notin \mathit{dom}(\chi) \wedge \chi' = \chi[\omega \mapsto (\texttt{C}, \mathcal{F}s(\texttt{C}) \mapsto \mathit{null})]$
- $\mathit{New}(\chi, \texttt{A}, \texttt{k}, \overline{v}) = (\alpha, \chi')\ \mathit{where}$
  $\alpha \notin \mathit{dom}(\chi) \wedge \chi' = \chi[\alpha \mapsto (\texttt{A}, \mathcal{F}s(\texttt{A}) \mapsto \mathit{null}, (\texttt{k}, \overline{v}), \alpha, \varepsilon, [])]$

Figure 3.6: Auxiliary definitions

undefined fields or variables, or the construction of a partially defined object (one where some subset of fields remains undefined). Thus there is no *null*, in the sense of a bottom value that can inhabit any type, in the full language.

The symbol `x` is used to indicate a source identifier, `t` to indicate a temporary identifier, and `y` and `z` to indicate identifiers which may be either. Temporary identifiers are used in the semantics instead of values in order to associate a type, including a reference capability, with intermediate expression results. Importantly, temporary identifiers are accounted for in well-formedness.

A call stack consists of an actor address α followed by a sequence of frames φ. A frame consists of the method identifier, a mapping of its parameters to values, and an expression hole. The latter is the continuation of the caller and will be executed by the previous frame when the current activation terminates.

The auxiliary judgement $\chi, \sigma, \texttt{e} \rightsquigarrow \chi', \sigma', \texttt{e}'$ expresses local execution within a *single* actor. $\mathcal{M}$ and $\mathcal{F}$ return method and field declarations, as defined in figure 3.10.

3.2.1 Notation

To simplify the presentation, some notation conventions are followed. The use of $\overline{x}$ indicates a sequence of x, whereas xs indicates a set of x. The projection $x\downarrow_k$ is used to express the k^{th} element of the tuple x. The use of $x\downarrow_k ... x\downarrow_{k+n}$ refers to a sequence of the k^{th} through $k + n^{th}$ elements of the tuple x.

3.2.2 Concurrent Execution

Local execution is defined in figure 3.4. These rules define local execution within an actor. ExprHole allows execution to propagate to the context, and also determines the order of evaluation for complex expressions. Fld, DeclLocal, and Seq are as expected.

AsnLocal and AsnFld combine assignment with a destructive read, returning the previous value of the left-hand side. The resulting value is *unaliased*: while there may be other paths pointing to the value in the program, this one no longer does. In effect, one alias to the value has been discarded. The existence of unaliased values is used in the type system, where T-AsnLocal and T-AsnField both return an *unaliased type*, as explained in chapter 4.

Sync and Return describe synchronous method call and return. In Sync, method `m` is called on object or actor ι. The method parameters $\overline{\texttt{x}}$ and the method body `e` are looked up using the method `m` and the type `S` of ι from the heap. A new frame is pushed on to the stack, consisting of `m`, the address of the receiver, the values of the arguments, and the continuation. In Return, the topmost frame is popped from the stack and execution continues.

Async and Behave describe asynchronous method calls and execution. In Async, a message consisting of the behaviour identifier `b` and the arguments is appended to the receiver's message queue. In Behave, an actor with an empty call stack and a non-empty message queue removes the oldest message from the queue, and pushes a new frame on the stack.

Ctor and Ator describe the construction of new objects and actors. In Ctor, a new address ω is allocated on the heap and the fields are initialised to *null*. A new frame is pushed on the stack in the same way as for Sync. In Ator, instead of pushing a new frame on the stack, the new actor's queue is initialised with a constructor message containing the constructor identifier `k` and the arguments. The first local execution rule for a new actor will be Behave, which will execute the body of the constructor `k`.

Recover and Consume are effectively no-ops in the operational semantics, but have an impact in the type system, where T-Recover and T-Consume affect the reference capability of the result of the expression.

Global execution is defined in figure 3.5. Global says that if an actor can execute, then its stack and next expression to execute will be updated.

3.2.3 Distributed Execution

In the distributed setting, atomic delivery of messages is not feasible. To account for this, the definition of an actor is extended in figure 3.7 to include a logical pairwise (actor-to-actor) FIFO queue of messages sent by this actor that have

$$\begin{array}{rcl} \mathit{Actor} & = & \mathit{ActorID} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \\ & \times & \overline{\mathit{Message}} \times \mathit{Stack} \times \mathit{Expr} \\ & \times & (\mathit{ActorID} \rightarrow \overline{\mathit{Message}}) \end{array}$$

Figure 3.7: Distributed runtime entities. Elements that are unchanged are greyed out.

$$\frac{\begin{array}{c}\alpha' = \varphi(\mathtt{z}) \\ \chi' = \chi[\alpha, \alpha' + +(\mathtt{b}, \varphi(\overline{\mathtt{y}})]\end{array}}{\chi, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{z.b}(\overline{\mathtt{y}}) \rightsquigarrow \chi', \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{z}} \ \textsc{Async}$$

$$\frac{\exists \alpha'.\chi(\alpha') \downarrow_6 (\alpha) = \mu \cdot \overline{\mu}}{\chi \rightarrow \chi[\alpha', \alpha \mapsto \overline{\mu}][\alpha + +\mu]} \ \textsc{Deliver}$$

Figure 3.8: Distributed execution

not yet been delivered.

In figure 3.8, the Async rule is changed to place the newly created message into the pairwise FIFO queue of undelivered messages between the sender and the receiver, rather than directly into the message queue of the receiver. The new Deliver rule moves a message from a pairwise FIFO queue to the receiver's message queue.

The purpose of the separate pairwise FIFO queues is to model an arbitrary delay between some actor α sending a message and the delivery of that message to the destination actor α'. On a single node, Async atomically enqueues the message in the receiver's message queue. This guarantees message causality on a single node.

However, in the distributed context, the message produced by Async is not placed in the receiver's queue until some future execution of Deliver. This models arbitrary delay, including network delays, and breaks causality. Pairwise FIFO ordering is maintained, but it is possible for a third actor to appear to violate causality. Distributed causality can be built on top of this model using tree-structured networks, which will be addressed in chapter 7.

In order to consider all messages bound for some actor α, $Q(\chi, \alpha)$ is defined to be the concatenation of all message queues that will be delivered to α, including the already enqueued messages and the FIFO queue from all other actors that have not yet been enqueued for α.

- $\chi[\alpha, \alpha' + +\mu] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1 ... \chi(\alpha) \downarrow_5, \chi(\alpha) \downarrow_6 [\alpha' \mapsto \chi(\alpha) \downarrow_6 (\alpha') \cdot \mu])$
- $\chi[\alpha, \alpha' \mapsto \mu] = \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1 ... \chi(\alpha) \downarrow_5, \chi(\alpha) \downarrow_6 [\alpha' \mapsto \overline{\mu}])$

Figure 3.9: Auxiliary definitions for distributed execution

Definition 3.1. Queues for α

$$Q(\chi, \alpha) = \{\chi(\alpha) \downarrow_3\} \cup \{\overline{\mu} \,|\, \alpha' \in \mathit{dom}(\chi) \wedge \overline{\mu} = \chi(\alpha') \downarrow_6 (\alpha)\}$$

$$\begin{array}{rcl}
Q(\chi, \alpha) & = & Q(\chi, \alpha, \{\alpha' \,|\, \alpha' \in \mathit{dom}(\chi)\} \\
Q(\chi, \alpha, \{\alpha'\} \cup \alpha s) & = & Q(\chi, \alpha, \alpha') \cdot Q(\chi, \alpha, \alpha s) \\
Q(\chi, \alpha, \emptyset) & = & \emptyset \\
Q(\chi, \alpha, \alpha') & = & \begin{cases} \chi(\alpha) \downarrow_3 & \mathit{if}\ \alpha = \alpha' \\ \chi(\alpha') \downarrow_6 (\alpha) & \mathit{otherwise} \end{cases}
\end{array}$$

In the concurrent setting, where enqueuing messages is atomic, only the actor's message queue will contain messages. However, in the distributed setting, where causality is not enforced, the FIFO queues may also contain messages. The concatenation produced by $Q(\chi, \alpha)$ is non-deterministic, as the evaluation of $\alpha' \in \mathit{dom}(\chi)$ is unordered.

Note that, unlike Async, Ator does not enqueue the constructor message. Instead, the new actor is created with a single message in its queue, as if Deliver had already taken place. This is to avoid a situation in a distributed context where a message could arrive before the actor had been constructed, while still allowing asynchronous construction of actors in both concurrent and distributed settings.

3.3 Lookup Functions

Lookup functions are defined in fig. 3.10. Function $\mathcal{P}$ returns a type definition for a class identifier `C` or actor identifier `A`. This contains the fields $\overline{\texttt{F}}$, constructors $\overline{\texttt{K}}$, functions $\overline{\texttt{M}}$, and behaviours $\overline{\texttt{B}}$ defined for that type. Since classes have no asynchronous behaviour, the last entry in $\mathcal{P}(\texttt{C})$ is empty, i.e. ε.

Function $\mathcal{F}s$ returns the identifiers of all fields defined in a type `S`, and function $\mathcal{F}$ returns the type of field `f` in `S`. Function $\mathcal{M}$ returns method information for some method in `S`. This is overloaded on both the method identifier and the type identifier in order to handle class constructors, actor constructors, synchronous methods (functions) and asynchronous methods (behaviours). The result is a tuple of: the receiver type, the names and types of the parameters, the body of the method, and the return type.

The lookup functions mention both types and *reference capabilities*, such as **ref**, **ref**$\circ$, and **tag**. These will be explained in chapter 4.

3.4 Actor model implementation

This operational semantics is intended to conform to the design decisions in chapter 2. Actors as active objects, single node causality, and distributed pair-

$$\frac{\mathtt{P} = \overline{\mathtt{CT}}\,\overline{\mathtt{AT}} \quad \mathtt{class\,C}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}} \in \overline{\mathtt{CT}}}{\mathcal{P}(\mathtt{C}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\varepsilon \quad \mathtt{C} \in \mathtt{P}}$$

$$\frac{\mathtt{P} = \overline{\mathtt{CT}}\,\overline{\mathtt{AT}} \quad \mathtt{actor\,A}\,\overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \in \overline{\mathtt{AT}}}{\mathcal{P}(\mathtt{A}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad \mathtt{A} \in \mathtt{P}}$$

$$\frac{\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}}}{\mathcal{F}s(\mathtt{S}) = \{\mathtt{f} \mid \mathtt{var\,f} : \mathtt{T} \in \overline{\mathtt{F}}\}}$$

$$\frac{\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad \mathtt{var\,f} : \mathtt{T} \in \overline{\mathtt{F}}}{\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{T}}$$

$$\frac{\mathcal{P}(\mathtt{C}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}} \quad (\mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}}{\mathcal{M}(\mathtt{C}, \mathtt{k}) = (\mathtt{C\,ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{C\,ref}\circ)}$$

$$\frac{\mathcal{P}(\mathtt{A}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad (\mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{K}}}{\mathcal{M}(\mathtt{A}, \mathtt{k}) = (\mathtt{A\,ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A\,tag})}$$

$$\frac{\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad (\mathtt{fun}\,\kappa\,\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{ET} \Rightarrow \mathtt{e}) \in \overline{\mathtt{M}}}{\mathcal{M}(\mathtt{S}, \mathtt{m}) = (\mathtt{S}\,\kappa, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{ET})}$$

$$\frac{\mathcal{P}(\mathtt{A}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \quad (\mathtt{be\,b}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}) \in \overline{\mathtt{B}}}{\mathcal{M}(\mathtt{A}, \mathtt{b}) = (\mathtt{A\,ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A\,tag})}$$

Figure 3.10: Lookup functions

wise FIFO message ordering are directly provided by the operational semantics.

However, many of the design goals cannot be provided for with the semantics alone. For example, logically atomic behaviours rely on the operational semantics to order messages and run behaviours to completion without interleaving other messages or allowing the message queue to be examined, but this is insufficient. Without a type system that can enforce that communication between actors happens only via message passing, it remains possible for two actors to exchange information by modifying fields of the same object, resulting in behaviours that are no longer logically atomic.

Chapter 4

Reference Capabilities

Capabilities were introduced to support protection across processes [38], and have been adopted into several branches of computing since. The term *object capabilities* was coined by Mark Miller [49, 48] to describe the set of operations an object is allowed to apply on some other object. Mark Miller proposes that in order to restrict this set, one should create a new object which only offers these capabilities, and which delegates to the original object. This is a critical insight: in a memory safe language, the methods on an object are the capabilities available on that object.

A *reference capability* is a type qualifier annotation on a reference, rather than an object, that modifies the capabilities available on the underlying object when it is accessed through that reference. In Pony, reference capabilities are used to statically ensure data-race freedom. The term *reference capability* was introduced specifically to distinguish these Pony annotations from object capabilities [17].

In previous work, type qualifiers for data-race freedom have been expressed as *permissions* [12], *fractional permissions* [11], *uniqueness* [15], *immutability* [55], and *isolation* [26] (a refinement of *separate uniqueness* [30], which is a refinement of *external uniqueness* [15]).

These approaches use type qualifiers to describe what a reference is *allowed* to do. Pony's type system takes a different approach and uses reference capabilities to describe what other aliases are *denied* [21] by the existence of a reference. This is accomplished using a matrix of *deny properties*, with notions such as isolation, mutability, and immutability all being derived from these properties. What aliases to the object are allowed to do is explicit rather than implied, whereas what the reference is allowed is derived. This change in approach gives a derivation for properties previously considered intrinsic, and models a reduction in reference capabilities as a weaker guarantee.

Other approaches have combined actors with data-race freedom, such as minimal ownership for active objects [16], capabilities for uniqueness and borrowing in Scala [30], and Kilim [65]. However, various useful patterns have not been supported, e.g. traversing and modifying an isolated data structure, or updating an object and then sending it in a message while keeping read access to it. By taking a more fundamental view of reference capabilities, a more flexible type system that supports such patterns became possible.

The matrix of deny properties exposes two novel reference capability types, `tag` and `trn` (*transition*). A `tag` reference capability allows identity comparison and *asynchronous* method call, but does not allow reading from or writing to the reference. Actors are typed as `tag`, which allows them to be integrated into the object type system and passed in messages. A `trn` reference capability is a new form of uniqueness, *write uniqueness*, that describes objects that can only be written to through a single reference, but can be read from through many aliases.

This work also extends *viewpoint adaptation* [19, 26] to apply to every reference capability and introduce the concept of *safe to write*, which, taken together, allow reading from and writing to both unique objects and unique fields. The types of *temporary identifiers* are treated differently from those of permanent paths, which allows the traversal of unique structures, something that is not possible using other approaches [26, 30, 16].

In this system, an alias may have a different reference capability from the initial reference. This addresses a key issue in reference capability systems, namely that sub-typing is not reflexive: an isolated type cannot be assigned to a field or local variable unless the source reference is eliminated with a technique such as *destructive read* or *alias burying* [10]. As a part of this, the type system supports *unaliased types*, which provide static alias tracking without alias analysis.

Reference capabilities based on deny properties also provide a static *region* system [28], requiring no additional annotation. The `trn` reference capability provides a new form of *write region*, in which a region boundary applies to write operations but not read operations.

4.1 Deny Properties

Rather than indicate which operations are allowed on a reference[1], reference capabilities indicate what operations are *denied* on aliases (other references) to the same object. Deny properties distinguish what is denied to the actor that

[1]The term *reference* is used to mean the path currently being considered, and *alias* to mean any other path to the same object.

	Deny global read/write aliases	Deny global write aliases	Allow all global aliases
Deny local read/write aliases	*Isolated (iso)*		
Deny local write aliases	Transition (`trn`)	*Value (val)*	
Allow all local aliases	Reference (`ref`)	Box (`box`)	*Tag (tag)*
	(Mutable)	(Immutable)	(Opaque)

Table 4.1: Reference capability matrix

holds a reference (local aliases) from what is denied to all other actors (global aliases). Each reference capability stands for a pair of local and global deny properties. These are shown in table 4.1. For example, `ref` denies global aliases that can read from or write to the object, but it allows local aliases to both read from and write to it.

No reference capability can deny local aliases that it allows globally. Therefore, some cells in the matrix are empty. For example, there is no reference capability that denies local read and write aliases, but denies only write aliases globally. This is because local execution is a subset of global execution. As such, local deny properties can be weaker than global deny properties, but not stronger.

These deny properties are used to derive the operations permitted on a reference. A reference that denies global read and write aliases is safe to both read and write, i.e. is *mutable*, since it guarantees that no other actor can read from or write to the object. A reference that denies only global write aliases is only safe to read, i.e. *immutable*, since it guarantees no other actor will write to the object, but does not guarantee no other actor will read from it. A reference that allows all global aliases is not safe to either read or write, i.e. it is opaque.

When the local deny properties and the global deny properties of a reference are the same, the reference can be safely sent as an argument to an asynchronous method call to another actor, i.e. it is *sendable*. In other words, when the local alias deny properties are the same as the global alias deny properties, it does not matter which actor holds the reference. Reference capabilities in italics in table 4.1 are *sendable*.

4.2 Short Examples

The following examples are compact and artificial. A more complete example, using a sample program from the compiler distribution, is in section 2.4, along with a line by line explanation.

A `ref` reference to an object denies global read/write aliases. As a result, it is safe to mutate the object, since no other actor can read from it. This is effectively a traditional object-oriented *reference type*.

If an actor has a `box` reference to an object, no alias can be used by other actors to write to that object. This means that other actors may be able to read the object, and aliases in the same actor may be able to write to it (although not both: if the actor can write to the object, other actors cannot read from it). Using `box` for immutability allows a program to enforce read-only behaviour, similar to `const` in C/C++. For example:

```
class List
  fun box size1(): Int => ...
  fun val size2(): Int => ...
```

Note that the receiver reference capability is specified after the keyword `fun`. In `size1`, by indicating that the receiver has `box` reference capability, we can be certain that `this` will not be mutated when calculating its size. In addition, immutability is transitive, so no readable fields of `this` will be mutated either. Since `box` denies global write aliases but does not deny local write aliases, it is possible for `this` to be mutated through some alias if that alias is held by the same actor. The `box` reference functions as a *black box*: the underlying object may be mutable through an alias or it may be immutable through any alias, but in any case it is immutable through this reference. This form of immutability is referred to as *local immutability*.

In `size2`, by indicating that the receiver has `val` reference capability, we make a stronger guarantee: we deny both local and global write aliases. As a result, it is not possible for `this` (and all its readable fields) to be mutated, regardless of other aliases, nor will it be mutated at any time in the future. This form of immutability is referred to as *global immutability*.

Since a `val` reference has the same local and global deny properties, it is possible to *send* a `val` reference to another actor. A `val` reference is effectively a *value type*, similar to values in functional languages.

```
actor Dataflow
  be calculate1(list: List val) => ...
  be calculate2(list: List box) // Not allowed
```

The keyword `actor` is used to indicate a class that can have *behaviours* (asynchronous methods), and the keyword `be` is used to define behaviours. A behaviour is executed asynchronously by the receiving actor, and a given actor executes only one behaviour at a time, making behaviours *atomic*. While executing a behaviour, the receiver sees itself (i.e. `this` in the behaviour) as `ref`, and is able to freely read from and write to its own fields. However, at the call-site, a behaviour does not read from or write to the receiver, and so a behaviour can be called on a `tag` receiver.

In `calculate1`, the `list` parameter is guaranteed to be deeply and globally immutable, because a `val` is guaranteed to have no local or global write aliases. As a result, it is safe to share this object amongst actors. Denying local and global write aliases means no actor can write to the object, regardless of how many actors have an alias to `list`, making concurrent reads safe without copying, locks, or any other runtime safety mechanism. In `calculate2`, a parameter of type `List box` is rejected by the type system, as a `box` does not deny local write aliases, making it unsafe to send a `box` to another actor as the sending actor could retain a mutable alias.

A `tag` reference has no deny properties, but it can be used for *asynchronous* method calls, i.e. calling behaviours. A reference capability with no permissions has appeared in previous work [54], but without allowing asynchronous method calls.

```
actor Dataflow
  be step(list: List val, flow: Dataflow tag) => ...
```

Here, we can call behaviours on `flow`, but we cannot read or write the fields of `flow`. However, when `flow` executes those behaviours asynchronously, it will see itself as a `ref`, allowing it to mutate its own state. As such, `tag` allows actors themselves to have a reference capability, thus integrating them into the type system and allowing threads of control (in the form of actors) to be treated as first-class values. In contrast to existing systems [26], this allows for the formalisation of both dynamic thread creation (actor constructors) and communicating actor graphs of any shape (including cycles).

In order to pass mutable data between actors, we use `iso` references. All mutable reference capabilities deny global read/write aliases, allowing them to be written to because no other actor can read from the object. An `iso` reference also denies local read/write aliases, which means if the `iso` reference is sent to another actor, we are guaranteed that the sending actor no longer holds either read or write aliases to the object sent.

```
actor Dataflow
  be step(list: List iso, flow: Dataflow tag) => ...
```

Here, by passing an `iso` reference, a `Dataflow` actor can mutate the `list` before sending it to the `flow` actor. In order to do this, we must be certain the sending actor does not retain a read or write alias. To this end we use an *aliasing* type system wherein a newly created alias to an object cannot violate the deny properties of the reference being aliased. For example, a newly created alias of an `iso` reference must be neither readable nor writeable (i.e. a `tag`).

To *move* deny properties, we *consume* a reference or use a *destructive read*, both with the expected semantics.

```
actor Dataflow
  be step(list: List iso, flow: Dataflow tag) =>
    flow.step(list, this) // Not allowed
    flow.step(consume list, this)
```

The type system introduces the concept of *unaliased types,* annotated with $\circ$, in order to type values for which an alias has been removed. Here, the `consume` produces a `List iso`$\circ$ which is aliased as a `List iso` when the behaviour is called. The non-destructive read produces a `List iso` which is aliased as a `List tag`, which is rejected by the type system.

Aliases which outlive the execution of an expression are treated differently from *temporary identifiers*, which do not. The use of *temporary identifiers*, combined with *viewpoint adaptation*, allows reading from and writing to isolated objects and isolated fields. Earlier work on isolation and external uniqueness systems [15, 26, 30] does not provide this.

```
actor Dataflow
  be step(list1: List iso, list2: List iso,
      flow: Dataflow tag) =>
    list1.next = consume list2
    flow.step(consume list1)
```

Here, we mutate `list1` by assigning `list2` to its `next` field, maintaining isolation for both `list1` and `list1.next`. Similarly, we could read from or write to fields of `list1.next`, since path traversal is allowed. This also allows calling methods on isolated references and fields of any path depth.

Unsafe reads are prevented by *viewpoint adaptation*, and unsafe writes are prevented by *safe-to-write* rules. For example:

```
actor Dataflow
  fun ref append(list1: List iso,
      list2: List ref) =>
    list1.next = list2 // Not allowed
```

Even if `list1.next` had the type `List ref`, this assignment is rejected. As a result, isolated references form *static regions*, wherein mutable references reachable by the `iso` reference can only be reached via the `iso` reference and immutable references reachable by the `iso` reference are either globally immutable or can only be reached via the `iso` reference.

A `trn` reference makes a novel guarantee: *write uniqueness* without *read uniqueness.* By denying global read/write aliases, but only denying local write aliases, it allows an object to be written to only via the `trn` reference, but read from via other aliases held by the same actor. This allows the object to be mutable while still allowing it to *transition* to an immutable reference capability in the future, in order to share it with another actor.

```
class BookingManager
  var accountant: Accountant
  var all: Map[Date, Booking box]
  var future: Map[Date, Booking trn]
  fun ref close(date: Date) =>
    accountant.account(future.remove(date))

actor Accountant
  be account(booking: Booking val) => ...
```

Here[2] we use a `trn` reference to model bookings that remain mutable until they are closed and sent for accounting. All bookings are in the `all` map, but only mappings that have not been closed out and are still mutable are in the `future` map. When a booking is closed, it is removed from the `future` map, returning a `Booking trn`$\circ$, which is aliased as a `Booking trn`, which is a subtype of `Booking val` and can be shared with the `Accountant` actor. Without a *write unique* type, namely `trn`, this would require copying the `Booking`.

A `trn` reference also forms a *static region*, but with a looser guarantee than an `iso` reference. Mutable references reachable by the `trn` reference can only be reached via the `trn` reference, but immutable references, whether global or local, are not contained in the resulting *write region*.

4.3 Type System

The type system has the format $\Gamma \vdash \texttt{e} : \texttt{ET} \,|\, \Gamma'$ and is defined in figure 4.1. A new environment is generated in order to allow local variable declaration and `consume` to be simply expressed.

Because this simplified type system does not formalise traits, interfaces, type expressions, or generic types, the T-Subsume rule and the subtyping rules account only for reference capabilities.

4.4 Well-Formedness

The rules for a well-formed program are presented in figure 4.3. The WF-Program rule indicates a program is well-formed if all types in the program

[2]In this example, generic types and default reference capabilities (`ref` for objects and `tag` for actors) are used. While the full Pony language supports these, they are not formalised here.

$$\frac{\mathtt{x} \in \Gamma}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \mid \Gamma}\ \textsc{T-Local} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa \mid \Gamma' \quad \mathcal{F}(\mathtt{S},\mathtt{f}) = \mathtt{S}'\,\kappa'}{\Gamma \vdash \mathtt{e.f} : \mathtt{S}'\,\kappa \triangleright \kappa' \mid \Gamma'}\ \textsc{T-Fld}$$

$$\frac{\mathtt{x} \notin \Gamma}{\Gamma \vdash \mathtt{var}\,\mathtt{x} : \mathtt{ET} \mid \Gamma[\mathtt{x} \mapsto \mathtt{ET}]}\ \textsc{T-DeclLocal} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{ET} \mid \Gamma' \quad \Gamma' \vdash \mathtt{e}' : \mathtt{ET}'\,\Gamma''}{\Gamma \vdash \mathtt{e};\mathtt{e}' : \mathtt{ET}' \mid \Gamma''}\ \textsc{T-Seq}$$

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa \quad \Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{S}\,\kappa \mid \Gamma'}{\Gamma \vdash \mathtt{x} = \mathtt{e} : \mathcal{U}(\mathtt{S}\,\kappa) \mid \Gamma'}\ \textsc{T-AsnLocal} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\mathcal{A}} \mathtt{e}' : \mathtt{S}'\,\kappa' \mid \Gamma' \\ \Gamma' \vdash \mathtt{e} : \mathtt{S}\,\kappa \mid \Gamma'' \\ \mathcal{F}(\mathtt{S},\mathtt{f}) = \mathtt{S}'\,\kappa'' \quad \kappa' \leq \kappa'' \\ \vdash \kappa \triangleleft \kappa' \vee \vdash \kappa \triangleleft \kappa'' \end{array}}{\Gamma \vdash \mathtt{e.f} = \mathtt{e}' : \mathcal{U}(\mathtt{S}'\,\kappa \triangleright \kappa'') \mid \Gamma''}\ \textsc{T-AsnFld}$$

$$\frac{\begin{array}{c} \mathcal{M}(\mathtt{S},\mathtt{m}) = (\mathtt{T}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{ET}) \\ \Gamma_0 = \Gamma \quad \Gamma_{\mathtt{i}-1} \vdash_{\mathcal{A}} \mathtt{e}_{\mathtt{i}} : \mathtt{T}_{\mathtt{i}} \mid \Gamma_{\mathtt{i}} \\ \Gamma_{|\overline{\mathtt{e}}|} \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T} \mid \Gamma' \end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{ET} \mid \Gamma'}\ \textsc{T-Sync} \qquad \frac{\begin{array}{c} \mathcal{M}(\mathtt{A},\mathtt{b}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A}\,\mathtt{tag}) \\ \Gamma_0 = \Gamma \quad \Gamma_{\mathtt{i}-1} \vdash_{\mathcal{A}} \mathtt{e}_{\mathtt{i}} : \mathtt{T}_{\mathtt{i}} \mid \Gamma_{\mathtt{i}} \\ \Gamma_{|\overline{\mathtt{e}}|} \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{A}\,\mathtt{tag} \mid \Gamma' \end{array}}{\Gamma \vdash \mathtt{e.b}(\overline{\mathtt{e}}) : \mathtt{A}\,\mathtt{tag} \mid \Gamma'}\ \textsc{T-Async}$$

$$\frac{\begin{array}{c} \mathcal{M}(\mathtt{C},\mathtt{k}) = (\mathtt{C}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{C}\,\mathtt{ref}\circ) \\ \Gamma_0 = \Gamma \quad \Gamma_{\mathtt{i}-1} \vdash_{\mathcal{A}} \mathtt{e}_{\mathtt{i}} : \mathtt{T}_{\mathtt{i}} \mid \Gamma_{\mathtt{i}} \end{array}}{\Gamma \vdash \mathtt{C.k}(\overline{\mathtt{e}}) : \mathtt{C}\,\mathtt{ref}\circ \mid \Gamma_{|\overline{\mathtt{e}}|}}\ \textsc{T-Ctor} \qquad \frac{\begin{array}{c} \mathcal{M}(\mathtt{A},\mathtt{k}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{A}\,\mathtt{tag}) \\ \Gamma_0 = \Gamma \quad \Gamma_{\mathtt{i}-1} \vdash_{\mathcal{A}} \mathtt{e}_{\mathtt{i}} : \mathtt{T}_{\mathtt{i}} \mid \Gamma_{\mathtt{i}} \end{array}}{\Gamma \vdash \mathtt{A.k}(\overline{\mathtt{e}}) : \mathtt{A}\,\mathtt{tag} \mid \Gamma_{|\overline{\mathtt{e}}|}}\ \textsc{T-Ator}$$

$$\frac{\mathtt{x} \in \Gamma}{\Gamma \vdash \mathtt{consume}\,\mathtt{x} : \mathcal{U}(\Gamma(\mathtt{x})) \mid \Gamma \backslash \{\mathtt{x}\}}\ \textsc{T-Consume} \qquad \frac{\Gamma \backslash \{\mathtt{x} \mid \neg \mathit{Sendable}(\Gamma(\mathtt{x}))\} \vdash \mathtt{e} : \mathtt{ET} \mid \Gamma'}{\Gamma \vdash \mathtt{recover}\,\mathtt{e} : \mathcal{R}(\mathtt{ET}) \mid \Gamma \cap \Gamma'}\ \textsc{T-Rec}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{ET}' \mid \Gamma' \quad \mathcal{A}(\mathtt{ET}') \leq \mathtt{T}}{\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{T} \mid \Gamma'}\ \textsc{T-Alias} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa\circ \mid \Gamma'}{\Gamma \vdash \mathtt{e} : \mathtt{S}\,\kappa \mid \Gamma'}\ \textsc{T-Subsume}$$

Figure 4.1: Expression typing

$$\frac{\mathtt{ET} \leq \mathtt{ET}'' \quad \mathtt{ET}'' \leq \mathtt{ET}'}{\mathtt{ET} \leq \mathtt{ET}'} \qquad \frac{}{\mathtt{S}\,\kappa\circ \leq \mathtt{S}\,\kappa} \qquad \frac{\kappa \leq \kappa'}{\mathtt{S}\,\kappa \leq \mathtt{S}\,\kappa'}$$

$$\mathtt{iso} \leq \mathtt{trn} \leq \{\mathtt{ref}, \mathtt{val}\} \leq \mathtt{box} \leq \mathtt{tag}$$

$$\mathit{Sendable}(\mathtt{T}) \ \mathit{iff}\ \mathtt{T} = \mathtt{S}\,\kappa \wedge \kappa \in \{\mathtt{iso}, \mathtt{val}, \mathtt{tag}\}$$

Figure 4.2: Sub-types and auxiliary type definitions

$$\frac{\forall \mathtt{S} \in \mathtt{P}.\ \vdash \mathtt{S}\diamond}{\vdash \mathtt{P}\diamond}\ \text{WF-PROGRAM}$$

$$\frac{\begin{array}{c}\mathcal{P}(\mathtt{S}) = \overline{\mathtt{F}}\,\overline{\mathtt{K}}\,\overline{\mathtt{M}}\,\overline{\mathtt{B}} \\ \forall \mathtt{var\,f} : \mathtt{S}\,\kappa \in \overline{\mathtt{F}}.\ \vdash \mathtt{S} \diamond \quad \forall \mathtt{K} \in \overline{\mathtt{K}}.\mathtt{S} \vdash \mathtt{K}\diamond \\ \forall \mathtt{M} \in \overline{\mathtt{M}}.\mathtt{S} \vdash \mathtt{M} \diamond \quad \forall \mathtt{B} \in \overline{\mathtt{B}}.\mathtt{S} \vdash \mathtt{B}\diamond\end{array}}{\vdash \mathtt{S}\diamond}\ \text{WF-TYPE}$$

$$\frac{[\mathtt{this} \mapsto \mathtt{C\,var}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{C\,ref}\circ}{\mathtt{C} \vdash \mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}\diamond}\ \text{WF-CTOR}$$

$$\frac{[\mathtt{this} \mapsto \mathtt{S}\kappa_{\mathtt{r}}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{ET}}{\mathtt{S} \vdash \mathtt{fun}\,\kappa\,\mathtt{m}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) : \mathtt{ET} \Rightarrow \mathtt{e}\diamond}\ \text{WF-SYNC}$$

$$\frac{\begin{array}{c}\mathit{Sendable}(\mathtt{T_i}) \\ [\mathtt{this} \mapsto \mathtt{A\,var}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{A\,tag}\end{array}}{\mathtt{A} \vdash \mathtt{new\,k}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}\diamond}\ \text{WF-ATOR}$$

$$\frac{\begin{array}{c}\mathit{Sendable}(\mathtt{T_i}) \\ [\mathtt{this} \mapsto \mathtt{A\,var}, \overline{\mathtt{x}} \mapsto \overline{\mathtt{T}}] \vdash \mathtt{e} : \mathtt{A\,tag}\end{array}}{\mathtt{A} \vdash \mathtt{be\,b}(\overline{\mathtt{x}} : \overline{\mathtt{T}}) \Rightarrow \mathtt{e}\diamond}\ \text{WF-ASYNC}$$

Figure 4.3: Well-formed programs

- $\mathtt{z} \in \varphi$ iff $\mathtt{z} \in \mathit{dom}(\varphi\downarrow_2)$
- $\alpha \in \chi$ iff $\alpha \in \mathit{dom}(\chi)$
- $\Delta \vdash \alpha \in \chi$ iff $\alpha \in \mathit{dom}(\chi)$
- $\Delta \vdash \iota \in \chi$ iff $\exists \iota'$ such that $\Delta \vdash \iota' \in \chi$ and $\Delta, \chi, \iota' \vdash \iota : \_$
- $\mathcal{M}(\varphi, \chi) = \mathcal{M}(\chi(\varphi(\mathtt{this})\downarrow_1, \varphi\downarrow_1)$

Figure 4.4: Auxiliary well-formedness definitions

are well-formed. The WF-Type rule indicates that a type is well-formed if the types of all of its fields are well-formed, its constructors are well-formed, and its synchronous and asynchronous methods are well-formed. The WF-Ctor, WF-Sync, and WF-Async rules indicate that a method is well-formed when the body of the method in results in a subtype of the return type of the method. Additionally, WF-Async requires that behaviour parameters are *Sendable*. The body of the method is evaluated using an environment composed of the receiver and the method parameters, each mapped to their type, as shown in figure 4.1.

4.4.1 Constructor Reference Capabilities

The reference capability of the receiver and the return type can vary for synchronous methods, but not for constructors or asynchronous methods. Constructors always operate on a `ref` receiver, since the constructor must write to the new object's fields, and return a `ref`$\circ$ result, since the new object is initially mutable but also unaliased, as the constructor's reference to the receiver (`this`) is discarded when the constructor returns. This allows a constructor that is passed only sendable references as parameters to be embedded in a `recover` expression, which allows constructing an object with any reference capability.

In the full language, a class constructor may specify an alternate reference capability for the constructed object.

4.4.2 Behaviour Reference Capabilities

Asynchronous methods always operate on a `ref` receiver. This is because the receiver of an asynchronous method is always an actor; when the body is executed, a new stack with the receiver as the root actor is created. Since each actor executes the body of a single behaviour (or asynchronous constructor) at any given time, every behaviour body can read from and write to the receiver.

Since an asynchronous method cannot, by definition, perform any operations at the call site to construct a return value before returning, the receiver is returned to allow chaining method calls. For the same reason, a behaviour can be called on a `tag` actor (i.e. an actor of any reference capability).

4.5 Viewpoint Adaptation

When reading a field `f` from an object ι we obtain a temporary identifier. The reference capability of this temporary identifier must be a combination of κ, the reference capability of the path leading to ι, and κ', the reference capability with which ι sees the field. This is expressed through the operator $\triangleright$, defined in table 4.2. When reading a field through an origin, the result must not violate

$\kappa \triangleright \kappa'$	κ'					
κ	iso	trn	ref	val	box	tag
iso	iso	tag	tag	val	tag	tag
trn	iso	trn	box	val	box	tag
ref	iso	trn	ref	val	box	tag
val	val	val	val	val	val	tag
box	tag	box	box	val	box	tag
tag	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Table 4.2: Viewpoint adaptation. Given an object reference of capability κ with a field of capability κ', reading the field via that reference produces a reference of capability $\kappa \triangleright \kappa'$.

$\kappa \triangleleft \kappa'$	κ'					
κ	iso	trn	ref	val	box	tag
iso	$\surd$			$\surd$		$\surd$
trn	$\surd$	$\surd$		$\surd$		$\surd$
ref	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
val						
box						
tag						

Table 4.3: Safe to write. Given an object reference of capability κ, the relation $\kappa \triangleleft \kappa'$ expresses which reference capabilities κ' are safe to store into a field of an object via that reference.

the deny properties of either the origin or the field. For example, reading a `ref` field from an `iso` reference returns `tag` — thus we do not violate the deny properties of the origin or the field itself.

Storing a reference into a field of an object ι is legal if the type of the reference is both a subtype of the type of the field and also *safe to write* into the origin. The relation $\kappa \lhd \kappa'$, as defined in table 4.3, expresses which reference capabilities κ' are safe to write into origin κ. When writing to a field through an origin, no alias of the object being written may exist that would violate the deny properties of the origin. Therefore, all entries for `val`, `box` and `tag` are empty. Moreover, only `iso`, `val` or `tag` references may be stored into an `iso` origin; all other writes would violate the region introduced by the `iso` origin.

4.6 Reference Capability Recovery

The evaluation of an expression which has access only to sendable variables (i.e. `iso`, `val`, and `tag`) can return a sendable type, even if the result of the expression would otherwise not be sendable. This is an extension of previous work on *recovery* [26], which is related to work on *borrowing* [30]. Such expressions are introduced using the `recover` keyword (T-REC). The return type of `recover e` is the sendable version of the return type of `e`. For example, if `e` has type `ref`, then `recover e` has type `iso`, and if `e` has type `ref∘`, then `recover e` has type `iso∘`.

Definition 4.1. Reference capability recovery

$$\mathcal{R}(\mathtt{S}\,\kappa\,\phi) = \begin{cases} \mathtt{S}\;\mathtt{iso}\,\phi & \textit{iff}\ \kappa \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{ref}\} \\ \mathtt{S}\;\mathtt{val} & \textit{iff}\ \kappa \in \{\mathtt{val}, \mathtt{box}\} \\ \mathtt{S}\;\mathtt{tag} & \textit{otherwise} \end{cases}$$

$\mathcal{R}(\mathtt{ET})$ is the sendable reference capability that retains the same local read and/or write guarantee. In other words, a writeable reference capability can become `iso` and a readable reference capability can become `val`. In Pony, explicit `recover` expressions are used along with implicit recovery detected by the compiler.

4.7 Treatment of Actors

Actors introduce the question of who may read or update the actor's fields, the possibility of synchronous calls on actors, and the type required for asynchronous calls.

Field read and write requires that the actor should see itself as a `ref`. As a result, any other actor will see it as `tag`. Therefore no other actor except the current one will be allowed to observe an actor's fields — a nice consequence of the type system.

By a similar argument, because the actor sees itself as `ref`, any other paths that point to it will do so as `box`, `ref`, or `tag`, and this means that the actor may call synchronous methods on itself, provided that the receiver reference capability of the method declaration is `ref`, `box`, or `tag`. Interestingly, for asynchronous (behaviour) calls, the receiving actor only needs to be seen as a `tag`, even though the receiver reference capability in the behaviour is `ref`. This is in contrast to method calls, where the receiving object or actor has to be seen as a reference capability which is a subtype of the receiver reference capability in the method declaration. The looser requirement for actors is sound, because, as discussed above, no other actor may obtain access to the actor's state.

4.8 Aliasing and Unaliasing

Assignment operations discard aliases, as they return the previous value of the left-hand side (AsnLocal and AsnField) after overwriting it. The fact that an alias has been discarded is important in the cases where the reference capability is unique (`iso` or `trn`). This is indicated through the unaliased annotation $\circ$, which expresses that there is no stable path to the corresponding object.

Because unaliasing is of importance only when the underlying reference capability is `iso`, `trn` or `ref`, the unaliasing operation $\mathcal{U}$ is defined, which takes a type and returns an extended type, as seen in definition 4.2. This operator is used whenever an alias is discarded, for example in the type system rules T-AsnLocal and T-AsnFld. Object constructors also introduce unaliased values, as indicated in the type system rule T-Ctor.

Some operations introduce stable aliases (e.g. assignment), while others introduce only unstable ones (e.g. field read). The distinction is expressed in the type system through the difference between the type judgments $\Gamma \vdash \mathtt{e} : \mathtt{ET}$ and the *aliased* type judgment $\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{ET}$. For example, when assigning an expression `e` to a variable `x`, the right-hand side is typed in the judgment $\vdash_{\mathcal{A}}$ (cf. T-AsnLocal). The aliasing judgement is also applied to the receiver and arguments of method calls and asynchronous behaviours (T-Sync and T-Async), the arguments to object and actor constructors (T-Ctor and T-Ator), and the right-hand side of a field assignment (T-AsnFld).

The aliased type judgment $\Gamma \vdash_{\mathcal{A}} \mathtt{e} : \mathtt{ET}$ is defined in terms of the unaliased type judgment $\Gamma \vdash \mathtt{e} : \mathtt{ET}'$, where ET has to be a super-type of the aliased version of $\mathtt{ET}'$, i.e. $\mathcal{A}(\mathtt{ET}') \leq \mathtt{ET}$. The operation $\mathcal{A}(\mathtt{ET})$ gives the type that an alias of ET would have. When aliasing an unaliased type there is no previous alias to consider, and therefore $\mathcal{A}(\mathtt{S}\,\kappa\circ) = \mathtt{S}\,\kappa$. For other types, the result must be the minimal super-type of the underlying type which is locally compatible with it, i.e. $\mathcal{A}(\mathtt{S}\,\kappa) = \mathtt{S}\,\kappa'$ where $\kappa' \leq \mathcal{A}(\kappa')$ and $\mathcal{A}(\kappa')$ does not locally deny κ'.

Definition 4.2. Aliasing and unaliasing.

- $\mathcal{A}(\mathtt{S}\,\kappa\circ) = \mathtt{S}\,\kappa$
- $\mathcal{A}(\mathtt{S}\,\kappa) = \begin{cases} \mathtt{S}\,\mathtt{tag} & \textit{iff } \kappa = \mathtt{iso} \\ \mathtt{S}\,\mathtt{box} & \textit{iff } \kappa = \mathtt{trn} \\ \mathtt{S}\,\kappa & \textit{otherwise} \end{cases}$
- $\mathcal{U}(\mathtt{S}\,\kappa) = \begin{cases} \mathtt{S}\,\kappa\circ & \textit{iff } \kappa \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{ref}\} \\ \mathtt{S}\,\kappa & \textit{otherwise} \end{cases}$

Thus, through a combination of aliasing and unaliasing, the programmer can obtain unique types when needed. For example, for $\mathtt{x}$ and $\mathtt{y}$ of type $\mathtt{C}\,\mathtt{trn}$, the assignment $\mathtt{x} = \mathtt{y}$ is illegal, because the aliased type of $\mathtt{y}$ is $\mathtt{C}\,\mathtt{box}$ and $\mathtt{C}\,\mathtt{box} \not\leq \mathtt{C}\,\mathtt{trn}$. However, the assignment $\mathtt{x} = \mathtt{consume}\,\mathtt{y}$ is legal, because the type of $\mathtt{consume}\,\mathtt{y}$ is $\mathtt{C}\,\mathtt{trn}\circ$, and the alias of $\mathtt{C}\,\mathtt{trn}\circ$ is $\mathtt{C}\,\mathtt{trn}$.

4.9 Consistent heap visibility

The core of the soundness of this approach is *consistent heap visibility*, which requires that aliasing in the heap must satisfy all the deny properties specified by the capabilities attached to fields and variables. This leads to the notions of local and global compatibility. Namely, two capabilities are *locally compatible* $\kappa \sim_\ell \kappa'$ if neither has a local deny property that prevents the existence of the other. Similarly, they are *globally compatible*, $\kappa \sim_g \kappa'$, if neither has a global deny property that prevents the existence of the other. These relationships are defined in table 4.4, e.g. $\mathtt{ref} \sim_\ell \mathtt{ref}$ but $\mathtt{ref} \not\sim_g \mathtt{ref}$. Both relations are symmetric. It is interesting to note that global compatibility is a subset of local compatibility.

Figure 4.5 shows a diagrammatic representation of a heap χ_0 which contains actors α_1 and α_2, and objects $\iota_{10}...\iota_{19}$. The top rectangles indicate stack frames, for example $\chi_0(\alpha_1) = (\_, \_, \alpha_1 \cdot \varphi_1 \cdot \varphi_2, \_)$ and $\varphi_1(\mathtt{this}) = \iota_{10}$ and $\varphi_2(\mathtt{t_2}) = \iota_{18}$. The objects are in rounded boxes, and the annotated arrows indicate the contents of their fields, e.g. $\chi_0(\iota_{14}, \mathtt{f10}) = \iota_{19}$. The annotations next to the field identifiers (ref, val, etc.) give types to the variables. Note that $\alpha_1 = \iota_{10}$ and $\alpha_2 = \iota_{14}$.

For consistent heap visibility we require that different paths originating from the same actor and pointing to the same object have locally consistent visibility, while paths originating from different actors and pointing to the same

$\kappa \sim \kappa'$			κ'			
κ	iso	trn	ref	val	box	tag
iso						ℓ, g
trn					ℓ	ℓ, g
ref			ℓ		ℓ	ℓ, g
val				ℓ, g	ℓ, g	ℓ, g
box		ℓ	ℓ	ℓ, g	ℓ, g	ℓ, g
tag	ℓ, g	ℓ, g	ℓ, g	ℓ, g	ℓ, g	ℓ, g

Table 4.4: Compatible capabilities.

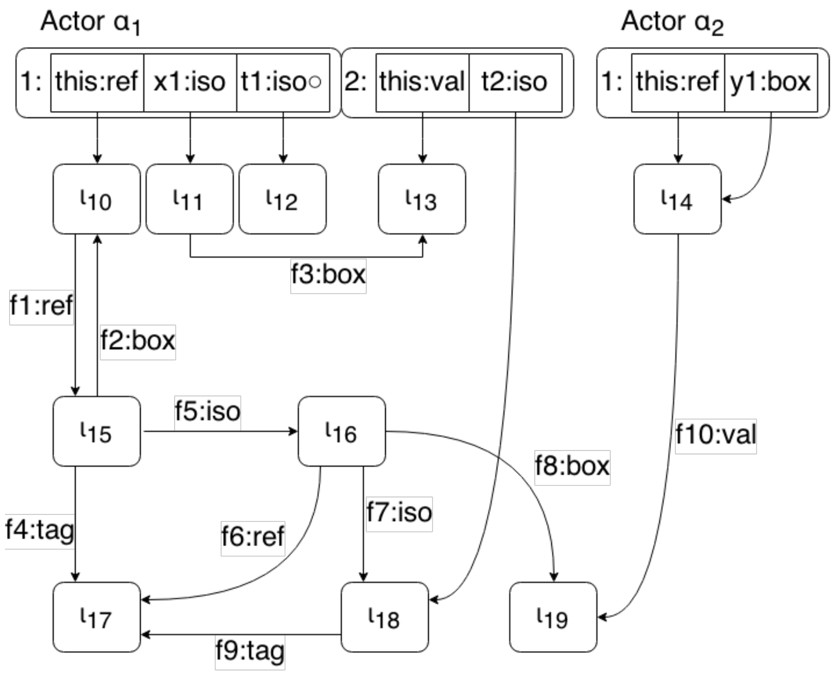


Figure 4.5: A representation of part of a heap.

$$\begin{array}{rcccc} \Gamma & \in & \mathit{Env} & = & \mathit{LocalID} \rightarrow \mathit{ExtType} \\ \Delta & \in & \mathit{GlobalEnv} & = & (\mathit{ActorAddr} \times \mathit{Integer}) \rightarrow \mathit{Env} \\ p & \in & \mathit{Path} & = & (\mathit{Integer} \times \mathit{LocalID}) \cdot \overline{\mathit{FieldID}} \end{array}$$

Figure 4.6: Global environments and paths.

- $\Delta, \chi, \iota \vdash \iota : \texttt{ref}, (0, \texttt{this})$
- $\Delta, \chi, \alpha \vdash \iota : \kappa, (i, \texttt{z})$ iff $\chi(\alpha, (i \cdot \texttt{z})) = \iota$ and $\Delta(\alpha, i, \texttt{z}) = \texttt{S}\, \kappa\, \phi$ and $\kappa \neq \texttt{tag}$
- $\Delta, \chi, \iota \vdash \iota' : \kappa \blacktriangleright \kappa', p \cdot \texttt{f}$ iff $\Delta, \chi, \iota \vdash \iota'' : \kappa, p$ and $\chi(\iota'', \texttt{f}) = \iota'$ and $\mathcal{F}(\chi(\iota'') \downarrow_1, \texttt{f}) = \texttt{S}\, \kappa'$ and $\kappa \blacktriangleright \kappa' \neq \texttt{tag}$
- $\Delta, \chi, \iota \vdash \iota' : \kappa$ iff $\exists p$ such that $\Delta, \chi, \iota \vdash \iota' : \kappa, p$

Figure 4.7: Visibility.

object have globally consistent visibility. For example, in figure 4.5 the path `this.f1.f5.f8` starting at the first frame of actor α_1 and the path `this.f10` at the first frame of actor α_2 are aliases, as they both reach object ι_{19}. The first path sees ι_{19} as `tag`, while the second sees it as `val`. These are globally compatible capabilities, and therefore these paths preserve consistent heap visibility. On the other hand, if we added a `ref` field to ι_{15}, such that it pointed to ι_{19}, the resulting capabilities would not be globally compatible.

For the formal definition of consistent heap visibility, we need notions of:

1. Paths p and global environments Δ, which give types to the local variables and temporaries in each frame or message, as defined in fig. 4.6.
2. Path visibility $\Delta, \chi, \iota \vdash \iota' : \kappa, p$, which says that the object or actor ι sees the object or actor ι' as capability κ through path p, as defined in fig. 4.7.
3. Topological properties of paths, as defined in figure 4.8.

Environments, Γ, map variables (i.e. local variables or temporaries) to extended types and global environments, Δ, map actor addresses and integers to environments. Figure 4.5 indicates the types assigned to local variables through the annotations. Thus, it has an implicit global environment Δ_0, such that $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{10} : \texttt{ref}, (1, \texttt{this})$, and $\Delta_0, \chi_0, \alpha_2 \vdash \iota_{19} : \texttt{val}, (1, \texttt{this}) \cdot \texttt{f10}$.

To define path visibility, the notion of deep viewpoint adaptation $\kappa \blacktriangleright \kappa'$ is used, which combines two capabilities as given in figure 4.8. The definition ensures that $\kappa \blacktriangleright \kappa' = \kappa'$ if κ is writeable (deep mutability), $\kappa \blacktriangleright \kappa' = \texttt{val}$ if either κ or κ' is `val` (deep immutability) and $\texttt{box} \blacktriangleright \kappa' = \texttt{box}$ unless $\kappa' \in \{\texttt{iso}, \texttt{val}, \texttt{tag}\}$. For example, $\texttt{iso} \blacktriangleright \texttt{ref} = \texttt{ref}$.

- $\kappa \blacktriangleright \kappa' = \begin{cases} \kappa' & \textit{if } \kappa \in \{\texttt{iso}, \texttt{trn}, \texttt{ref}\} \\ \texttt{val} & \textit{if } \kappa = \texttt{val} \lor \kappa' = \texttt{val} \\ \texttt{box} & \textit{if } \kappa = \texttt{box} \land \kappa' \notin \{\texttt{iso}, \texttt{val}, \texttt{tag}\} \\ \texttt{tag} & \textit{otherwise} \end{cases}$
- $\chi, \alpha \vdash p_1 \cdot \texttt{f} \sim p_2 \cdot \texttt{f}$ iff $\chi(\alpha, p_1) = \chi(\alpha, p_2)$
- $\chi, \alpha \vdash (i, \texttt{z}) \sim (i, \texttt{z})$
- $\chi, \alpha \vdash \iota \in p$ iff $\exists p', \bar{\texttt{f}}$ such that $p = p'.\bar{\texttt{f}}$ and $\chi(\alpha, p') = \iota$
- $\chi(\alpha, (i, \texttt{z}) \cdot \overline{\texttt{f}}) = \chi(\varphi_i(\texttt{z}), \overline{\texttt{f}})$ where $\chi(\alpha) \downarrow_4 = \alpha \cdot \overline{\varphi}$
- $\chi(\alpha, (-i, \texttt{x}_\texttt{j}) \cdot \overline{\texttt{f}}) = \chi(v_j, \overline{\texttt{f}})$ where $Q(\chi, \alpha) = \overline{\mu}$ and $\mu_i = (\_, \overline{v})$
- $\textit{Stable}(\Delta, \alpha, (i, \texttt{z}) \cdot \overline{\texttt{f}})$ iff $\Delta(\alpha, i, \texttt{z}) \notin \{\texttt{iso}, \texttt{trn}\}$ or $\texttt{z} \neq \texttt{t}$

Figure 4.8: Topological properties of paths.

The rules in figure 4.7 say that an address sees itself as `ref`, an actor sees a stack identifier as the capability provided by Δ, and an address sees another address as a deep viewpoint adapted capability. Note that, for visibility, `tag` types are not seen. Therefore, our example gives us:

- $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{10} : \texttt{ref}, (1, \texttt{this})$, but also
  $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{10} : \texttt{box}, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f2}$.
- $\Delta_0, \chi_0, \alpha_2 \vdash \iota_{19} : \texttt{val}, (1, \texttt{this}) \cdot \texttt{f10}$, but also
  $\Delta_0, \chi_0, \alpha_1 \vdash \iota_{19} : \texttt{tag}, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f5} \cdot \texttt{f8}$.

In figure 4.8, two paths are compatible if they share the last step or they are the same identifier with no fields, an address ι is in a path if some prefix of the path points to ι, and a path is stable, $\textit{Stable}(\Delta, \alpha, p)$, if its initial identifier is not a unique temporary. For example, $\chi_0, \alpha_2 \vdash (1, \texttt{this}) \cdot \texttt{f10} \sim (1, \texttt{y1}) \cdot \texttt{f10}$. Also, $\textit{Stable}(\Delta_0, \alpha_1, (1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f4})$ and $\neg\textit{Stable}(\Delta_0, \alpha_1, (2, \texttt{t2}) \cdot \texttt{f9})$, even though the two paths are aliases.

Consistent heap visibility is defined in figure 4.9. It requires:

1. Global compatibility. Any two distinct actors that can see the same address must see that address with globally compatible capabilities.
2. Local compatibility. An actor that sees an address in multiple ways must either see compatible paths or locally compatible capabilities.
3. Containment properties of `iso` and `trn`. Given α that sees ι as some unique κ and sees ι' as κ' via some stable p', and given that ι sees ι' as κ'':

$\mathit{WFV}(\Delta, \chi)$ *iff*
$\forall \alpha, \alpha', \iota, \iota' \in \chi . \forall \kappa, \kappa', p, p', \mathtt{t}$ where $\mathit{Stable}(\Delta, \alpha, p)$ and $\mathit{Stable}(\Delta, \alpha, p')$

1. If $\Delta, \chi, \alpha \vdash \iota : \kappa$ and $\Delta, \chi, \alpha' \vdash \iota : \kappa'$ and $\alpha \neq \alpha'$ then $\kappa \sim_g \kappa'$
2. If $\Delta, \chi, \alpha \vdash \iota : \kappa, p$ and $\Delta, \chi, \alpha \vdash \iota : \kappa', p'$ then
   (a) $\chi, \alpha \vdash p \sim p'$ or
   (b) $\kappa \sim_\ell \kappa'$
3. If $\Delta, \chi, \alpha \vdash \iota : \kappa$ and $\Delta, \chi, \alpha \vdash \iota' : \kappa', p'$ and $\Delta, \chi, \iota \vdash \iota' : \kappa''$ and $\kappa \in \{\mathtt{iso}, \mathtt{trn}\}$ then
   (a) $\chi, \alpha \vdash \iota \in p'$ or
   (b) $\kappa'' \in \{\mathtt{val}, \mathtt{box}\}$ and $\kappa' \sim_g \mathtt{val}$ or
   (c) $\kappa'' \in \{\mathtt{iso}, \mathtt{trn}, \mathtt{ref}\}$ and $\kappa \sim_\ell \kappa'$
4. If $\Delta(\alpha, i, \mathtt{t}) = \mathtt{S}\,\kappa$ and $\kappa \in \{\mathtt{iso}, \mathtt{trn}\}$ and $\chi(\alpha, i, \mathtt{t}) = \chi(\alpha, p_1) = \iota$ then
   (a) $p_1 = (i, \mathtt{t})$ or
   (b) $\exists \iota', \kappa', p_2, \overline{\mathtt{f}}$ such that
      i. $\kappa \leq \kappa'$
      ii. $\kappa' \in \{\mathtt{iso}, \mathtt{trn}\}$
      iii. $p_1 = p_2 \cdot \overline{\mathtt{f}}$
      iv. $\Delta, \chi, \alpha \vdash \iota' : \kappa', p_2$
      v. $\Delta, \chi, \iota' \vdash \iota : \kappa, \overline{\mathtt{f}}$

Figure 4.9: Well-formed visibility.

   (a) ι' must be contained by ι, or
   (b) neither ι nor α can write to ι', or
   (c) ι can write to ι' and α sees ι' as locally compatible with κ.

4. Properties of unique temporary identifiers. Given `t` that points to ι , some other path p_1 to the same ι must be either:
   (a) also `t` or
   (b) that path p_1 must have a prefix p_2 that sees some ι' with a unique capability κ' less precise than κ and ι' must see ι as κ.

An implication of well-formed visibility is that if two variables (temporary or otherwise) are aliases and one of them has unique type (aliased or unaliased) then 1) they come from the same actor and 2) they are either the same variable or they have locally compatible capabilities, cf. lemmas 4.8 and 4.9. Note that *WFV.1 − 3* are concerned with stable paths only, while *WFV.4* is about unstable paths. In particular, *WFV.4* allows a unique temporary to break the requirements from *WFV.3* and alias something writeable from a unique.

The heap from figure 4.5 has consistent visibility. The paths $(1, \texttt{this}) \cdot \texttt{f1} \cdot \texttt{f5} \cdot \texttt{f8}$ from α_1 and $(1, \texttt{this}) \cdot \texttt{f10}$ from α_2 satisfy *WFV.1*, while $(1, \texttt{x1}) \cdot \texttt{f4}$ from α_1 and $(2, \texttt{this})$ from α_1 satisfy *WFV.2* and *WFV.3*. On the other hand, the temporary $(2, \texttt{t2})$ is not stable, and therefore not restricted by *WFV.2* or *WFV.3*, but does adhere to *WFV.4*. Finally, the assignment `this.f1.f5.f6` = `this.f1.f5.f7` would break *WFV.2*, while setting `t2` to point to ι_{15} would break *WFV.4*.

4.10 Soundness

A heap χ is well-formed as defined in figure 4.10 if all objects in the heap are well-formed, all actors in the heap are well-formed, and visibility is well-formed. An object is well-formed if all its fields belong to the type defined in the object's class. An actor is well-formed if its stack frames and messages are well-formed. A stack frame is well-formed if 1) its receiver and arguments are well-formed, 2) all local identifiers are well-formed, 3) if it is the only stack frame, it has no continuation and the receiver is the actor, 4) if it is not the only stack frame, its return value and temporary identifiers are well-formed with regard to the previous frame, and 5) if it is the last frame, temporary identifiers are well-formed and the expression has the expected type.

- $\Delta \vdash \chi\diamond$ iff $\forall \iota, \alpha \in dom(\chi)$, $\chi \vdash \iota\diamond$ and $\Delta, \chi \vdash \alpha\diamond$ and $WFV(\Delta, \chi)$
- $\chi \vdash \iota\diamond$ iff $\forall \mathtt{f}$, $\mathcal{F}(\chi(\iota)\downarrow_1, \mathtt{f}) = \mathtt{S}\,\kappa$ implies $\chi(\iota, \mathtt{f})\downarrow_1 = \mathtt{S}$
- $\Delta, \chi \vdash \alpha\diamond$ iff $\chi(\alpha) = (\_, \_, \bar{\mu}, \alpha \cdot \overline{\varphi}, \mathtt{e})$ and $\forall i$, $\Delta, \chi, \alpha \vdash \varphi_i, i\diamond$ and $\forall j$, $\Delta, \chi \vdash \mu_j, j\diamond$
- $\Delta, \chi, \alpha \vdash \varphi, i\diamond$ iff given $\varphi = (\mathtt{m}, \_, \mathtt{E}[\cdot])$ and $\mathcal{M}(\varphi, \chi) = (\mathtt{T}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{ET})$ and $\Delta(\alpha, i) = \Gamma$ then

  1. $\Gamma(\mathtt{this}) = \mathtt{T}$ and $\forall j \in 1..|\overline{\mathtt{T}}|.\Gamma(\mathtt{x_j}) = \mathtt{T_j}$
  2. $\forall \mathtt{z} \in \varphi$, $\Gamma(\mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ and $\chi(\varphi(\mathtt{z}))\downarrow_1 = \mathtt{S}$
  3. If $i = 1$ then $\mathtt{E}[\cdot] = \cdot$ and $\varphi(\mathtt{this}) = \alpha$
  4. If $i > 1$, given $\chi(\alpha)\downarrow_4 = \alpha \cdot \overline{\varphi}$ and $\Gamma' = \Delta(\alpha, i-1)$ and $\mathtt{t} \notin \Gamma'$ and $\Gamma'' = \Gamma'[\mathtt{t} \mapsto \mathtt{ET}]$ then
     (a) $\Gamma'' \vdash \mathtt{E}[\mathtt{t}] : \mathcal{M}(\varphi_{i-1}, \chi)\downarrow_4$
     (b) $WFT(\Delta[(\alpha, i) \mapsto \Gamma''], \chi, \alpha, i, \mathtt{E}[\mathtt{t}])$
  5. If $i = |\chi(\alpha)\downarrow_4|$ then $WFT(\Delta, \chi, \alpha, i, \mathtt{e})$ and $\Gamma \vdash \mathtt{e} : \mathtt{ET}$

- $\Delta, \chi, \alpha \vdash \mu, i\diamond$ iff given $\mu = (\mathtt{b}, \overline{v})$ and $v_j = \iota$ and $\mathcal{M}(\chi(\alpha)\downarrow_1, \mathtt{b}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_)$ and $\Delta(\alpha, -i) = \Gamma$ then

  1. $\chi(\iota)\downarrow_1 = \mathtt{S_j}$
  2. $\Gamma(\mathtt{x_j}) = \mathtt{S}\,\kappa$

Figure 4.10: Well-formed heaps.

4.10.1 Treatment of temporaries

Temporaries with unique capabilities, `iso` or `trn`, are fragile: on the one hand they may break the encapsulation of other `iso` or `trn` objects. For example, because $\texttt{iso} \rhd \texttt{iso} = \texttt{iso}$, a field read (FLD) may return a temporary pointing within the encapsulation of `iso`. On the other hand, an assignment to another field or variable might break *their* encapsulation.

To be well-formed, it is required that in a frame, no more than one temporary has an `iso` or `trn` capability, and this temporary appears on a field assignment or a field read. In addition, any temporaries that appear within a recover expression are either inaccessible from any frame or are only accessible through sendable local variables.

Definition 4.3. Well-formed temporaries. $\mathit{WFT}(\Delta, \chi, \alpha, i, \texttt{e})$ iff:

1. No temporary appears more than once in $\texttt{e}$.
2. If $\mathcal{T}(\Gamma) \neq \emptyset$, then $\texttt{e} \equiv \texttt{E}[\texttt{e}']$, where $\texttt{e}'$ is a redex of the form $\texttt{t.f}$ or $\texttt{t.f} = \texttt{y}$, and $\mathcal{T}(\Gamma) = \{\texttt{t}\}$, where $\Gamma = \Delta(\alpha, i)$ and $\mathcal{T}(\Gamma) \equiv \{\texttt{t} \,|\, \Gamma(\texttt{t}) = \texttt{S}\,\kappa \wedge \kappa \in \{\texttt{iso}, \texttt{trn}\}\}$.
3. If $\texttt{e} = \texttt{E}[\texttt{recover}\,\texttt{e}']$ and $\Delta, \chi, \alpha \vdash \iota : \_, (i, \texttt{t})$ and $\Delta, \chi, \alpha \vdash \iota : \kappa', (i', \texttt{z}) \cdot \overline{\texttt{f}}$ where $\texttt{t}$ is free in $\texttt{e}'$then either $\mathit{Sendable}(\Delta(\alpha, i', \texttt{z}))$ or $(i, \texttt{z}) = (i', \texttt{t}')$ and $\texttt{z}$ is not free in $\texttt{E}[\cdot]$.

The requirements above do not apply to *unaliased unique* capabilities, e.g. $\texttt{iso}\circ$, or $\texttt{trn}\circ$. When proving type preservation, we maintain the property $\mathit{WFT}(\Delta, \chi, \alpha, i, \texttt{e})$ by turning the types of temporaries with unique capabilities $\kappa \in \{\texttt{iso}, \texttt{trn}\}$ into their aliases, $\mathcal{A}(\kappa)$, as soon as the temporary is no longer involved in field reads or updates in the current redex.

Definition 4.4. We call an expression $\texttt{e}$ a *redex* if it has one of the following forms:

$$\texttt{e} \quad ::= \quad \texttt{z.f} \,|\, \texttt{z.f} = \texttt{y} \,|\, \texttt{z.m}(\overline{\texttt{y}}) \,|\, \texttt{z.b}(\overline{\texttt{y}}) \,|\, \texttt{S.k}(\overline{\texttt{z}})$$

The type of the expression is preserved despite this change, because the type rules from fig. 4.1 require the alias of a type $(\ldots \vdash_{\mathcal{A}} \ldots)$ in all such situations. This is explained further in lemma 4.10.

Theorem 4.1. *A well-formed heap ensures data race freedom.*
$\forall \Delta, \chi, \alpha_1, \alpha_2, \texttt{f}, \texttt{g}$, *if*

1. $\Delta \vdash \chi\diamond$, *and*
2. $\chi(\alpha_1) = (\_, \_, \sigma_1, \_, \texttt{E}_1[\texttt{z}_1\texttt{.f} = \texttt{z}_3])$, *and*

3. $\chi(\alpha_2) = (\_\,, \_\,, \sigma_2, \_\,, \mathtt{E}_2[\mathtt{z}_2.\mathtt{g}])$

then $\chi(\alpha_1, |\sigma_1| \cdot \mathtt{z}_1) \neq \chi(\alpha_2, |\sigma_2| \cdot \mathtt{z}_2)$.

Proof. Follows from the type system and the application of *WFV*.1 (global consistency). □

Theorem 4.2. *Well-formedness is preserved.*
$\forall \Delta, \chi$, *if* $\Delta \vdash \chi\diamond$ *and* $\chi \to \chi'$ *then* $\exists \Delta'.\Delta' \vdash \chi'\diamond$.

Proof. Follows from lemmas 4.17-4.20. □

4.10.2 Atomicity

Because the type of any entity does not change, any readable reference is always readable, and so guarantees no other actor can write to it. This holds not just for methods, but for behaviours. As a result, theorem 4.1 guarantees that behaviours are *logically atomic*, a stronger guarantee than data-race freedom.

4.11 Soundness Argument

The property central to any soundness argument is the preservation of the well-formed visibility property, $\mathit{WFV}(\Delta, \chi)$, and the well-formed temporaries property $\mathit{WFT}(\Delta, \chi, \alpha, i, \mathtt{e})$ for all expressions and continuations. To study the former, we need properties about the creation of new paths, while for the latter, we need to control the types we assign to the temporaries in each step.

While this section is, by necessity, detailed and somewhat verbose, the argument presented is standard and unsurprising. Its approach is based on that taken in existing work on the Java type system, particularly in [22] and [33]. I personally found both works invaluable as aids to both understanding and formulating type soundness arguments.

Lemma 4.1. *Uniqueness of contexts.*
For any expressions $\mathtt{e}_1$, $\mathtt{e}_2$ *and contexts* $\mathtt{E}_1[\cdot]$, $\mathtt{E}_2[\cdot]$, *if* $\mathtt{E}_1[\mathtt{e}_2] \equiv \mathtt{E}_2[\mathtt{e}_2]$ *and* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are redexes then* $\mathtt{E}_1[\cdot] \equiv \mathtt{E}_2[\cdot]$ *and* $\mathtt{e}_1 \equiv \mathtt{e}_2$.

Lemma 4.2. *Context lemma.*

1. $\Gamma \vdash \mathtt{E}[\mathtt{e}] : \mathtt{ET} \Rightarrow \exists \mathtt{ET}'.\Gamma, \mathtt{y} \mapsto \mathtt{ET}' \vdash \mathtt{E}[\mathtt{y}] : \mathtt{ET} \land \Gamma \vdash \mathtt{e} : \mathtt{ET}' \land \mathtt{y} \notin \mathit{dom}(\Gamma)$

2. $\mathtt{y} \notin \mathit{dom}(\Gamma) \land \Gamma, \mathtt{y} \mapsto \mathtt{ET}' \vdash \mathtt{E}[\mathtt{y}] : \mathtt{ET} \land \Gamma \vdash \mathtt{e} : \mathtt{ET}' \Rightarrow \Gamma \vdash \mathtt{E}[\mathtt{e}] : \mathtt{ET}$

Lemma 4.3. *Properties of capability operators.*
$\forall \kappa, \kappa_1, \kappa_2 :$

1. *If* $\kappa_1 \sim_g \kappa_2$*, then* $\kappa_1 \sim_l \kappa_2$.
2. *If* $\kappa_1 \leq \kappa_2$*, then*
   (a) $\kappa_1 \sim_l \kappa \Rightarrow \kappa_2 \sim_l \kappa$
   (b) $\kappa_1 \sim_g \kappa \Rightarrow \kappa_2 \sim_g \kappa$
3. *If* $\kappa_1 \sim_l \kappa_2$*, and both* $\kappa_1 \triangleright \kappa$ *and* $\kappa_2 \triangleright \kappa$ *are defined, then* $\kappa_1 \triangleright \kappa \sim_l \kappa_2 \triangleright \kappa$.
4. *If* $\kappa_1 \sim_g \kappa_2$*, and both* $\kappa_1 \triangleright \kappa$ *and* $\kappa_2 \triangleright \kappa$ *are defined, then* $\kappa_1 \triangleright \kappa \sim_g \kappa_2 \triangleright \kappa$
5. $\kappa_2 \leq \kappa_1 \triangleright \kappa_2$ *or* $\kappa_1 = \texttt{val}$ *or* $\kappa_1 \triangleright \kappa_2 = \perp$
6. *If* $\mathcal{A}(\kappa_1) \leq \kappa_2$ *then*
   (a) $\kappa_1 \sim_l \kappa \Rightarrow \kappa_2 \sim_l \kappa$
   (b) $\kappa_1 \sim_g \kappa \Rightarrow \kappa_1 \sim_g \kappa$
   (c) $\mathcal{A}(\kappa_1 \triangleright \kappa) \leq \kappa_2 \triangleright \kappa$
7. *If* $\mathcal{A}(\kappa_1) \leq \kappa_2$ *and* $\mathcal{A}(\kappa_2) \leq \kappa_4$ *then*
   (a) $\kappa_1 \sim_l \kappa_2 \Rightarrow \kappa_3 \sim_l \kappa_4$
   (b) $\kappa_1 \sim_g \kappa_2 \Rightarrow \kappa_3 \sim_g \kappa_4$

Proof. By case analysis on κ_1 and κ_2. □

On the other hand, $\kappa_1 \leq \kappa_2$does not imply that $\kappa \triangleright \kappa_2 \leq \kappa \triangleright \kappa_2$. For example, $\texttt{iso} \leq \texttt{trn}$, but $\texttt{box} \triangleright \texttt{iso} = \texttt{tag} \not\leq \texttt{box} \triangleright \texttt{trn} = \texttt{box}$. Similarly, $\kappa_1 \leq \kappa_2$does not imply that $\kappa_1 \triangleright \kappa \leq \kappa_2 \triangleright \kappa$; take $\texttt{iso} \triangleright \texttt{trn} = \texttt{tag} \not\leq \texttt{trn} \triangleright \texttt{trn} = \texttt{trn}$. Finally, the $\triangleright$ operator is not associative, i.e. $(\kappa_1 \triangleright \kappa_2) \triangleright \kappa_3 \neq \kappa_1 \triangleright (\kappa_2 \triangleright \kappa_3)$. For example, $(\texttt{iso} \triangleright \texttt{trn}) \triangleright \texttt{val} = \perp \neq \texttt{iso} \triangleright (\texttt{trn} \triangleright \texttt{val}) = \texttt{val}$.

Lemma 4.4. *Properties of deep viewpoint adaptation.*
$\forall \kappa, \kappa_1 ..., \kappa_n$:

1. *If* $\kappa_1 \leq \kappa_2$*then* $\kappa_1 \blacktriangleright \kappa \leq \kappa_2 \triangleright \kappa$*, or* $\kappa_2 = \texttt{val}$.
2. $\kappa_1 \blacktriangleright \kappa_2 = \texttt{val}$ *iff* $\kappa_1 \triangleright \kappa_2 = \texttt{val}$.
3. $\kappa_1 \blacktriangleright \kappa_2 \leq \kappa_1 \triangleright \kappa_2$
4. $(...(\kappa_1 \blacktriangleright \kappa_2) \blacktriangleright \kappa_3 ...) \blacktriangleright \kappa_n \leq (...(\kappa_1 \triangleright \kappa_2) \triangleright \kappa_3 ...) \triangleright \kappa_n$
5. $(...(\kappa_1 \blacktriangleright \kappa_2) \blacktriangleright \kappa_3 ...) \blacktriangleright \kappa_n = \texttt{val}$ *iff* $(...(\kappa_1 \triangleright \kappa_2) \triangleright \kappa_3 ...) \triangleright \kappa_n = \texttt{val}$
6. *If* $\kappa_1 \sim_l \kappa_2$*and* $\kappa_1, \kappa_2 \neq \texttt{tag}$*, then* $\kappa_1 \blacktriangleright \kappa \sim_l \kappa_2 \blacktriangleright \kappa$ *or* $\kappa_1 = \kappa_2 = \texttt{ref}$
7. *If* $\kappa_1 \sim_g \kappa_2$*and* $\kappa_1, \kappa_2 \neq \texttt{tag}$ *then* $\kappa_1 \blacktriangleright \kappa \sim_g \kappa_2 \blacktriangleright \kappa$

8. *If* $\mathcal{A}(\kappa_1) \leq \kappa_2$ *and* $\kappa_1 \neq \kappa_2 \neq \texttt{tag}$ *then* $\mathcal{A}(\kappa_1 \blacktriangleright \kappa) \leq \kappa_2 \blacktriangleright \kappa$

Lemma 4.5. *Capabilities are preserved along paths.*
If $\Delta, \chi, \alpha \vdash \iota : \kappa, p$ *and* $\Delta, \chi, \alpha \vdash \iota : \kappa', p$ *then* $\kappa = \kappa'$.

Proof. By induction over the structure of p. □

4.11.1 New paths

Lemma 4.6. *Simplification.*
If

1. $\mathcal{A}(\kappa_1\, \phi) \leq \kappa_2$

2. $\kappa_2 \leq \kappa_3$

3. $\kappa_4 \lhd \kappa_2$ *or* $\kappa_4 \lhd \kappa_3$

Then

4. $\mathcal{A}(\kappa_1\, \phi) \leq \kappa_3$

5. $\kappa_4 \lhd \mathcal{A}(\kappa_1\, \phi)$ *or* $\kappa_4 \lhd \kappa_3$

Proof. (4) follows from (1) and (2). For (5), if $\kappa_4 \lhd \kappa_3$, done. Otherwise, $\kappa_2' = \mathcal{A}(\kappa_1\, \phi)$. If $\kappa_4 = \texttt{ref}$, then for all $\kappa_1\, \phi$, $\kappa_4 \lhd \kappa_2'$. If $\kappa_4 = \texttt{trn}$, then $\kappa_2 \in \{\texttt{iso}, \texttt{trn}, \texttt{val}, \texttt{tag}\} \not\ni \kappa_3$. If $\kappa_2' \in \{\texttt{iso}, \texttt{trn}, \texttt{val}\}$ then $\kappa_1\, \phi \in \{\texttt{iso}\circ, \texttt{trn}\circ, \texttt{val}\}$ and $\texttt{trn} \lhd \kappa_2'$. If $\kappa_2' = \texttt{tag}$ then $\kappa_3 = \texttt{tag}$, which contradicts $\kappa_4 \not\lhd \kappa_3$. If $\kappa_4 = \texttt{iso}$, the same holds, except κ_2 cannot be $\texttt{trn}$. □

Definition 4.5. Unaliased types can be treated as base types.
$\texttt{ET}' \sqsubseteq \texttt{ET}$ iff $\texttt{ET}' = \texttt{ET}$, or $\texttt{ET}' = \texttt{S}\,\kappa\circ$ and $\texttt{ET} = \texttt{S}\,\kappa$

Definition 4.6. An identifier $\texttt{z}$ is *aliased* in a runtime expression $\texttt{e}$ iff
$\exists \texttt{E}[\cdot], \texttt{e}', \texttt{f}, \overline{\texttt{y}}, \overline{\texttt{e}}, \texttt{n}, \texttt{S}$ such that

- $\texttt{e} \equiv \texttt{E}[\texttt{x} = \texttt{z}]$ or
- $\texttt{e} \equiv \texttt{E}[\texttt{e}'.\texttt{f} = \texttt{z}]$ or
- $\texttt{e} \equiv \texttt{E}[\texttt{e}'.\texttt{n}(\overline{\texttt{y}}, \texttt{z}, \overline{\texttt{e}})]$ or
- $\texttt{e} \equiv \texttt{E}[\texttt{z}.\texttt{n}(\overline{\texttt{y}})]$ or
- $\texttt{e} \equiv \texttt{E}[\texttt{S}.\texttt{k}(\overline{\texttt{y}}, \texttt{z}, \overline{\texttt{e}})]$

Lemma 4.7. *Inversion.*
If $\Gamma \vdash \texttt{e} : \texttt{ET}$ *then*

1. *If* $\mathtt{e} \equiv \mathtt{x}$ *then* $\Gamma(\mathtt{x}) \sqsubseteq \mathtt{ET}$

2. *If* $\mathtt{e} \equiv \mathtt{e}_1.\mathtt{f}$ *then* $\exists \mathtt{S}, \mathtt{S}', \kappa, \kappa'$ *such that* $\Gamma \vdash \mathtt{e}_1 : \mathtt{S}\,\kappa$ *and* $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'$ *and* $\mathtt{ET} = \mathtt{S}'\,\kappa \triangleright \kappa'$.

3. *If* $\mathtt{e} \equiv \mathtt{null}$ *then* $\exists \mathtt{S}$ *such that* $\mathtt{S}\;\mathtt{iso}\circ \sqsubseteq \mathtt{ET}$

4. *If* $\mathtt{e} \equiv \mathtt{e}_1; \mathtt{e}_2$ *then* $\exists \mathtt{ET}_1$ *such that* $\Gamma \vdash \mathtt{e}_1 : \mathtt{ET}_1$ *and* $\Gamma \vdash \mathtt{e}_2 : \mathtt{ET}$

5. *If* $\mathtt{e} \equiv \mathtt{x} = \mathtt{e}_1$ *then* $\exists \mathtt{S}, \kappa, \kappa', \phi$ *such that* $\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa$ *and* $\Gamma \vdash \mathtt{e}_1 : \mathtt{S}\,\kappa'\,\phi$ *and* $\mathcal{A}(\kappa'\,\phi) \leq \kappa$ *and* $\mathcal{U}(\mathtt{S}\,\kappa) \sqsubseteq \mathtt{ET}$

6. *If* $\mathtt{e} \equiv \mathtt{e}_1.\mathtt{f} = \mathtt{e}_2$ *then* $\exists \mathtt{S}_1, \mathtt{S}_2, \kappa_1, \kappa_2, \phi$ *such that*
   $\Gamma \vdash \mathtt{e}_1 : \mathtt{S}_1\,\kappa_1$ *and* $\Gamma \vdash \mathtt{e}_2 : \mathtt{S}_2\,\kappa_2\,\phi$ *and*
   $\mathcal{F}(\mathtt{S}_1, \mathtt{f}) = \mathtt{S}_2\,\kappa_3$ *and* $\mathcal{A}(\kappa_2\,\phi) \leq \kappa_3$,
   either $\kappa_1 \triangleleft \kappa_3$ *or* $\kappa_1 \triangleleft \mathcal{A}(\kappa_2\,\phi)$,
   and $\mathcal{U}(\mathtt{S}_2\,\kappa_1 \triangleright \kappa_3) \sqsubseteq \mathtt{ET}$

7. *If* $\mathtt{e} \equiv \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})$ *then* $\exists \mathtt{S}_0, \kappa_0, \kappa_0', \phi, \overline{\mathtt{T}}, \overline{\mathtt{ET}}, \mathtt{ET}'$ *such that*
   $\Gamma \vdash \mathtt{e}_0 : \mathtt{S}_0\,\kappa_0\,\phi$ *and* $\mathcal{A}(\kappa_0\,\phi) \leq \kappa_0'$ *and*
   $\mathcal{M}(\mathtt{S}_0, \mathtt{m}) = (\mathtt{S}_0\,\kappa_0', \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{ET}')$ *and*
   $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$ *and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{ET}' \sqsubseteq \mathtt{ET}$

8. *If* $\mathtt{e} \equiv \mathtt{e}_0.\mathtt{b}(\overline{\mathtt{e}})$ *then* $\exists \mathtt{A}, \kappa_0, \kappa_0', \phi, \overline{\mathtt{T}}$ *such that*
   $\Gamma \vdash \mathtt{e}_0 : \mathtt{A}\,\kappa_0\,\phi$ *and* $\mathcal{A}(\kappa_0\,\phi) \leq \kappa_0'$ *and*
   $\mathcal{M}(\mathtt{A}, \mathtt{b}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{A}\,\mathtt{tag})$ *and*
   sendable$(\mathtt{T}_\mathtt{i})$ *and* $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$*and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{A}\,\mathtt{tag} = \mathtt{ET}$

9. *If* $\mathtt{e} \equiv \mathtt{C}.\mathtt{k}(\overline{\mathtt{e}})$*then* $\exists \overline{\mathtt{ET}}, \overline{\mathtt{T}}$ *such that*
   $\mathcal{M}(\mathtt{C}, \mathtt{k}) = (\mathtt{C}\,\mathtt{ref}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{C}\,\mathtt{ref}\circ)$ *and*
   $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$*and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{C}\,\mathtt{ref}\circ \sqsubseteq \mathtt{ET}$

10. *If* $\mathtt{e} \equiv \mathtt{A}.\mathtt{k}(\overline{\mathtt{e}})$*then* $\exists \overline{\mathtt{ET}}, \overline{\mathtt{T}}$ *such that*
    $\mathcal{M}(\mathtt{A}, \mathtt{k}) = (\mathtt{A}\,\mathtt{ref}, \overline{\mathtt{x} : \mathtt{T}}, \_, \mathtt{A}\,\mathtt{tag})$ *and*
    sendable$(\mathtt{T}_\mathtt{i})$ *and* $\Gamma \vdash \mathtt{e}_\mathtt{i} : \mathtt{ET}_\mathtt{i}$*and* $\mathcal{A}(\mathtt{ET}_\mathtt{i}) \leq \mathtt{T}_\mathtt{i}$ *and* $\mathtt{A}\,\mathtt{tag} = \mathtt{ET}$

11. *If* $\mathtt{e} \equiv \mathtt{recover}\,\mathtt{e}'$ *then* $\exists \mathtt{ET}'$ *such that*
    $\Gamma' = \Gamma \backslash \{\mathtt{x} \,|\, \neg \textit{sendable}(\Gamma(\mathtt{x}))\}$ *and*
    $\Gamma' \vdash \mathtt{e}' : \mathtt{ET}'$ *and* $\mathcal{R}(\mathtt{ET}') \sqsubseteq \mathtt{ET}$

Proof. By induction on the typing of $\Gamma \vdash \mathtt{e} : \mathtt{ET}$. For case 6 (field assignment), apply lemma 4.6. □

Lemma 4.8. *Temporaries and variables with unique capabilities are unique.*
If

1. $\mathit{WFV}(\Delta, \chi)$

2. $\chi(\alpha, i, \mathtt{z}) = \chi(\alpha', i', \mathtt{z}') = \iota$

3. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$

4. $\kappa \in \{\mathtt{iso}, \mathtt{trn}\}$

5. $\Delta, \chi, \alpha' \vdash \iota : \kappa', (i', \mathtt{z}')$

Then $\alpha = \alpha'$ *and either* $\kappa \sim_\ell \kappa'$ *or* $(i, \mathtt{z}) = (i', \mathtt{z}')$*.*

Proof. Assume that $\alpha \neq \alpha'$. Then, by *WFV.1*, $\kappa \sim_g \kappa'$. This implies $\kappa' = \mathtt{tag}$, which contradicts 5. Therefore, $\alpha = \alpha'$ and $\Delta, \chi, \alpha' \vdash \iota : \kappa, (i, \mathtt{z})$. If *Stable*$(\Delta, \alpha, (i, \mathtt{z}))$ then by *WFV.2*, either $\kappa \sim_\ell \kappa'$ (done) or $\chi, \alpha \vdash (i, \mathtt{z}) \sim (i', \mathtt{z}')$, which requires $(i, \mathtt{z}) = (i', \mathtt{z}')$ (done). If $\neg$*Stable*$(\Delta, \alpha, (i, \mathtt{z}))$ then $\mathtt{z} = \mathtt{t}$ and by *WFV.4* either $(i, \mathtt{z}) = (i', \mathtt{z}')$ (done) or $\exists \iota', \kappa'', p', \overline{\mathtt{f}}$ such that $\kappa \leq \kappa''$ and $\kappa'' \in \{\mathtt{iso}, \mathtt{trn}\}$ and $(i', \mathtt{z}') = p' \cdot \overline{\mathtt{f}}$ and $\Delta, \chi, \alpha \vdash \iota' : \kappa'', p'$ and $\Delta, \chi, \iota' \vdash \iota : \kappa, (0, \mathtt{this}) \cdot \overline{\mathtt{f}}$, so $\overline{\mathtt{f}} = \epsilon$ and $p' = (i', \mathtt{z}')$ and $\iota = \iota'$. This gives us $\Delta, \chi, \iota \vdash \iota : \kappa, (0, \mathtt{this})$, which by the definition of visibility gives us $\kappa = \mathtt{ref}$, which contradicts (4) (done). □

Lemma 4.9. *Isolation in well-formed visibility.*
If

1. $\mathit{WFV}(\Delta, \chi)$

2. $\Delta(\alpha, i, \mathtt{t}) = \mathtt{S}\,\kappa\,\phi$ *and* $\chi(\alpha, i, \mathtt{t}) = \iota$ *and* $\kappa \in \{\mathtt{iso}, \mathtt{trn}\}$

3. $\Delta, \chi, \alpha' \vdash \iota : \kappa', p$

Then

4. *If* $\kappa\,\phi = \mathtt{iso}\circ$ *then* $\alpha = \alpha'$ *and* $p = (i, \mathtt{t})$*.*

5. *If* $\kappa\,\phi = \mathtt{trn}\circ$ *then* $\alpha = \alpha'$ *and either* $p = (i, \mathtt{t})$ *or* $\kappa' = \mathtt{box}$*.*

6. *If* $\kappa\,\phi = \mathtt{iso}$ *then* $\alpha = \alpha'$ *and either* $p = (i, \mathtt{t})$ *or* $\exists \iota', \kappa'', p', \overline{\mathtt{f}}$ *such that* $\kappa \leq \kappa''$ *and* $\kappa'' \in \{\mathtt{iso}, \mathtt{trn}\}$ *and* $p = p' \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota' : \kappa'', p'$ *and* $\Delta, \chi, \iota' \vdash \iota : \mathtt{iso}, \overline{\mathtt{f}}$*.*

7. *If* $\kappa\,\phi = \mathtt{trn}$ *then* $\alpha = \alpha'$ *and either* $p = (i, \mathtt{t})$ *or* $\kappa' = \mathtt{box}$ *or* $\exists \iota', p', \overline{\mathtt{f}}$ *such that* $p = p' \cdot \overline{\mathtt{f}}$ *and* $\Delta, \chi, \alpha \vdash \iota' : \mathtt{trn}, p'$ *and* $\Delta, \chi, \iota' \vdash \iota : \mathtt{trn}, \overline{\mathtt{f}}$*.*

Proof. (4) and (5) follow from lemma 4.8. (5) and (6) follow from lemma 4.8 and *WFV.4*. □

Lemma 4.10. *Aliasing and replaceability.*
If

1. $\Gamma \vdash \mathtt{e} : \mathtt{ET}$ *and* $\mathtt{z}$ *is aliased in* $\mathtt{e}$

2. $\mathtt{z}$ *does not appear more than once in* $\mathtt{e}$

3. $\Gamma(\mathtt{z})$ *is not unaliased*

4. $\Gamma' = \Gamma[\mathtt{z} \mapsto \mathcal{A}(\Gamma(\mathtt{z}))]$

Then $\Gamma' \vdash \mathtt{e} : \mathtt{ET}$

Proof. By induction over the structure of $\mathtt{e}$. We apply lemma 4.7. Moreover, we use the fact that $\forall\kappa.\mathcal{A}(\mathcal{A}(\kappa)) = \mathcal{A}(\kappa)$. The base cases are expressions that can alias $\mathtt{z}$.

- If $\mathtt{e} \equiv \mathtt{x} = \mathtt{z}$ then, by lemma 4.7, we obtain $\Gamma(\mathtt{x}) = \mathtt{S}\,\kappa$ and $\Gamma(\mathtt{z}) = \mathtt{S}\,\kappa'\,\phi$ and $\phi \neq \circ$ and $\mathcal{A}(\kappa') \leq \kappa$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa')) \leq \kappa$ and so $\Gamma' \vdash \mathtt{x} = \mathtt{z} : \mathtt{ET}$.

- If $\mathtt{e} \equiv \mathtt{e}'.\mathtt{f} = \mathtt{z}$ then, by lemma 4.7, we obtain $\Gamma \vdash \mathtt{e}' : \mathtt{S}\,\kappa$ and $\mathcal{F}(\mathtt{S},\mathtt{f}) = \mathtt{S}'\,\kappa'$ and $\Gamma(\mathtt{z}) = \mathtt{S}'\,\kappa''\,\phi$ and $\phi \neq \circ$ and $\mathcal{A}(\kappa'') \leq \kappa'$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa'')) \leq \kappa'$ and so $\Gamma' \vdash \mathtt{e}'.\mathtt{f} = \mathtt{z} : \mathtt{ET}$.

- If $\mathtt{e} \equiv \mathtt{e}'.\mathtt{n}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}})$ then, by lemma 4.7, we obtain $\Gamma \vdash \mathtt{e}' : \mathtt{S}\,\kappa$ and $\mathcal{M}(\mathtt{S},\mathtt{n}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_)$ and $\Gamma(\mathtt{z}) = \mathtt{S_i}\,\kappa'_{\mathtt{i}}\,\phi$ and and $\phi \neq \circ$ and $\mathcal{A}(\kappa'_i) \leq \kappa_i$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa'_i)) \leq \kappa_i$ and so $\Gamma' \vdash \mathtt{e}'.\mathtt{n}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}}) : \mathtt{ET}$.

- If $\mathtt{e} \equiv \mathtt{z}.\mathtt{n}(\overline{\mathtt{y}})$ then, by lemma 4.7, we obtain $\Gamma(\mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ and $\phi \neq \circ$ and $\mathcal{M}(\mathtt{S},\mathtt{n}) = (\mathtt{S}\,\kappa')$ and $\mathcal{A}(\kappa) \leq (\kappa')$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa)) \leq \kappa'$ and so $\Gamma' \vdash \mathtt{z}.\mathtt{n}(\overline{\mathtt{y}}) : \mathtt{ET}$.

- If $\mathtt{e} \equiv \mathtt{S}.\mathtt{k}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}})$then, by lemma 4.7, we obtain $\mathcal{M}(\mathtt{S},\mathtt{k}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_)$ and $\Gamma(\mathtt{z}) = \mathtt{S_i}\,\kappa'_{\mathtt{i}}\,\phi$ and and $\phi \neq \circ$ and $\mathcal{A}(\kappa'_i) \leq \kappa_i$. Therefore, we have $\mathcal{A}(\mathcal{A}(\kappa'_i)) \leq \kappa_i$ and so $\Gamma' \vdash \mathtt{S}.\mathtt{k}(\overline{\mathtt{y}}, \mathtt{z}, \overline{\mathtt{e}}) : \mathtt{ET}$.

For the inductive step, if $\mathtt{e} \equiv \mathtt{E}[\mathtt{e}']$and $\mathtt{z}$ is aliased in $\mathtt{e}$, then, by lemma 4.2, we obtain that $\exists \mathtt{ET}', \mathtt{y} \notin \Gamma$ such that $\Gamma \vdash \mathtt{e}' : \mathtt{ET}'$ and $\Gamma[\mathtt{y} \mapsto \mathtt{ET}'] \vdash \mathtt{E}[\mathtt{y}] : \mathtt{ET}$, and so $\Gamma' \vdash \mathtt{e}' : \mathtt{ET}'$. Therefore, by lemma 4.2, we obtain $\Gamma' \vdash \mathtt{E}[\mathtt{e}'] : \mathtt{ET}$. □

Lemma 4.11. *Origins of temporary identifiers.*
If

1. $\mathtt{z}$ *appears once in expression* $\mathtt{e}$

2. $\mathtt{z}$ *is not aliased in* $\mathtt{e}$

Then $\exists \mathtt{E}'$ *such that*

3. $\mathtt{e} \equiv \mathtt{E}'[\mathtt{z.f}]$, *or*

4. $\mathtt{e} \equiv \mathtt{E}'[\mathtt{z.f} = \mathtt{e}']$, *or*

5. $\mathtt{e} \equiv \mathtt{E}'[\mathtt{recover\,z}]$, *or*

Proof. By application of definition 4.6. □

Lemma 4.12. *If* $\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{e} : \mathtt{ET}_1$ *and* $\mathcal{A}(\mathtt{T}_2) \leq \mathtt{T}_1$ *then* $\exists \mathtt{ET}_2 . \Gamma, \mathtt{x} : \mathtt{T}_2 \vdash \mathtt{e} : \mathtt{ET}_2$ *and* $\mathtt{ET}_1 = \mathtt{ET}_2$ *or* $\mathcal{A}(\mathtt{ET}_2) \leq \mathtt{ET}_1$

Proof. By structural induction on the typing and lemma 4.3. □

Lemma 4.13. *New paths through field read.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$

2. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$

3. $\chi(\iota, \mathtt{f}) = \iota'$ *and* $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa'$

4. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}'$, $\mathtt{S}\,\kappa$ *otherwise*

5. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha, i, \mathtt{t}) \mapsto \mathtt{S}'\,\kappa \triangleright \kappa')]$

6. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}\}$

Then

7. $\forall \alpha', \iota'', \kappa'', p'$ *if* $\Delta', \chi, \alpha' \vdash \iota'', \kappa'', p'$ *then*
   (a) $\Delta, \chi, \alpha' \vdash \iota'' : \kappa'', p'$ *or*
   (b) $\alpha' = \alpha$ *and* $\exists \overline{\mathtt{f}}, \overline{\kappa}$ *such that*
      i. $p' = (i, \mathtt{t}) \cdot \overline{\mathtt{f}}$
      ii. $\kappa'' = \kappa \triangleright \kappa' \overline{\blacktriangleright \kappa}$
      iii. $\Delta, \chi, \alpha \vdash \iota'' : \kappa \blacktriangleright \kappa' \overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$

8. *If* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi)$ *and* $\mathcal{T}(\Delta'(\alpha, i)) \subseteq \{\mathtt{t}\}$

Lemma 4.14. *New paths through local assignment.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$ *and* $\chi(\alpha, i, \mathtt{x}) = \iota'$ *and* $\mathtt{t} \notin \chi(\alpha, i)$

2. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ *and* $\Delta(\alpha, i, \mathtt{x}) = \mathtt{S}\,\kappa'$

3. $\chi' = \chi[(\alpha, i, \mathtt{x}) \mapsto \iota, (\alpha, i, \mathtt{t}) \mapsto \iota']$

4. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}'$, $\mathtt{S}\,\kappa$ *otherwise*

5. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha, i, \mathtt{t}) \mapsto \mathcal{U}(\mathtt{S}\,\kappa')]$

6. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}\}$

Then

7. $\forall \alpha', \iota'', \kappa'', p$ *if* $\Delta', \chi', \alpha' \vdash \iota'' : \kappa'', p$ *then*

    (a) $\Delta, \chi, \alpha' \vdash \iota'' : \kappa'', p$ *or*

    (b) $\alpha = \alpha'$ *and* $\exists \overline{\mathtt{f}}, \overline{\kappa}$ *such that*

        i. $p = (i, \mathtt{x}) \cdot \overline{\mathtt{f}}$ *and* $\kappa'' = \kappa' \overline{\blacktriangleright \kappa}$ *and* $\Delta, \chi, \alpha \vdash \iota'' : \kappa \overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \overline{\mathtt{f}}$, *or*

        ii. $p = (i \cdot \mathtt{t}) \cdot \overline{\mathtt{f}}$ *and* $\kappa'' = \kappa' \overline{\blacktriangleright \kappa}$ *and* $\Delta, \chi, \alpha \vdash \iota'' : \kappa' \overline{\blacktriangleright \kappa}, (i, \mathtt{x}) \cdot \overline{\mathtt{f}}$

8. *If* $\mathcal{A}(\kappa\,\phi) \leq \kappa'$ *and* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi')$ *and* $\mathcal{T}(\Delta'(\alpha, i)) \subseteq \{\mathtt{t}\}$

Lemma 4.15. *New paths through field assignment.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$ *and* $\chi(\alpha, i, \mathtt{z}') = \iota'$

2. $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$ *and* $\Delta(\alpha, i, \mathtt{z}') = \mathtt{S}'\,\kappa'\,\phi'$

3. $\chi(\iota, \mathtt{f}) = \iota''$ *and* $\mathcal{F}(\mathtt{S}, \mathtt{f}) = \mathtt{S}'\,\kappa''$

4. $\chi' = \chi[(\iota, \mathtt{f}) \mapsto \iota', (\alpha, i, \mathtt{t}) \mapsto \iota'']$

5. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}'$, $\mathtt{S}\,\kappa$ *otherwise*

6. $\mathtt{T}' = \bot$ *if* $\mathtt{z}' = \mathtt{t}''$, $\mathtt{S}'\,\kappa'$ *otherwise*

7. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha, i, \mathtt{z}') \mapsto \mathtt{T}', (\alpha, i, \mathtt{t}) \mapsto \mathcal{U}(\mathtt{S}'\,\kappa \triangleright \kappa'')]$

8. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}, \mathtt{z}'\}$

Then

9. $\forall \alpha', \iota''', \kappa''', p'$ *if* $\Delta', \chi', \alpha' \vdash \iota''' : \kappa''', p'$ *then*

    (a) $\Delta, \chi, \alpha' \vdash \iota''' : \kappa''', p'$ *or*

    (b) $\alpha' = \alpha$ *and* $\exists \overline{\mathtt{f}}, \overline{\kappa}$ *such that*

i. $\kappa''' = \kappa \blacktriangleright \kappa''\overline{\blacktriangleright \kappa}$ and $p' = (i, \mathtt{z}) \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$ and $\Delta, \chi, \alpha \vdash \iota''' : \kappa'\overline{\blacktriangleright \kappa}, (i, \mathtt{z}') \cdot \overline{\mathtt{f}}$, or

ii. $\kappa''' = \mathcal{U}(\kappa \triangleright \kappa'')\overline{\blacktriangleright \kappa}$ and $p' = (i, \mathtt{t}) \cdot \overline{\mathtt{f}}$ and $\Delta, \chi, \alpha \vdash \iota''' : \kappa \blacktriangleright \kappa''\overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$, or

iii. $\exists \kappa'''', p \neq (i, \mathtt{z})$ such that $\kappa''' = \kappa'''' \blacktriangleright \kappa''\overline{\blacktriangleright \kappa}$ and $p' = p \cdot \mathtt{f} \cdot \overline{\mathtt{f}}$ and $\Delta, \chi, \alpha \vdash \iota : \kappa'''', p$

10. *If* $\mathcal{A}(\kappa'\,\phi') \leq \kappa''$ *and* ($\kappa \triangleleft \kappa'$ *or* $\kappa \triangleleft \kappa''$) *and* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi')$ *and* $\mathcal{T}(\Delta'(\alpha, i)) = \emptyset$

Lemma 4.16. *New paths through message passing.*
If

1. $\chi(\alpha, i, \mathtt{z}) = \iota$ *and* $\Delta(\alpha, i, \mathtt{z}) = \mathtt{S}\,\kappa\,\phi$

2. $\chi' = \chi[(\alpha', -j, \mathtt{x}) \mapsto \iota]$

3. $\mathtt{T} = \bot$ *if* $\mathtt{z} = \mathtt{t}$, $\mathtt{S}\,\kappa$ *otherwise*

4. $\Delta' = \Delta[(\alpha, i, \mathtt{z}) \mapsto \mathtt{T}, (\alpha', -j, \mathtt{x}) \mapsto \mathtt{S}\,\kappa']$

5. $\mathcal{T}(\Delta(\alpha, i)) \subseteq \{\mathtt{z}\}$

Then

6. $\forall \alpha'', \iota'', \kappa'', p$ *if* $\Delta', \chi', \alpha'' \vdash \iota'' : \kappa'', p$ *then*

   (a) $\Delta, \chi, \alpha'' \vdash \iota'' : \kappa'', p$ *or*

   (b) $\alpha'' = \alpha'$ *and* $\exists \overline{\mathtt{f}}, \overline{\kappa}$ *such that*

      i. $p = (-j, \mathtt{x}) \cdot \overline{\mathtt{f}}$

      ii. $\kappa'' = \kappa'\overline{\blacktriangleright \kappa}$

      iii. $\Delta, \chi, \alpha \vdash \iota'' : \kappa\overline{\blacktriangleright \kappa}, (i, \mathtt{z}) \cdot \overline{\mathtt{f}}$

7. *If* $\mathcal{A}(\kappa\,\phi) \leq \kappa'$ *and sendable*(κ') *and* $WFV(\Delta, \chi)$ *then* $WFV(\Delta', \chi')$ *and* $\mathcal{T}(\Delta'(\alpha, i)) = \emptyset$

4.11.2 Preservation of well-formedness

Lemma 4.17. *Type preservation on same frame.*

For all heaps χ*, actors* α*, global type environments* Δ*, frames* φ*, stacks* σ *and expressions* $\mathtt{e}$*, if*

1. $\chi(\alpha) = (\_, \_, \_, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{E}[\mathtt{e}])$ *and* $|\bar{\varphi}| = i - 1$

2. $\chi, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{e} \rightsquigarrow \chi'', \alpha \cdot \overline{\varphi} \cdot \varphi', \mathtt{e}'$

3. $\chi' = \chi''[\alpha \mapsto (\alpha \cdot \overline{\varphi} \cdot \varphi, \texttt{E}[\texttt{e}'])]$

4. $\Delta(\alpha, i) \vdash \texttt{e} : \texttt{ET}$

5. $\Delta \vdash \chi\diamond$

Then $\exists\Delta'$ *such that*

1. $\Delta'(\alpha, i) \vdash \texttt{e}' : \texttt{ET}$

2. $\Delta' \vdash \chi'\diamond$

Lemma 4.18. *Type preservation for method call.*
For all heaps χ *and actors* α*, if*

1. $\chi(\alpha) = (\_, \_, \_, \sigma \cdot \varphi, \texttt{E}[\texttt{e}])$

2. $\chi, \sigma \cdot \varphi, \texttt{e} \rightsquigarrow \chi'', \sigma \cdot \varphi \cdot \varphi', \texttt{e}'$

3. $\chi' = \chi''[\alpha \mapsto (\sigma \cdot \varphi \cdot \varphi', \texttt{E}[\texttt{e}'])]$

4. $\Delta \vdash \chi\diamond$

Then $\exists\Delta'$ *such that* $\Delta' \vdash \chi'\diamond$.

Lemma 4.19. *Type preservation upon method return*
For all heaps χ *and actors* α*, if*

1. $\chi(\alpha) = (\_, \_, \_, \sigma \cdot \varphi \cdot \varphi', \texttt{z})$

2. $\texttt{t} \notin \varphi$ *and* $\varphi'' = \varphi[\texttt{t} \mapsto \varphi'(\texttt{z})]$

3. $\varphi' = (\_, \_, \texttt{E}[\cdot])$

4. $\chi' = \chi[\alpha \mapsto (\sigma \cdot \varphi'', \texttt{E}[\texttt{t}])]$

5. $\Delta \vdash \chi\diamond$

Then $\exists\Delta'$ *such that* $\Delta' \vdash \chi'\diamond$.

Lemma 4.20. *Type preservation upon message handling.*
For all heaps χ *and actors* α*, if*

1. $\chi(\alpha) = (\texttt{A}, fs, (\texttt{n} \cdot \overline{v}) \cdot \overline{\mu}, \alpha, \epsilon)$

2. $\mathcal{M}(\texttt{A}, \texttt{n}) = (\_, \overline{\texttt{x} : \texttt{T}}, \texttt{e}, \_)$

3. $\varphi = (\texttt{n}, [\texttt{this} \mapsto \alpha, \bar{\texttt{x}} \mapsto \overline{v}], \cdot)$

4. $\chi' = \chi[\alpha \mapsto (\texttt{A}, fs, \overline{\mu}, (\alpha \cdot \varphi), \texttt{e})]$

5. $\Delta \vdash \chi\diamond$

Then $\exists\Delta'$ *such that* $\Delta' \vdash \chi'\diamond$.

Chapter 5

Actor Collection

The disposal of dead actors (those no longer required for computation or I/O) in actor-model languages is as important as disposal of unreachable objects in object-oriented languages. Actor-model languages must know when an actor has terminated in order to free resources dedicated to the actor. Most existing actor-model languages and libraries do not attempt to solve this problem, instead requiring the programmer to explicitly manage every actor's lifetime [6, 29, 69, 72, 65].

The very problems that actor-model programming excels at addressing (including concurrency, scalability, and simplicity) have made actor garbage collection problematic, due to the difficulty of observing the global state of a program. As a result, actor-model systems in applications which create many short-lived actors become either more difficult to program (when they require manually terminating actors) or encounter performance problems (when they have actor garbage collection that is not fully concurrent).

A language which does not provide garbage collection of actors will require a facility to explicitly terminate actors. This will also require the language to provide a default behaviour when a message is sent to a terminated actor, the ability to distinguish at runtime between terminated and non-terminated actors, and possibly notification mechanisms for actor termination.

Pony uses a novel technique for garbage collection of actors, called Message-based Actor Collection (MAC), that satisfies the following goals:

1. *Soundness:* the technique collects only dead actors.
2. *Completeness:* the technique collects all dead actors eventually.
3. *Concurrency:* the technique does not require a stop-the-world step, thread coordination, actor introspection, shared memory, read/write barriers or cache coherency.

When an actor has completed local execution and has no pending messages on its queue, it is *blocked*. An actor is *dead* if it is blocked and all actors that have a reference to it are blocked, transitively. Collection of dead actors depends on being able to collect closed cycles of blocked actors.

This approach is inspired by previous work on distributed garbage collection of passive objects using distributed reference counting and a secondary mechanism to collect cyclic garbage [41, 7, 51, 50]. Detection of cycles of objects is based on their *topology,* which is essentially the number of incoming references and the identities of all outgoing references. As such, the topology of an actor consists of the number of incoming references from actors, the set of outgoing references to actors, and a flag indicating whether the actor is blocked. A dedicated actor, called the *cycle detector*, keeps track of the actor topology and detects any cycles.

The core challenge is that the true topology of an actor is a concept distributed across all of the actors: it changes not only when the actor mutates, but also when other actors mutate. An actor's view of its topology may be out of sync with the true topology, and the cycle detector's view of an actor's topology may be out of sync with the actor's view of its topology. This differs radically from previous work on distributed object cycle detection, where objects must either be immutable or cycle detection must monitor mutation [40, 34, 20].

This technique uses the message passing paradigm at the heart of the actor-model: when an actor blocks, it sends a snapshot of its view of its topology to the cycle detector. The cycle detector in turn detects cycles based on its view of the topology of blocked actors. Because the cycle detector operates on its own view of the blocked actor topology rather than stopping execution or monitoring mutation, cycles may be detected based on a view of the topology that is out of date. This is overcome with a confirmation protocol that allows the cycle detector to determine whether or not its view of the blocked actor topology is the same as the true topology, without stopping execution, monitoring mutation, or examining any actor's state.

5.1 Background on Garbage Collection of Actors

Existing actor-model languages and libraries use three approaches to garbage collection of actors.

The first approach is to require the programmer to manually terminate actors. Many existing actor-model languages and libraries, such as Erlang [6], Scala [29], AmbientTalk [69], SALSA 2.0 [72], Kilim [65], and Akka, do not garbage collect actors at all. All of these except Kilim support actors on distributed nodes, although only SALSA supports manual migration of actors to new nodes. None support distributed scheduling or automatic migration.

The second approach is to transform the actor graph into an object graph and use a tracing garbage collector to collect actors [35, 71, 76], as done in ActorFoundry. This requires shared memory, cache coherency, and a stop-the-world step. This approach allows actors to be collected using the same collector used for passive objects, but cannot be used across distributed nodes.

The third approach, used in SALSA 1.0 [77], uses reference listing (whereby an actor keeps a complete list of every other actor that references it) and monitoring of actor mutation to build conservative local snapshots which are assembled into a global snapshot. This requires write barriers for actor mutation (which requires shared memory and cache coherency), a global synchronisation agent, and coordination of local snapshots within an overlapping time range. These snapshots are used with the pseudo-root algorithm, which additionally requires acknowledgement messages for all asynchronous messages, inverse reference listing, and a multiple-message protocol for reference passing [78, 75]. Like SALSA 2.0, SALSA 1.0 supports distributed nodes and manual actor migration.

None of these approaches provides a fully concurrent method for garbage collection of actors.

5.1.1 Distributed Passive Object Collection

The literature on distributed passive object collection is vast, and so only key differences will be briefly mentioned here. Pony's *MAC* has been inspired not just by previous work in actor collection, but also by work in concurrent cycle detection [7] and distributed reference counting [51, 50, 34, 40, 20] for passive object collection. Some of these approaches do not address cyclic garbage [51, 50]. Others require either immutable passive objects or a synchronisation mechanism between the cycle detector and the mutator, which makes them inapplicable to actor collection [7, 34, 40, 20]. Others use dynamic process groups to partially trace the distributed object graph, trading completeness for promptness, and requiring mutators to pause only during the local portion of such a partial trace [59, 60].

5.1.2 Differences From Existing Approaches

MAC differs significantly from previous work. Unlike distributed passive object collection, no restriction on actor mutation or monitoring of mutation is required in order to detect cyclic garbage, and no reference listing, indirection cells, or diffusion trees (a technique whereby nodes keep a trail of object references they have passed, which can lead to zombie nodes) are required. Unlike the pseudo-root approach, acknowledgement messages are only required when actors are actually collected, no reference listing is required, no message round-trips are required, and no snapshot integration or time ranges are required. As a result, *MAC* requires significantly less overhead. In addition, because *MAC* does not require thread coordination or cache coherency, it does not become less efficient as core count increases.

Some approaches to distributed passive object collection are fault-tolerant [57]. In order to make distributed garbage collection fault-tolerant, it is necessary to detect and handle failure and often also to track a global view of time.

5.2 Message-based Actor Collection Algorithm

The *MAC* algorithm extends the operational semantics from chapter 3 to allow actors to be garbage collected when they have no pending messages and will never again have pending messages, that is, when the actor is *dead*. To do so, firstly, the definition of an actor is extended to keep track of its own *local reference count*, a map of actor addresses to *foreign reference counts*, and a flag to indicate whether or not the actor is *blocked*. Secondly, new message types are added to allow reference counts to be changed in an asynchronous way. Finally, a *cycle detector* actor is added to the system that can correctly detect and collect isolated cyclic graphs of dead actors.

Effectively, *MAC* is an eventual consistency algorithm for for communicating the *true topology* of an actor graph to a cycle detector without any actor (including the cycle detector) examining another actor's state.

Note that without causality, *MAC* does not hold without modification. This is discussed in section 5.6, where an extension of *MAC* is proposed that does not require causal ordering.

5.2.1 Topology

The *true topology* of the system is the directed graph of actor reachability. Because actors execute concurrently, it is not possible to efficiently track the true topology. Instead, each actor maintains a view of its own topology, consisting of a reference count (indicating the number of incoming graph edges) and an *external map* of potentially reachable actors (the outgoing edges) to *foreign reference counts* for those actors.

The actor's view can disagree with the true topology. When an actor ι_1 sends a reference to itself to another actor ι_2, it can immediately update its reference count, maintaining agreement with the true topology. However, if ι_2 drops its reference to ι_1, ι_2 cannot directly mutate ι_1's reference count. Now

ι_1's reference count is out of sync. To correct this, ι_2 sends a reference count decrement message (*DEC*) to ι_1. When ι_1 processes that message, it updates its view to restore agreement with the true topology.

Similarly, if ι_2 sends a reference to ι_1 to a third actor ι_3, it may first send a reference count increment message (*INC*) to ι_1, allowing ι_2 to increase its foreign reference count, without requiring ι_2 to mutate any other actor's state or wait for a message in reply.

These *INC* and *DEC* messages allow the actor's view of its topology to be eventually consistent with the true topology.

5.2.2 Deferred Reference Counting

The *external map* is an over-approximation of the set of actor references reachable from some actor's local state. It differs from the local state in order to allow reference counting to be lazy. Rather than tracking all references from ι_1 to ι_2, a reference exists if ι_2 appears one or more times in ι_1's local state. The external map contains all actors that have been in the actor's local state or were received in a message since the last local garbage collection pass. When an actor performs local garbage collection, the external map is compacted so as to contain only the actors still reachable from the local state. Actors removed from the external map when it is compacted represent dropped references, and are sent *DEC* messages.

Similarly, when an actor ι_1 receives another actor ι_2 in a message, ι_1 adds ι_2 to its external map. If ι_2 is not present in ι_1's external map, the reference held by the message is transferred to ι_1, giving ι_1 a foreign reference count of 1 for ι_2, and ι_2's view of its topology remains in agreement with the true topology. If ι_2 is already present in ι_1's external map, ι_1 already has an outgoing edge to ι_2. To maintain the reference count invariant of ι_2, ι_1 increments its foreign reference count for ι_2 by 1.

This is based on, but differs from, deferred increments [8], where ephemeral reference count updates can be skipped, and update coalescing, where redundant reference count updates are combined for efficiency [39].

In *MAC*, reference counts are not updated when references are created or destroyed on the stack or in the heap, but only when references are sent and received in messages and when local garbage collection indicates no references to an actor are reachable from actor's local state. The messages act as the mechanism for deferring increments, and the external map, in combination with local garbage collection, acts as the mechanism for coalescing updates.

5.2.3 Cycle Detection

As in any reference counting system, cyclic garbage cannot be collected by reference counting alone. *MAC* uses a *cycle detector* that has a message queue like an actor, and can both send and receive messages.

When an actor has no pending messages on its queue, it is *blocked*. When an actor blocks, it sends a block message (*BLK*) to the cycle detector containing the actor's view of its topology, i.e. its reference count and its external map. When a blocked actor processes a message, it becomes *unblocked* and sends an unblock message (*UNB*) to the cycle detector, informing the cycle detector that its view of that actor's topology is invalid and that actor is no longer blocked.

This allows the cycle detector to maintain a view of the topology of all blocked actors that is eventually consistent with each actor's view of its topology, which is in turn eventually consistent with the true topology. However, a naive implementation can result in actors emptying their message queue during every scheduling turn. This would result in a block message being sent at the end of the scheduling turn and, similarly, an unblock message being sent at the beginning of the next scheduling turn, which in turn results in the cycle detector spending excessive time handling block and unblock messages. The solution is for an actor to delay sending a block message until the runtime is confident that the actor will not be rescheduled soon. There are several strategies for such a heuristic, including delaying by a number of scheduling turns, using processor cycle count timestamps to track previous rescheduling delays, or using a timer on a central service, such as the cycle detector itself, to poll for blocked actors.

It would be possible but not efficient for application actors to perform cycle detection when no messages are pending on their queue (i.e. just before blocking): this would require every actor in the system to maintain a view of every other actor's topology, which for n actors would require n messages upon each block and unblock and duplication of blocked actor topology in every actor. A separate cycle detector reduces this to one message upon block or unblock regardless of the number of actors.

5.2.4 Application Messages

Application-level messages are presented in examples as a single message type (*APP*) that allows an actor ι_1 to send a set of actors ιs to another actor ι_2. In fact, there are multiple application message types, which can contain passive objects as well as actors. However, since actor collection relies only on the topology of actors, and not objects, the set of actors ιs is used to present all explicit actor references contained in the message, as well as all implicit actor references (i.e. the allocating actors for any passive objects included in the message).

This is done for convenience in examples, but the operational semantics and associated definitions express actual messages and define reachability fully.

5.2.5 Dead Actors

An actor is *dead* if it is blocked and all actors that have a reference to it are blocked, transitively. Because messaging is required to be causal on a single node, a blocked actor with a reference count of zero is unreachable by any other actor and is therefore dead (acyclic garbage).

For cyclic garbage, the cycle detector uses a standard cycle detection algorithm to find isolated cycles in its view of the topology of blocked actors. However, the cycle detector's view of the topology may disagree with an actor's view of its topology (when a *BLK* or *UNB* message is on the cycle detector's queue but as yet unprocessed), and the actor's view of its topology may in turn disagree with the true topology (when an *INC* or *DEC* message is on the actor's queue but as yet unprocessed). If cyclic garbage is detected on the basis of a view of the topology that disagrees with the true topology, that cycle must not be collected. A cycle that has been detected is termed a *perceived cycle*, and a cycle that has been detected using a view of the topology that agrees with the true topology is termed a *true cycle.*

Example 5.1. A perceived cycle that is not a true cycle. This is shown in figure 5.1.

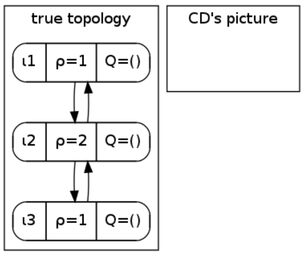
1. Given three actors (ι_1, ι_2 and ι_3), ι_1 and ι_2 reference each other and ι_2 and ι_3 reference each other, all with foreign reference counts of 1.

2. ι_1 blocks, sending $BLK(\iota_1, 1, [\iota_2 \mapsto 1])$ to the cycle detector. When the cycle detector processes this, its view of the topology becomes $[\iota_1 \mapsto (1, [\iota_2 \mapsto 1])]$.

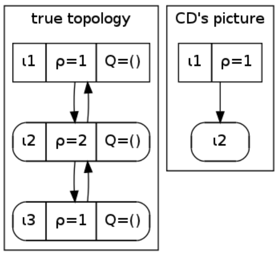
3. ι_2 wishes to send a reference to ι_1 to ι_3. It sends $INC(1)$ to ι_1 and then $APP(\iota_1)$ to ι_3. ι_2 then drops its reference to ι_3 , collects garbage locally, and sends $DEC(1)$ to ι_3. The cycle detector's view of the topology does not change.

4. ι_3 processes $APP(\iota_1)$, adding ι_1 to its external map. ι_3 then drops its reference to ι_2 , collects garbage locally, and sends $DEC(1)$ to ι_2. The cycle detector's view of the topology does not change.

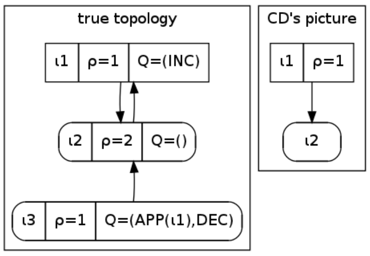
5. ι_2 processes $DEC(1)$, then blocks, sending $BLK(\iota_2, 1, [\iota_1 \mapsto 1])$ to the cycle detector. When the cycle detector processes this, its view of the topology becomes $[\iota_1 \mapsto (1, [\iota_2 \mapsto 1]), \iota_2 \mapsto (1, [\iota_1 \mapsto 1])]$.



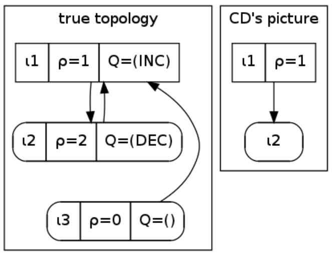
(a) Initial state, as in step 1



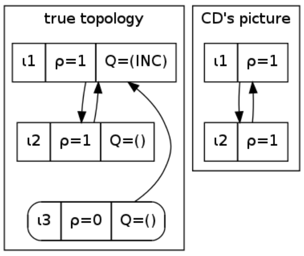
(b) ι_1 blocks, as in step 2



(c) ι_2 sends $\iota_3 \leftarrow APP(\iota_1)$ and drops ι_3, as in step 3

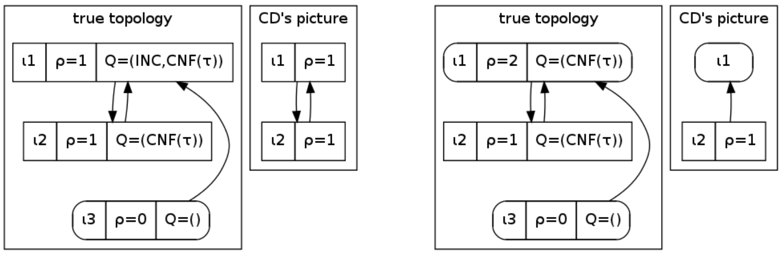


(d) ι_3 processes $APP(\iota_1)$ and drops ι_2, as in step 5



(e) ι_2 blocks, as in step 5. The perceived cycle is incorrect due to ι_1's pending INC.

Figure 5.1: Diagram of example 5.1. Boxes display the reference count (ρ), and queue (Q) of actors, with round corners indicating unblocked and square corners indicating blocked. The arrows indicate references, e.g. ι_1 references ι_2, which implicitly shows the external map.



(a) Cycle detector sends *CNF*, as in step 6.

(b) ι_1 unblocks, as in step 2. The perceived cycle is correctly cancelled, as in step 3.

Figure 5.2: Diagram of example 5.2. Boxes display the reference count (ρ), and queue (Q) of actors, with round corners indicating unblocked and square corners indicating blocked. The arrows indicate references, e.g. ι_1 references ι_2, which implicitly shows the external map.

6. The cycle detector perceives a cycle $\{\iota_1, \iota_2\}$, even though ι_1 is reachable from ι_3. This is because ι_1 has a pending *INC*(*1*) that it has not processed.

5.2.6 Conf-Ack Protocol

When a perceived cycle is detected, the cycle detector must determine whether or not the view of the topology used to detect the cycle agrees with the true topology. To do so, a *conf-ack* step is introduced to the protocol. When the cycle detector detects a perceived cycle, it sends a confirm message (*CNF*) with a token uniquely identifying the perceived cycle to each actor in the cycle. When an actor receives *CNF*, it sends an acknowledgement message (*ACK*) with the token to the cycle detector without regard to the actor's view of its topology.

If the cycle detector receives *ACK* from an actor in a perceived cycle without receiving *UNB*, then that actor did not unblock between blocking and the detection of the perceived cycle. This means that the actor's view of its topology when the perceived cycle was detected was the same as the cycle detector's view of that actor's topology used to detect the perceived cycle. Such an actor is *confirmed.* Conversely, if an actor in a cycle changes state, it will send *UNB* before it sends *ACK*. Because messaging is causal, the cycle detector will receive the *UNB* before it receives the *ACK*. When the cycle detector receives *UNB* for an actor, it cancels all perceived cycles containing the newly unblocked actor, since they were detected with an incorrect view of that actor's topology.

Further, if all actors in a perceived cycle are confirmed, then, at the time the cycle was detected, each actor in the cycle had a view of its topology that agreed with the true topology. As a result, the perceived cycle is a true cycle and can be collected.

Example 5.2. Expanding example 5.1 with the conf-ack protocol. This is shown in figure 5.2.

1. The cycle detector sends $\mathit{CNF}(\tau)$ to ι_1 and ι_2, where τ is a token uniquely identifying this perceived cycle.

2. ι_1 processes the pending *INC* from example 5.1 before $\mathit{CNF}(\tau)$, due to causal messaging, and sends $\mathit{UNB}(\iota_1)$ to κ.

3. ι_1 processes $\mathit{CNF}(\tau)$ and sends $ACK(\iota_1, \tau)$ to the cycle detector.

4. κ processes $\mathit{UNB}(\iota_1)$ before $ACK(\iota_1, \tau)$, due to causal messaging, and correctly cancels the perceived cycle.

The conf-ack protocol works by providing the cycle detector with confirmation that the view of the topology used to detect a cycle (which was sent to the cycle detector as a snapshot of each actor's view of its topology) agreed with the true topology when the cycle was detected. This approach allows the cycle detector to work concurrently with other actors, without shared memory, locks, read/write barriers, cache coherency, or any other form of thread coordination.

5.2.7 Causal Messaging

In order to maintain the actor's reference count invariant, message delivery must be *causal.* When an actor ι_1 sends *INC* to an actor ι_2 before including it in a message to an actor ι_3, ι_2 must process that *INC* before any *DEC* message sent by ι_3. Each message is an *effect,* and every message the sending actor has previously sent or received is a *cause* of that effect. Messaging is causal if every cause is enqueued before the effect. Causality propagates forward: the causes of an effect are also causes for any secondary effect.

Example 5.3. Causal messaging.

1. ι_1 sends μ_1 to ι_2.

2. ι_1 sends μ_2 to ι_3 .

3. After receiving μ_2, ι_3 sends μ_3 to ι_2.

4. To preserve causality, ι_2 must receive μ_1 before μ_3.

Causality is easy and efficient to achieve on a single node, even in a many-core setting. Sending a message and enqueuing it at the destination can be done with a single atomic operation. As a result, causality is a natural consequence of lock-free, wait-free FIFO message queues, and has no overhead.

$$
\begin{array}{rcll}
 & & \mathit{Actor} & = \mathit{ActorID} \times (\mathit{FieldID} \rightarrow \mathit{Value}) \\
 & & & \times\ \overline{\mathit{Message}} \times \mathit{Stack} \times \mathit{Expr} \\
 & & & \times\ (\mathit{ActorID} \rightarrow \overline{\mathit{Message}}) \\
 & & & \times\ \mathit{RefCount} \times \mathit{ExMap} \times \mathit{Blocked} \\
\mu & \in & \mathit{Message} & = (\mathit{MethodID} \times \overline{\mathit{Value}}) \\
 & & & \mid\ INC(\mathit{Integer}) \mid DEC(\mathit{Integer}) \\
 & & & \mid\ BLK(\mathit{ActorAddr}, \mathit{RefCount}, \mathit{ExMap}) \\
 & & & \mid\ UNB(\mathit{ActorAddr}) \\
 & & & \mid\ CNF(\mathit{Token}) \mid ACK(\mathit{ActorAddr}, \mathit{Token}) \\
CD & \in & \mathit{CycleDetector} & = \mathit{PerceivedTopo} \times \mathit{PerceivedCycles} \times \mathit{Token} \\
PT & \in & \mathit{PerceivedTopo} & = \mathit{ActorAddr} \rightarrow (\mathit{RefCount} \times \mathit{ExMap}) \\
PC & \in & \mathit{PerceivedCycles} & = \mathit{Token} \rightarrow (\mathit{ActorAddr} \rightarrow \mathit{Boolean}) \\
\xi & \in & \mathit{ExMap} & = \mathit{Addr} \rightarrow \mathit{Integer} \\
\rho & \in & \mathit{RefCount} & = \mathit{Integer} \\
\beta & \in & \mathit{Blocked} & = \mathit{Boolean} \\
\tau & \in & \mathit{Token} & = \mathit{Integer}
\end{array}
$$

Figure 5.3: Runtime entities for actor GC. Elements that are unchanged are greyed out.

5.2.8 Consistency Model

MAC requires only weak memory consistency. In particular, when a message is sent, all writes to the contents of the message must be visible to the receiver of the message. This can be implemented with a release barrier on message send. On the x86 architecture, this release barrier is implicit on all writes, so no fence is required. Moreover, because MAC requires no shared memory other than the contents of messages, no consistency model is necessary for other writes, e.g. when the cycle detector updates its view of blocked actor topology.

5.3 Formal Model

The formal model is expressed as an operational semantics for MAC. Types and identifier conventions are presented in figure 5.3, and the steps that rewrite the configuration are presented in figures 5.4 and 5.6. The operational semantics extends the runtime entities in figure 3.3 and the semantics in figures 3.4 and 3.5.

Actors are extended with a local reference count, an external map, and a blocked flag. Messages are extended with a collection of garbage collection messages that are not available to the programming language.

The cycle detector is modelled as an actor whose local state is composed of the cycle detector's view of the blocked actor topology (PT), the set of perceived cycles that are awaiting confirmation (PC), and the next token that will be used to identify a perceived cycle (τ). The cycle detector's identifier is α_{CD}, which is instantiated when a program begins as $(\emptyset, \emptyset, 0)$, is globally accessible, but is not available to expressions in the operational semantics in chapter 3.

5.3.1 Ownership

In Pony, each actor is responsible for garbage collecting any objects it has allocated. The runtime provides a mapping of $\mathit{Addr} \rightarrow \mathit{ActorAddr}$ that returns the allocating actor for an object address, or returns the actor itself for an actor address. This owner function is referred to as $\mathcal{O}$. The implementation of the owner function is described in section A.4.

5.3.2 Reachability

In the operational semantics for garbage collection, it is necessary to determine which addresses are *reachable* from some initial address in the heap. To do so requires not just the heap and the address, but also a *reference capability*. This is because a `tag` is an opaque reference capability, so the fields of an object seen as a `tag` are not reachable.

For garbage collection purposes, the owner of an address is implicitly reachable if the address is reachable. This is because an actor α must not be prematurely garbage collected if some other actor α' can reach an object allocated by α, as otherwise such an object would either never be garbage collected (since the allocating actor is responsible for collection) or would itself be prematurely collected.

Definition 5.1. Reachability

$$\begin{array}{lcl}
\mathit{Reach}(\chi, \mathit{null}, \kappa) & = & \emptyset \\
\mathit{Reach}(\chi, \iota, \mathtt{tag}) & = & \{\iota, \mathcal{O}(\iota)\} \\
\mathit{Reach}(\chi, \iota, \kappa) & = & \{\iota, \mathcal{O}(\iota)\} \\
 & \cup & \{\iota' \mid \exists \mathtt{f}.\mathcal{F}(\chi(\iota) \downarrow_1, \mathtt{f}) = \mathtt{S}\,\kappa' \wedge \iota' \in \mathit{Reach}(\chi, \chi(\iota, \mathtt{f}), \kappa')\} \\
\mathit{Reach}(\chi, \overline{v}, \overline{\kappa}) & = & \bigcup_{i \in 1..|\overline{v}|} \mathit{Reach}(\chi, v_i, \kappa_i) \\
\mathit{Reach}(\chi, \alpha, (\mathtt{n}, \overline{v})) & = & \mathit{Reach}(\chi, \overline{v}, \overline{\kappa}) \ \mathit{where}\ \mathcal{M}(\chi(\alpha) \downarrow_1, \mathtt{n}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \mathtt{e}, \_)
\end{array}$$

Here, the reachable set is defined as all actors and objects that can be *read* from some starting point (including the starting point), plus the owners of all such reachable objects.

Reachability is related to liveness. Actors and objects are live if they are reachable from some actor α, expressed as $\mathit{Reach}(\chi, \alpha, \mathtt{ref})$, or if they are reachable from some as yet undelivered message $(\mathtt{n}, \overline{v})$. Importantly, it is not necessary in MAC to determine global liveness in order to correctly collect actors.

Note that for garbage collection, accounting for viewpoint adaptation is not necessary. Viewpoint adaptation governs how a reference capability κ sees another reference capability κ', i.e. $\kappa \triangleright \kappa'$, but in the context of garbage collection, κ for some path to some ι must account for possible future paths to ι. Thus, while $\texttt{iso} \triangleright \texttt{box} = \texttt{tag}$, during garbage collection we must account for the fact that the `iso` origin may in the future be seen as some κ such that $\texttt{iso} \leq \kappa$. In other words, if the origin is later viewed as a `val`, we must not have previously garbage collected the contents of the field, as now $\texttt{val} \triangleright \texttt{box} = \texttt{val}$, and the field's contents are now readable.

5.3.3 Reference Count Invariant

To ensure that no actor is prematurely collected, but all actors are eventually collected, a reference count invariant is maintained. This invariant has the form:

$$\mathit{LRC}(\alpha) + \mathit{INC}(\alpha) - \mathit{DEC}(\alpha) = \mathit{AMC}(\alpha) + \mathit{FRC}(\alpha)$$

This invariant is evaluated in the context of some heap χ. This states that the actor's local reference count, *LRC*, plus the sum of the values of all increment messages for that actor, *INC*, is always equal to the number of application messages that can reach that actor, *AMC*, plus the sum of all foreign reference counts for the actor, *FRC*, plus the sum of the values of all decrement messages for that actor, *DEC*.

Definition 5.2. Reference count invariant components

$$\begin{array}{lcl}
\mathit{LRC}(\alpha) & = & \chi(\alpha)\downarrow_7 \\
\mathit{INC}(\alpha) & = & \sum_{i=1}^{|Q(\chi,\alpha)|} \begin{cases} \rho & \mathit{if}\ Q(\chi,\alpha)_i = \mathit{INC}(\rho) \\ 0 & \mathit{otherwise} \end{cases} \\
\mathit{DEC}(\alpha) & = & \sum_{i=1}^{|Q(\chi,\alpha)|} \begin{cases} \rho & \mathit{if}\ Q(\chi,\alpha)_i = \mathit{DEC}(\rho) \\ 0 & \mathit{otherwise} \end{cases} \\
\mathit{AMC}(\alpha) & = & \sum_{\alpha' \in dom(\chi)} \sum_{i=1}^{|Q(\chi,\alpha')|} \begin{cases} 1 & \mathit{if}\ \alpha \in \mathit{Reach}(\chi, \alpha, Q(\chi,\alpha')_i) \\ 0 & \mathit{otherwise} \end{cases} \\
\mathit{FRC}(\alpha) & = & \sum_{\alpha' \in dom(\chi)} \chi(\alpha')\downarrow_8(\alpha)
\end{array}$$

The reference count invariant is maintained because reference counts (or, more properly, reference count weights) are *created* when an actor sends a reference to itself, *destroyed* when an actor received a reference to itself, and *moved* when an actor sends a reference to another actor. This is detailed in sections 5.3.4 and 5.3.5.

Additionally, each actor maintains another invariant: if an actor α is reachable from the local state of another actor α', then α has a local reference count of 1 or more, and α' will hold a foreign reference count for α of 1 or more.

Definition 5.3. Reachability invariant

$$\forall \alpha, \alpha'. \alpha \neq \alpha' \wedge \alpha \in \mathit{Reach}(\chi, \alpha', \texttt{ref}) \Rightarrow \chi(\alpha) \downarrow_7 > 0 \wedge \chi(\alpha') \downarrow_8 (\alpha) > 0$$

5.3.4 Sending a Reference

When an actor α sends a reference to an actor α' to some actor α'', the sending actor α must change either its reference count or its external map to reflect that the resulting message has an implicit reference count of 1 for α'. This is the case even when some of these actors are the same actor.

When $\alpha = \alpha'$, α must increment its reference count by 1. This balances the implicit reference count the message has, maintaining the reference count invariant.

When $\alpha \neq \alpha'$, α must decrement its external map reference count for α' by 1, effectively *moving* a reference from α to the message. In order to do so, α must have a reference count for α' that is greater than 1, since there is no guarantee that the local state of α no longer contains a reference to α', and it is important to not require examining all local state when sending a message.

If α has a reference count of 1 for α' (it cannot be 0 or less, since α' must be accessible to α in order for α' to be included in a message), then α cannot execute the Ator or Async rule. Instead, the Acquire rule must be executed first. Note that the Acquire rule allows α to increase its reference count for α' without thread coordination or requiring a message in response.

This is expressed in the definition of *Adj*, the rule for adjusting reference counts, in figure 5.5, which requires that α have a reference count greater than $-n$ for α'. The definition of *Send* uses -1 for n, establishing this requirement.

Definition 5.4. Sending a reference

$$\begin{array}{lcl}
\mathit{Send}(\chi, \alpha, \{v\} \cup \mathit{vs}) & = & \mathit{Send}(\mathit{Send}(\chi, \alpha, v), \alpha, \mathit{vs}) \\
\mathit{Send}(\chi, \alpha, \emptyset) & = & \chi \\
\mathit{Send}(\chi, \alpha, \mathit{null}) & = & \chi \\
\mathit{Send}(\chi, \alpha, \omega) & = & \chi \\
\mathit{Send}(\chi, \alpha, \alpha') & = & \mathit{Adj}(\chi, \alpha, \alpha', 1)
\end{array}$$

Note that for actor garbage collection, *Send* ignores object references. This will be extended to account for object references in chapter 6. *Adj* is defined such that if α sends a reference to itself, then α adds n (here, -1) to its local reference count. On the other hand, if α sends a reference to another actor α', then α subtracts n from its foreign reference count for α'.

5.3.5 Receiving a Reference

When an actor α receives a reference to an actor α', the receiving actor α must change either its reference count or its external map to reflect that the message no longer exists, and its implicit reference counts have been *moved* to α.

When $\alpha = \alpha'$, α must decrement its reference count by 1. The implicit reference in the message has been destroyed, so the actor's reference count is decremented to maintain the reference count invariant.

When $\alpha \neq \alpha'$, α must increment its external map reference count for α' by 1, effectively *moving* the message's implicit reference count to α. The definition of *Recv* uses 1 for n, which is a simpler case than *Send*, as *Adj* will always be possible. No adjusting messages are ever required when receiving a message.

Definition 5.5. Receiving a reference

$$\begin{array}{lcl} \mathit{Recv}(\chi, \alpha, \{v\} \cup vs) & = & \mathit{Recv}(\mathit{Recv}(\chi, \alpha, v), \alpha, vs) \\ \mathit{Recv}(\chi, \alpha, \emptyset) & = & \chi \\ \mathit{Recv}(\chi, \alpha, \mathit{null}) & = & \chi \\ \mathit{Recv}(\chi, \alpha, \omega) & = & \chi \\ \mathit{Recv}(\chi, \alpha, \alpha') & = & \mathit{Adj}(\chi, \alpha, \alpha', -1) \end{array}$$

As with *Send*, here *Recv* ignores object references. This will be extended to account for object references in chapter 6.

5.3.6 Closed Cycles

A set of actors is considered a *closed set* by the cycle detector when the cycle detector's perceived topology indicates that all references to actors in the set are accounted for by other actors in the set.

Definition 5.6. Closed cycles

$$\begin{array}{lcl} \mathit{Closed}(\chi, \alpha_{CD}, \alpha s) & \mathit{iff} & PT = \chi(\alpha_{CD}, \mathtt{f}_1) \wedge \\ & & \forall \alpha \in \alpha s. \\ & & \sum_{\alpha' \in \alpha s} PT(\alpha') \downarrow_2 (\alpha) = PT(\alpha) \downarrow_1 \end{array}$$

5.3.7 Actor Local Execution

The extended small-step operational semantics in figure 5.4 describe the effects of actor local execution that includes garbage collection of actors. Rules from figure 3.4 that are not listed remain unchanged. As usual in concurrency, execution is non-deterministic. The identifier n is used to represent an arbitrary integer such than $n > 0$. This is used when an actor is created with some initial reference count, or when an actor creates additional reference count for another actor by sending an *INC* message. The value of n could vary, or the implementation could use a fixed value.

The ATOR rule is modified such that the creating actor α begins with a foreign reference count that balances the initial local reference count of α'.

$$\frac{\begin{array}{c}(\alpha', \chi') = \mathit{New}(\chi, \alpha, \mathtt{A}, \mathtt{k}, \varphi(\overline{\mathtt{y}}))\\ \mathtt{t} \notin \varphi \quad \varphi' = \varphi[\mathtt{t} \mapsto \alpha]\end{array}}{\chi, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{A.k}(\overline{\mathtt{y}}) \rightsquigarrow \chi', \alpha \cdot \overline{\varphi} \cdot \varphi', \mathtt{t}}\ \textsc{Ator}$$

$$\frac{\begin{array}{c}\alpha' = \varphi(\mathtt{z})\\ \chi' = \mathit{Send}(\chi, \alpha, \mathit{Reach}(\chi, \alpha', (\mathtt{n}, \varphi(\overline{\mathtt{y}}))))\\ \chi'' = \chi'[\alpha, \alpha' +\!+ (\mathtt{b}, \varphi(\overline{\mathtt{y}}))]\end{array}}{\chi, \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{z.b}(\overline{\mathtt{y}}) \rightsquigarrow \chi'', \alpha \cdot \overline{\varphi} \cdot \varphi, \mathtt{z}}\ \textsc{Async}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha) \downarrow_9 \quad (\mathtt{n}, \overline{v}) \cdot \overline{\mu} = \chi(\alpha) \downarrow_3\\ \chi' = \mathit{Recv}(\chi, \alpha, \mathit{Reach}(\chi, \alpha, (\mathtt{n}, \overline{v})))\\ \varphi = (\mathtt{n}, [\mathtt{this} \mapsto \alpha, \overline{\mathtt{x}} \mapsto \overline{v}], \cdot)\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'[\alpha \mapsto \overline{\mu}], \alpha \cdot \varphi, \mathtt{e}}\ \textsc{Behave}$$

$$\frac{\begin{array}{c}\mathit{CNF}(\tau) \cdot \overline{\mu} = \chi(\alpha) \downarrow_3\\ \chi' = \chi[\alpha, \alpha_{CD} +\!+ \mathit{ACK}(\alpha, \tau)]\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'[\alpha \mapsto \overline{\mu}], \alpha, \varepsilon}\ \textsc{Cnf}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha) \downarrow_9 \quad \mathit{INC}(\rho) \cdot \overline{\mu} = \chi(\alpha) \downarrow_3\\ \chi' = \mathit{Adj}(\chi, \alpha, \alpha, \rho)\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'[\alpha \mapsto \overline{\mu}], \alpha, \varepsilon}\ \textsc{Inc}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha) \downarrow_9 \quad \mathit{DEC}(\rho) \cdot \overline{\mu} = \chi(\alpha) \downarrow_3\\ \chi' = \mathit{Adj}(\chi, \alpha, \alpha, -\rho)\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'[\alpha \mapsto \overline{\mu}], \alpha, \varepsilon}\ \textsc{Dec}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha) \downarrow_9 \quad () = \chi(\alpha) \downarrow_3\\ \mu = \mathit{BLK}(\alpha, \chi(\alpha) \downarrow_7, \chi(\alpha) \downarrow_8)\\ \chi' = \chi[\alpha, \alpha_{CD} +\!+ \mu]\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'[\alpha \mapsto \mathit{true}], \alpha, \varepsilon}\ \textsc{Block}$$

$$\frac{\begin{array}{c}\chi(\alpha) \downarrow_9\\ \mu \cdot \overline{\mu} = \chi(\alpha) \downarrow_3 \quad \mu \neq \mathit{CNF}(\tau)\\ \chi' = \chi[\alpha, \alpha_{CD} +\!+ \mathit{UNB}(\alpha)]\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'[\alpha \mapsto \mathit{false}], \alpha, \varepsilon}\ \textsc{Unblock}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha) \downarrow_9 \quad \alpha \neq \alpha'\\ \chi(\alpha) \downarrow_8 (\alpha') > 0\\ \chi' = \chi[\alpha, \alpha' +\!+ \mathit{INC}(n)]\\ \chi'' = \mathit{Adj}(\chi', \alpha, \alpha', -n)\end{array}}{\chi, \alpha \cdot \overline{\varphi}, \mathtt{e} \rightsquigarrow \chi'', \alpha \cdot \overline{\varphi}, \mathtt{e}}\ \textsc{Acquire}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha) \downarrow_9 \quad \alpha \neq \alpha'\\ \chi(\alpha) \downarrow_8 (\alpha') > 0\\ \alpha' \notin \mathit{Reach}(\chi, \alpha, \mathtt{ref})\\ \chi' = \chi[\alpha, \alpha' +\!+ \mathit{DEC}(\chi(\alpha) \downarrow_8 (\alpha'))]\\ \chi'' = \mathit{Rel}(\chi', \alpha, \alpha')\end{array}}{\chi, \alpha, \varepsilon \rightsquigarrow \chi'', \alpha, \varepsilon}\ \textsc{Release}$$

Figure 5.4: Operational semantics of actor GC

$$\begin{array}{lcl}
\mathit{New}(\chi, \alpha, \mathtt{A}, \mathtt{k}, \overline{v}) & = & (\alpha', \chi'')\ \mathit{where}\\
 & & \quad \alpha' \notin \mathit{dom}(\chi) \wedge\\
 & & \quad \mathcal{M}(\mathtt{A}, \mathtt{k}) = (\_, \overline{\mathtt{x} : \mathtt{S}\,\kappa}, \_, \_) \wedge\\
 & & \quad \chi' = \mathit{Send}(\chi, \alpha, \mathit{Reach}(\chi, \overline{v}, \overline{\kappa})) \wedge\\
 & & \quad \chi'' = \chi'[\alpha \mapsto \mathit{Inc}(\chi, \alpha, \alpha', n),\\
 & & \qquad \alpha' \mapsto (\mathtt{A}, \mathcal{F}\mathtt{s}(\mathtt{A}) \mapsto \mathit{null}, (\mathtt{k}, \overline{v}), \alpha, \varepsilon, \emptyset, n, \emptyset, \mathit{false})]\\
\mathit{Adj}(\chi, \alpha, \alpha', n) & = & \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1 ... \chi(\alpha) \downarrow_7,\\
 & & \quad \chi(\alpha) \downarrow_8 [\alpha' \mapsto (\chi(\alpha) \downarrow_8 (\alpha') - n)], \chi(\alpha) \downarrow_9)]\\
 & & \quad \mathit{if}\ \alpha \neq \alpha' \wedge \chi(\alpha) \downarrow_8 (\alpha') > n\\
\mathit{Adj}(\chi, \alpha, \alpha', n) & = & \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1 ... \chi(\alpha) \downarrow_6,\\
 & & \quad \chi(\alpha) \downarrow_7 + n, \chi(\alpha) \downarrow_8, \chi(\alpha) \downarrow_9)]\\
 & & \quad \mathit{if}\ \alpha = \alpha'\\
\mathit{Rel}(\chi, \alpha, \iota) & = & \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1 ... \chi(\alpha) \downarrow_7,\\
 & & \quad \chi(\alpha) \downarrow_8 [\iota \mapsto 0], \chi(\alpha) \downarrow_9)]\\
\chi[\alpha \mapsto \beta] & = & \chi[\alpha \mapsto (\chi(\alpha) \downarrow_1 ... \chi(\alpha) \downarrow_8, \beta)]
\end{array}$$

Figure 5.5: Auxiliary definitions for actor GC

The Async rule is modified such that the reachable contents of the message μ have their reference counts adjusted to account for the implicit references held by μ. Note that if it is not possible to execute this rule, due to the sending actor α holding a foreign reference count of 1 for some actor α' referenced in μ, the Acquire rule can be executed to add weight to the foreign reference count α holds for α' before μ is sent.

In Behave, reference count adjustment is also accounted for, moving the implicit references held by the received message μ to the receiving actor α.

The Cnf rule shows how an actor α that receives $\mathit{CNF}(\tau)$ simply echoes $\mathit{ACK}(\alpha, \tau)$ to the cycle detector, without needing to examine its own state or track any state transitions.

In Inc and Dec, the receiving actor α adjusts its own local reference count to account for changes in foreign reference counts due to weight being added to allow sending a reference to α (producing *INC*) or α no longer being reachable in some other actors local state (producing *DEC*).

Block is executed when an actor in currently *unblocked* and it has an empty queue. The cycle detector is notified of the actor's view of its own topology and the actor is marked as *blocked*.

When *blocked*, an actor that has a non-empty queue must execute Unblock before it can process any messages on its queue. This notifies the cycle detector that the actor is no longer blocked and its previous view of its own topology is now invalid.

The Acquire rule allows an actor α to add weight to its foreign reference count for some other actor α'. The semantics allows this rule to be executed non-deterministically, but it is strictly required in order to send a reference to α' in a message if α holds a foreign reference count of 1 for α' . Note that Acquire uses $-n$ rather than n, as α wishes to add to, rather than subtract from, its foreign reference count for α'. Since *Adj* is defined in terms of sending and receiving, it must be reversed for Acquire.

The converse of Acquire is the Release rule, which is executed when an actor α cannot reach some other actor α' in its local state, but holds a foreign reference count for α'. This is a garbage collection rule, allowing α to return that weight to α' in the form of a *DEC* message.

5.3.8 Cycle Detector Execution

The rules in figure 5.6 describe local execution of the cycle detector α_{CD}. Note that the cycle detector is never sent application level messages, so the rules from figure 3.4 do not apply to it. These rules model the perceived topology as field $\mathtt{f}_1$, the perceived cycles as field `f2`, and the next token as field $\mathtt{f}_3$.

$$\frac{\begin{array}{c}\mathit{BLK}(\alpha,\rho,\xi)\cdot\overline{\mu}=\chi(\alpha_{CD})\downarrow_3\\ \chi'=\chi[\alpha_{CD},\alpha\mapsto(\rho,\xi)]\end{array}}{\chi,\alpha_{CD},\varepsilon\rightsquigarrow\chi'[\alpha_{CD}\mapsto\overline{\mu}],\alpha_{CD},\varepsilon}\ \textsc{Blk}\qquad\frac{\begin{array}{c}\mathit{UNB}(\alpha)\cdot\overline{\mu}=\chi(\alpha_{CD})\downarrow_3\\ \chi'=\chi[\alpha_{CD}\backslash\{\alpha\}]\end{array}}{\chi,\alpha_{CD},\varepsilon\rightsquigarrow\chi'[\alpha_{CD}\mapsto\overline{\mu}],\alpha_{CD},\varepsilon}\ \textsc{Unb}$$

$$\frac{\begin{array}{c}\mathit{ACK}(\alpha,\tau)\cdot\overline{\mu}=\chi(\alpha_{CD})\downarrow_3\\ \tau\in\chi(\alpha_{CD},\mathtt{f_2})\\ \chi'=\chi[\alpha_{CD},\tau,\alpha\mapsto\mathit{true}]\end{array}}{\chi,\alpha_{CD},\varepsilon\rightsquigarrow\chi'[\alpha_{CD}\mapsto\overline{\mu}],\alpha_{CD},\varepsilon}\ \textsc{Ack}\qquad\frac{\begin{array}{c}\mathit{ACK}(\alpha,\tau)\cdot\overline{\mu}=\chi(\alpha_{CD})\downarrow_3\\ \tau\notin\chi(\alpha_{CD},\mathtt{f_2})\end{array}}{\chi,\alpha_{CD},\varepsilon\rightsquigarrow\chi[\alpha_{CD}\mapsto\overline{\mu}],\alpha_{CD},\varepsilon}\ \textsc{NoAck}$$

$$\frac{\begin{array}{c}\mathit{Closed}(\chi,\alpha_{CD},\alpha s)\quad\tau=\chi(\alpha_{CD},\mathtt{f_3})\\ \chi'=\chi[\alpha_{CD},\tau,\alpha s\mapsto\mathit{false}]\\ \chi''=\chi'[\alpha_{CD},\mathtt{f_3}\mapsto\tau+1]\\ \chi'''=\chi''[\alpha_{CD},\alpha s++\mathit{CNF}(\tau)]\end{array}}{\chi,\alpha_{CD},\varepsilon\rightsquigarrow\chi''',\alpha_{CD},\varepsilon}\ \textsc{Detect}\qquad\frac{\begin{array}{c}\alpha s=\chi(\alpha_{CD},\mathtt{f_2})(\tau)\\ \forall\alpha\in\alpha s.\chi(\alpha_{CD},\mathtt{f_2})(\tau)(\alpha)\\ \chi'=\mathit{Rel}(\chi,\alpha s)\\ \chi''=\chi'[\alpha_{CD}\backslash\alpha s]\backslash\alpha s\end{array}}{\chi,\alpha_{CD},\varepsilon\rightsquigarrow\chi''[\alpha_{CD}\mapsto\overline{\mu}],\alpha_{CD},\varepsilon}\ \textsc{Collect}$$

Figure 5.6: Operational semantics of cycle detection

$$\begin{array}{lcl}
\chi[\alpha_{CD},\alpha\mapsto(\rho,\xi)] & = & \chi[\alpha_{CD},\mathtt{f_1}\mapsto\chi(\alpha_{CD},\mathtt{f_1})[\alpha\mapsto(\rho,\xi)]]\\
\chi[\alpha_{CD}\backslash\alpha s] & = & \chi[\alpha_{CD},\mathtt{f_1}\mapsto\chi(\alpha_{CD},\mathtt{f_1})\backslash\alpha s,\\
 & & \quad\alpha_{CD},\mathtt{f_2}\backslash\{\tau\,|\,\alpha\in\alpha s\wedge\alpha\in\chi(\alpha_{CD},\mathtt{f_2})(\tau)\}]\\
\chi[\alpha_{CD},\tau,\alpha\mapsto\beta] & = & \chi[\alpha_{CD},\mathtt{f_2}\mapsto\chi(\alpha_{CD},\mathtt{f_2})(\tau)[\alpha\mapsto\beta]]\\
\chi[\alpha_{CD},\tau,\{\alpha\}\cup\alpha s\mapsto\beta] & = & \chi[\alpha_{CD},\mathtt{f_2}\mapsto\chi(\alpha_{CD},\mathtt{f_2})(\tau)[\alpha\mapsto\beta]][\alpha,\alpha s\mapsto\beta]\\
\chi[\alpha_{CD},\{\alpha\}\cup\alpha s++\mu] & = & \chi[\alpha_{CD},\alpha++\mu][\alpha_{CD},\alpha s++\mu]\\
\\
\mathit{Rel}(\chi,\{\alpha\}\cup\alpha s) & = & \mathit{Rel}(\mathit{Rel}(\chi,\alpha),\alpha s)\\
\mathit{Rel}(\chi,\alpha) & = & \mathit{Rel}(\chi,\alpha,\mathit{Reach}(\chi,\alpha,\mathtt{ref}))\\
\mathit{Rel}(\chi,\alpha,\{\iota\}\cup\iota s) & = & \mathit{Rel}(\mathit{Rel}(\chi,\alpha,\iota),\iota s)
\end{array}$$

Figure 5.7: Auxiliary definitions for cycle detection

The Blk rule updates the cycle detector's view of the topology of some actor α upon receipt of a *BLK* message. Similarly, the Unb rule invalidates the cycle detector's view of the topology of some actor α upon receipt of an *UNB* message. It also invalidates all perceived cycles that α is a member of, as those perceived cycles were detected using an out of date view of the topology of α.

The Ack rule updates a perceived cycle to indicate that some actor α has acknowledged the confirmation message it was sent. If the cycle detector receives an *ACK* message for an invalidate perceived cycle, the NoAck rule is executed instead, discarding the message.

In Detect, the cycle detector finds a close cycles of blocked actors and sends a *CNF* message to each actor in the cycle. The cycle is recorded as a perceived cycle, and no further action is taken until the cycle is either acknowledged by all members of the cycle, or the cycle is invalidated by an *UNB* message from some member of the cycle.

The Collect rule is executed when a perceived cycle is fully acknowledged. Any remaining foreign reference counts held by actors in the cycle are released, generating *DEC* messages, and the actors in the cycle are removed from the heap. The generated *DEC* messages are important when a closed cycle holds references to actors outside that cycle. If the *DEC* messages were not sent, such actors could not be collected in the future.

5.4 Completeness

If a cycle of blocked actors exists, each actor will have sent *BLK* to the cycle detector. The cycle detector will eventually execute Blk for each blocked actor, and will eventually execute Detect and begin a confirmation process that will result in executing Collect. This process is non-deterministic, but it is theoretically possible to detect a cycle as soon as it appears. If all actors are blocked, the system will find all cycles.

The program terminates when it is not possible to apply any rule. This occurs when no actors are executing (preventing any actor local execution rules from being applied), the queue is empty (preventing any actor or cycle detector message receipt rules from being applied), and no cycles are detected (preventing any cycle detector local execution rules from being applied).

In practice, early termination can be achieved by detecting *quiescence*, without waiting for the cycle detector to collect all actors. Similarly, it is possible for an actor with $LRC(\alpha) = 0$ and an empty queue to collect itself, without waiting for the cycle detector, as this indicates that no other actor can send a message to α and α has no pending work. In this circumstance, α will never have pending work, as α must be executing in order to send a message to itself.

5.5 Robustness

As presented, *MAC* is sound and does not have exceptional conditions. However, the protocol is robust even if failure is introduced. If the cycle detector fails, cycles of dead actors will not be collected, but no live actor will be collected.

If an actor fails, the result depends on whether or not the cycle detector's view of the failed actor's topology is in agreement with the failed actor's view of its own topology. If it is, the failed actor can be considered blocked, and the system will function normally. If the cycle detector's view of the failed actor's topology is not in sync, then there is no way to determine what other actors the failed actor referenced. As a result, actors the failed actor held a reference to will not receive *DEC* messages for those references and will not be collected. However, it remains the case that the cycle detector will continue to collect other dead cycles, and no live actor will be collected.

Moreover, failure of actors or the cycle detector does not jeopardise termination of the overall system. Namely, collection of all actors is not required in order to reach a *quiescent* state where no rules can be applied. This allows the program to terminate even when some dead actors have not been collected. As a result, failure results in uncollected dead actors but does not impact soundness or robustness.

Failure of individual messages, where a message is sent but not received while future messages from the same sender are successful, impacts the system differently depending on the message type. A failed *DEC* results in an actor with an excess reference count that will not be collected. A failed *CNF* or *ACK* message that pertains to a dead cycle results in the failure to collect that dead cycle, but if the message pertains to a live cycle, there is no impact on the system. A failed *BLK* message results in an actor never being collected if the actor is blocked from that point on, but has no impact on the system if the actor ever unblocks. A failed *APP* message will result in excess reference counts for actors in the message, with the result that those actors will not be collected.

The two messages that can impact soundness on failure are *INC* and *UNB*. A failed *INC* message results in an actor that has a reference count that is too low. As a result, the cycle detector may find perceived cycles that are smaller than the true cycle. If the actors in the perceived cycle are all blocked, the cycle may be collected while an unblocked actor retains a reference to a collected actor. A failed *UNB* message for an actor in a perceived cycle can cause the cycle to be incorrectly collected if all other actors in the cycle are blocked. The sender of the failed *UNB* message will now respond with an *ACK* without having unblocked, and the cycle detector will incorrectly perceive it as having confirmed.

However, the actor-model requires guaranteed message delivery [3]. Failure of an individual message that cannot be corrected with buffering, retries, or other techniques, can thus be treated as failure of the sending actor. If a failed message results in all future messages from the sender also failing, no form of failure impacts either soundness or robustness.

In Pony, neither message nor actor failure occurs in a running program. The program itself may fail (due to, for example, a bug in the runtime implementation, or a class of bug that Pony cannot guarantee against, such as running out of memory), but individual actors and messages will not.

This does not hold true in the distributed setting, which will be discussed in section 5.6.

5.6 Distributed Actor GC

In the distributed setting, the DELIVER rule need not be executed immediately after ASYNC. This breaks causal ordering of messages, but maintains FIFO ordering of messages between pairs of actors. Under these conditions, *MAC* as described is not sound, as a *DEC* message may overtake an *INC* message, resulting in premature collection of an actor.

However, a small change to the protocol allows *MAC* to function without causal ordering of messages. This change will be described briefly and informally, leaving a a complete and formal development of the protocol change for future work. It is important to note that distributed actor collection has not yet been implemented in the runtime. As such, this section represents a sketch of a possible implementation, rather than a description of completed work.

5.6.1 Additional Runtime Entities

Nodes in the distributed system are referred to with the identifier $\mathcal{N}$. An actor α that is referenced by a node $\mathcal{N}$, but is executing on a different node $\mathcal{N}'$, has a *proxy actor* on $\mathcal{N}$, referred to as $\alpha : \mathcal{N}$. The proxy actor $\alpha : \mathcal{N}$ is used as the destination address for messages to α sent by actors on $\mathcal{N}'$. Such messages are forwarded to α on $\mathcal{N}$.

Each node in the distributed system runs its own cycle detector. Each actor keeps track of an additional value: a per-node *distributed reference count*, which is an approximation of *distributed weight*, in the same way the *local reference count* is an approximation of weight on a single node. This distributed reference count is sent to the cycle detector, along with the local reference count and the external map, when an actor blocks.

Proxy actors keep a per-node *proxy reference count*, which is the counterbalance to the distributed reference count, in the same way the *foreign reference* *count* is the counter-balance to the local reference count on a single node. Note that both the distributed reference count and the proxy reference count are maps (*Node* $\rightarrow$ *RefCount*) rather than scalar values.

5.6.2 Sending a Local Reference Locally

No change to the protocol is necessary when an actor sends a reference to a local actor (i.e. an actor running on the same node as the sender) to a local receiver, nor when an actor receives a reference to a local actor from a local sender.

5.6.3 Sending a Local Reference Remotely

When an actor α_1 on node $\mathcal{N}_1$ sends a reference to a local actor α_2 to another actor α_3 on a remote node $\mathcal{N}_2$, α_1 also sends a message to α_2 telling it to increment its distributed reference count for $\mathcal{N}_2$. When α_3 receives the reference to the remote actor α_2 (remote from the perspective of α_3), it sends a message to the local proxy $\alpha_2 : \mathcal{N}_2$, telling it to increment its proxy reference count for $\mathcal{N}_2$.

5.6.4 Sending a Remote Reference Locally

Expanding on the example above, when α_3 sends a reference to $\alpha_2 : \mathcal{N}_2$ to another actor α_4 on the local node $\mathcal{N}_2$, the reference to $\alpha_2 : \mathcal{N}_2$ is treated exactly as if it were a local actor. Effectively, the proxy reference count is the node's approximation of its distributed weight, and so it is not altered when references to the remote actor are sent in local messages.

5.6.5 Sending a Remote Reference Remotely

Expanding further on the example, when α_3 sends a reference to $\alpha_2 : \mathcal{N}_2$ to another actor α_5 on the remote node $\mathcal{N}_3$, α_3 also sends a message to the local proxy $\alpha_2 : \mathcal{N}_2$ telling it to *decrement* its proxy reference count for $\mathcal{N}_3$. Thus the proxy $\alpha_2 : \mathcal{N}_2$ will have a *negative* proxy reference count for $\mathcal{N}_3$. When α_5 receives the reference to the remote actor α_2, it sends a message to the local proxy $\alpha_2 : \mathcal{N}_3$, telling it to increment its proxy reference count for $\mathcal{N}_3$.

5.6.6 Releasing Remote References

When no references to α_2 remain in the local state of any actor on $\mathcal{N}_2$, the proxy actor $\alpha_2 : \mathcal{N}_2$ can be collected. When it is collected, its proxy reference count map is sent to α_2 on $\mathcal{N}_1$. When this is received, it is subtracted from the distributed reference count of α_2. In this example, α_2 has a distributed reference count of $[\mathcal{N}_2 \to 1]$ and subtracts the proxy reference count $[\mathcal{N}_2 \to 1, \mathcal{N}_3 \to -1]$, resulting in a distributed reference count of $[\mathcal{N}_2 \to 0, \mathcal{N}_3 \to 1]$.

To ensure that α_2 is not prematurely collected when another node may still be able to send it a message, it is sufficient for the cycle detector on $\mathcal{N}_1$ not to collect α_2 if it has a non-zero distributed reference count for any node. Allowing $\mathcal{N}_2$ to keep a negative proxy reference count for $\mathcal{N}_3$ in $\alpha_2 : \mathcal{N}_2$ is sufficient to protect α_2 from collection.

For example, if $\mathcal{N}_3$ release $\alpha_2 : \mathcal{N}_3$ first, it would send the proxy reference count $[\mathcal{N}_3 \to 1]$ to α_2, resulting in a distributed reference count of $[\mathcal{N}_2 \to 1, \mathcal{N}_3 \to -1]$. Later, when $\mathcal{N}_2$ released $\alpha_2 : \mathcal{N}_2$ and sent a proxy reference count of $[\mathcal{N}_2 \to 1, \mathcal{N}_3 \to -1]$, the distributed reference count for α_2 would be adjusted to $[\mathcal{N}_2 \to 0, \mathcal{N}_3 \to 0]$ and collection would be allowed (assuming α_2 was no longer referenced locally on $\mathcal{N}_1$).

Note that no additional network messages are required for this protocol extension. Additional local messages are required, but such messages are relatively inexpensive. For example, in the Pony runtime, such a message is a single atomic operation.

5.6.7 Distributed Cycle Detection

As described, this protocol extension will only successfully collect closed cycles of actors that are non-cyclic across nodes. That is, if actors in the cycle on node $\mathcal{N}_1$ (possibly cyclic amongst themselves) reference actors in the cycle on node $\mathcal{N}_2$ (also possibly cyclic amongst themselves), the cycle can still be collected, but it cannot if any of the actors on node $\mathcal{N}_2$ reference any of the actors on $\mathcal{N}_1$.

The solution is to extend the cycle detection phase to allow cycle detectors on a collection of nodes to cooperate. To do so, each cycle detector independently detects perceived cycles, without accounting for distributed reference counts. Once a perceived cycle PC_1 is confirmed on node $\mathcal{N}_1$, it can be immediately collected if all distributed reference counts are zero. If they are not, PC_1 is sent to each node that any actor in PC_1 references.

In the two node example, $\mathcal{N}_1$ sends PC_1 to $\mathcal{N}_2$ and $\mathcal{N}_2$ sends PC_2 to $\mathcal{N}_1$. When $\mathcal{N}_2$ receives PC_1 , the cycle detector on $\mathcal{N}_2$ extends PC_2 with the information from PC_1. The resulting extended perceived cycle PC'_2 is collected if all distributed reference counts can be accounted for via the proxy reference counts contained in PC'_2. Otherwise, the extended cycle PC'_2 is again sent to each node that any actor in PC'_2 references.

In this simple two node example, the cycles are collected after being extended once. In a more complicated multi-node example, it may be necessary to forward extended cycles several times, but the information will eventually propagate to all nodes and allow each actor to be collected by the cycle detector on its host node. If, during this process, any actor unblocks, that will cause the relevant perceived cycles on the local node to be discarded, preventing the cycle from being prematurely collected.

This is effectively a form of *gossip protocol* that allows cycle detection across the distributed system to be eventually consistent, without any actor being prematurely collected.

5.6.8 Distributed Node Failure

In the event of node failure, no actor will be prematurely collected, but some actors will never be collected, due to distributed reference counts that can no longer be accounted for. While any distributed reference count attributable to a failed node can be safely discarded, the failed node may hold negative proxy reference counts for other nodes.

To reduce the risk of node failure resulting in a set of actors that cannot be collected, nodes can periodically clear negative proxy reference counts for other nodes by sending distributed reference count increment messages to host nodes. This does not eliminate the risk, as even if such messages were sent before sending a remote reference to a remote node it would be possible for the decrement message not to arrive due to network failure, even if the remote node was able to receive the remote reference.

5.7 Using *MAC* in Other Actor-Model Languages

The core of *MAC* is that an actor is dead when it has no work to perform and will never again have work to perform, and only then can an actor be safely terminated. To determine if an actor has or will ever have work, *MAC* relies on message queues being the only mechanism by which an actor receives work.

As a result, while *MAC* can be used in any actor-model language, it will only provide safe garbage collection if the language does not allow actors to receive work by other means. This is not universally true for actor-model languages. Erlang and Elixir, for example, both make this guarantee, but Akka (being a library on top of Scala and Java) does not.

To use *MAC* for another language, it may be necessary to modify the operational semantics to accommodate non-atomic behaviours. For example, Behave only accepts application-level messages when an actors stack is empty, but Erlang allows messages to be received using an expression in a function.

In addition, the semantics would need to be modified to account for pattern matching on an actor's queue. Such pattern matching allows causal ordering to be broken. To account for this, MAC messages on the queue would have to be handled in order, and before application level messages that appear after them.

The final wrinkle is the notion of a blocked actor in the presence of non-atomic behaviours. Instead of blocking when a message queue is empty, an actor would only block when the message queue is empty *and* the actor is performing a receive without a timeout.

Chapter 6

Object Collection

As discussed in chapters 3 and 4, Pony, like many actor-model languages, has non-actor entities, in the form of passive (synchronous) objects, and such objects can be passed between actors in messages. In order to pass such messages by reference, in order to have zero-copy message passing, objects that have been passed or shared between actors must be garbage collected, and this garbage collection must be more efficient than copying objects in messages. Garbage collecting such objects in an actor-model language faces similar problems to garbage collecting actors themselves. The inability to examine the global state of the program without pausing the program results in existing approaches to object garbage collection suffering performance problems, particularly when stop-the-world pauses cause the program to become unresponsive.

Existing approaches to reducing stop-the-world pauses in garbage collection [18, 67] significantly reduce such program pauses. However, such general approaches cannot leverage the properties of an actor-model language, particularly the isolation of state within actors. As a result, they rely on techniques such as *read barriers*, which adversely affect mutator thread performance, in order to reduce pause times.

Erlang achieves fully concurrent passive object garbage collection by copying passive objects sent in messages to the *process-local heap* of the destination. This comes at a cost: copying the passive objects can be expensive when large data structures are passed between actors, both when the message is sent (due to the time taken to copy the message) and over time (due to the resulting increased memory usage). Copying message contents also means that *object identity* must be encoded in the data structure by the programmer, rather than being implicitly derived from the object's memory address. While this is less important for a functional language such as Erlang, it is important for an *object capability* language such as Pony.

In Pony, a novel technique for passive object collection has been developed based on Message-based Actor Collection (*MAC*), described in chapter 5. This approach, termed Ownership and Reference Counting for Actors (*ORCA*) extends *MAC* to allow synchronous objects allocated by some actor α and subsequently shared with other actors to be efficiently and safely garbage collected by the allocating actor α without requiring any synchronisation mechanism other than message passing. This combines *actor-local heaps* with an *ownership* model wherein an actor α may be able to reach an object ω in the heap of another actor α', but the allocating actor α' (i.e. the owner) remains responsible for garbage collecting ω.

6.1 Background on Actor-Model Object GC

Existing approaches to collecting passive objects in actor-model languages and libraries fall into four basic categories:

1. *Manual actor termination with global heap tracing.* This approach relies on the programmer manually terminating actors and leverages an existing garbage collection mechanism, such as is present on the JVM or the CLR, as is done in Scala [29], Akka, Kilim [65], AmbientTalk [69], and SALSA 2.0 [72]. Garbage collection in such systems does not leverage actor isolation, but can leverage existing well-known garbage collection techniques for non-actor languages.

2. *Manual actor termination with local heap tracing.* This approach also relies on the programmer manually terminating actors, but the tracing mechanism takes advantage of actor isolation to provide local heaps. To preserve heap locality, messages are copied into the heap of the receiving actor. This approach is taken by BEAM-based languages such as Erlang [6] and Elixir. By accepting the costs of message copying, this technique allows garbage collection to be fully concurrent.

3. *Actor graph transformation with global heap tracing.* By transforming the actor graph into a passive object graph and using a tracing collector for both actors and objects [71, 76], this approach allows actors themselves to be garbage collected. ActorFoundry uses this technique. While this approach requires significant stop-the-world pauses, it is also the first approach to allow actors themselves to be garbage collected.

4. *Snapshots and reference listing with global heap tracing.* This technique, used in SALSA 1.0 [77, 78, 75], is able to collect both local and distributed actors. While the approach is heavyweight and was removed in SALSA

2.0, it provided a mechanism for collecting actors that did not rely on transforming the actor graph to a passive object graph. This approach uses an existing garbage collecting runtime, in this case the JVM, to collect passive objects.

The purpose of *ORCA* is to provide an approach that combines garbage collected actors with local heap tracing while not requiring messages to be copied to the destination heap.

6.2 Ownership and Reference Counting for Actors Algorithm

ORCA extends *MAC* by tracking object references in external maps as well as actor references. Through message receipt, actors may acquire references to objects allocated by other actors, which we term ***foreign objects***, and during execution they may release those references. Therefore, to keep track of this, they acquire and release reference counts. Importantly, such reference counts are not dependent on the shape of the heap, but instead change only when messages are sent and received.

To do so, the operational semantics are altered to perform reference count adjustments for passive objects as well as for actor references when sending and receiving messages. Similarly, actors can acquire and release reference counts for objects owned by other actors. To account for that, *INC* and *DEC* messages are changed to carry a map of address to reference count (that is, an *ExMap*) instead of a scalar reference count.

6.2.1 Local Heap Tracing

The model for *ORCA* does not require a specific method of tracing a local heap. In Pony, a mark-and-don't-sweep collector is used, as described in section A.10. However, alternate mechanisms for collecting a local heap, such as an incremental garbage collector, or even a concurrent collector that examines a single actor's local state and local heap at any given time, could be used.

The operational semantics accounts for this with a high level description of when objects are unreachable by an actor α. If such objects are owned by α, they may be collected if they have a local reference count of zero (or have never had a local reference count). If such objects are owned by some other actor, the foreign reference count held by α can be released.

$$\begin{array}{rcll} \mu & \in & \mathit{Message} & = (\mathit{MethodID} \times \overline{\mathit{Value}}) \\ & & & \mid\ \mathit{INC}(\mathit{ExMap}) \mid \mathit{DEC}(\mathit{ExMap}) \\ & & & \mid\ \mathit{BLK}(\mathit{ActorAddr}, \mathit{RefCount}, \mathit{ExMap}) \\ & & & \mid\ \mathit{UNB}(\mathit{ActorAddr}) \\ & & & \mid\ \mathit{CNF}(\mathit{Token}) \mid \mathit{ACK}(\mathit{ActorAddr}, \mathit{Token}) \end{array}$$

Figure 6.1: Runtime entities for object GC. Elements that are unchanged are greyed out.

6.2.2 Cyclic Passive Object Garbage

Importantly, *ORCA* does not require a cycle detector to collect passive objects. This is because of the *ownership* model, wherein only actors hold reference counts for passive objects. The topology of an object graph, that is, the number of paths by which an object is reachable, does not affect the local or foreign reference count for an object. As such, a cyclic graph of passive objects does not result in reference counts that will never reach zero.

6.2.3 Local Heap Collection

The *ownership* model results in local heaps that are independently traceable and collectable, without synchronisation. Effectively, passive objects for which the owning (i.e. allocating) actor maintains a positive local reference count are added to the reachable set, preventing premature collection.

This local heap collection takes place without synchronisation, even though the algorithm uses message passing to increment and decrement deferred distributed weighted reference counts. Importantly, none of the *ORCA* messages requires a reply, so there is never a situation in which an actor α must wait for another actor α' to make progress before α can itself make progress.

6.3 Formal Model

The formal model for *ORCA* is expressed as an extension of the model for message-based actor collection (*MAC*) presented in chapter 5. Where a rule or a definition is not changed, it is used unmodified.

The only change in the runtime entities is a modification in the information carried by *INC* and *DEC* messages. Instead of a scalar reference count, those messages now carry a mapping of address to reference count. This allows a single message to communicate reference count changes for a collection of objects allocated by some actor, and possibly the actor itself.

6.3.1 Reachability

The rules for reachability remain unchanged from those delineated for *MAC* in section 5.3.2. Because *MAC* must find actor references anywhere in a passive object graph, the same rules can be used to allow *ORCA* to find passive object references in the same graph. Just as *MAC* does not have to determine global liveness for actor collection, *ORCA* does not need to determine global liveness for object collection.

6.3.2 Reference Count Invariant

The reference count invariant for passive objects takes the same form as the invariant for actors, and so can be stated for both as:

$$LRC(\iota) + INC(\iota) - DEC(\iota) = AMC(\iota) + FRC(\iota)$$

This invariant is evaluated in the context of some heap χ. The reference count components for passive objects are effectively the same as for actors, but use the external map exclusively for tracking reference counts, without using the owning actor's local reference count.

Definition 6.1. Reference count invariant components

$$\begin{array}{lcl}
LRC(\omega) & = & \chi(\mathcal{O}(\omega)) \downarrow_8 (\omega) \\
INC(\omega) & = & \sum_{i=1}^{|Q(\chi,\mathcal{O}(\omega))|} \begin{cases} \rho & \text{if } Q(\chi,\mathcal{O}(\omega))_i = INC(\xi) \wedge \xi(\omega) = \rho \\ 0 & \text{otherwise} \end{cases} \\
DEC(\omega) & = & \sum_{i=1}^{|Q(\chi,\mathcal{O}(\omega))|} \begin{cases} \rho & \text{if } Q(\chi,\mathcal{O}(\omega))_i = DEC(\xi) \wedge \xi(\omega) = \rho \\ 0 & \text{otherwise} \end{cases} \\
AMC(\omega) & = & \sum_{\alpha \in dom(\chi)} \sum_{i=1}^{|Q(\chi,\alpha)|} \begin{cases} 1 & \text{if } \omega \in Reach(\chi,\alpha,Q(\chi,\alpha)_i) \\ 0 & \text{otherwise} \end{cases} \\
FRC(\omega) & = & \sum_{\alpha \neq \mathcal{O}(\omega)} \chi(\alpha) \downarrow_8 (\omega)
\end{array}$$

The reference count invariant is maintained in the same way for passive objects as for actors.

The reachability reference count invariant for objects differs from the actor reachability reference count invariant in several ways. When determining which queues should be examined, the owning actor $\alpha = \mathcal{O}(\omega)$ is considered. Rather than an object ω tracking its own local reference count, the owning actor α for ω tracks the local reference count for ω in the external map of α. When examining *INC* and *DEC* messages, external sets are examined rather than a scalar reference count value.

Definition 6.2. Reachability invariant

$\forall \alpha, \omega . [\alpha \neq \mathcal{O}(\omega) \wedge \omega \in \mathit{Reach}(\chi, \alpha, \texttt{ref}) \Rightarrow \chi(\mathcal{O}(\omega)) \downarrow_8 (\omega) > 0 \wedge \chi(\alpha) \downarrow_8 (\omega) > 0]$

6.3.3 Sending a Reference

When a sending a message that can reach some object ω, *ORCA* must perform a reference count adjustment. This is accounted for by changing the definition of *Send* when a passive object is being sent (previously defined in definition 5.4) to perform reference a reference count adjustment, as defined by *Adj* in figure 6.3.

Definition 6.3. Sending a reference

$$\mathit{Send}(\chi, \alpha, \omega) \quad = \quad \mathit{Adj}(\chi, \alpha, \omega, 1)$$

Note that *Adj* is defined such that if α is the owner of ω, then α adds n (here, -1) to its local reference count for ω. On the other hand, if α is not the owner of ω, then α subtracts n from its foreign reference count for ω.

6.3.4 Receiving a Reference

As with sending a reference to an object, when receiving a reference to some object ω, *ORCA* must also perform a reference count adjustment. The definition of *Recv*, previously defined in definition 5.5, is altered in the same way as *Send*, performing a reference count adjustment for passive objects.

Definition 6.4. Receiving a reference

$$\mathit{Recv}(\chi, \alpha, \omega) \quad = \quad \mathit{Adj}(\chi, \alpha, \omega, -1)$$

6.3.5 Actor Local Execution

The operational semantics in figure 6.2 extends the semantics presented in figure 5.4. Rules from figure 5.4 that are not listed remain unchanged from *MAC*.

The Inc and Dec rules are modified such that the message carries an *ExMap*, a mapping of address to reference count, rather than a scalar value. This allows an *INC* or *DEC* message to carry reference count adjustment information for a collection of addresses. In practice, this is some set of passive objects all owned by the same actor α, and possibly α itself. Passive objects owned by other actors, and indeed other actors themselves, will not appear in *INC* or *DEC* messages delivered to α, as they are not owned by α.

Note that both Inc and Dec require the actor to Unblock if they are blocked. This is to ensure that an actor α does not send a *MAC ACK* message to the cycle detector when in fact the cycle detector's view of the reference count of α is out of date. This requirement can be optimised away by keeping

$$\frac{\begin{array}{c}\neg\chi(\alpha)\downarrow_9 \quad INC(\xi)\cdot\overline{\mu} = \chi(\alpha)\downarrow_3 \\ \chi' = Adj(\chi,\alpha,\xi)\end{array}}{\chi,\alpha,\varepsilon \rightsquigarrow \chi'[\alpha\mapsto\overline{\mu}],\alpha,\varepsilon}\ \textsc{Inc} \qquad \frac{\begin{array}{c}\neg\chi(\alpha)\downarrow_9 \quad DEC(\xi)\cdot\overline{\mu} = \chi(\alpha)\downarrow_3 \\ \chi' = Adj(\chi,\alpha,-\xi)\end{array}}{\chi,\alpha,\varepsilon \rightsquigarrow \chi'[\alpha\mapsto\overline{\mu}],\alpha,\varepsilon}\ \textsc{Dec}$$

$$\frac{\begin{array}{c}\neg\chi(\alpha)\downarrow_9 \quad \alpha\neq\alpha' \\ \iota s\subseteq dom(\chi(\alpha)\downarrow_8) \\ \forall\iota\in\iota s.[\mathcal{O}(\iota)=\alpha'\wedge\chi(\alpha)\downarrow_8(\iota)>0] \\ \xi=[\iota\mapsto n\,|\,\iota\in\iota s] \\ \chi' = Adj(\chi[\alpha,\alpha'++INC(\xi)],\alpha,-\xi)\end{array}}{\chi,\alpha\cdot\overline{\varphi},\mathtt{e} \rightsquigarrow \chi',\alpha\cdot\overline{\varphi},\mathtt{e}}\ \textsc{Acquire} \qquad \frac{\begin{array}{c}\neg\chi(\alpha)\downarrow_9 \quad \alpha\neq\alpha' \\ \iota s\subseteq dom(\chi(\alpha)\downarrow_8)\backslash Reach(\chi,\alpha,\mathtt{ref}) \\ \forall\iota\in\iota s.[\mathcal{O}(\iota)=\alpha'\wedge\chi(\alpha)\downarrow_8(\iota)>0] \\ \xi=[\iota\mapsto \chi(\alpha)\downarrow_8(\iota)\,|\,\iota\in\iota s] \\ \chi' = Rel(\chi[\alpha,\alpha'++DEC(\xi)],\alpha,\iota s)\end{array}}{\chi,\alpha,\varepsilon \rightsquigarrow \chi',\alpha,\varepsilon}\ \textsc{Release}$$

$$\frac{\begin{array}{c}\omega\in dom(\chi) \quad \omega\notin Reach(\chi,\alpha,\mathtt{ref}) \\ \mathcal{O}(\omega)=\alpha \quad \chi(\alpha)\downarrow_8(\omega)\not>0\end{array}}{\chi,\alpha\cdot\overline{\varphi},\mathtt{e} \rightsquigarrow \chi\backslash\omega,\alpha\cdot\overline{\varphi},\mathtt{e}}\ \textsc{Collect}$$

Figure 6.2: Operational semantics of object GC

$$\begin{array}{lcl}
Adj(\chi,\alpha,\xi) & = & Adj(Adj(\chi,\alpha,\iota,\xi(\iota)),\alpha,\xi\backslash\iota)\ \mathit{where}\ \iota\in dom(\xi) \\
Adj(\chi,\alpha,\emptyset) & = & \chi \\
Adj(\chi,\alpha,\omega,n) & = & \chi[\alpha\mapsto(\chi(\alpha)\downarrow_1 ...\chi(\alpha)\downarrow_7, \\
 & & \quad \chi(\alpha)\downarrow_8[\omega\mapsto(\chi(\alpha)\downarrow_8(\omega)-n)],\chi(\alpha)\downarrow_9)] \\
 & & \quad \mathit{if}\ \alpha\neq\mathcal{O}(\omega)\wedge\chi(\alpha)\downarrow_8(\omega)>n \\
Adj(\chi,\alpha,\omega,n) & = & \chi[\alpha\mapsto(\chi(\alpha)\downarrow_1 ...\chi(\alpha)\downarrow_7, \\
 & & \quad \chi(\alpha)\downarrow_8[\omega\mapsto(\chi(\alpha)\downarrow_8(\iota)+n)],\chi(\alpha)\downarrow_9)] \\
 & & \quad \mathit{if}\ \alpha=\mathcal{O}(\omega) \\
-\xi & = & [\iota\mapsto-\rho\,|\,\xi(\iota)=\rho] \\
\chi(\alpha)\downarrow_8(\omega)\not>0 & \mathit{iff} & \omega\notin dom(\chi(\alpha)\downarrow_8)\vee\chi(\alpha)\downarrow_8(\omega)=0
\end{array}$$

Figure 6.3: Auxiliary definitions for object GC

a per-actor flag indicating when the actor has altered its reference count due to an *INC* or *DEC* while blocked. When this flag is set, *ACK* messages are suppressed, and the flag is unset when the actor unblocks.

The Acquire and Release rules are modified to allow the executing actor to acquire and release reference counts for passive objects as well as actors. Each rule allows generating a single message (*INC* and *DEC*, respectively) to a single destination. As such, only the destination actor and passive objects owned by the destination actor are considered. As with *MAC*, the executing actor must hold a foreign reference count of 1 or more for an address in order to acquire more reference count weight, and must both hold a foreign reference count weight and not be able to reach the address in order to release that weight.

Note that Acquire uses $-\xi$ rather than ξ, as α wishes to add to, rather than subtract from, its foreign reference counts for ιs. Since *Adj* is defined in terms of sending and receiving, it must be reversed for Acquire.

The Collect rule allows for removal of a passive object ω from the heap when ω is not reachable from the owning actor and the owning actor holds no local reference count weight for ω. This allows local heap collection by the owning actor using any mechanism for local heap tracing.

Note that in the definition of *Adj* in figure 6.3, although the choice of ι in the first case is arbitrary, the definition is deterministic. The order in which adjustments to addresses in ξ are applied does not affect the result.

6.4 Completeness

The presented object garbage collection semantics allow the Collect rule to be executed non-deterministically. This models real world garbage collection scenarios, wherein there can be many heuristics to determine at what point garbage collection should occur. As such, it is possible for a program to run to completion (i.e. achieve actor quiescence) without collecting any passive object garbage. Indeed, some small real world programs written in Pony do exactly this.

However, these semantics allow all globally unreachable passive objects to be collected, while not strictly requiring that such collection takes place. Objects that are not sent in messages are collectable via Collect, and objects that have been sent in messages will eventually be collectable after receiving actors either no longer reference the object and send a *DEC* message via Release, or are themselves collected (as in figure 5.6), which also generates a *DEC* message.

6.5 Robustness

The semantics for *ORCA* are sound and do not have exceptional conditions. However, as with *MAC*, *ORCA* is robust even when failure is introduced.

The robustness conditions are similar to those for *MAC*, described in section 5.5. The failure of a some actor α will cause objects α has allocated to never be collected, even if they become globally unreachable. Similarly, any objects for which α held a foreign reference count will not be collected by their owners, as α will never send *DEC* messages for those objects. However, again similar to *MAC*, such failures impact completeness, but do not result in objects being prematurely collected.

The impact of individual message failure depends on the message type. For *APP* and *DEC* messages, an object ω contained in such a messages will never be collected, as the implicit reference to ω in *APP*, or the explicit reference count in *DEC*, will be lost. However, a lost *INC* message can result in premature object collection, making *ORCA* unsound in the face of individual message loss.

However, the mediating factors described in section 5.5 remain true for *ORCA*: due to the actor-model requirement of guaranteed delivery, individual message failure can be treated as actor failure, reducing the soundness failure to a completeness failure. In Pony, individual actors and messages in the concurrent setting (or on a single node in the distributed setting) do not fail, removing even the completeness impact.

Interestingly, another possible failure mode is integer overflow in reference counts. In the formal model, unbounded integers are used, and this problem does not occur. In the implementation, bounded integers are used, and it might be possible to overflow a reference count, for example if the owner α of ω sends ω to some other actor α' 2^{64} times without α' performing a RELEASE for ω. In this circumstance, the implementation should treat the reference count of ω as infinite, preserving robustness, but not completeness.

6.6 Distributed Object GC

The reduction in the ordering guarantee from causal order in the concurrent setting to pairwise FIFO order in the distributed setting would initially appear to make *ORCA* as described unsound. However, the data-race free type system described in chapter 4 combined with the necessity for copying objects when they are sent between distributed nodes (generally by serialising the object over a network, as distributed nodes do not share memory between nodes) means that *ORCA* remains sound in the distributed setting. As with distributed actor collection, it is important to note that distributed object collection has not yet been implemented in the runtime. As such, this section represents a sketch of a possible implementation, rather than a description of completed work.

6.6.1 Locally Fulfilled Object Capabilities

WF-Async in figure 4.3 requires that behaviour arguments are *Sendable*, i.e. `iso`, `val`, or `tag`. Each of these cases will be examined.

When an actor α_1 on node $\mathcal{N}_1$ sends an `iso` reference to object ω to another actor α_2 on node $\mathcal{N}_2$, the deny properties of `iso` guarantee that no actor other than α_2 can read from or write to ω. As a result, the deserialised copy of ω on $\mathcal{N}_2$ is the authoritative representation of ω. As such, if ω becomes unreachable on $\mathcal{N}_2$, without accounting for reachability on any other node, it is safe for α_2 to collect ω. References on other nodes may survive, but they will be `tag` references, so the collection of ω is sound.

When an actor α_1 on node $\mathcal{N}_1$ sends a `val` reference to object ω to another actor α_2 on node $\mathcal{N}_2$, the deny properties of `val` guarantee that no actor can write to ω. As such, any copy of ω on any node is an authoritative representation of ω. As such, if ω becomes unreachable on any node, without accounting for reachability on any other node, it is safe for ω to be collected on that node. References on other nodes may survive, and will remain authoritative.

Finally, when an actor α_1 on node $\mathcal{N}_1$ sends a `tag` reference to object ω to another actor α_2 on node $\mathcal{N}_2$, that reference cannot be used to read from or write to ω. The fields of ω need not be serialised when sending such a reference. If ω becomes unreachable on $\mathcal{N}_2$, without accounting for reachability on any other node, it is safe for ω to be collected on $\mathcal{N}_2$. References on other nodes may survive, and, if they allow reading from or writing to ω, will remain authoritative.

The combination of the introduction requirement and deny properties results in a distributed system wherein object references are always able to fulfil their interfaces (i.e. provide capabilities allowed by the type system) locally, without communication with other nodes. Asynchronous messages to actors, i.e. behaviour invocations, may result in communication with other nodes, but synchronous calls to actors or objects, i.e. function invocations, will not.

6.6.2 Distributed Object Identity

An object's identity in the concurrent setting, or on a single node in the distributed setting, is the object's address in the heap. However, when an actor α_1 on node $\mathcal{N}_1$ sends a reference to some object ω to α_2 on node $\mathcal{N}_2$, it is not feasible to require that α_2 instantiate ω on $\mathcal{N}_2$ with the same heap address that ω has on $\mathcal{N}_1$. The language must either accept that an object loses its identity when sent in a message to an actor that happens to be executing on another node, which would break the equivalence of the concurrent and distributed operational semantics, or some mechanism for mapping the identity of ω on $\mathcal{N}_1$ to its identity on $\mathcal{N}_2$ must be provided.

It is important that such a mechanism not introduce a new garbage collection problem in the form of garbage collecting assigned global identities. For example, if $\mathcal{N}_2$ were to provide a local mapping of $\mathcal{N}_1$ heap addresses to $\mathcal{N}_2$ heap addresses, $\mathcal{N}_2$ would need to inform all nodes in the distributed system when any address on $\mathcal{N}_2$ that had been sent to any node was garbage collected, in order to prevent out-of-date mappings resulting in a new object at address ω on $\mathcal{N}_1$ having the same identity as the previous object at address ω on $\mathcal{N}_1$ that had been sent to $\mathcal{N}_2$.

Similarly, any system of non-random identities must cope with namespace issues ($\mathcal{N}_1$ must not assign an identity that $\mathcal{N}_2$ might assign). Assigning namespaces is complex in a distributed system due to network splits and joins. That is, if $\mathcal{N}_1$ and $\mathcal{N}_2$ each take on the namespace assignment role after a network split, if the network split is healed, $\mathcal{N}_1$ and $\mathcal{N}_2$ may have assigned the same namespace to different nodes.

One possible approach is to assign a sufficiently long and sufficiently random identifier to any object sent to another node. This provides a probabilistic approach to distributed object identity that requires no garbage collection. The probability of a distributed object identity collision can be approximated by treating the distributed system as if it were executing a birthday attack on itself, which gives a probability p for a number of distributed objects n with a random identifier length of k bits as $p(n, k) = 1 - e^{-n(n-1)/2^{k+1}}$. For example, if the acceptable risk of an object identity collision is $p = 10^{-12}$, a 128-bit object identity allows approximately 2^{45} objects to be created, or a 256-bit object identity allows approximately 2^{109} objects to be created.

To reduce the computational overhead of calculating such random numbers for every object sent to another node, each node $\mathcal{N}$ can create a random identity for itself, $\mathcal{N}_{id}$, when the node joins the distributed system, and use a monotonic counter c to assign object identities of the form $\mathcal{N}_{id} \cdot c$. For the same $p = 10^{-12}$ risk, a 128-bit node identifier and a 64-bit monotonic counter allows approximately 2^{45} nodes and 2^{64} objects per node.

6.7 Using *ORCA* in Other Actor-Model Languages

As an extension of *MAC*, using *ORCA* in another actor-model language would be possible with the same caveats described in section 5.7. Additionally, *ORCA* requires that the *introduction requirement* [3] of the actor model, described in section 2.1, not be violated. If the introduction requirement were to be violated, then it would be possible for some actor α to be able to reach some object ω allocated by some other actor α' without α having either created ω or received ω in a message. As a result, α might not hold a positive foreign reference count for ω, and α' might prematurely collect ω.

The BEAM runtime, being built specifically for as an actor-model runtime, does not violate the introduction requirement, and so languages built on top of it, such as Erlang and Elixir, could use *ORCA*, again with the same caveats regarding causality and non-atomic behaviours as for *MAC*. On the other hand, Akka, being built on JVM languages that allow non-actor concurrency, does not enforce the introduction requirement, and so *ORCA* would be unsound.

Chapter 7

Conclusion and Further Work

This thesis has presented a new programming language, Pony, that is the result of co-designing a data-race free type system and a scalable, lock-free runtime. We have presented several novel type system and runtime elements, and combined them in a working open source compiler and runtime. Specifically, we have detailed a unified actor-model operational semantics for concurrent and distributed execution, a novel type system for data-race freedom based on *reference capabilities*, a novel no-stop-the-world garbage collection algorithm for actors, a novel no-stop-the-world garbage collection algorithm for zero-copy message passing, and a complete runtime implementation.

The operational semantics define a unified semantics for concurrent and distributed execution of an actor-model language that accounts for differences in message causality. The semantics provides an extensible framework for the remainder of the thesis.

Reference capabilities, and the related concepts of viewpoint adaptation, safe-to-write, and aliased and unaliased types, provide a statically data-race free type system in which concepts such as isolation and immutability are derived rather than fundamental. This approach both enables and informs the design of the runtime, for example by allowing safe zero-copy message passing and no-stop-the-world garbage collection.

The *MAC* protocol, message-based actor collection, allows actors themselves to be safely garbage collected fully concurrently. The underlying Conf-Ack protocol, which establishes that a possibly out-of-date view of global state was up-to-date at the time it was generated, is also used for quiescence detection and program termination. In the distributed setting, the Conf-Ack protocol can be extended for distributed actor collection and migration.

The *ORCA* protocol, ownership and reference counting for actors, is an extension of *MAC* that allows for no-stop-the-world garbage collection of passive objects that have been shared by reference across actors with independent heaps. This enables both zero-copy message passing and scalable garbage collection.

The runtime implementation is also described, showing the impact of the type system and the garbage collection protocols on other areas, such as the memory allocator, the scheduler, and asynchronous I/O. The feedback loop between type system and runtime design decisions is shown through the examination of details of the implementation.

7.1 Further Work

There are several avenues of further work that present themselves.

The extensions for distributed computing discussed for *MAC* and *ORCA* are a starting point for a distributed runtime. Other essential elements for a distributed runtime include a machine-specific data type system that can distinguish data that is meaningful when sent to another node from data which is not (for example, a file descriptor is serialisable but meaningless or worse to the receiver). This also requires a notion of locality: that is, machine-specific data can be sent to an actor that is on the same node, but that actor must *remain* on the same node in the future.

As in all distributed computation projects, a strong approach to node failure must be adopted, whether in the form of Erlang-like monitors and supervision, or EROS-like orthogonal persistence, or a combination of both. Particularly interesting is the possibility of using non-volatile memory to achieve object persistence. This raises significant issues, such as how to reconstitute objects from non-volatile memory in the presence of partially completed operations or corrupted data.

Similarly difficult is the development of heuristics for distributed work-stealing that are useful for many classes of workload. For example, in the distributed setting, an actor with a ten gigabyte working would be an unlikely candidate for migration to another node due to the communication cost that would be incurred. However, migrating such an actor might alleviate memory pressure on a node and improve performance once the communication cost had been paid.

An interesting open question is the nature of reflection in a capabilities-secure environment. Mirror-based reflection [13] gives a solid foundation, but does not address capabilities-security. Reflection in AmbientTalk [53] supports capabilities-security in a dynamically typed language, and reflection in Wyvern [74] applies similar principles to achieve capability-secure reflection in a statically typed language. In the context of Pony, additional issues remain to be addressed, such as introspection over mirrors of objects that contain isolated state.

Adding live programming in the form of a REPL would involve revisiting the compiler's approach to virtual dispatch and pattern matching on structural types, as discussed in section 2.3.3. A possible approach is to use perfect hashing [23] for both dispatch and subtype tests.

Any technique that enabled a Pony REPL could also be extended to enable hot code loading, wherein not only can types be added to a running system, but the definition of a type can be replaced at runtime. This is used in Erlang to allow some programs to be updated while running, under some conditions. There is significant work to be done here in the form of validating whether the replacement definition is acceptable in a statically typed setting, and providing a transformation function from the previous definition to the new definition that can be applied to a running concurrent system.

At the type system level, continued formalisation is needed, particularly in the area of pattern matching over generic types. New type system features may also be possible, such as an annotation ensuring that an identifier will not be consumed, possibly combined with explicit region annotation, which would allow more instances of isolated references being temporarily usable as non-isolated mutable references while maintaining isolation.

At the runtime level, some features have been implemented in the runtime that require formalisation, such as generalised actor-model backpressure. Other features have been formalised, but require implementation, such as compile-time expression evaluation and simple value-dependent types.

As with all programming languages, tooling support is as important as language and runtime features. Pony can currently be debugged using `gdb` and `lldb`, but improved `DWARF` debugging information support and a Pony expression parser would improve the experience. Additionally, there are debugging features that are specific to actor-model programming, such as stepping through asynchronous message passing rather than by instruction, that would be worth investigating [42].

7.2 Benchmarking

Benchmarking a new programming language is a daunting task. When making changes to an existing language, whether at the language feature level or in the runtime, the impact of those changes can be benchmarked against the unmodified language using any existing corpus of programs. However, a new programming language suffers from two problems: initially, there is no existing corpus of programs to test, and, more importantly, even once such a corpus exists, it is rarely possible to directly compare the performance of such programs

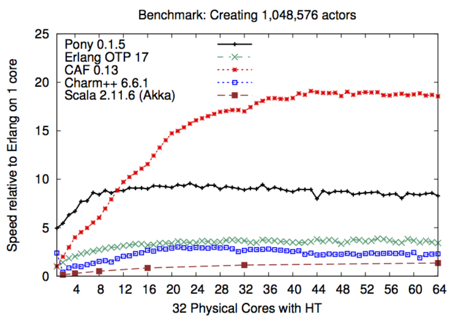


Figure 7.1: CAF actor creation benchmark

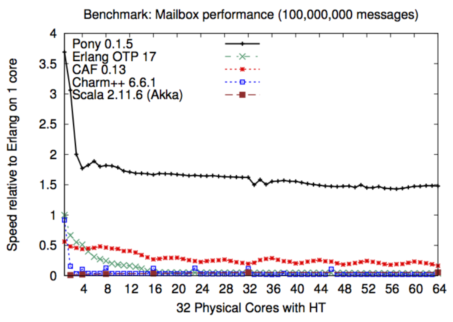


Figure 7.2: CAF actor mailbox benchmark

against the performance of programs written in another language. Similarly, implementing alternate versions of programs written in existing languages in the new language may not only be infeasible due to time and effort concerns, but often results in fundamentally rethinking how a program is written, due to differing language semantics. The end result, even when this is attempted, is rarely comparable to the original program.

That said, an early version of Pony was benchmarked against other actor-model languages using the CAF benchmark suite [2], which provides sample microbenchmarking workloads that attempt to model aspects of real-world program behaviour. Results are the average of 100 runs, normalised against Erlang performance on a single core such that performance improvement linear to core count would be shown as a straight line sloping up. We chose to normalise against Erlang because it is a mature system with consistent performance across core counts, with little jitter.

In figure 7.1, we show actor creation performance when creating an interconnected tree of actors that cannot be collected until the program completes, which is the worst case for Pony. Here, we are garbage collecting actors themselves as well as objects, but still outperforms existing systems other than CAF, which is neither garbage collected nor data-race free. In figure 7.2, we show

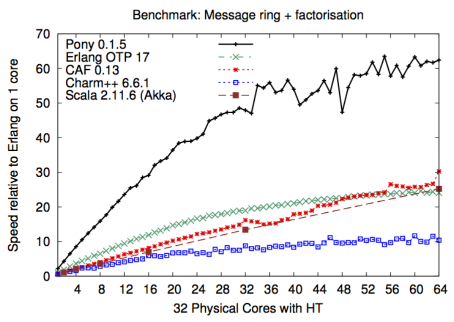


Figure 7.3: CAF actor mixed workload benchmark

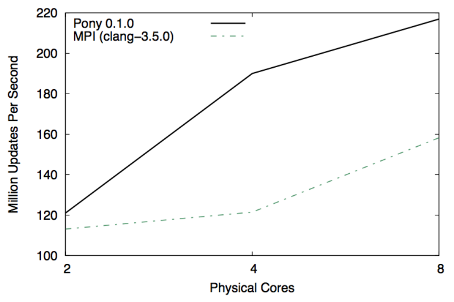


Figure 7.4: HPCC RandomAccess benchmark

performance of a highly contended mailbox, where additional cores tend to degrade performance. In figure 7.3, we show performance of a mixed case, where a heavy message load is combined with brute force factorisation of large integers.

In figure 7.4, we show a benchmark that is not tailored for actors: we take the HPCC RandomAccess benchmark from high-performance computing [1], which tests random access memory subsystem performance, and demonstrate that our implementation is significantly faster than the highly optimised MPI implementation. We show only power-of-two core counts because the MPI implementation is optimised for this case.

While all benchmarking is to some degree snake oil, we chose these benchmarks because a) they were designed by others, b) they are hopefully representative of some common actor-model programming idioms, and c) they have optimised implementations in existing languages and frameworks provided by programmers expert with those tools.

One possible approach to future benchmarking is to use the Computer Language Benchmark Game [27] as a corpus of cross-language programs. While the available CLBG benchmarks may not be representative of real-world performance, they have the advantage of existing. By applying rigorous statistical methods [36] and correcting for warmup vs. steady state performance [9] it may be possible to extract a useful cross-language comparison.

Another approach is to carefully select benchmarks that use only language features common amongst the set of tested languages [44], allowing effectively identical cross-language microbenchmarks. While this approach does not test overall language performance, it provides insight into the implementation efficiency of specific language features, which is not possible using the CLBG approach.

7.3 Continuing Efforts

Finally, we would like to quote Mark S. Miller: “Any dissertation necessarily reflects the state of an ongoing project at a particular point in time.” [49] Work on Pony is continuing, both academically and in the open source community, and while it is our hope that this thesis is valuable to the reader, it is also our hope that it quickly becomes outdated.

Appendix A

Runtime Implementation

The implementation of the ideas presented in this thesis, in the form of the Pony compiler and the Pony runtime, represents a significant amount of software engineering as well as a certain amount of systems research. The purpose of this appendix is to serve as a guide to some of the subtler aspects of the implementation of the Pony runtime library. The elements of the runtime explained here are those most influenced by the type system and garbage collection work that represent the bulk of this thesis. The dissection of the subtler aspects of the Pony compiler is left as an exercise for the reader.

A.1 Component Architecture

The Pony runtime consists of several core components. Figure A.1 shows the structure of those components. At the bottom is the pool allocator, which handles memory allocation and deallocation. On top of this is built the Single-Producer Multiple-Consumer (SPMC) queue, the Multiple-Producer Single-Consumer (MPSC) queue, and the page map. SPMC and MPSC queues are used to build the scheduler, an MPSC queue is used to build the asynchronous I/O handler, and the page map is used to build per-actor heaps. Another MPSC queue and the heap are used to build the actors themselves, which depend on both the scheduler and the asynchronous I/O handler. Tracing garbage collection (GC), sharing GC, and actor GC are built on top of actors, heaps, the page map, and the pool allocator.

Sharing GC, as described in chapter 6, uses tracing GC during message send and receive. Actor GC, as described in chapter 5, relies on sharing GC in order to track implicit references from some actor α_1 to some actor α_2 that arise due to α_1 having a reference to some object ω that is owned by α_2, as shown in section 5.3.2. In addition, tracing GC invokes sharing GC when objects owned by other actors are encountered during the trace.

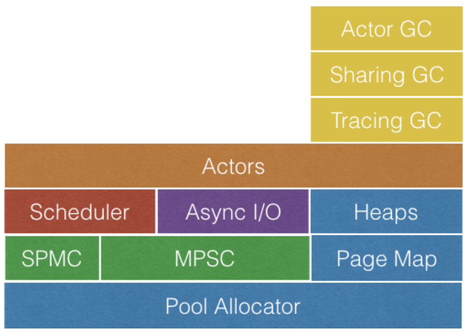


Figure A.1: Runtime architecture

Every component's design is influenced by the type system. This influence is particularly felt in the garbage collection components, but the impact is felt at every level, including the memory allocator.

A.2 Naming Conventions

The Pony runtime is implemented in C, and maintains a C API and ABI in order to be usable from programming languages other than Pony. To avoid conflicts with symbols exported from other libraries, functions and types intended to be used at the language level are prefixed with `pony_`, while functions intended to be used only internally by the runtime itself are prefixed with `ponyint_`. Types that are only used internally have no prefix, since they do not appear in the symbol table or any publicly accessible header, and so do not cause conflicts.

A.3 Pool Allocator

General purpose memory allocation is a complex problem. Manual memory management allocators have seen significant research in malloc-like allocators that scale better as core counts increase, including `tcmalloc`[25], `jemalloc`[24], the `Intel TBB` allocator[58], and most recently `scalloc`[5]. This category of memory allocator faces several related problems: finding free memory large enough to fulfil the requested allocation size without wasting excessive memory, coalescing freed memory to allow larger allocations to reuse memory from multiple smaller freed memory blocks, and coordinating allocation across threads. Similar problems emerge with automatic memory management allocators, with the additional complication that the garbage collector must interact with the allocator, and that a compacting garbage collector may move objects in memory.

The Pony memory allocator is divided into three parts: an underlying pool allocator, a page map, and a heap implementation built on top. We'll begin by examining the pool allocator. It is important to note that the pool allocator is not directly used for language-level allocation. That is, when a new object is allocated in Pony, it is allocated from the currently executing actor's heap, rather than directly from the pool allocator. The pool allocator is used for runtime data structures, including actor heaps themselves.

To reduce contention, the pool allocator operates as a thread-local allocator. When we examine the scheduler, we'll see that this results in each core having a pool allocator that is pinned to that core. This pool allocator is responsible for pulling new pages from the kernel when necessary, managing size classed free lists, and making free lists available to other thread-local pool allocators when possible.

When a pool allocator is asked to allocate memory, it tries three sources, in order:

1. The thread-local size classed free list for amount of memory requested, rounded to the next higher size class.

2. The global size classed list of free lists for that size class, which is used to prevent producer-consumer allocation starvation.

3. The thread-local free block.

4. The operating system, by allocating new pages.

A.3.1 Size Classes

Size classes are used to simplify free list management. This approach segregates free lists by the size of the allocation, and limits the available allocation sizes. The implementation currently uses power-of-two sized classes ranging from 2^5 to 2^{20} bytes, but these values are easily tuneable. Allocations exceeding the largest size class begin with the thread-local free block.

Such segregated free lists have a cost, in that allocations sizes are rounded up to a size class. For example, using power-of-two size classes, a 65 byte allocation request will instead allocate 128 bytes. However, in return, allocations do not require that memory in the allocated block be used to indicate the allocated size, and allocations are always cache line aligned.

In addition, no free list searching is required on allocation. Using the head of the free list for a size class approximates a best-fit search of all free memory, which improves memory utilisation and reduces fragmentation.

```c
typedef struct pool_item_t {
  struct pool_item_t* next;
} pool_item_t;
```

Figure A.2: A thread-local free list node

```c
typedef struct pool_central_t {
  pool_item_t* next;
  uintptr_t length;
  struct pool_central_t* central;
} pool_central_t;
```

Figure A.3: A global list of free lists node

A.3.2 Thread-local Size Classed Free List

Each pool allocator keeps a linked list of free memory for each size class. Initially, these free lists are empty, which is encoded as a `null` pointer.

When the pool allocator is told to free memory within a size class, it prepends that memory to the appropriate free list. The memory being freed is reused as the linked list node, so that there is no memory overhead associated with free memory, as shown in figure A.2.

The memory is prepended, following a last-in first-out strategy, in order to improve cache locality: if memory of that size class is quickly allocated and used again, it is more likely that the memory remains in cache.

When memory is freed to a thread-local free list, the provenance of the memory is not examined. Any pool allocator can place memory allocated by any other pool allocator on its thread-local free list for that size class. As a result, it is not necessary to perform any checks as to the origin of memory when it is freed. This differs from *arena* based allocators, such as `jemalloc`[24], that must return free memory to the allocating arena. For the Pony runtime, this behaviour is critical, as it allows both runtime data structures and heap chunks to be efficiently freed regardless of which thread an actor executes on. On the other hand, this approach makes coalescing free memory, either to reuse as a different size class or to return to the operating system, more difficult. The pool allocator relies on a stable state for a program rather than coalescing, which is problematic for some applications.

A.3.3 Global List of Size Classed Free Lists

The system as a whole also keeps a single global list of free lists for each size class. This is to avoid memory starvation in a producer-consumer scenario where one scheduler thread is consistently allocating memory and another scheduler thread is consistently freeing it.

When a thread-local free list exceeds a certain size, defaulting to 2^{20} bytes, that free list is pushed to a global list of free lists. The maximum size of a thread-local free list is a global compile-time constant, rather than being adaptive. It is possible that an adaptive approach could yield additional performance benefits, at the cost of evaluating some heuristic.

The first linked list node in the thread-local free list is reinterpreted as the first node in a global list of lists. This involves reusing available space in the node *after* the pointer to the next node in the thread-local free list to record both the length of the current list and a pointer to the next global list, as shown in figure A.3. Note that a `pool_central_t` maintains the same initial structure as a `pool_item_t`.

To make this possible, the minimum allocation size must be sufficient to hold all three values, i.e. 24 bytes in a 64-bit runtime or 12 bytes in a 32-bit runtime. The minimum allocation size is a compile-time constant that defaults to 32 bytes.

An atomic compare-and-swap loop is used to change the global list of free lists pointer for the size class being pushed, so that any number of pool allocators can safely attempt to push a free list to the global list of lists simultaneously.

Similarly, when a pool allocator's thread-local free list is empty, it examines the global list of free lists. Crucially, if that global list is empty, which is encoded as a `null` pointer, no atomic operation is required, and the pool allocator tries its next source of free memory. If the global list of free lists contains one or more lists, an atomic compare-and-swap loop is used to remove one list from global list. That list is then used as the thread-local free list, allowing memory to be returned immediately.

A.3.4 Thread-local Free Block

A pool allocator also keeps a thread-local free block. This is a list of free memory, insert-sorted by size, that is initially empty. If no size classed memory is available, the first free block in this list large enough to fulfil the allocation is used. If a block is available, it is fragmented into the requested size and any remaining memory, with any remaining memory being insert-sorted back into the free block.

This approach allows large allocations to be reused for future large allocations, or fragmented into future size classed allocations. It also allows freed large allocations to coexist with new pages pulled from the operating system.

A.3.5 New Pages

If the thread-local and global size classed pools are empty, and the free block is either empty or contains no memory block large enough to fulfil the allocation request, the pool allocator uses an underlying system call (`mmap` on UNIX-like operating systems, `VirtualAlloc` on Windows) to request a number of pages mapped as a contiguous chunk of address space. Some empirical testing with various work loads has lead to choosing a 128 megabyte chunk in a 64-bit address space and a 16 megabyte chunk in a 32-bit address space as the default. These values can be easily changed in the runtime.

This address space is mapped using huge pages, where available, to minimise the number of address translations, reducing pressure on the translation lookaside buffer (TLB). This also reduces the TLB lookup pressure on the L2 cache, leaving more cache for application data. However, this address space is not pre-faulted. That is, no data is written to the mapped address space. This prevents unused pages from being mapped to physical memory, and significantly reduces allocation-time jitter, particularly on NUMA machines.

The mapped address space is then fragmented into the requested allocation size and any remaining size, with the remaining memory kept in the thread-local free block for future allocations.

A.3.6 Fragmentation

The pool allocator as currently implemented has no mechanism for coalescing free memory. As a result, small allocations can result in fragmentation, such that free memory is available, but a request for a single large contiguous address space fails.

This deficiency is addressed in the per-actor heap, rather than in the pool allocator, as explained in section A.5.5. This allows the pool allocator itself to avoid jitter caused by attempting to coalesce free memory.

A key area of future work is to add a coalescing mechanism for pool allocator fragmentation that does not overly adversely affect common cases that do not require coalescing, such as repeatedly allocating and freeing buffers used for I/O operations. In particular, the page map (see section A.4) can be extended to cheaply track free contiguous allocations. The challenge is to cheaply coalesce these even when they are allocated and freed by different threads.

Allocations that exceed the size class limit of the pool allocator are added to the insert-sorted list of free blocks for the local pool allocator. For these very large allocations, the pool allocator performs no better than existing allocators.

A.3.7 Role of the Type System

The type system has an indirect effect on the design of the pool allocator. The use of size classed allocations that do not record their size within the allocated memory is made possible because the program always knows the size of any memory being freed. This approach, which does not conform to the `malloc` interface, could be used seamlessly in any language that knows allocation sizes, either statically or dynamically. It can also be used in languages where sizes are not known if the program explicitly tracks allocation sizes. For example, the Pony compiler, although implemented in `C`, uses the pool allocator.

A.4 Page Map

The runtime keeps a radix tree (two-level on 32-bit architectures, three-level on 64-bit architectures). Page map levels are described in figure A.4. The bottom level of the radix tree refers to 2^{10} byte regions, allowing the page map to associate a pointer to a chunk descriptor with each one. These regions are the *pages* for the Pony runtime, although they are smaller than either the standard (4 kilobyte) or huge (2 megabyte) pages used by some CPUs, such as `ARM64` and `x86`. Using a radix tree allows the page map to associate a pointer with every 2^{10} byte region with a total memory overhead of 0.39% on a 32-bit architecture, or 0.78% on a 64-bit architecture.

The alignment of the memory areas managed by chunks, as described in section A.5, is exploited to give a simple mask-based lookup to find the chunk descriptor for any address. This is used during garbage collection to find the chunk descriptor for an object, as detailed in sections A.8 and A.9.

A page map is used rather than embedding a chunk descriptor in allocated memory as an allocation header. Doing so significantly reduces the memory overhead for small allocations, as described in section A.5.1. In addition, it allows the runtime to allocate and garbage collect objects that conform to `C/C++` layout conventions on a particular platform, which is important for an efficient foreign function interface. Pony also follows these platform specific layout conventions.

The page map is a global rather than a thread-local data structure, with the root of the page map being a single global pointer. Because it can be written to for any allocation, and is read from during garbage collection, thread contention must be minimised.

In figure A.5, we see that page map reads require no thread coordination and no atomic operations. In figure A.6, we see that writing to the page map may result in a non-looping atomic compare-and-swap. This is used to extend the

```
typedef struct pagemap_level_t {
  int shift;
  int mask;
  size_t size;
  size_t size_index;
} pagemap_level_t;
```

Figure A.4: Page map levels

```
void* pagemap_get(const void* m) {
  void** v = root;
  for(int i = 0; i < PAGEMAP_LEVELS; i++) {
    if(v == NULL)
      return NULL;
    uintptr_t ix = ((uintptr_t)m >> level[i].shift)
      & level[i].mask;
    v = (void**)v[ix];
  }
  return v;
}
```

Figure A.5: Reading from the page map

page map. This results in worst case behaviour of three atomic instructions on write, which happens only when the page map is entirely empty. The common case involves no atomic instructions and no thread coordination.

A.4.1 Role of the Type System

The type system does not allow the allocating actor for an object to be known statically. As a result, when the allocating actor is needed, for example during garbage collection as detailed in section A.9.3, it must be determined dynamically. The page map provides an efficient way to do this.

The page map points to a chunk descriptor, which in turn points to the allocating actor, in order to allow the size class for an arbitrary pointer to memory to be determined. This allows internal pointers to be easily distinguished from external pointers, and allows the runtime to efficiently determine the external pointer for any internal pointer. This is used to distinguish between deep and shallow marking, as shown in section A.8.1.

```
void pagemap_set(const void* m, void* v) {
  void*** pv = &root;
  void* p;
  for(int i = 0; i < PAGEMAP_LEVELS; i++) {
    if(*pv == NULL)
    {
      p = pool_alloc(level[i].size_index);
      memset(p, 0, level[i].size);
      void** prev = NULL;
      if(!_atomic_cas(pv, &prev, p))
      {
        pool_free(level[i].size_index, p);
        *pv = prev;
      }
    }
    uintptr_t ix = ((uintptr_t)m >> level[i].shift)
      & level[i].mask;
    pv = (void***)&((*pv)[ix]);
  }
  *pv = (void**)v;
}
```

Figure A.6: Writing to the page map

A.5 Per-Actor Heaps

All memory allocated by the runtime is handled by the thread-local pool allocator. This includes all runtime data structures that are not visible to the program, such as message queue nodes, scheduler queue nodes, and garbage collection management data, as well as memory allocated for actors themselves. However, when an actor allocates objects, it does so via a per-actor heap, rather than by using the pool allocator directly.

Each actor has its own heap, composed of lists of size classed chunks, a list of oversized chunks, and information used to determine if the heap may be in need of garbage collection, as shown in figure A.7. Each standard chunk is a 2^{10} byte area of contiguous memory that is used to allocate multiple objects, each of the same size. An oversized chunk is an area of contiguous memory larger than 2^{10} bytes that is used to allocate a single large object. In both cases, an allocation may represent an array of objects as well as a single object. An allocation is a *small allocation* if it is less than the standard chunk size, otherwise it is a *large allocation*.

A chunk descriptor is a small (24 bytes on a 32-bit architecture, 40 bytes on a 64-bit architecture) structure that stores the owning actor for chunk (i.e. the actor whose heap the chunk is associated with), a base memory address, a size

```
typedef struct heap_t {
  chunk_t* small_free[HEAP_SIZECLASSES];
  chunk_t* small_full[HEAP_SIZECLASSES];
  chunk_t* large;
  size_t used;
  size_t next_gc;
} heap_t;
```

Figure A.7: A heap

```
typedef struct chunk_t {
  // immutable
  pony_actor_t* actor;
  char* m;
  size_t size;
  // mutable
  uint32_t slots;
  uint32_t shallow;
  struct chunk_t* next;
} chunk_t;
```

Figure A.8: A chunk descriptor

class or byte size, and garbage collection information, as shown in figure A.8. Chunk descriptors are used for both small allocations and large allocations.

The garbage collection information is contained in the `slots` and `shallow` fields. The use of these fields as bitmaps of free memory, and the need for separate fields, will be discussed in section A.8.1.

A.5.1 Small Allocation

A heap keeps two singly-linked lists of chunks for each small allocation size class. The first is a list of chunks with memory available for allocation, and the second is a list of chunks with no available memory. Initially, both lists are empty for every size class.

When the program requests memory less than the configured small allocation size (defaulting to 2^{10} bytes), the heap uses size classed memory to fulfil the allocation. The amount of requested memory is turned into a size class using a bit shift and a table lookup on a small (16-byte) static table, as in figure A.9. The resulting size class index is then used to determine the correct linked list of chunks with free memory, also in figure A.9.

The `small_free` and `small_full` lists are segregated to ensure that the heap never has to walk a linked list when allocating memory. To do so, an

```c
static const uint8_t sizeclass_table[] = {
  0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4
};

uint32_t heap_index(size_t size) {
  return sizeclass_table[(size - 1) >> HEAP_MINBITS];
}
```

Figure A.9: Determining a size class

important invariant is maintained:

Definition A.1. Chunk validity

- All chunks allocated by this heap with free space are in the `small_free` list for their size class.
- All chunks allocated by this heap without free space are in the `small_full` list for their size class.

As a result, when an allocation for some size class is requested, it will come from the first element of the `small_free` list, if there is one. Otherwise, a new chunk will be allocated and the allocation will come from this new chunk.

This invariant is important for performance. Without it, small allocation would be an $O(n)$ operation, with n being the amount of memory allocated in a size class. By keeping segregated lists, small allocation is $O(1)$, with at most one pointer dereference at the heap level.

A.5.2 Small Chunk Bitmaps

Each chunk keeps a bitmap of free memory addresses in the memory it manages. This is called `slots` in figure A.8. The default configuration uses a single 32-bit integer to map a 2^{10} byte memory area. This gives enough bits to describe free memory for a 32 objects of the minimum allocation size of 32 bytes, which is size class `0`. Similarly, a chunk can hold 16 objects with a 64 byte allocation size, which is size class `1`.

The bitmap uses set bits to indicate addresses in the memory area that are both free and valid for the size class, and unset bits to indicate addresses that are either in use or invalid for the size class. For example, using the default configuration, a chunk managing a 32 byte size class has an empty bitmap that consists of all bits being set, whereas a chunk managing a 64 byte size class has an empty bitmap consisting of every other bit being set.

To find a free and valid address, the runtime finds the first set bit in the bitmap (a cheap instruction on a modern processor), unsets the bit in the bitmap,

```
uint32_t bit = __pony_ffs(chunk->slots) - 1;
chunk->slots &= ~(1 << bit);
void* m = chunk->m + (bit << HEAP_MINBITS);
```

Figure A.10: Finding free small allocation memory

and returns the base memory address offset by the bit position multiplied by the minimum allocation size, as in figure A.10. Note that `__pony_ffs` ("find-first-set") is a cross-platform macro for the platform specific code that finds the first set bit. Also note that no conditional branches are required.

A.5.3 Allocating on an Existing Chunk

When a small allocation is performed and there is a non-empty linked list of chunks with free memory for that size class, i.e. `heap->small_free[sizeclass] != NULL`, the allocation can always be fulfilled using memory available in the first element of that linked list. That is, there is guaranteed to be at least one bit set in the chunk's bitmap.

If allocation results in all memory in the chunk being exhausted (i.e. there was only one bit set before allocation, and so no bits set after allocation), the chunk is moved from the linked list of chunks with available memory to the linked list of chunks with no available memory. This is a cheap operation: it involves popping the head of a singly-linked list and pushing it to the head of another singly-linked list. This maintains the chunk validity invariant in definition A.1.

A.5.4 Allocating on a New Chunk

If there are no non-empty chunks for a given size class, the linked list head will be `NULL`. This indicates that a new chunk is required, without a pointer dereference.

At this point, two pool allocations are required. The first is for a chunk descriptor, and the second is for the chunk itself, i.e. the memory area to be managed by the new chunk descriptor. These are allocated separately in order to maintain address alignment for memory areas (a requirement for the page map, as detailed in section A.4) while also avoiding wasting memory. To illustrate the advantage of this approach, a 2^{10} byte memory area used for a 2^8 byte size class has space for four objects. If the chunk descriptor were embedded in the chunk memory, it would use the space one object would otherwise occupy, which in this case would require using 2^8 bytes to store the chunk descriptor. With split allocation, a chunk descriptor is a fixed size.

An additional concern is to keep chunk descriptors in separate cache lines from program data, and, where efficient, on separate operating system pages. This is important to reduce cache pollution and page faults during garbage collection, as detailed in section A.8. The pool allocator ensures that chunk descriptors are cache-line aligned, and will prefer placing different size classes on separate operating system pages where practicable, accomplishing both goals.

The new chunk is initialised as an empty chunk and registered with the page map (c.f. section A.4). As an optimisation, the first valid bit in `slots` is initialised as unset, so that the base memory address for the chunk can be used to fulfil the allocation without first checking for the first set bit.

A.5.5 Small Allocation Fragmentation

A major concern in memory allocators is avoiding fragmentation. This problem has two manifestations: internal fragmentation and external fragmentation.

Internal fragmentation occurs when an allocator must return more memory than is requested by the application in order to fulfil an allocation. This can have two causes: headers related to the allocation itself (as opposed to object headers in the programming language) and the need to align memory.

Because the type system is integrated with the runtime, the Pony allocator does not require memory allocation headers, and so avoids the first problem on a per-object basis. However, a chunk descriptor is still required for each memory area. This descriptor is allocated separately from the memory area to avoid internal fragmentation, although it still imposes a worst case memory size overhead of 3.125% (32-bit architecture) or 6.25% (64-bit architecture). This overhead can be configured by using a chunk size larger than 2^{10} bytes. The trade off is that an actor reserves memory for future small allocations in these block sizes. If the actor does not perform those future allocations, some fraction of that space is wasted, although this is always less than the memory area size multiplied by the number of size classes (i.e. it is $O(1)$ rather than $O(n)$ on allocation count).

Internal fragmentation due to memory alignment is sometimes a net performance benefit, due to cache line alignment and machine word alignment (e.g. 64-bit floating point or SSE data alignment). The Pony runtime aligns all small allocations on the size of the allocation to take best advantage of this. This creates a worst case internal fragmentation cost approaching 50%. For example, if all allocated objects in a program were 65 bytes, they would all be fulfilled with 128 byte allocations. Some allocators, `tcmalloc` *inter alia*, use more finely grained size classes to reduce this form of internal fragmentation. The trade off is that more data structure overhead is required to keep track of the size classes themselves. For a thread-specific allocator such as `tcmalloc`, this overhead is only multiplied by the number of threads in a program (where a large number is on the order of thousands), whereas for the Pony runtime, the overhead would be multiplied by the number of actors in a program (where a large number is on the order of tens of millions).

External fragmentation occurs when free memory is interspersed with allocated memory, resulting in a situation where sufficient memory is available to fulfil an allocation request, but that memory is non-contiguous, and the allocation fails. In a garbage collected language, this can be addressed with a compacting collector. Pony does not use a compacting collector, as explained in section A.8. Instead, external fragmentation is addressed in two ways: chunk bitmap management and pool allocator reuse.

By using a chunk bitmap, free memory of a given size class is reused when a new allocation occurs, without the need to search through or coalesce a free list. This addresses the most common cause of external fragmentation: repeated allocation and freeing of objects of a finite number of sizes in an interleaved manner. Essentially, space in a chunk is back-filled when new objects of that size class are allocated.

In addition, when all of the memory in a chunk is marked as free, that chunk descriptor and the memory area it manages are returned to the pool allocator. This allows a chunk to be reused for a different size class. As a result, repeated small allocations of any size are always defragmented, even if those allocations are temporally disparate or have an unpredictable layout. The worst case behaviour for a given size class is to free all but one object per chunk, preventing the chunk itself from being returned to the pool, without ever allocating additional memory in that size class. However, even this worst case is strictly bounded: any additional allocation in that size class will use memory available in the existing chunks.

A.5.6 Large Allocation

When a program requests contiguous memory larger than the maximum size that can be handled by a small allocation, by default 2^{10} bytes, the heap performs a large allocation. This includes allocating a chunk descriptor and a single contiguous space for the full allocation, both using the underlying pool allocator. The requested contiguous memory area size is rounded up to be a multiple of the chunk size. This is done to allow large allocations that are later freed to be efficiently subdivided into chunks by the pool allocator if the pool allocator determines that doing so would be more efficient than pulling new pages from the operating system to fulfil an allocation request.

Aligning large allocations in this way also allows the memory used by a large allocation to be efficiently represented in the page map, as described in section A.4.

Unlike the chunk descriptors for small allocations, there is no list of chunk descriptors for large allocations that contain free space, as each large allocation is either entirely in-use or entirely free. When garbage collection determines that a large allocation is no longer required, both the chunk descriptor and the contiguous memory area is describes are returned to the pool allocator.

Large allocations occur primarily for contiguous arrays, including strings. It is unusual for a single object to require more than 2^{10} bytes of storage.

A.5.7 Large Allocation Fragmentation

Since the contiguous memory areas required for large allocations are handled by the pool allocator, and each chunk descriptor represents a single allocation, there is no chunk bitmap and no cross-size defragmentation performed by the heap. Instead, fragmentation is handled by the pool allocator.

A.5.8 Role of the Type System

The type system distinguishes between active objects, in the form of actors, and passive objects. This allows the runtime to distinguish possibly concurrent allocations (i.e. those occurring in different actors) from allocations guaranteed to be sequential (i.e. those occurring in the same actor). Per-actor heaps allow both allocation and garbage collection without contention, even when memory is shared by pointer across actors, as detailed in section A.9.

A.6 MPSC Queues

Each actor has its own message queue, and all messages to an actor are delivered through that single queue. A message queue can therefore be a bottleneck in a program, particularly if a program uses a pattern whereby many actors send messages to a single receiving actor.

In the Pony runtime, an actor's message queue is implemented as a multi-producer single-consumer (MPSC) queue. Any actor can push a message on to a queue, but only the owning actor can pop a message off the queue. The point of contention is the push, where many actors could simultaneously push messages to the same queue.

In figure A.11, we see that pushing a message on to a queue is accomplished with a single atomic exchange operation (the atomic stores are logically atomic, but on most common architectures, e.g. X86, X86-64, AArch32, AArch64, et

```
bool ponyint_messageq_push(messageq_t* q, pony_msg_t* m)
    {
  atomic_store_explicit(&m->next, NULL,
      memory_order_relaxed);
  pony_msg_t* prev = atomic_exchange_explicit(&q->head, m
      , memory_order_relaxed);
  bool was_empty = ((uintptr_t)prev & 1) != 0;
  prev = (pony_msg_t*)((uintptr_t)prev & ~(uintptr_t)1);
  atomic_store_explicit(&prev->next, m,
      memory_order_release);
  return was_empty;
}
```

Figure A.11: Pushing to a message queue

```
pony_msg_t* ponyint_messageq_pop(messageq_t* q) {
  pony_msg_t* tail = q->tail;
  pony_msg_t* next = atomic_load_explicit(&tail->next,
      memory_order_acquire);
  if(next != NULL) {
    q->tail = next;
    ponyint_pool_free(tail->index, tail);
  }
  return next;
}
```

Figure A.12: Popping from a message queue

al., all stores are inherently atomic). Critically, there is no compare-and-swap loop or other waiting construct. As a result, sending a message is lock-free and wait-free. The use of the low bit for empty queue detection will be discussed in section A.6.2.

In figure A.12, we see that popping a message from a queue is accomplished with zero atomic operations (the atomic load is logically atomic, but like the store when pushing, results in a normal load instruction on most common architectures), and without any loop or wait. This means that popping a message in order to begin a behaviour is lock-free, wait-free, and uses no atomic operations. The memory management of queue nodes is discussed in section A.6.3.

In addition to being a safe MPSC queue, an actor's queue must provide two additional features: it must be unbounded, and it must atomically detect when a message is pushed on to an empty queue.

Although there has been extensive research on queues, usefully sumarized in [32], the MPSC queue in the Pony runtime has more in common with a CLH queue lock [43]. While it is asynchronous and non-blocking, it uses a similar mechanism for maintaining a queue. Unlike the classic Michael and Scott queue [45], the Pony queue does not require atomic compare-and-swap operations, and so does not suffer from performance scalability problems due to CAS failures on the x86/x64 architecture [52].

A.6.1 Unbounded Message Queues

Unbounded message queues are required to prevent both deadlock and incorrectly reporting resource exhaustion. If an actor's queue had any bound, even a large one, then some actor α_1 that sent a message to an actor α_2, where α_2 had a full queue, would have to handle the inability to send a message. There are two strategies for this: α_1 could either block until α_2's queue is not full, or fail to send the message (a retry and time-out mechanism is simply a combination of these two approaches).

- If α_1 adopts the strategy of blocking while attempting to push a message on to α_2's queue, then the system could deadlock. This occurs when α_1 also has a full queue and α_2 is attempting to send a message to α_1. More complex message sending patterns could also lead to a deadlock. The key observation is that α_1 cannot pop a message from its own queue while it is waiting for α_2's queue to no longer be full, as doing so would cause behaviours on α_1 to be interleaved, resulting in inconsistent state.

- If α_1 adopts the strategy of failing to send a message when α_2's queue is full, all asynchronous behaviour calls would have to be able to propagate an error, and that error would have to be checked in the program. The error would relate to resource exhaustion (a full queue) but unlike other forms of resource exhaustion (e.g. running out of memory, reaching file handle limits, etc.), the condition is inherently temporary. That is, α_2 would eventually pop a message from its queue, and the program would need to restart or retry actions in response. Effectively, this would transform any behaviour that sent one or more messages into a state machine depending on message sending progress. To keep consistent state, all of an actor's behaviours would have to be integrated into a single state machine that tracked this progress. This is effectively interleaved behaviours, but with a complex mechanism for the programmer to manually resolve inconsistent state.

Unbounded message queues transform a temporary resource exhaustion problem into a less common permanent resource exhaustion problem, i.e. an unrecoverable out of memory error when a queue grows so large that it exhausts all available memory. This results in a simpler error handling mechanism (the program terminates), but, more importantly, in an error condition that represents an actual unrecoverable program error, rather than a transient and recoverable condition.

A.6.2 Empty Queue Detection

An important aspect of the scheduler is that an actor with no pending work must not consume scheduler resources. That is, it must not be present on any scheduler queue. In order to accomplish this, a scheduler thread that is executing an actor must be able to detect when that actor's queue is empty in order to not reschedule it. In addition, whenever a message is sent to an actor, it must be possible to atomically detect that the queue was empty when the message was sent, in order to allow the receiving actor to be scheduled on some scheduler thread.

When a scheduler thread executes an actor, it is possible that the actor will attempt to pop a message from its queue, as in figure A.12, and no message will be available. When this happens, the actor will attempt to set the low bit on the stub message (stub messages are discussed in section A.6.3) remaining in its queue, as shown in figure A.13. This requires a single atomic compare-and-swap, with no loop, i.e. it is a lock-free and wait-free operation. It will succeed only if the queue is truly empty.

If the actor is able to mark its queue as empty, the scheduler thread will not reschedule the actor. At this point, the actor will not be referenced by any scheduler thread.

```
bool ponyint_messageq_markempty(messageq_t* q) {
  pony_msg_t* tail = q->tail;
  pony_msg_t* head = atomic_load_explicit(&q->head,
      memory_order_relaxed);
  if(((uintptr_t)head & 1) != 0)
    return true;
  if(head != tail)
    return false;
  head = (pony_msg_t*)((uintptr_t)head | 1);
  return atomic_compare_exchange_strong_explicit(&q->head
      , &tail, head,
    memory_order_release, memory_order_relaxed);
}
```

Figure A.13: Marking a message queue as empty

Returning to figure A.11, we can see that when some actor α_1 sends a message to some actor α_2 and α_2's queue is empty at the moment when the message is sent, it will be atomically detected. When this happens, the sending actor α_1 will reschedule α_2, as α_2 now has pending work. Rescheduling involves pushing the actor onto the scheduler queue of the currently executing scheduler thread, as described in section A.12.

Critically, this empty queue detection and rescheduling will happen only once, regardless of contention on α_2's queue, and will not occur again until α_2 marks its own queue as empty again. As a result, any given actor will always be present on either zero or one scheduler queue.

A.6.3 Queue Node Memory Management

When a message is pushed to a queue, the sender implicitly uses the pool allocator to allocate a new message queue node that will contain the arguments. This is the message queue node that is pushed on to the receiver's queue, as seen in figure A.11.

However, when the receiver pops a message from its queue, the receiver reads the next node from the tail, rather than the tail, as seen in figure A.12. If the pop is successful, the previous tail is freed, and the message node containing the popped message remains on the queue.

This has two key advantages. The first is that the message queue node is not freed until after the behaviour that handles the message has finished executing, which simplifies memory management. The second, more critical, advantage is that the message queue is never empty. It always contains at least one message queue node, which may or may not point to another node that represents a

```
typedef struct pony_actor_t {
  pony_type_t* type;
  messageq_t q;
  uint8_t flags;
  __pony_spec_align__(heap_t heap, 64); // 52/104 bytes
  gc_t gc; // 44/80 bytes
} pony_actor_t;
```

Figure A.14: Internal actor description

pending message. This allows the code that pushes a new message on the queue to separate the atomic exchange on the queue head and the setting of the next pointer without causing a race condition.

When the push code executes the atomic store, it will always write the correct next node, even if the head has in the meantime changed due to another message being pushed on to the queue. This can result in a message pop returning `NULL` even though a new message has been partially pushed on to the queue (that is, the queue head has been changed, but the tail does not yet point to the new head). However, this is not a race condition. Under these circumstances, the actor will report that is has no pending messages, but when it is asked to mark its queue as empty, it will fail to do so, due to the head no longer being the same as the tail. In fact, the actor will determine the queue is non-empty without an atomic operation. As a result, the actor will be rescheduled, and the newly pending message will eventually be handled, as expected.

A.7 Actors

In the runtime, an actor is represented as an object header `type`, a message queue `q`, a set of operational `flags`, a `heap`, and information for tracking cross-actor garbage collection `gc` (as discussed in sections A.9 and A.10), as seen in figure A.14.

The heap and garbage collection structure are aligned on a 64 byte boundary to keep them on a separate cache line from the type, queue, and flags, to reduce false sharing in the cache coherency protocol. The actor description, including padding, is 156 bytes on a 32-bit architecture and 240 bytes on a 64-bit architecture.

For a concrete actor type, this structure is extended with the fields of the actor. For example, if the concrete actor type has 16 bytes of field data on a 64-bit architecture, an instance of the type will be allocated with a single 256 byte allocation, i.e. 240 bytes of actor description plus 16 bytes of field data.

This is done to avoid a memory indirection when reading from or writing to the fields of an actor.

It is important to note that an actor has no stack associated with it. Instead, only scheduler threads have stacks. This allows actors to consume significantly less memory, while avoiding the execution speed penalties associated with segmented stacks. In addition, the foreign-function interface (FFI) can use the same unsegmented stack that the actor executing the FFI call is currently using, providing a simple and efficient FFI mechanism.

A.7.1 Message Handling

When an actor is scheduled for execution by a scheduler thread, the actor will pop messages from its own queue and handle each one in turn. These messages are divided between internal garbage collection messages, as described in sections A.9 and A.10, and messages explicitly generated by the program, which are termed application messages, as in chapter 6.

An application message is handled by executing a behaviour on an actor. That behaviour is executed to completion by the scheduler thread. Since the type system enforces isolation and immutability, as detailed in chapter 4, this results in every actor behaviour being logically atomic. That is, each behaviour on an actor can reach only the actor's state and the arguments to the behaviour, and the behaviour cannot witness any mutation other than mutations it performs itself.

A.7.2 Message Batching

Popping an actor from a scheduler queue requires an atomic compare-and-swap loop, as discussed in section A.12.1, whereas popping a message from a message queue requires no atomic operations and is wait-free, as discussed in section A.12. As a result, it is beneficial to handle more than one message each time an actor is scheduled for execution.

Actors are not asked to handle every message in their queue entirely when they are scheduled as this would lead to starvation. For example, an actor could send itself one or more messages in a behaviour, resulting in the actor never yielding execution, or a producer-consumer pattern could result in a consumer receiving messages as or more quickly than it could handle them, again resulting in starvation.

In the current implementation, actors are asked by the scheduler thread to handle up to a fixed number of application messages (defaulting to 100) each time they are scheduled, amortising the cost of a scheduler queue pop across many behaviours. In addition, any number of interleaved internal, garbage collection related, messages can be processed without counting against the batch limit. In future work, the number of application messages an actor is asked to handle will fluctuate dynamically based on back-pressure.

A.7.3 Blocking and Unblocking

When an actor believes it has no pending messages, it sets a flag indicating that it is blocked and sends a message to the cycle detector (described in section A.10.3) indicating that it should be considered blocked. It then attempts to mark its own message queue as empty, as discussed in section A.6.2. Because the actor may be rescheduled on any scheduler thread as soon as its queue is marked as empty, the actor must set the blocked flag and send the message to the cycle detector before it marks its own message queue as empty. As a result, the state reported to the cycle detector may already be out of date by the time it is sent. The implications of this are discussed in section A.10.

When an actor handles an application message and its own blocked flag is set, it unsets the flag and sends a message to the cycle detector indicating that is should be considered unblocked. However, internal garbage collection messages do not result in the actor unblocking. Instead, if these are received while blocked, the actor sets a flag indicating that its garbage collection status has changed. This flag influences the way the actor responds to the cycle detector, as discussed in section A.10.9.

If an actor has handled one or more application messages when scheduled, as an optimisation it will not attempt to block or mark its message queue as empty even if it believes its message queue to be empty. Instead, it will reschedule as if it had pending work. This is an optimisation that results in fewer block and unblock messages, significantly reducing state churn and message load on the cycle detector.

A.7.4 Rescheduling

If an actor handles any application messages or is unable to mark its queue as empty (indicating there are new pending messages), it asks the scheduler thread to reschedule it. However, if the actor successfully marks its queue as empty, the actor is not rescheduled. This results in the actor not being in any scheduler queue, and so consuming no resources (memory or CPU time), other than the memory required to hold its own state, when it is not scheduled.

A.8 Tracing GC

The core of the Pony garbage collector is the per-actor tracing garbage collection step. This step traces and collects actor heaps independently, without requiring coordination between actors. The algorithm used is a mark-and-don't-sweep collector that has a time complexity of $O(m + n)$, where n is the number of objects reachable by the actor, and m is the number of chunks (as described in section A.5) on the actor's heap.

The algorithm begins by setting all the valid bits in every chunk descriptor on the heap. This tentatively frees all memory allocated by the actor, without releasing chunk descriptors or the memory areas managed by those chunks. The counter for the amount of memory in use by the actor is also reset to zero.

While this phase involves the chunk descriptors for all memory allocated on the actor's heap, it does not actually touch the allocated memory itself. This is important as it avoids unnecessary cache pollution and page faults that can result from reading unreachable memory.

In the next phase, the garbage collection roots are marked as reachable, and any root with a trace function, as detailed in section A.8.2, is pushed on to a stack of objects that have not yet been traced. If the trace function can be determined statically, the code generator inserts it directly. Otherwise, it is dynamically determined by reading from the object header. While additional roots are introduced by the sharing protocol in section A.9, the only root introduced here is the actor itself.

The stack of as yet untraced objects is then processed. When an object is traced, its memory address is looked up in the page map (described in section A.4). This gives the address of the chunk descriptor that manages the memory area the object has been allocated in. The offset of the object's address from the base address of the memory area gives the bit that should be unset to represent that this memory is reachable.

When the stack of objects and their trace functions is empty, every reachable object in the actor's heap has been traced. At this point, the chunk descriptors are examined again. A chunk descriptor with no available memory is placed on the linked list of full chunks for its size class, and a chunk descriptor that has some available memory, but is not empty, is placed on the linked list of chunks with available memory for its size class. An empty chunk, i.e. one with all valid bits set indicating the memory is free, is returned to the pool allocator along with the memory area it manages. This maintains the chunk validity invariant in definition A.1.

A.8.1 Deep vs. Shallow Marking

A distinction is made when clearing bits (to indicate that memory is in use) between deep and shallow marking. A deep mark is used when the object is reachable with a reference capability that allows reading from the object. This indicates not only that the object is reachable, but also that it has already been placed on the reachable object stack with its trace function. On the other hand, when an object is reachable as a `tag`, and so its fields are not reachable, a shallow mark is used. This indicates that the object is reachable, but it has not yet been placed on the reachable object stack. In this case, if the object is reachable via some other path with a readable reference capability, the object's fields will be correctly traced. The distinction between deep and shallow mark bits is shown in figure A.8.

This approach treats internal pointers (pointers inside a small allocation as opposed to a pointer to the beginning of a small allocation) as shallow marks, even if their fields are reachable. This is because internal pointers do not have bitmap entries of their own, but rather all internal pointers within an allocation share the bitmap entry of the external pointer. By using shallow marks, other internal pointers, or the enclosing external pointer, will be correctly traced if they are reached after an internal pointer has been traced.

Correctly accounting for internal pointers allows optimisations at the language level. Array and string slices, which use internal pointers to arrays of objects, can be naturally expressed. In addition, complex object layouts, such as embedded fields (in a similar manner as would be done in C by defining a field as a `struct` instead of a pointer to a `struct`), or the transformation of an array of `structs` to a `struct` of arrays, can be handled without specialising the garbage collector.

A.8.2 Trace Functions

Each concrete type generated by the compiler has a trace function. The trace function describes how to trace the fields of a reachable and readable reference to an instance of the type.

Fields are traced based on the field type visible from the object, without taking into account any viewpoint adaptation in the type system, as detailed in sections 5.3.2 and 6.3.1. In other words, the trace function for a type implicitly treats the instance being traced as a `ref`. This allows the trace function to reach the maximal set of objects that could be reached from the program, without requiring reference capabilities to be encoded at runtime. Doing so is safe, because garbage collection does not mutate objects: all garbage collection related state is kept in parallel, actor-specific, data structures, rather than in

```
primitive P
interface I
class C1 // No fields
class C2
  let f1: P val // No trace
  let f2: C2 ref // Static trace function
  let f3: I ref // Dynamic trace function
  let f4: C2 tag // Shallow mark
  let f5: I tag // Shallow mark or actor
  let f6: (P val, C2 ref, I ref) // Destructured
  let f7: (C2 ref | I tag) // Shallow mark or actor
  let f8: ((C1 ref, C2 tag) | (C2 tag, C1 ref))
    // Dynamically destructured
```

Figure A.15: Classes, fields, and associated tracing

the objects themselves.

The action taken by a trace function for each field depends on the type of the field. Some examples are given in figure A.15. Note that this example covers more of the Pony type system than is described in chapter 4, and covers `primitive` types (singleton types with no state, also usable as first-class parameterised modules), `interface` types (structural types, also usable as type classes), tuples, and unions. The full Pony type system also includes `traits` (nominal types), generic types and functions, as well as intersection types. Their behaviour during tracing is a natural extension of the behaviour presented here.

For primitive types, no action is taken, as in `C2.f1`. For types with a known `tag` reference capability, such as `C2.f4`, the object is shallowly marked, and is not placed on the trace stack, as described in section A.8.1. If the type could be an actor, either because it is an interface or trait type or because a concrete actor type is present in an algebraic data type, such as `C2.f5`, then the object's header is examined at runtime to determine if it is actually an actor. If so, it is handled via the actor garbage collector, as described in section A.10, instead of being marked. Otherwise, the trace function of the object is called dynamically.

If the field has a known concrete type, that is, a `struct` or `class`, and does not have a `tag` reference capability, such as `C2.f2`, the object is marked and placed on the trace stack with its statically known trace function. If the field has an unknown concrete type, either because it is a trait or interface or because it is an algebraic data type, but it can be determined statically that the object will not have a `tag` reference capability, such as `C2.f3`, the object is marked and placed on the trace stack with a dynamically determined trace function that is looked up in the object's header.

A field that is statically known to be a tuple, such as `C2.f6`, is de-structured and each field is traced independently. The tuple itself does not have to be marked, as statically known tuple fields are embedded in the parent instance rather than being reached via a pointer indirection.

The complex case comes when it is not possible to determine statically whether or not a field has a `tag` reference capability. This case requires the object to be examined at runtime. The issue is not just that tracing a `tag` reference as if it was readable would lead to unreachable objects not being collected (which would be sub-optimal but still correct), but that tracing a `tag` reference as if it was readable in some actor α_1 could result in α_1 reading the fields of an object (for tracing purposes) while some other actor α_2 modifies those fields.

Handling this case requires the code generator to use the type system to determine which types a field could be visible as to the program. For example, `C2.f7` is treated as an `I tag`, i.e. the same as `C2.f5`, because `C2` is a subtype of `I`, so that it is not possible for the program to pattern match on `C2.f7` as a `C2 ref`, or indeed as any readable capability.

This particular example is somewhat counterintuitive, as it results in the fields of `C2.f7` not being traced, and so possibly garbage collected if they are not reachable via some other path. This initially appears to be unsound. However, the inability to pattern match on `C2.f7` as any readable capability, which is enforced by the type checker, prevents the fields of `C2.f7` from being read or written to. As such, it is safe to garbage collect those fields if they are not otherwise reachable.

When tuples are contained in algebraic data types, the code generator must determine the set of possible ways each element of each tuple could be viewed. For example, in `C2.f8`, each element must be dynamically examined, as each may or may not have a `tag` reference capability at runtime.

Generating trace functions that can make these distinctions at runtime is important because reference capabilities, being based on viewpoints, cannot be encoded in objects. That is, any given object may be viewed with a collection of reference capabilities, depending on the path used to reach the object. As a result, if reference capabilities were to be encoded at runtime, every object reference would have to include both a pointer and a capability, which would impose both a memory overhead and an execution speed overhead. In addition, such an encoding would still require adapted capabilities to be calculated at runtime based on the encoded field or local variable capabilities.

A.8.3 Garbage Collecting During Behaviours

The current implementation performs per-actor garbage collection on an actor in between behaviours. As a result, garbage collection does not occur during actor execution. This approach reduces performance jitter within a behaviour. In addition, it means that object references on the stack do not have to be tracked as garbage collection roots, so that no stack map need be compiled into the program.

However, this approach has a significant downside: behaviours that allocate large amounts of short-lived memory (i.e. memory that is not further referenced within the behaviour) will not collect that short-lived memory until the behaviour finishes executing.

In future work, the code generator will be extended to optionally generate a stack map. This will allow object references on the stack to function as garbage collection roots. To do so, the stack map will have to encode the same ability to distinguish possible types at runtime as trace functions encode on fields, as described in section A.8.2.

The stack map on its own is sufficient to allow garbage collection during behaviour execution, without the addition of read or write barriers, or of safe-points. This is possible because each actor heap is independently collectable (no safe-points required), and because concurrent garbage collection is handled through the message-based protocol described in chapter 6 rather than with a stop-the-world phase.

In addition, the design decision to not use a copying collector means that stack references do not need to be changed to point to objects moved to new addresses by the copying collector. This is also true of heap pointers, but is arguably more significant for stack pointers. Changing stack pointers requires the single static assignment (SSA) intermediate representation (IR) of the program used by LLVM to generate machine code to account for relocations, which can impede certain compiler optimisations.

A.8.4 Role of the Type System

The tracing phase of the garbage collector is heavily dependent on the type system. The use of a data-race free type system enables independent heap tracing. It also allows normal program execution to proceed without read or write barriers.

In addition, the existence of object references that do not allow reading from the object (i.e. the `tag` reference capability) requires the distinction between shallow and deep marking, as well as the generation of trace functions which can distinguish, either statically or dynamically, between readable and unreadable object references.

```c
typedef struct gc_t {
  uint32_t mark;
  uint32_t rc_mark;
  size_t rc;
  objectmap_t local;
  actormap_t foreign;
  deltamap_t* delta;
} gc_t;
```

Figure A.16: Tracking local and foreign reference counts

```c
typedef struct object_t {
  void* address;
  size_t rc;
  uint32_t mark;
  bool immutable;
} object_t;
```

Figure A.17: Object GC information

A.9 Sharing GC

The per-actor tracing phase of the garbage collector, described in section A.8, is unsafe if used alone: it would garbage collect memory allocated by some actor α_1 that is no longer reachable by α_1, but is reachable by some other actor α_2. The message-based protocol described in chapter 6 is used to protect such objects from premature collection. This description goes beyond the formal model in chapter 6 and describes both the overall implementation and specific optimisations, such as generational marks, that are necessary to achieve the desired performance.

The implementation of the protocol requires keeping a data structure to track local and foreign reference counts, as shown in figure A.16. Each actor has a `gc_t` to track its view of shared objects, as shows in figure A.14. The `local` map associates locally owned pointers with a reference count, as shown

```c
typedef struct actorref_t {
  pony_actor_t* actor;
  size_t rc;
  uint32_t mark;
  objectmap_t map;
} actorref_t;
```

Figure A.18: Actor GC information

in figure A.17. The foreign map associates actors with objects they own, as shown in figure A.18. Note that this is a map of actors to a map of objects, i.e. $[Actor \mapsto (RC, Mark, [Object \mapsto (RC, Mark, Immutable)])]$, allowing foreign object GC information to be kept segregated by owner.

This per-object and per-actor GC information is kept outside of the object itself, i.e. in a separately allocated data structure rather than in fields of the object, because it is not intrinsic to the object: it is information about an object from the perspective of some actor. As such, each actor has GC map entries only for the local objects it has sent and the foreign actors and objects it has received.

The map algorithm used for both actormap_t and objectmap_t is a linear probing Robin Hood hash map where the key is embedded in the same data structure as the value. In an objectmap_t, the key is the address field, and in an actormap_t, the key is the actor field.

A.9.1 Mark, Send, and Receive Phases

There are three distinct phases when the gc_t for an actor is used. The first is during the mark phase of the tracing garbage collection described in section A.8. The second is when a message is sent, and the third is when a message is received.

In the second and third phases, which correspond to definitions 6.3 and 6.4 from chapter 6, the tracing algorithm from section A.8 is used, but the roots are the arguments to the behaviour rather than the fields of the actor. That is, when a message is sent or received, the arguments to the message are traced to determine what garbage collection information must be updated. This is used to determine reachability as defined in section 6.3.1.

A.9.2 Generational Marks

The mark field in the data structures in figures A.16, A.17, and A.18 is used to track whether an object has been marked in the current garbage collection phase. Its purpose is to allow cyclic structures to be correctly traced, with each object being examined only once.

Because the number of objects tracked in an actor's gc_t structure could be significantly larger than the number of objects being sent or received in a message, a generational mark is kept for each object rather than a simple mark bit. The mark field of gc_t indicates the current generation, with this number being incremented after the completion of any mark, send, or receive phase.

When an actor or object is visited in a garbage collection phase, its mark is set to the current generation. Thus, an actor or object entry with a mark field that differs from the `mark` field in the `gc_t` has not yet been visited.

The purpose of this is to avoid the need to sweep the actors and objects being tracked. Instead, the generational mark is incremented, which implicitly invalidates the mark for every tracked actor or object. This not only reduces the time complexity of a phase, it also avoids the cache pollution that would result from sweeping these data structures.

A.9.3 Determining Ownership

When an object is encountered in any garbage collection phase, the first step is to determine the owner of the object. To do so, the object's address is looked up in the page map, as described in section A.4. This returns the chunk descriptor for the memory area the object is allocated in. The chunk descriptor, in turn, contains a pointer to the owning actor.

Because the chunk descriptor contains both the mark bits for the managed memory area and the owning actor, the chunk descriptor lookup performed in the tracing garbage collector described in section A.8 also serves to distinguish between local and foreign object ownership.

A.9.4 Sending and Receiving Local Objects

When a locally owned object is sent in a message, the sending actor must protect that object from garbage collection, as described in chapter 6. To do so, the object's address is looked up in the `local` map. If the object is not present in the map, it is added with an `rc` of one and the object's `mark` is set to the current mark. If the object is present in the map and the object's `mark` is not current, the object's `rc` is incremented by one, and its `mark` is set to the current mark.

The result of this is that an object's local `rc` is incremented by one each time it is sent in a message, where being sent in a message means that the object is a message argument or is reachable from some message argument, as defined in section 6.3.1.

Similarly, when a locally owned object is received in a message, the object's address is looked up in the `local` map and, if the object's `mark` is not current, its `rc` is decremented by one. Because of the invariants described in chapter 6, the object will always be present in the `local` map and will always have an `rc` greater than zero.

If an object's local reference count drops to zero, it is not removed from the `local` map at this point. That bookkeeping is done when the local map is traced during garbage collection, as described in section A.9.7. This is done to avoid adding and removing an object's `local` map entry multiple times in between garbage collection phases.

A.9.5 Sending and Receiving Foreign Objects

For foreign objects, the simpler case is when they are received. When this happens, the object's owning actor address is looked up in the `foreign` map, and is added if it is not present. Then a second lookup, in the resulting `map` for that actor, is performed using the object's address, again adding the object if it is not present. At this point, if the object's `mark` is not current, the object's foreign `rc` is incremented by one and its `mark` is set to the current mark.

As a result, while a local object's `rc` is incremented when it is sent, a foreign object's `rc` is incremented when it is received.

The more complex case occurs when a foreign object is sent in a message. In this case, the owning actor and then the object are looked up, just as when a foreign object is received. If the object's `mark` is not current, its `rc` is decremented by one. The complexity occurs when the sending actor's foreign `rc` for the object is one. In this case, the foreign `rc` cannot be decremented without breaking the invariant that an actor that can reach a foreign object, either through its fields or on the stack, must hold a foreign `rc` for that object of at least one, as required by definition 6.2.

When this occurs, the sending actor increases its foreign `rc` for the object by an arbitrary amount (256 in the default configuration). To maintain all of the reference count invariants described in chapter 6, a message must be sent to the owning actor informing it of the increased reference count. To do so, the object is added to the set of objects for which an *acquire message* will be sent when the current garbage collection phase finishes, as detailed in section A.9.9. This is the implementation of the ACQUIRE rule from figure 6.2.

Note that an *acquire message* consists of a set of `object_t`, as defined in figure A.17. This is equivalent to the *ExMap* defined in figure 5.3. The `mark` field is ignored, but the `immutable` field is used in section A.9.6. When the `immutable` field is `false`, an *acquire message* behaves exactly as detailed in chapter 6.

A.9.6 Sending and Receiving Immutable Objects

The process of tracing message arguments leads to an $O(n)$ step both when sending and receiving the message, where n is the number of objects reachable from those arguments. For large data structures, this can be a significant cost.

The purpose of tracing the message arguments is to protect the contents from premature garbage collection by adjusting the distributed weighted reference counts. The end result is that the owner of an object with a positive local `rc` will not collect that object.

The type system can be used to provide an important optimisation: when some immutable object ω is sent in a message, either as an argument or reachable from an argument, any objects reachable from ω do not have to be protected from premature garbage collection. This is possible because the owner of ω can safely trace the fields of ω when it traces its local object map, as described in section A.9.7. The fields of ω can be traced by the owner precisely because ω is immutable: it is safe for the owner to read the fields because no other actor will write to those fields.

As a result, sending and receiving an immutable object in a message requires $O(1)$ tracing instead of $O(n)$. To achieve this, the per-object garbage collection information in figure A.17 includes the flag `immutable`, which indicates the object is immutable and that the owner knows this. When that flag is `true`, the owner traces the fields of the object.

Maintaining this flag is trivial if the owner initially sends the immutable object. However, it is possible for some actor α_1 that owns some object ω to send it as an isolated object, and then for some other actor α_2 to later send ω as an immutable object to some receiver α_3. In this case, α_1 will have protected the fields of ω against premature garbage collection by tracing them when the message was sent, but α_2 may have written to those fields before sending ω as an immutable object.

When α_2 initially receives ω, the flag will not be set, since α_1 has sent ω as an isolated object. When α_2 later sends ω as an immutable object, this discrepancy will be detected. At this point, α_2 will mark ω as immutable in its foreign map, but it will also send an acquire message for ω to α_1 with the `immutable` flag set to `true`, as described in section A.9.9, and it will trace the fields of ω for the current message. Note that this process is recursive: all objects ω' reachable from ω that α_2 sees as mutable are also marked as immutable, and the owners of such objects ω' are sent acquire messages with the `immutable` flag set to `true`.

This will result in α_1 receiving an acquire message that tells α_1 to mark ω as immutable in its local map *before* it receives any reference count decrement for ω from α_2. At this point, α_2 can safely send ω to α_3 without tracing ω. The object graph reachable by ω will be protected from premature collection because α_1 will trace the fields of ω during garbage collection as long as ω itself is live, i.e. has a positive reference count.

When α_3 receives ω as an immutable object from α_2, α_3 can safely mark ω as immutable in its foreign map, and not trace the fields of ω when receiving ω. Indeed, all future sends of ω, any corresponding receives, and future garbage collection phases by any actor other than α_1 (the owner of ω) need not trace ω.

Effectively, *freezing* some isolated ω, such that ω becomes deeply immutable, requires a single $O(n)$ tracing phase in order to replace future $O(n)$ tracing phases with an $O(1)$ change in the reference count of ω.

A similar situation arises if α_3 receives ω in a message, then reads some field ω' from ω. It is then possible for α_3 to drop its reference to ω while continuing to hold a reference to ω'. To protect ω' from premature collection, α_3 treats ω' as described above: an acquire message for ω' is sent to the owner of ω' (which may or may not be α_1). This message is sent before the release message for ω, as detailed in section A.9.9, which correctly protects ω' while allowing α_1 to possibly collect ω.

A.9.7 Tracing the Local Map

The local map is used by an actor to mark memory on its heap that it cannot reach as in-use because it may be reachable from some other actor. To do so, the map is iterated over, and every object with an `rc` greater than zero is marked. If the object is also marked as `immutable` in the describing `object_t`, it is placed on the trace stack with its trace function, so that its fields are traced as well, as described in section A.9.6.

During this iteration, an object is removed from the local map if its local `rc` is zero. If the object has a finaliser, then it is only removed from the local map if it is also unreachable from the owning actor, at which point its finaliser is run as well. This is described in detail in section A.11.

A.9.8 Tracing the Foreign Map

After an actor's reachable objects and local map are traced, the weighted reference count held for any objects in the actor's foreign map that have not been marked can be released. The object garbage collection information for an unreachable foreign object is removed from the foreign map and sent to the owning actor as a *release message*, as described in section A.9.9. This is the implementation of the Release rule from figure 6.2.

The organisation of the foreign map into a map of actors, each of which has a map of objects, allows these release messages to be easily aggregated. All of the unreachable foreign objects owned by some actor α are combined in a single message to α.

A.9.9 Acquire and Release Messages

An owning actor adjusts the local `rc` for an object both implicitly when sending or receiving the object in a message, and explicitly when it receives acquire and release messages generated by other actors.

Acquire messages are generated when some actor α sends some foreign object ω_1 for which it has a foreign `rc` of one or less, or when α traces its reachable objects and discovers an immutable object ω_1 that is not in α's foreign map because it was previously reachable only via some other immutable object ω_2 that is in α's foreign map. In both these cases, α is free to increase its weighted reference count for ω_1 by an arbitrary amount and inform the owner of ω_1 that it has done so.

This is accomplished by building an alternate foreign map during a garbage collection phase that consists of the objects that need to be acquired. The structure used is the same as the foreign map. When the phase completes, each actor in the acquire map is sent the accumulated actor garbage collection information, as described in figure A.18, as an acquire message. When that acquire message is received by the owning actor, the `rc` for each object is added to its local `rc`.

Release messages are generated when an actor finishes a garbage collection pass. Unmarked objects in the foreign map are removed and sent to their owning actor. Like acquire messages, these are aggregated, so that each owning actor is sent at most a single release message as the result of a garbage collection pass. When that release message is received by the owning actor, the `rc` for each object is subtracted from its local `rc`.

A.10 Actor GC

The garbage collection mechanism described in sections A.8 and A.9 collects objects allocated on actor heaps, even when those objects are shared across actors. The same mechanism is extended in the runtime to garbage collect actors themselves.

Each actor keeps track of a local `rc` for itself in the structure described in figure A.16. This local actor `rc` works in the same fashion as a local `rc` for an object. Similarly, actors keep a foreign `rc` for every other actor they can reach, in the structure described in figure A.18. These weighted reference counts are managed in the same way as for objects.

As a result, an actor can be garbage collected when it is both blocked, as discussed in section A.7.3, and its local `rc` is zero. This condition is sufficient, but not necessary: an isolated graph of blocked actors may be collectable even if each actor's local `rc` is not zero, as described in chapter 5. The implementation is described in section A.10.3.

```
typedef struct delta_t {
  pony_actor_t* actor;
  size_t rc;
} delta_t;
```

Figure A.19: Topology delta information

A.10.1 Implicit Actor Reachability

An actor is implicitly reachable from an object if it owns that object. This prevents an actor from being collected, and its heap destroyed, if any object on its heap is reachable. To account for this, during each garbage collection phase, if a foreign object is marked, the owner of that foreign object is also marked.

This means that it is possible to mark an actor that does not appear in the foreign map. This happens when an immutable object that is not in the foreign map is encountered, as described in section A.9.6. When this occurs, the same mechanism is used: the actor is inserted into the foreign map and an acquire message is generated for that actor.

A.10.2 Topology Deltas

In addition to the local and foreign maps, each actor keeps an additional data structure that describes the change in the actor's view of its own topology since the last time that actor blocked. This is a simple map of actor address to a reference count, as shown in figure A.19. Note that these are stored in the `deltamap_t` contained in each actor's `gc_t`, as shown in figure A.16.

When some actor α_1 changes its local `rc` for some actor α_2, the new `rc` is stored both in the foreign map and in the topology delta. When α_1 blocks, its topology delta is sent to the cycle detector, and it begins again with an empty topology delta.

As a result, an actor that blocks sends the cycle detector only the minimal information that the cycle detector needs to recreate the actor's view of its own topology. This eliminates an $O(n)$ step to gather local topology when an actor blocks, where n is the number of other actors that actor has in its foreign map. This is replaced with an $O(1)$ step for each local topology change, no cost when sending a block message, and an $O(m)$ step when the cycle detector receives a block message, where m is the number of actors for which a topology change occurred. Importantly, $m < n$, possibly significantly less, and the step is performed in the cycle detector rather than in a program actor.

```c
struct view_t {
  pony_actor_t* actor;
  size_t rc;
  uint32_t view_rc;
  bool blocked;
  bool deferred;
  uint8_t color;
  viewrefmap_t map;
  deltamap_t* delta;
  perceived_t* perceived;
};

typedef struct viewref_t {
  view_t* view;
  size_t rc;
} viewref_t;
```

Figure A.20: The cycle detector's representation of a view of an actor's topology

A.10.3 The Cycle Detector

Unlike shared object collection, it is possible to have an isolated cyclic graph of actors where all the actors in the graph are blocked, and their local `rc` is held entirely by other blocked actors in the graph, but the actors are not collectable because their local `rc` is not zero, as detailed in section 5.2.3.

The cycle detector is implemented as an actor in the runtime. It is instantiated when a program begins, and its address is kept as global data with ambient authority. That is, all actors can implicitly send messages to the cycle detector. However, the cycle detector does not itself take part in garbage collection. The memory it uses is manually managed, and its references to actors that have reported their topology do not influence local or foreign reference counts.

The cycle detector's representation of a view of an actor's topology is shown in figure A.20. Here, the `rc` is the actor's last reported view of its own local reference count. The `view_rc` is used internally to manually manage the view structure memory. The `blocked` flag indicates the actor's last report of whether or not it believed it had an empty message queue. The `deferred` flag is set if the actor is in the `deferred` map, as described in section A.10.4. The `color` is used in a tri-colour tracing scheme for finding isolated graphs of actors, as described in section A.10.7. The `map` is a map of `viewref_t`, tracking the actor's last reported references to other actors, and the `rc` held for each. The `delta` pointer keeps the last topology delta reported by the actor. This is applied to the `map` in a deferred manner, as described in section A.10.4. Finally, the `perceived` pointer is a reference to the perceived cycle the actor is a

```c
typedef struct detector_t {
  pony_actor_pad_t pad;
  size_t next_token;
  size_t min_deferred;
  size_t max_deferred;
  size_t conf_group;
  size_t next_deferred;
  size_t since_deferred;
  viewmap_t views;
  viewmap_t deferred;
  perceivedmap_t perceived;
  size_t attempted;
  size_t detected;
  size_t collected;
  size_t created;
  size_t destroyed;
} detector_t;

struct perceived_t {
  size_t token;
  size_t ack;
  size_t last_conf;
  viewmap_t map;
};
```

Figure A.21: Cycle detector state

member of, if any, as described in section A.10.7.

The cycle detector's view of the actor topology is tracked in the `views` map of actor addresses to topology views, as shown in figure A.21. In addition, the cycle detector keeps track of the next confirmation-acknowledgment token, detailed in section A.10.9, state related to deferred checks for isolated graphs of blocked actors, detailed in section A.10.7, a map of tokens to perceived cycles, and some useful debugging statistics.

A.10.4 Block Messages

When an actor believes it has blocked, as described in section A.7.3, it sends a message to the cycle detector indicating that it is blocked, and also indicating the actor's view of its own topology. This is sent as the actor's view of its own local reference count, from the structure in figure A.16, and its topology delta, as described in section A.10.2.

When the cycle detector receives a block message from an actor, it first retrieves the actor from the `views` map in figure A.21, or adds it if the actor is not yet present. The cycle detector updates the view `rc` with the value reported by the actor. Any previous topology delta that has been cached in the `delta` field of the view is applied to the actor, as described in section A.10.5. The current topology delta is then cached in the `delta` field. This delayed application of the topology delta is used to avoid unnecessary work by the cycle detector.

The actor is then marked as `blocked`, and any perceived cycles (as described in section A.10.7) that the actor is a member of are cancelled. Note that perceived cycles are cancelled due to block messages as well as due to unblock messages (as described in section A.10.6). This allows actors that update their view of their own `rc` while blocked, due to receiving acquire and release messages as described in section A.9.9, to send a new block message to the cycle detector to update the cycle detector's view of the actor's topology. An actor with a new view of its topology is not collectable on the basis of its old view of its topology, and so any perceived cycle detected based on the old topology must be cancelled.

If the actor's perceived reference count is zero, it is removed from the `deferred` map and immediately collected. Otherwise, the actor is added to the `deferred` map. The `deferred` map is used to reduce the total work done by the cycle detector. Rather than finding isolated graphs of blocked actors every time an actor blocks, the process is deferred until some number of actors have blocked. At that point, some actor in the `deferred` map is chosen as a starting point, and an attempt is made. If the attempt touches actors in the `deferred` map, they are removed from the map, effectively reducing the total number of isolated graph searches. Finding or not finding an isolated graph is used as a feedback mechanism to the number of deferred actors required to trigger a detection attempt.

A.10.5 Applying Topology Deltas

Topology deltas are applied lazily, in order to avoid the work required to update a topology until the blocked actor's view of its own topology is required for a cycle detection attempt, or until the blocked actor sends another topology update to the cycle detector. The topology delta map shown in figure A.19 is applied to the `viewrefmap_t` from figure A.20. Each entry in the delta map sets a new reference count that some actor α whose topology is being updated holds for some other actor α'.

If the cycle detector did not previously see α as holding a reference count for α', then α' is added to the `viewrefmap_t` of α. Doing so increments the `view_rc` field of the `view_t` the cycle detector holds for α'. This provides a simple acyclic reference count for `view_t` structures. The purpose of the reference count is to allow a `view_t` to hold an out-of-date reference to an actor that has already been collected.

For example, suppose α_1 holds the only reference to α_2 and both actors block. Later, α_1 unblocks and drops its reference to α_2, which remains blocked. At this point, α_2 can be collected, even though the cycle detector has an out-of-date topology for α_1 in which α_1 still holds a reference for α_2. By using a reference counted `view_t` structure, the cycle detector avoids keeping inverse reference lists for actors or searching for all actors with out-of-date references to an actor that is being collected.

A.10.6 Unblock Messages

When an actor α unblocks, it notifies the cycle detector. When the cycle detector receives an unblock message from α, it removes α from the `deferred` map if it is present and cancels all perceived cycles that α is a member of. Rather than search all perceived cycles, each `view_t` keeps a map of perceived cycle that the actor is a member of. This allows the cycles to be cancelled directly.

A.10.7 Finding Isolated Graphs of Blocked Actors

Detecting isolated graphs is done using a tri-colour algorithm that combines finding graphs with determining if the cycle's view of the graph topology indicates that the graph is isolated. The algorithm divides actors into three sets:

- The white set are actors that are candidates for collection. The cycle detector's view of the actor topology indicates that these actors have no incoming edges from unblocked actors.
- The black set are actors that cannot be collected. The cycle detector's view of the actor topology indicates that these actors have incoming edges from unblocked actors.
- The grey set are actors that have not yet been scanned for outgoing edges.

The algorithm preserves the strong tri-colour invariant [56]: there is no edge from a black node to a white node. To achieve this, all `view_t` structures begin coloured black. A `view_t` from the `deferred` map is selected as the entry point and placed on a processing stack with an incoming reference count of zero. Elements are popped from the stack and processed until the stack is empty.

For each element, when it is popped from the stack with some incoming reference count:

1. If the `view_t` has a deferred topology delta, it is applied.
2. If the `view_t` is in the `deferred` map, it is removed. This is the key step in allowing the `deferred` map to reduce the overall work performed by the cycle detector.
3. The incoming reference count on the stack is subtracted from the cycle detector's view of the actor's view of it's own reference count.
4. If the `view_t` is not coloured grey, then it is coloured grey and the elements of its `viewref` map are pushed on the stack. The outgoing reference count from the current actor is pushed as the incoming reference count on the processing stack.

When this process is complete, all actors reachable from the initially selected `view_t` are marked grey, including the initial actor. Another scan is performed, this time looking for `view_t` entries with zero reference counts. The process begins with the same initial `view_t` being pushed on the processing stack. Stack processing in this phase consists of:

1. If the `view_t` is not marked grey, it is skipped.
2. If the `view_t` has a reference count of zero, it is marked white.
3. Otherwise, the `view_t` is marked black, and a scan using a new stack is begun starting from that `view_t`. This scan marks each reachable `view_t` black.

Upon completion, all actors reachable from the initially selected `view_t` will be marked black or white. The actors marked white form a *perceived cycle* and are candidates for collection. The perceived cycle is stored as a `perceived_t`, as shown in figure A.21, and is added to the map of tokens to perceived cycles kept by the cycle detector.

If no actors are marked white (i.e. no perceived cycle is detected), the cycle detector will increase the threshold of deferred actors required before another cycle detection attempt is made. Similarly, if a perceived cycle is detected, the cycle detector will decrease the threshold. The result is an adaptive cycle detector that will collect more often when a program has short-lived actors and less often when a program has long-lived actors.

A.10.8 Confirmation Messages

When a perceived cycle is detected, confirmation messages are sent to the members of the perceived cycle. Rather than send confirmation messages to all actors in the perceived cycle, an initial subset is sent messages. Additional messages are sent when acknowledgment messages arrive for a perceived cycle that has not been cancelled. This deferment reduces the number of confirmation messages sent when a perceived cycle is cancelled.

Confirmation messages carry a token identifying the perceived cycle. Rather than use the perceived cycle's memory address, which could be reused if a perceived cycle were cancelled and a new one detected, a 64 bit integer is used as a monotonic counter. A perceived cycle is simply assigned the next integer.

A.10.9 Acknowledgement Messages

Actors respond to confirmation messages by echoing an acknowledgement message with the perceived cycle token back to the cycle detector. When the cycle detector receives an acknowledgement message, the token echoed back is used to look up the `perceived_t` that represents the perceived cycle in the `perceived` map. The acknowledgment count, `ack`, for the perceived cycle is incremented. In addition, if any actors in the perceived cycle have not yet been sent a confirmation message, an additional confirmation message is sent.

If the cycle detector receives an unblock message from any member of a perceived cycle, that cycle is cancelled, as in section A.10.6. If an acknowledgement message arrives for a cancelled cycle, it is ignored.

When a perceived cycle has received acknowledgement messages from all actors in the perceived cycle, it is a *true cycle*, and all actors in the cycle can be collected.

A.10.10 Actor Collection

Actors are collected by the cycle detector when a perceived cycle is fully acknowledged. Each actor in the cycle is marked as *pending destruction* and its finaliser is run, as described in section A.11. Following this, each actor generates release messages for actors in its foreign map that are not marked as pending destruction. This avoids sending unnecessary release messages within the cycle, while correctly releasing held reference count for actors and objects outside the cycle.

Each actor is then destroyed, freeing everything allocated on its heap as well as the heap structure itself, any garbage collection data such as local and foreign maps, and the remaining stub message in its message queue. Importantly, before destruction of some actor α can proceed, it is necessary to wait in a tight loop until the message queue head for α has had its lower bit marked, indicating that it is empty, as described in section A.6.2. This is required because it is possible for α to send a block message but not yet have marked its queue as empty in some thread t_1 while the cycle detector is collecting it in another thread t_2. Waiting until the queue is marked empty prevents α from failing to mark its queue as empty because the queue has itself been collected. Such a failure would result in spuriously rescheduling α for execution after it has been destroyed.

This race condition can occur even though α marks its queue as empty immediately after sending the block message.

A.11 Finalisation

The runtime supports both actor and object finalisation. This was a difficult decision to take, as finalisation can cause significant problems, such as resurrection. In addition, the non-deterministic finalisation that occurs in a garbage collected environment interacts poorly with releasing non-memory resources, which is the most common task of a finaliser.

However, for interacting with non-memory system resources, a finaliser provides a guarantee that the resource will be *eventually* released, which is critical. Designing interfaces that also encourage *explicit* resource release, using a `dispose` method, allows the programmer to achieve deterministic resource management, which combines well with a finaliser that acts as a backstop.

Pony provides a less problematic finalisation sequence by using a combination of reference capabilities and pessimistic static analysis. Finalisers on both actors and objects run with a `box` capability, with no arguments. As a result, finalisers cannot cause resurrection by mutating the object graph: the object being finalised can reach only itself and the transitive closure of objects reachable via its fields, and due to viewpoint adaptation has no write access to any part of that graph. It is possible for finalisers to *create* new objects, and to mutate those objects, but such objects cannot outlive the finaliser unless they are sent in a message to another actor, so they do not result in resurrection.

It is not only new objects that could outlive the finaliser if sent in a message: any part of the object graph could be sent as a `tag` in a message, resulting in resurrection. To curtail this, pessimistic static analysis is used when compiling finaliser methods. Any expression that could result in sending a message is prohibited in a finaliser. This is extremely strict, as it prevents method calls on objects with an `interface` or `trait` type, as the concrete implementation of the method is not known at compile time. However, in practice, objects and actors being finalised should be freeing non-memory resources held by the instance itself, rather than interacting with other parts of the objects graph (which can be independently finalised if necessary). As a result, this restriction has not so far proven to be problematic in the standard library or in end-user code.

A.12 Scheduler

The Pony runtime uses a work-stealing scheduler built around single-producer multi-consumer queues of actors with pending work. The runtime can be started with any number of scheduler threads, defaulting to the number of cores on the machine. Threads are pinned to cores, and, on a NUMA machine, thread stacks are allocated on the local NUMA node.

A scheduler thread has a queue of actors, described in section A.12.1, a message queue that is used to detect quiescence, described in section A.12.3, and a victim indicator that is used for round-robin work-stealing, as described in section A.12.2.

Each scheduler thread also keeps a context, which holds per-thread runtime information that would otherwise be encoded as thread-local storage. This is done to improve performance on platforms where thread-local storage is less efficient.

A.12.1 SPMC Queues

Each scheduler has a single-producer multi-consumer queue of actors that may have pending work. These queues are effectively the inverse of actor message queues. They are single-producer because only the owning scheduler thread will place work (in the form of an actor that may have pending work) on the queue, but any scheduler thread can remove work from the queue, in order to enable work-stealing, as detailed in section A.12.2.

```c
struct scheduler_t {
  // These are rarely changed.
  pony_thread_id_t tid;
  uint32_t cpu;
  uint32_t node;
  bool terminate;
  bool asio_stopped;

  // These are changed primarily by the owning scheduler
      thread.
  __pony_spec_align__(struct scheduler_t* last_victim,
      64);

  pony_ctx_t ctx;
  uint32_t block_count;
  int32_t ack_token;
  uint32_t ack_count;

  // These are accessed by other scheduler threads. The
      mpmcq_t is aligned.
  mpmcq_t q;
  messageq_t mq;
};
```

Figure A.22: Scheduler thread state

```c
void ponyint_mpmcq_push_single(mpmcq_t* q,
  void* data) {
  mpmcq_node_t* node = POOL_ALLOC(mpmcq_node_t);
  atomic_store_explicit(&node->data, data,
    memory_order_relaxed);
  atomic_store_explicit(&node->next, NULL,
    memory_order_relaxed);
  mpmcq_node_t* prev = atomic_load_explicit(
    &q->head, memory_order_relaxed);
  atomic_store_explicit(&q->head, node,
    memory_order_relaxed);
  atomic_store_explicit(&prev->next, node,
    memory_order_release);
}
```

Figure A.23: Pushing to a scheduler queue

```
void* ponyint_mpmcq_pop(mpmcq_t* q) {
  mpmcq_node_t* cmp, xchg, next, tail;
  cmp = atomic_load_explicit(&q->tail,
    memory_order_acquire);
  uintptr_t mask = UINTPTR_MAX ^
    ((1 << (POOL_MIN_BITS +
      POOL_INDEX(sizeof(mpmcq_node_t*)))) - 1);
  do {
    uintptr_t aba = (uintptr_t)cmp & ~mask;
    tail = (mpmcq_node_t*)((uintptr_t)cmp & mask);
    next = atomic_load_explicit(&tail->next,
      memory_order_acquire);
    if(next == NULL)
      return NULL;
    xchg = (mpmcq_node_t*)((uintptr_t)next
      | ((aba + 1) & ~mask));
  } while(!atomic_compare_exchange_weak_explicit(
      &q->tail, &cmp, xchg, memory_order_acq_rel,
      memory_order_relaxed));

  void* data = atomic_load_explicit(&next->data,
    memory_order_relaxed);
  atomic_store_explicit(&next->data, NULL,
    memory_order_release);

  while(atomic_load_explicit(&tail->data,
    memory_order_acquire) != NULL)
    ponyint_cpu_relax();

  POOL_FREE(mpmcq_node_t, tail);
  return data;
}
```

Figure A.24: Popping from a scheduler queue

As seen in figure A.23, pushing to a scheduler queue is lock-free, wait-free, and requires zero atomic operations (aligned stores always being atomic on common architectures). However, in figure A.24 we see that popping from a scheduler queue is more complex. A compare-and-swap loop is necessary, to allow multiple schedulers to pop actors from a scheduler queue. This necessitates an ABA counter to avoid a situation where a CAS succeeds on queue node, but should fail, as the queue node has actually been popped and replaced on the queue. In addition, there is a second loop, where the data pointer of the queue node must be `NULL` in order to continue. This is because the pop operation frees the *previous* queue node. For that to be safe, any previous consumer must be finished reading the data pointer from the queue node. By setting the data pointer to `NULL` and testing for the previous data pointer being `NULL`, consumers can coordinate memory management without locks.

If this step were to happen on every queue pop, the coordination cost would be very high. However, it occurs only when a scheduler thread successfully steals work from another thread. Otherwise, a scheduler thread will only be coordinating with itself, and no cost will be paid.

Like message queues, it is important that scheduler queues are unbounded. An actor α executing on a scheduler thread t may generate an unbounded number of messages. In doing so, an unbounded number of previously unscheduled actors may need to be scheduled. A bounded scheduler queue would make this impossible, resulting in situations similar to those outlined for message queues in section A.6.1.

The SPMC queue uses a compare-and-swap loop to pop, and so it could suffer from CAS-failure related scalability problems on the x86/x64 architecture [45]. These problems could be reduced while maintaining an unbounded queue by using a linearizable concurrent ring queue (LCRQ), also in [45]. However, an LCRQ requires additional overhead in the form of an atomic fetch-and-add in addition to the CAS on pop, and requiring an atomic fetch-and-add and CAS on push, as well as requiring individual concurrent ring queues (CRQs) to be used as nodes in a Michael and Scott list. This overhead reduces CAS failures by reducing CAS contention. The Pony SPMC queues reduce CAS contention by requiring scheduler threads to pop from SPMC queues in a round robin fashion, as described in section A.12.2. This is an interesting research area, and it may be possible to use a combination of these techniques to further reduce contention.

A.12.2 Work-Stealing

If a scheduler thread's queue is empty, it will attempt to steal an actor from another thread's queue. The victim thread is selected in round robin fashion. Other selection algorithms, such as preferring scheduler threads running on cores in the same NUMA node or preferring scheduler threads running on cores with the same L3 cache, have not as yet performed as well as simple round robin selection.

Once a victim is selected, an actor is popped from the victim's scheduler queue. This actor is then processed by the stealing scheduler thread. If the victim's queue is empty, the stealing thread checks for quiescence (as detailed in section A.12.3). If the program is not quiescent, the stealing thread may pause in order to avoid excessive work-stealing attempts.

A scheduler thread's back-off pause length is based on the time since the scheduler thread last performed useful work. In practice, this is estimated with a processor cycle count when possible, using `RDTSC` on x86/x64, or a fast monotonic system clock if `RDTSC` is not available. This pause also allows the runtime to avoid spinning a core when there is no work to do, avoiding unnecessary energy expenditure.

A.12.3 Quiescence

Rather than waiting for the cycle detector to collect all actors in the program, the runtime determines quiescence by detecting when all scheduler threads have empty queues and no actors are awaiting asynchronous I/O (as detailed in section A.13). To do this without locks, scheduler threads have message queues: effectively, the scheduler threads are also actors.

When a scheduler thread t has an empty queue, before t attempts to steal work from another scheduler thread t', it sends a message to the the *coordinating thread* t_0 indicating that it is blocked. The coordinating thread is simply the first thread spawned by the runtime. Similarly, if t then successfully steals work, it will send a message to t_0 indicating that it has unblocked.

When t_0 has received block messages from all other scheduler threads and is itself blocked, it engages in the same CNF-ACK protocol that the cycle detector uses, sending CNF messages to the other scheduler threads and expecting ACK messages. If an unblock messages is received before all ACK messages (or if t_0 itself unblocks), the attempt at quiescence is cancelled.

If ACK messages from all scheduler threads are received, t_0 will attempt to shut down the asynchronous I/O thread. If this is not possible, due to actors still awaiting asynchronous I/O, then the attempt at quiescence is cancelled. Otherwise, another CNF-ACK series is run. The second CNF-ACK ensures that a scheduler thread that was woken up by asynchronous I/O before it was shut down has now reached quiescence.

At this point, the runtime can terminate without collecting the remaining actors. The quiescence protocol improves program termination performance by short-circuiting the actor garbage collection algorithm with a scheduler thread collection algorithm, which operates on fewer participants.

A.12.4 Cache Locality

When some actor α is sent a message and the actor is already scheduled on some scheduler thread t, α remains on t. This is because the working set of α is more likely to be in the cache on t than on any other thread. Conversely, if α is not currently scheduled on any thread, it is unlikely that the working set of α is in the cache of any core. In this case, α is scheduled on the thread where the message send takes place, as the message is in the cache of the sending core.

A.13 Asynchronous I/O

The Pony runtime uses a separate, non-scheduler, thread to transform operating system events into messages to the actors that will handle those events. The underlying mechanism is operating system dependent (`epoll` on Linux, `kqueue` on OSX and FreeBSD, and `IOCP` on Windows), but the approach is the same: a single thread is used to block on a unified view of pending events. When that thread is woken up, notification of pending events is dispatched to the appropriate actors. Importantly, the event itself is not handled in the non-scheduler thread.

When the asynchronous I/O thread sends a message to some actor α with an empty message queue, α must be scheduled. The asynchronous I/O thread cannot directly push α on to a scheduler queue, as they are single-producer queues. Instead, α is placed on a separate queue that is checked by all scheduler threads. This helps to reduce event response latency.

While the initial implementation of asynchronous I/O handling was intended for network activity, it has been extended to cover signals, timers, `stdin` input, and reading both `stdout` and `stderr` of any spawned child processes. Each of these functions as input in the sense that they represent events that are not directly generated by the program. Translating those events asynchronously to actor messages allows the runtime to treat external events in a unified way.

This approach is primarily used for *input* events. However, some *output* is also treated in an asynchronous manner. For example, socket writes that result in back-pressure result in the actor handling the socket ceasing to try to write to the socket until an event is received from the asynchronous I/O thread that indicates the socket is once again writeable.

File system access is not treated as fundamentally asynchronous in the runtime. The interfaces presented by operating systems for file system access are largely synchronous, and where asynchronous interfaces exist, they can be restrictive. In Pony, a synchronous operating system interface can easily be made asynchronous by wrapping it in an actor. On the other hand, this still results in a scheduler thread being used to call synchronous OS APIs. An adaptive scheduler that could create new scheduler threads when an existing thread was waiting on a kernel operation would address this, without requiring a flexible, high performance, asynchronous file system API to be provided by the operating system.

Bibliography

[1] http://icl.cs.utk.edu/hpcc/.

[2] https://github.com/actor-framework/benchmarks/.

[3] Gul Agha. Actors: a model of concurrent computation in distributed systems, series in artificial intelligence. *MIT Press*, 11(12):12, 1986.

[4] Gul Agha and Carl Hewitt. Concurrent programming using actors. In *Object-oriented concurrent programming*, pages 37–53. MIT Press, 1987.

[5] Martin Aigner, Christoph M Kirsch, Michael Lippautz, and Ana Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. *OOPSLA'15*, 50(10):451–469, 2015.

[6] Joe Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

[7] David F Bacon and VT Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP 2001—Object-Oriented Programming*, pages 207–235. Springer, 2001.

[8] Henry G Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *ACM Sigplan Notices*, 29(9):38–43, 1994.

[9] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *OOPSLA'17*, 1(OOPSLA):52, 2017.

[10] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[11] John Boyland. Checking interference with fractional permissions. In *Static Analysis*, pages 55–72. Springer, 2003.

[12] John Boyland, James Noble, and William Retert. Capabilities for sharing. In *ECOOP'01*, pages 2–27. Springer, 2001.

[13] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. *OOPSLA'04*, 50(8):35–48, 2004.

[14] Luke Cheeseman. Value-dependent types: Efficient, flexible containers in Pony. Master's thesis, Imperial College, 2016.

[15] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. *ECOOP 2003–Object-Oriented Programming*, pages 59–67, 2003.

[16] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. *Programming Languages and Systems*, pages 139–154, 2008.

[17] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM, 2015.

[18] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *VEE'05*, pages 46–56. ACM, 2005.

[19] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J Summers. Universe types for topology and encapsulation. In *Formal Methods for Components and Objects*, pages 72–112. Springer Berlin Heidelberg, 2008.

[20] Andrei de Araújo Formiga and Rafael Dueire Lins. A new architecture for concurrent lazy cyclic reference counting on multi-processor systems. *J. UCS*, 13(6):817–829, 2007.

[21] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP'09*, pages 363–377. Springer, 2009.

[22] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *TAPOS*, 5(1):3–24, 1999.

[23] Roland Ducournau and Floréal Morandat. Perfect class hashing and numbering for object-oriented implementation. *Software: Practice and Experience*, 41(6):661–694, 2011.

[24] Jason Evans. Scalable memory allocation using jemalloc. *Notes by Facebook Engineering*, 2011.

[25] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2009.

[26] Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA'12*, volume 47, pages 21–40. ACM, 2012.

[27] Isaac Gouy. The computer language benchmarks game. *http://icl.cs.utk.edu/hpcc/*.

[28] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI'02*, volume 37, pages 282–293. ACM, 2002.

[29] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[30] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP'10*, pages 354–378. Springer, 2010.

[31] Norman Hardy. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[32] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[33] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[34] Richard E Jones and Rafael D Lins. Cyclic weighted reference counting without delay. In *International Conference on Parallel Architectures and Languages Europe*, pages 712–715. Springer, 1993.

[35] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. *Garbage collection of actors*, volume 25. ACM, 1990.

[36] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *ISMM'13*, volume 48, pages 63–74. ACM, 2013.

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[38] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[39] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01*, volume 36, pages 367–380. ACM, 2001.

[40] Rafael Dueire Lins. Lazy cyclic reference counting. *J. UCS*, 9(8):813–828, 2003.

[41] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In J. Halpern, editor, *5th Annual ACM Symposium on the Principles on Distributed Computing*, pages 29–39, Calgary, August 1986.

[42] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. Towards advanced debugging support for actor languages. *AGERE'16*, 2016.

[43] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171. IEEE, 1994.

[44] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In *Proceedings of the Dynamic Languages Symposium*, volume 52, pages 120–131. ACM, 2016.

[45] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[46] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized Javascript. *Google, Inc., Tech. Rep*, 2008.

[47] Mark S Miller, E Dean Tribble, and Jonathan Shapiro. Concurrency among strangers. In *International Symposium on Trustworthy Global Computing*, pages 195–229. Springer, 2005.

[48] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[49] Mark Samuel Miller and Jonathan S Shapiro. *Robust composition: towards a unified approach to access control and concurrency control.* PhD thesis, Johns Hopkins University, 2006.

[50] Luc Moreau, Peter Dickman, and Richard Jones. Birrell's distributed reference listing revisited. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1344–1395, 2005.

[51] Luc Moreau and Jean Duprat. A construction of distributed reference counting. *Acta Informatica*, 37(8):563–595, 2001.

[52] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *PPoPP'13*, volume 48, pages 103–112. ACM, 2013.

[53] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in ambienttalk. *Software: Practice and Experience*, 39(7):661–699, 2009.

[54] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *POPL'12*, 47(1):557–570, January 2012.

[55] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness, and immutability. *Objects, Components, Models and Patterns*, pages 178–197, 2008.

[56] Pekka P Pirinen. Barrier techniques for incremental tracing. In *ISMM'98*, volume 34, pages 20–25. ACM, 1998.

[57] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, pages 211–249. Springer, 1995.

[58] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[59] Helena C. C. D. Rodrigues and Richard E. Jones. Cyclic distributed garbage collection with group merger. pages 249–273. Also UKC Technical report 17–97, December 1997.

[60] Helena C.C.D. Rodrigues. *Cyclic Distributed Garbage Collection.* PhD thesis, 1998.

[61] Jonathan S Shapiro and Norman Hardy. EROS: A principle-driven operating system from the ground up. *IEEE software*, 19(1):26–33, 2002.

[62] Jonathan S Shapiro, Eric Northup, M Scott Doerrie, Swaroop Sridhar, Neal H Walfield, and Marcus Brinkmann. Coyotos microkernel specification. *The EROS Group, LLC, 0.5 edition*, 2007.

[63] JS Shapiro, Swaroop Sridhar, and MS Doerrie. BitC language specification, 2008.

[64] Clay Spinuzzi. The methodology of participatory design. *Technical communication*, 52(2):163–174, 2005.

[65] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.

[66] George Steed. A principled design of capabilities in Pony. Master's thesis, Imperial College, 2016.

[67] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. *ISMM'11*, 46(11):79–88, 2011.

[68] E Dean Tribble, Mark S Miller, Norm Hardy, and David Krieger. Joule: Distributed application foundations. *Online at http://www. agorics. com/-joule. html*, 1995.

[69] Tom Van Cutsem. Ambient references: Object designation in mobile ad hoc networks. *Programming Technology Lab, Faculty of Sciences, Vrije Universiteit Brussel*, 2008.

[70] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*, pages 3–12. IEEE, 2007.

[71] Abhay Vardhan and Gul Agha. Using passive object garbage collection algorithms for garbage collection of active objects. In *ISMM'02*, volume 38, pages 106–113. ACM, 2002.

[72] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *OOPSLA'01*, 36(12):20–34, 2001.

[73] Jan Vitek and R Nigel Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *International Conference on Compiler Construction*, pages 309–325. Springer, 1996.

[74] Esther Wang and Jonathan Aldrich. Capability safe reflection for the wyvern language. In *Workshop on Meta-Programming Techniques and Reflection*, 2016.

[75] Wei-Jen Wang. Conservative snapshot-based actor garbage collection for distributed mobile actor systems. *Telecommunication Systems*, 52(2):647–660, 2013.

[76] Wei-Jen Wang, Carlos Varela, Fu-Hau Hsu, and Cheng-Hsien Tang. Actor garbage collection using vertex-preserving actor-to-object graph transformations. In *Advances in Grid and Pervasive Computing*, pages 244–255. Springer, 2010.

[77] Wei-Jen Wang and Carlos A Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Advances in Grid and Pervasive Computing*, pages 360–372. Springer, 2006.

[78] Wei-Jen Wang and Carlos A Varela. A non-blocking snapshot algorithm for distributed garbage collection of mobile active objects. Technical report, Technical Report 06-15, Dept. of Computer Science, RPI, 2006.