IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Modelling ORCA and the Concurrent Garbage Collection in Pony

MENG THESIS

JOINT MATHEMATICS AND COMPUTER SCIENCE

Daniel SLOCOMBE

Supervisor
Sophia DROSSOPOULOU

Second Marker
Anthony FIELD

June 2018

Abstract

Pony is a performant, actor-based programming language with a type system that statically guarantees freedom from data races. Pony uses a novel garbage collection algorithm *ORCA* designed to leverage the type safety of Pony.

It is a crucial to have a model for ORCA. We have proofs that the actors in Pony are free from data races, but this is meaningless if the memory they are reading from could be trashed. A soundness proof for ORCA guarantees memory safety in the langauge and any programs written in it. The current model $ORCA^0$ is complex and misses key optimisations, we address both of these problems in this thesis.

The model $ORCA^0$ is unsatisfactory for two main reasons.

1. The algorithm accounts for fully concurrent actor execution, and contains invariants that are preserved under each step of this execution model. But this results in many technical and syntactical definitions, obscuring a lot of intuition.

2. The model is a composition of many disparate elements. $ORCA^0$ contain a complete theory for the accessibility of objects, reference counts and coherence of the workings sets. This means that the definitions are cluttered and the proofs difficult to follow.

We present $ORCA^{\text{Ghost}}$, a model of ORCA that adds a ghost state used to addresses our first issue. This gives rise to some complexity but also simpler definitions making the model easier to reason about. On top of this we define $ORCA^{\text{Ghost+Val}}$ which includes the missing optimisations. We give a proof of soundness of $ORCA^{\text{Ghost+Val}}$ and completeness up to object cycles.

We tackle the second problem by building a submodel - *CALF* - that reasons only about the reference counts and how they are altered by execution. We then take a novel approach to proving properties on *CALF* . Inspired by linearisability theory, we break up actions the model can perform into *events* and look at how the events by different actors are interlaced. Then at a configuration, we reason about what properties would hold if all ongoing actions were finished. This gives us an alternate solution to the problem of fine grained concurrency, and we look at other problems this technique could be applied to.

Finally we define a relation between a larger model $ORCA^1$ and *CALF* . This forms a bisimulation between them and we look at what we can prove about $ORCA^1$ by its relation with *CALF* .

Acknowledgements

The project was made possible by the dedication of my supervisor Sophia Drossopoulou. The weekly meetings we had were invaluable, and her feedback and patience allowed me to develop my ideas into what I present here. I would also like to thank Juliana Franco for her insight into the project, and Toby Shaw and Csongor Kiss who were willing to discuss my ideas. The thesis itself was much improved by the generous feedback from Paul Liétar and Shinu Cho, and the 24-hour moral support from Michael Radigan.

Finally I would like to thank my family and friends who have supported me through my four years at Imperial. In particular, my parents, Kate and Lottie, and Tom Grigg, Dominic Englebright and Leosoc to whom I credit most of my sanity.

"I warned Garth [Marenghi], I said, 'I'm not an actor'. And he said that he didn't want an act, he wanted the truth."

- Dean Learner

Contents

1 Introduction

The goal of modern language design is to aid the programmer and allow them to focus on higher level logic free from the conceptual burden of reasoning about underlying systems.

Many programming languages, such as C [Kernighan. and Ritchie, 1988] allow the programmer to directly allocate, manipulate and free memory. However, this unchecked access opens up a wide range of potential programming errors. Accessing unallocated memory or trying to free the same memory address twice may cause trigger an error from the operating system, corrupt data, or even expose a security vulnerability [Seacord, 2005].

Garbage collection is one approach moving responsibility of memory management onto the programming language. The language runtime is given responsibility of creating and managing memory, hidden from the programmer. An example of such a language is Java [Gosling et al., 2014] Using a garbage collected language gives up fine control of allocations and can often have significant performance overheads, but the freedom and safety it provides have caused garbage collection to be widespread in languages used today [Chen et al., 2005].

With the decline of Moore's law the computer hardware industry has moved from single cored CPUs to multiple cores. Because of this, concurrency is becoming more and more relevant when designing performant programs. However, if concurrency is implemented poorly there are many pitfalls the naive programmer can succumb to. There is a whole class of of data races that can occur when the same piece of data is accessed simultaneously in two different contexts. This often makes concurrent programming very difficult in traditional languages like C and Java mentioned earlier. Fortunately, language design has evolved and there are languages such as Rust that promise to eradicate these bugs simply by construction of the language and type system [Klabnik and Nichols, 2018].

Pony [Pony, 2018] is a language that follows the *actor paradigm* [Hewitt et al., 1973], consisting of many *actors* with strictly sequential logic but executed in parallel. The actors then communicate only through asynchronous message passing. Pony is also a performant language with a static type system that guarantees freedom from data races. The type system uses a *capabilities* system to restrict which actors are able to read and mutate data.

The Pony runtime uses the ORCA protocol, a concurrent, reference counting garbage collector. ORCA does not use *stop-the-world* or block execution. Instead ORCA has actors garbage collect independently and synchronises only through message passing. ORCA gives each actor a heap of objects and the responsibility of collecting that heap. Each actor keeps a set of reference counts for both the owned objects that foreign actors have access to, and the foreign objects that they have access to. Reference counts are updated during message sending, message receipt, and garbage collection. Because of the strong guarantees from the Pony type system it is often known that an object is immutable. The Pony runtime leverages this by keeping a single reference count to represent the whole structure of an immutable object graph.

There is an existing model for ORCA, $ORCA^0$ [Franco et al., 2018] is complex. $ORCA^0$ describes the algorithm and then proves invariants over its execution. These invariants have to hold under fine grained concurrent execution, so many of the ideas are muddied by the technical definitions.

1.1 Contributions

We first present $ORCA^{\text{Ghost}}$, a reformulation of $ORCA^0$ designed to be easier to reason about.

We then extend this to $ORCA^{\text{Ghost+Val}}$ including the immutable object optimisations. This model does not follow the Pony runtime exactly, and provides an alternate approach to implementing the same optimisation. We thus give an evaluation of the performance difference between the model and the actions taken in the current Pony implementation.

In both $ORCA^{\text{Ghost}}$ and $ORCA^{\text{Ghost+Val}}$ we introduce auxiliary concepts to reframe and generalise existing invariants. This allows us to keep the "structure" of the proofs the same. Using this we show that $ORCA^{\text{Ghost+Val}}$ is sound.

Further, we show that in $ORCA^{\text{Ghost+Val}}$ the only time a memory leak can occur is when there is a cycle in the structure, proving completeness up to object cycles.

Finally we revisit $ORCA^0$ from a more abstract setting. We build a submodel *CALF* focused entirely on the reference counts of a configuration of $ORCA^0$. We then split up actions take into *stories* composed of fine-grained *events* that are interleaved during execution. By looking at the histories formed by an execution we see that they are not directly linearisable, but by taking inspiration from work done on linearisability we make arguments about the *closure* of configurations and histories. If we are concerned with an object we reason about what would hold if all the currently executing events that effect the object were to finish. We define invariants that hold at closures and use these to prove properties in the larger model. This allows us to reconstruct the main results of $ORCA^0$ without much of the tedium. We look at some general results of this form an ask whether it is useful in other contexts.

2 Background

2.1 Actor Languages

The actor paradigm tries to provide safe concurrency by denying shared mutability but instead allowing threads to send asynchronous messages to each other. An individual actor is run sequentially on a single thread, but in a program multiple actors will be executed simultaneously with nondeterministic interleaving. Actors define a set of *behaviours* which can be triggered by a message from another actor.

Actors can also make synchronous calls to their own functions or to objects they own. Each actor has a single message queue that is processed first-in first-out, sequentially [Hewitt et al., 1973] [Greif, 1975].

2.2 Pony

Pony is a object-oriented, actor-based, programming language. Pony is equipped with a formally verified type system that gives the following guarantees: if a program compiles, the program will never crash, deadlock or encounter a data race [Pony, 2018].

Pony uses *capabilities* in its type system to ensure freedom from data races. From the pony website:

> A capability is an unforgeable token that:
>
> 1. Designates an object
> 2. Gives the program the authority to perform a specific set of actions on that object.

This system is used to achieve several goals:

- Prevent shared mutability that could cause concurrent access.
- Allow sharing immutable data, which is safe.
- Allow sending of isolated, mutable data; this is safe because the type system ensures that the sender gives up control of the data.

Every variable and field is assigned, along with the type of a class, a capability. We will now describe all of the capabilities except *transition* and *box*, which we will not need.

- **Iso** - *Isolated object*. If a variable is **Iso** then there are no other variables that can reference the object. This means it can be mutated and we know that it cannot form a race condition.
- **Val** - *Value*. If a variable is **Val** then it is immutable. This means that an actor is free to read its value and share it with other actors without forming a data race.
- **Ref** - *Reference*. References are used for mutable data that is not isolated. References cannot be sent to other actors as they could cause a data race, so all references to a given object must only be accessible from a single actor.
- **Tag** - *Tag object*. Tag is the least permissive capability, denying both reads and writes. However this does mean that we can safely send a variable with tag capability. While we cannot inspect the values of the variable, we are still able to perform pointer equality checks between two variables.

[Liétar, 2017] gives a formal model of the Pony language and type system.

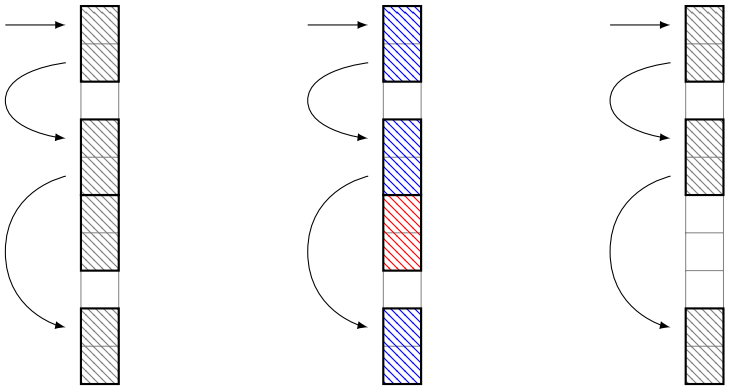


Figure 1: Mark and Sweep, left shows the initial heap, centre shows the traced heap and right shows the resultant heap.

2.3 Garbage Collection

During program execution, we generally have a heap of allocated memory, occupied by data from our program. We will refer to individual items within this heap as *objects*. In real-world machines this space will be finite, so if we are to execute a program with a potentially unbounded number of new object allocations, we will need some process to remove unused or unneeded objects. Our program will be able to reference some of these objects, and we allow objects themselves to reference other objects. We will use the term *garbage* to refer to objects that are no longer accessible to the program via a *path* of references from the variables. *Garbage collection* is the automated process of removing or reclaiming garbage.

In his paper on the implementation of the Lisp programming language, McCarthy describes the process of memory reclamation. The system holds a list of free registers to allocate objects to. When this list is exhausted, the system begins a trace of its memory. Starting at the "base registers", it walks recursively through the objects, marking them as accessible. The system can then add all registers not marked as accessible back onto the free register list[McCarthy, 1960]. This forms the most primitive form of the *Mark and Sweep* approach to garbage collection.

2.3.1 Mark and Sweep

If we consider program variables and objects as a directed graph and take the variables as *roots* of the graph, then we say that a node is reachable if there is some path through the directed edges to that node.

The Mark and Sweep approach to collection involves breaking the process down into two phases. The *mark phase*, involves a traversal of the above defined graph starting at its roots and marking all reachable nodes [Zorn, 1990] . The *sweep phase* can then act to reclaim all unmarked nodes. Optionally, a *compaction phase* can then be performed that reallocates the live objects to make future allocation easier.

No matter the chosen algorithms for the marking phase, sweeping the heap requires accessing all possible objects starting from the lowest memory address and iterating up to the highest.

2.3.2 Copying Collection

Copying divides the heap into two heaplets, a *from-space* and a *to-space*. During collection, the system traverses the from-space and builds an isomorphic copy of the traversed subgraph in the to-space. The new copy should be compact and occupy a contiguous block[w. Appel,]. While this approach requires copying all reachable objects, it means that we do not have to scan through the

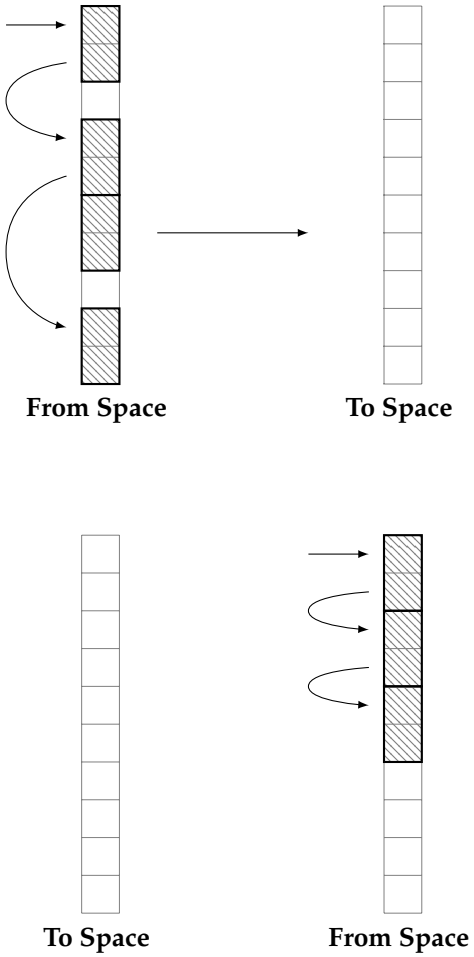


Figure 2: Copying Collection, top shows the initial state of the two heaplets and below shows the result with an isomorphic copy of the reachable subgraph. Then the two spaces are swapped over which is seen in the relabelling.

entire heap. After a copying collection cycle, the to-space and from-space swap roles. This is illustrated in figure 2.

A technique used by the JVM (Java Virtual Machine) is generational collection, where the heap is further divided into several regions for objects that have survived various collection cycles. This follows the heuristic that "young" objects are more likely to be collected, so collections on these heaps is performed more frequently.

2.3.3 Reference Counting

Instead of performing a search to determine what is reachable, reference counting collection instead keeps track of how many pointers exist to a given object, known as its *reference count* and stored inside the object [w. Appel,].

Most implementations operate by adding operations to variable assignments in order to deal with this bookkeeping. If we want to assign ω to $x.f$ we both need to decrement the reference count of the current object assigned to $x.f$ and then increment the count for ω. This can be very expensive, especially in concurrent systems where atomic reference counting is required.

If the reference of an object should ever become zero then it is unreachable so can be added to the free memory list and all of its fields can have their reference counts decremented.

The main problem with reference counting is dealing with cycles. If we have, for instance an object that references itself through one of its fields then even if that object is unreachable from every other point, its reference count will remain one, and so the object will not be collected. A solution to this is to require the programmer explicitly deal with cycles before they can assume an object will be collected. This is not as bad as explicit free calls but is a unsatisfactory solution for something that should automate the process of memory reclamation. Alternatively, reference counting could be supplemented with an occasional mark and sweep collection phase.

2.3.4 Concurrent Garbage Collection

In the most general scenario, we can view garbage collection as a graph problem. We consider our heap as a directed graph with some set of root nodes corresponding to local variables and other nodes as objects. A node is *reachable* if there exists a path from a root node to that node. Non-reachable nodes are *garbage nodes*.

[Dijkstra et al., 1978] first proposed a concurrent mark and sweep garbage collection scheme. They split the general description into a two parts. The *mutator* that alters the graph in one of a number of allowable ways. And a collector that can mark the reachable nodes and append them to a free list, but perform no other action that could effect the structure of the graph. They proposed a solution that would allow the mutator and collector to run in parallel with the smallest amount of synchronisation possible. They allow the mutator to perform the following actions:

1. Redirect an outgoing edge of a reachable node to an already reachable one.
2. Redirect an outgoing edge of a reachable node to an unreachable node with no outgoing edges.
3. Add an outgoing edge to a reachable node, pointing towards an already reachable one.
4. Add an outgoing edge to a reachable node, pointing towards a a fresh, but unreachable node with no outgoing edges (used to model allocation of a new object.)
5. Remove an outgoing edge from a reachable node.

Actions (1), (2) and (5) can cause nodes to become unreachable, garbage nodes.

The following example is given to show that some change to the mutator is necessary and that some overhead is required to facilitate concurrent collection. If we have three objects A, B, and C with A and B root nodes and initially with one edge from A to C:

1. B adds an edge to C
2. A removes its edge to C
3. A adds an edge to C
4. B removes its edge to C

Then it is possible that when tracing A, we find that A has no outgoing edges. Then tracing moves to B, where due to concurrent scheduling, B has no outgoing edges.

In order to combat this, marking is extended to tagging objects with one of three colours: black, grey and white. At the start of a gc cycle, all objects are reverted to white and then during collection nodes are only darkened. When the mutator acts on the object graph by redirecting

nodes it also shades the new targets of arcs. This two staged approach allows the collector to run simultaneously with the mutator but with a performance overhead.

Actor languages often use a variant of concurrent garbage collection. Akka [Akka, 2018] is an actor framework based on the JVM. Erlang [Erlang, 2018] uses an variant of the actor model where all objects are immutable. It then assigns each actor a heap where it keeps all of its own objects. When an object is sent, it is deeply copied onto the heap of the receiving actor. This means, as no object has any shared mutability, all collection can be done independently and concurrently by each actor on their own heap without risk of data races.

2.3.5 Correctness of Garbage Collection

Again we consider that collection is a problem on a directed graph, where we have a formal model of a mutator and a collector. We can prove correctness with three properties [Dijkstra et al., 1978]:

- Soundness - everything collected is garbage.
- Completeness - all garbage is eventually collected.
- The collector does not make any modifications to edges of reachable nodes in the graph. Note that this is satisfied in a copying collector, as it would preserve an isomorphic copy of the reachable subgraph.

2.4 ORCA Host Language

ORCA is a garbage collection protocol for actor oriented languages, designed to be parametric on any host language meeting some base assumptions. First we will describe the features a host language should have, all of which are present in Pony, which will be the target of later discussion.

A host language should conform to the actor paradigm by having both actors and objects, with objects simply being static data structures, not necessarily instances of classes. These objects need not contain more than a set of fields containing references to other objects.

Actors should contain a set of fields, set of synchronous methods, and a set of asynchronous behaviours. ORCA deals only with the collection of objects so we can assume that all actors have references to all other actors and may call behaviours on them. Calling a behaviour on an actor with some object involves sending a message with the behaviour and the parameter objects; doing so will effectively share or send the object to the target actor. Each actor has a queue of messages that can contain either these *application* messages or protocol-level ORCA messages. This queue is processed by the actor in a strict order for reasons discussed later.

An actor is either idle, executing a behaviour, or performing garbage collection. When idle an actor processes the top message of its message queue. This also ensures that an actor can never execute multiple behaviours concurrently.

ORCA assumes that the language has a type system that associates each field an access 'capability': **read**, **write** or **tag**. **write** allows both read and write access to the object , **read** allows only immutable access. **tag** gives only opaque access to the object, enough to make simple equality checks but without being able to examine it.

We define a path as a nonempty list of fields starting from an object or actor. We say that an object, ι is *reachable* from another object or an actor ω if there exists a path from that ω to ι.

We say that ι is *accessible* from ω if the capabilities on the path allow ω to access ι. Thus accessibility implies reachability but not necessarily the reverse.

ORCA uses messages to send protocol-related information, and because of this we require message sending in the language to be *causal* - messages must be delivered after any and all messages that caused them. We say If α is an actor and α receives a message m then afterwards, a message m', we say $Causes(m, m')$. If α sends a message x and then a message x', we also have $Causes(x, x')$.

Message passing should be the only way to share objects in the language, and message sending should comply with the capability system. When an actor shares an object to another actor, then either the first gives up all access to the object, or neither can modify the object. In the language we must ensure that heap mutation only decreases accessibility while message sending can transfer accessibility from sender to receiver.

2.5 ORCA Protocol - An Informal Description

The aims of ORCA are to compliment those of Pony and to form an idiomatic garbage collection protocol. While ORCA uses reference counting to decide whether an object is collectable these reference counts are local to each actor and only updated on message sending, receival and garbage collection. This removes some of the mentioned shortcomings of reference counting which gives a performance hit on expression evaluation. The reference counting does require communication between actors about changes to these reference counts but this is done by asynchronous message passing rather than by stopping execution. Reference counts on actors do not have to use atomic operations because they are only ever modified by the actor that controls them.

Actors are given a local heap that they assign objects to. An object is said to be *owned* by an actor if it exists on that actor's heap. Ownership cannot be modified but actors may have references to unowned objects. Each actor is then responsible for collecting objects on its own heap, so only the owner of an object may free it.

All actors keep two kinds of reference counts, to objects that they own and have sent other actors, and to objects they do not own which have been sent themselves. The local reference count of an object is the reference count of the owner to the object, which should be kept equal to the sum of the foreign reference counts - the reference counts from other actors. Then when an actor can no longer reach an object it owns and it has a local reference count of zero, the actor knows no other actor has a reference count so the object is safe to collect.

Specifically, an object is *collectable* if it is not reachable from its owner, and the owner's reference count for the object is zero. If an actor α has a reference to an non-owned object ω that it then drops, it is obligated to send an update to the owner of ω that there is one fewer foreign references. Similarly, if α sends the reference on to some other actor, it must tell the owner that there is one more foreign reference. These updates are done with *ORCA messages* that are processed separately to regular, application messages.

An example situation is shown in Figure 3. We have two actors α and α', and some object ι owned by α and α has a reference to ι. In this initial configuration, without any reference counts, ι is safe from collection as it is directly referenced by its owner. Now if α sends ι to α' and drops its own reference to the object, the act of sending will increase α's reference count for ι to 1, to represent that another actor has a reference and to prevent collection. When α' receives ι it increments its reference count to one.

Now, if α' replaces its reference to ι by a reference to a new object ι' nothing happens immediately. If α was to perform collection now it would not collect ι as garbage as it still has a positive reference count. However if α' performs collection it will trace its references and see that while it has a positive reference count for ι it is not reachable. It then zeros that reference count and sends an ORCA message to its owner, α, telling it to reduce its reference count by one. After α receives this message its reference count to ι will be zero, meaning that a collection cycle will reclaim ι.

We can see now why we require the message delivery of the language to be causal. If we could reorder incoming messages, it would be possible for a reference count for an object to dip below zero and then come back to a positive value. This is dangerous because during the period where the reference count was zero the object could have been collected.

It is a design decision of ORCA that collection does not interrupt the execution of behaviours of an actor. Throughout we will assume that behaviour execution of actors is finite.

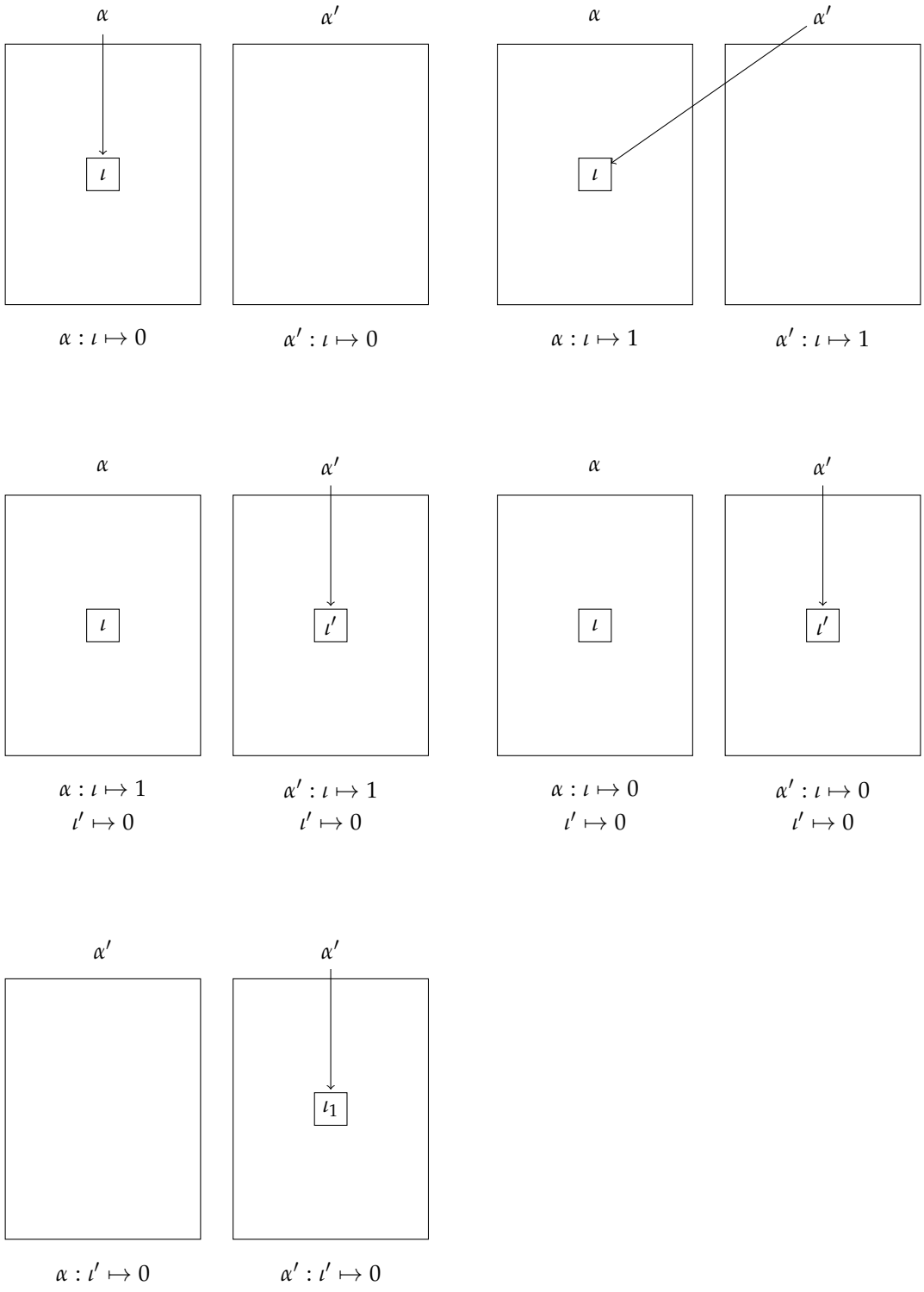


Figure 3: Example reference counting

2.6 ORCA Val Optimisation

ORCA requires tracing objects on sending and on receipt, so if we repeatedly send a complex object we will incur a significant performance cost.

Because immutability is deep in our host language, if we have an immutable object we know that none of its fields can change value, and we know that no paths from it can change value.

Because of this, we are able to optimise the sending and receiving of immutable objects. The current implementation of ORCA in the Pony runtime uses such an optimisation.

The runtime implements this by keeping track of which objects have been seen as immutable. Each reference count from an actor for an object has an attached field denoting whether the object has been seen as immutable from the actor. The first time it is seen immutable - sent or received with a `val` capability for example - a full trace is performed setting incrementing the message counts, and marking them as seen-immutable. For each object marked this way, a message is sent to the owner of the object, notifying them that the object is immutable.

Intuitively owners trace their own immutable objects and when an object graph crosses an ownership boundary, the owner of the child object is relied on to perform the tracing [Clebsch et al., 2017].

2.7 Order Theory

We recap some order theory as it will be used in some of our definitions and reasoning later on.

Definition 2.1. A *poset* or partially ordered set is a set X along with a binary relation $\leq$ that satisfies the following axioms:

1. Reflexivity: $\forall x \in X.\ x \leq x$
2. Antisymmetry: $\forall x, y \in X.\ x \leq y \wedge y \leq x \longrightarrow x = y$
3. Transitivity: $\forall x, y, z \in X.\ x \leq y \wedge y \leq z \longrightarrow x \leq z$

Definition 2.2. Let $(X, \leq)$ be a poset. a *chain* is a subset $S \subseteq X$ that is totally ordered with respect for $\leq$, i.e. for all elements $s, s' \in S$ either $s \leq s'$ or $s' \leq s$.

Definition 2.3. A chain $(S, \leq) \subseteq (X, \leq)$ is *maximal* if there is no other chain $(T, \leq)$ such that S is a proper subset of T.

Definition 2.4. Given two posets $\mathcal{X} = (X, \leq_X)$ and $\mathcal{Y} = (Y, \leq_Y)$ a function $f : X \to Y$ is *order-preserving* if:

$$\forall a, b \in X.\ a \leq_X b \longrightarrow f(a) \leq_Y f(b)$$

Definition 2.5. Let $\mathcal{X} = (X, \leq_X)$ and $\mathcal{Y} = (Y, \leq_Y)$ be two posets with order-preserving functions $\alpha : X \to Y$ and $\beta : Y \to X$. Then $(\mathcal{X}, \alpha, \beta, \mathcal{Y})$ forms a *galois connection* if:

1. $\alpha \circ \beta$ is *reductive* i.e. $\forall y \in Y, (\alpha \circ \beta)(y) \leq_Y y$
2. $\beta \circ \alpha$ is *extensive* i.e. $\forall x \in X, (\beta \circ \alpha)(x) \geq_X x$

3 $ORCA^0$

Here we present the original $ORCA^0$ model verbatim as described in [Franco et al., 2018], taking some of the examples given there. Our original work does not start until the next section. The reason we do this is because the rest of the project builds upon this model. Understanding it will give intuition into the decisions made later as well as give context for our evaluation.

Following the notation in the paper, we describe actors' actions through pseudocode procedures, which have the form:

procedure_name⟨α⟩:
  condition
→
{ instructions }

We start by defining the set of addresses as the disjoint union of actor addresses, object addresses and a single null address.

Definition 3.1 (Addr)**.**

$$Addr = ActorAddr \uplus ObjAddr \uplus \{Null\}$$

Definition 3.2 (Ownership)**.** An ownership function

$$\mathcal{O} : Addr \to ActorAddr$$

is one such that for all actors $\alpha \in ActorAddr$

this is defined such that for all actors $\alpha \in ActorAddr \mathcal{O}(\alpha) = \alpha$ and take some arbitrary value for $Null$

We will say that an object ι is *local* with respect to an actor α if $\mathcal{O}(\iota) = \alpha$ and say it is *foreign* otherwise.

We restrict ourselves to the following capabilities, which we assume are present in the host language

$$\kappa \in Capability = \{\mathsf{read}, \mathsf{write}, \mathsf{tag}\}$$

where write gives both read and write access to a object's fields, read gives read-only access and tag gives neither. So if an actor can access an object with a capability we have the following implications:

$$\mathsf{write} \longrightarrow \mathsf{read}$$
$$\mathsf{read} \longrightarrow \mathsf{tag}$$

We define an $ORCA^0$ runtime configuration $\mathcal{C}$ consisting of a heap, mapping addresses and field identifiers to addresses, and an actor map mapping actor addresses to actors.

Definition 3.3 ($ORCA^0$ configuration)**.**

$$\begin{aligned}
\mathcal{C} \in Config &= Heap \times Actors \\
\chi \in Heap &= (Addr \setminus \{\mathsf{null}\}) \times FId \to Addr \\
as \in Actors &= ActorAddr \to Actor \\
a \in Actor &= Frame \times Queue \times RefCountT \times State \times Workset \times Marks \times PC \\
\phi \in Frame &= \emptyset \cup (BId \times LocalMap) \\
\psi \in LocalMap &= VarId \to Addr \\
q \in Queue &= Message^\star \\
m \in Message &::= \mathsf{orca}(\iota : z) \mid \mathsf{app}(\phi) \\
\mathsf{rc} \in RefCountT &= Addr \to \mathbb{N}
\end{aligned}$$

We assume that multiple actors are executing concurrently at any point in program execution and individual actors transition between states as shown in Figure **??**.

Definition 3.4. $\mathcal{C}(\iota.f) \triangleq \mathcal{C}.\mathsf{heap}(\iota, f)$, and $\mathcal{C}(\iota.\overline{f}.f') \equiv \mathcal{C}.\mathsf{heap}(\mathcal{C}(\iota.\overline{f},)f')$

We introduce the distinction between *static paths* and *dynamic paths*. Static paths describe paths originating in the current actor or frame and dynamic paths include paths starting from messages. A static path is formed by the keyword this for a path starting from within the current actor, or a behaviour b and a local variable x indicating a path starting from a local variable in the context of a behaviour. This may then be followed by an arbitrary number of fields:

$$sp ::= \mathsf{this} \mid b.x \mid sp.f$$

This gives rise to the following axioms about the host language. **A1** and **A2** state that the capability of a static path is constant, so we have some strict type system in the language. **A3** and **A4** state that capabilities decrease along static paths, we cannot have a read capable path constructed from a path with only tag or a write capable path constructed from a path with read capability.

Definition 3.5. We use the typing judgement $\alpha \vdash_{\mathcal{C}} p : \kappa$ to denote that the actor α at the configuration $\mathcal{C}$ associates the capability κ to the path p.

Axiom 1. *For actors α, static path sp, field f, behaviour b, variable x, fields $\overline{f}$, capability κ, configurations $\mathcal{C}$ and $\mathcal{C}'$ which belong to the execution of the same program, we assume:*

A1 $\alpha \vdash_{\mathcal{C}} \mathsf{this}.\overline{f} : \kappa \iff \alpha \vdash_{\mathcal{C}'} \mathsf{this}.\overline{f} : \kappa.$
A2 $\alpha \vdash_{\mathcal{C}} b.x.\overline{f} : \kappa \iff \alpha \vdash_{\mathcal{C}'} b.x.\overline{f} : \kappa.$
A3 $\alpha \vdash_{\mathcal{C}} sp.f : \kappa \implies \exists \kappa' \neq \mathsf{tag}.\ \alpha \vdash_{\mathcal{C}} sp : \kappa'.$
A4 $\alpha \vdash_{\mathcal{C}} sp.f : \mathsf{write} \implies \alpha \vdash_{\mathcal{C}} sp : \mathsf{write}.$

Dynamic paths, or simply paths, are a superset of static paths as they can describe paths starting at an actor or a frame, but also they can describe paths starting at some message in an actor's message queue. The actor cannot access the paths from message queue objects but will be able to at some point in the future. We will use the notation $k.x$ where k is an integer $k \geq 0$ to refer to the k^{th} message on an actor's message queue and x is a variable in the frame contained in the message.

We will also later extend this for $k = -1$ which will be defined only while an actor is sending or receiving a message and refers to that message currently being processed.

$$p \in Path ::= lp \mid mp \qquad lp ::= \mathsf{this} \mid x \mid lp.f \qquad mp ::= k.x \mid mp.f$$

Accessibility is the partial function from configurations, actors and paths to the object and capability reached by the path.

Definition 3.6 (accessibility)**.** The partial function
$\mathcal{A} : Config \times ActorAddr \times Path \to (Addr \times Capability)$
is defined as

$\mathcal{A}_{\mathcal{C}}(\alpha, \mathsf{this}.\overline{f}) = (\iota, \kappa)$ iff $\mathcal{C}(\alpha.\overline{f}) = \iota \ \wedge \ \alpha \vdash_{\mathcal{C}} \mathsf{this}.\overline{f} : \kappa$
$\mathcal{A}_{\mathcal{C}}(\alpha, x.\overline{f}) = (\iota, \kappa)$ iff $\exists b.\psi.\ [\ \alpha.\mathsf{frame}_{\mathcal{C}} = (b, \psi) \ \wedge \ \mathcal{C}(\psi(x).\overline{f}) = \iota \ \wedge \ \alpha \vdash_{\mathcal{C}} b.x.\overline{f} : \kappa\ ]$
$\mathcal{A}_{\mathcal{C}}(\alpha, k.x.\overline{f}) = (\iota, \kappa)$ iff $k \geq 0 \ \wedge \ \exists b.\psi.\ [\ \alpha.\mathsf{qu}_{\mathcal{C}}[k] = app(b, \psi) \ \wedge \ \mathcal{C}(\psi(x).\overline{f}) = \iota \ \wedge \ \alpha \vdash_{\mathcal{C}} b.x.\overline{f} : \kappa\ ]$
$\mathcal{A}_{\mathcal{C}}(\alpha, -1.x.\overline{f}) = (\iota, \kappa)$ iff α is executing Sending or Receiving, and ... continued in Definition 3.24.
$\mathcal{A}_{\mathcal{C}}(\alpha, p.\mathsf{owner}) = (\alpha', \mathsf{tag})$ iff $\exists \iota.[\mathcal{A}_{\mathcal{C}}(\alpha, p) = (\iota, \_) \wedge \mathcal{O}(\iota) = \alpha'\ ]$

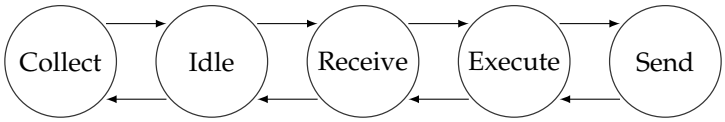


Figure 4: Actor state transitions, recreated from [Franco et al., 2018].

We use $\mathcal{A}_C(\alpha, p) = \iota$ as shorthand for $\exists\kappa.\ \mathcal{A}_C(\alpha, p) = (\iota, \kappa)$

Example 3.7. *$\mathcal{A}_{C_0}(\alpha_1, \mathsf{this}.f_1.f_2.f_3)$=$(\omega_3, \mathsf{write})$, and $\mathcal{A}_{C_0}(\alpha_2, \mathsf{this}.f_8)$=$(\omega_3, \mathsf{tag})$ give paths from actors' fields, while $\mathcal{A}_{C_0}(\alpha_2, x')$=$(\omega_8, \mathsf{write})$ gives a path from the actor's frame. $\mathcal{A}_{C_0}(\alpha_1, 0.x.f_5.f_7) = (\omega_8, \mathsf{tag})$ is a path from a message queue.*
Accessibility of objects determines what an actor can read or write. $\mathcal{A}_{C_0}(\alpha_1, \mathsf{this}.f_1.f_2.f_3) = (\omega_3, \mathsf{write})$, so actor α_1 can mutate object ω_3. However, this mutation is not visible by α_2, even though $C_0(\alpha_2.f_8)$=ω_3, because $\mathcal{A}_{C_0}(\alpha_2, \mathsf{this}.f_8) = (\omega_3, \mathsf{tag})$, which means that actor α_2 only knows of the existence of ω_3, not of the contents of its fields.
Accessibility plays a role in garbage collection: if the reference f_3 were to be dropped it would be safe to collect ω_4. Even though ω_4 is reachable from α_2, it is not accessible, as the path $\mathsf{this}.f_8.f_4$ leads to ω_4 but will never be navigated ($\mathcal{A}_{C_0}(\alpha_2, \mathsf{this}.f_8.f_4)$ is undefined). Further, $\mathcal{A}_C(\alpha_2, \mathsf{this}.f_8.\mathsf{owner}) = (\alpha_3, \mathsf{tag})$; thus, as long as ω_4 is accessible from some actor, e.g. through $C(\alpha_2.f_8)$=ω_4, actor α_3 will not be collected.

We assume the host language gives us freedom from data races as a result of the capabilities. This is characterised with the judgement $\vDash C\ \diamondsuit$, i.e. we say that C is free from data races if $\vDash C\ \diamondsuit$.

Definition 3.8 (Data-race freedom)**.** $\vDash C\ \diamondsuit \iff$
$\forall \alpha, \alpha', p, p', \kappa, \kappa'.$
$\quad \alpha \neq \alpha' \ \wedge\ \mathcal{A}_C(\alpha, p) = (\iota, \kappa) \ \wedge\ \mathcal{A}_C(\alpha', p') = (\iota, \kappa')$
$\quad \longrightarrow$
$\quad \kappa \sim \kappa'$
where we define
$\quad \kappa \sim \kappa' \iff [\ (\kappa = \mathsf{write} \longrightarrow \kappa' = \mathsf{tag}) \ \wedge\ (\kappa' = \mathsf{write} \longrightarrow \kappa = \mathsf{tag})\ ]$

3.1 Invariants

We now list the invariants of $ORCA^0$.

$\mathbf{I}_1$: if an object is accessible with write capability from an actor, then it is not accessible with read or write capability from any other actor.

This is guaranteed by the assumptions we put on the host language, and therefore, always holds in Pony. In particular, this holds from the Data-race freedom judgement $\vDash C\ \diamondsuit$.

$\mathbf{I}_2$: if an object is accessible via a message queue, or by a non-owner, then the owner's reference count is greater than zero.

Definition 3.9 ($\mathbf{I}_2$).

$$\mathbf{I}_2 \triangleq \forall \alpha, p, mp, \iota.\ \mathcal{A}_C(\alpha, p) = \iota \vee \mathcal{A}_C(\alpha, mp) = \iota \longrightarrow \mathcal{O}(\iota).rc_C(\iota) > 0$$

The notation $\mathcal{O}(\iota).rc_C(\iota)$ denotes the owner of ι's reference count at the configuration C for ι.

This means that when an actor has an inaccessible object with a reference count of zero they know that no actor can reach it, so it is safe to collect the object.

$\mathbf{I}_3$: if α can access an unowned object ι through a field or its call stack, then α's reference count for the object is greater than zero.

Definition 3.10 ($\mathbf{I}_3$)**.**

$$\mathbf{I}_3 \triangleq \mathcal{A}_C(\alpha, lp) = \iota \longrightarrow \alpha.rc_C(\iota) > 0$$

So actors must maintain positive reference counts for all foreign objects that are accessible to them.

Definition 3.11 (Local Reference Count)**.** The *Local Reference Count* of an object ι is defined as the reference count at the object's owner.

$$LRC_C(\iota) \triangleq \mathcal{O}(\iota).rc_C(\iota)$$

Definition 3.12 (Foreign Reference Count)**.** The *Foreign Reference Count* of an object ι is defined as the sum of the reference counts over all non-owning actors.

$$FRC_C(\iota) \triangleq \sum_{\alpha \neq \mathcal{O}(\iota)} \alpha.rc_C(\iota)$$

Definition 3.13 (Application Message Count)**.** The *Application Message Count* of ι is the sum across all message queues of behaviour invocations containing ι.

$$AMC_C(\iota) \triangleq \quad \#\{\ (\alpha, k) \mid k > 0 \wedge \exists x.\overline{f}.\mathcal{A}_C(\alpha, k.x.\overline{f}) = \iota\ \}$$

Definition 3.14 (Orca Message Count)**.** The *Orca Message Count* of ι, owned by α, is the sum of all increment and decrement messages sent by the ORCA protocol in α's message queue that refer to ι.

$$\mathrm{OMC}_{\mathcal{C}}(\iota) \triangleq \quad \sum_j \begin{cases} z & \text{if } \mathcal{O}(\iota).\mathsf{qu}_{\mathcal{C}}[j] = \mathsf{orca}(\iota : z) \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{I}_4$: We use these definitions to describe a relation between the distribution of reference counts throughout the system.

Definition 3.15 ($\mathbf{I}_4$)**.**

$$\mathbf{I}_4 \triangleq \forall \iota.LRC(\iota) + OMC(\iota) = FRC(\iota) + AMC(\iota)$$

Definition 3.16 ($\mathbf{I}_5$)**.** All reference counts are non-negative

$$\mathbf{I}_5 \triangleq \forall \alpha, \iota. \quad \alpha.rc_C(\iota) \geq 0$$

Definition 3.17 ($\mathbf{I}_6$)**.** Accessible paths don't dangle

$$\mathbf{I}_6 \triangleq \mathcal{A}_{\mathcal{C}}(\alpha, p) = \iota \longrightarrow \mathcal{C}.\mathsf{heap}(\iota) \neq \bot$$

3.2 Sending - $ORCA^0$

Sent objects are traced. Then for any object we find in the trace, if we own the object we increase the reference count by one. If we do not own an object in the trace, we decrease its reference count by one. This is done to preserve the equation of $\mathbf{I}_4$ when a message is sent.

If we would decrease an unowned object's reference count to zero, we instead set it to 256 and send an ORCA- level message to the owner notifying them of this increase. This means that we retain a positive reference count as we may still have a live reference to the object but we preserve $\mathbf{I}_4$.

```
1  α.st = EXECUTE && α.frame = (b, φ · φ′)
2  Sending<α>:
3  {
4    α.st := SEND
5
6    ws := trace_frame( α, (b, φ′))
7
8    for ι ∈ WS:
9      if 𝒪(ι) = α:
10       α.rc(ι) += 1
11     else if α.rc(ι) > 1:
12       α.rc(ι) −= 1
13     else:
14       𝒪(ι).qu.push(orca(ι:256))
15       α.rc(ι) := 256
16     ws := ws \ {ι}
17
18   α.frame := (b, φ)
19
20   α′.qu.push(app(b′, φ′))
21 }
```

3.3 Receiving - $ORCA^0$

When receiving an application message, we trace the incoming object and then if we own the object, we decrease its reference count by one, otherwise we increase it by one.

```
1  α.st = IDLE && top(α.qu) = app(ϕ)
2  Receiving<α>:
3  {
4    α.st := RECEIVE
5    ws := trace_frame(α, ϕ)
6
7    pop(α.qu)
8
9    for ι ∈ ws:
10     if 𝒪(ι) = α:
11       α.rc(ι) −= 1
12     else:
13       α.rc(ι) += 1
14     ws := ws \ {ι}
15
16   α.frame := ϕ
17
18   α.st := EXECUTE
19
20 }
```

3.4 Receiving orca message - $ORCA^0$

Upon receiving an orca message, we simply add the specified amount to our current reference count of the object.

```
1  α.st = IDLE && top(α.qu) = ORCA(ι : z)
2  Receiving<α>:
3  {
4    α.st := RECEIVE
5    α.rc(ι) += z
6    pop(α.qu)
7    α.st := EXECUTE
8  }
```

3.5 Garbage Collection - $ORCA^0$

```
α.st = IDLE && α.st = EXECUTE
GarbageCollection<α>:
{
  α.st := COLLECT
  ms := ∅

  // Marking as unreachable
  for ι such that 𝒪(ι) = α || α.rc > 0:
    ms := ms[ι → U]

  // Tracing and marking locally accessible as reachable
  for ι ∈ trace_this(α) ∪ trace_frame(α.frame):
    ms := ms[ι → R]

  // Marking owned and globally accessible as reachable
  for ι such that 𝒪(ι) = α && α.rc > 0:
    ms := ms[ι → R]

  // Collection
  for ι such that ms(ι) = U:
    if 𝒪(ι) = α:
      𝒞.heap := 𝒞.heap[ι → ⊥]
      α.rc := α.rc[ι → ⊥]
    else:
      α.rc(ι) := 0
      𝒪(ι).qu.push(orca(ι: −tmp))

  if α.frame = ∅:
    α.st := IDLE
  else:
    α.st := EXECUTE
}
```

```
GoIdle⟨α⟩:
  α.st = EXECUTE
→
{ α.frame := ∅; α.st := IDLE; }

Create⟨α⟩:
  α.st = EXECUTE  ∧  fresh ω ∧  𝒪(ω) = α
→
{
    heap :=
      heap[ω ↦ (f_1 ↦ null, ..., f_n ↦ null)]
    α.frame := α.frame[x ↦ ω]
}
```

```
MutateHeap⟨α⟩:
   α.st = EXECUTE ∧  𝒜_𝒞(α, lp1) = (ι_1, write)
    ∧  𝒜_𝒞(α, lp2) = ι_2
    ∧  ∀ι, κ, lp [ 𝒜_{𝒞[ι_1, f ↦ ι_2]}(α, lp) = (ι, κ)  ⟶
         (∃κ', lp' 𝒜_𝒞(α, lp') = (ι, κ') ∧ κ' ≤ κ ])
→
{
    heap := heap[ι_1, f ↦ ι_2]
}
```

Figure 5: Pseudo-code for synchronous operations.

3.6 Well Formed Queues

What we have defined so far has been quite concise and intuitive but we now begin a series of more technical definitions. In particular $\mathbf{I}_7$ which tells us about the effect of processing messages from message queues, and $\mathbf{I}_8$ which encompasses a series of definitions for well-formed states for each ORCA method we have seen.

Definition 3.18 (Reaches)**.**

$$\text{Reaches}_C(\alpha, \iota, k) \iff \exists x. \exists \overline{f}. \mathcal{A}_C(\alpha, k.x.\overline{f}) = \iota$$

Definition 3.19 (Weight)**.**

$$\text{Weight}_C(\alpha, \iota, j) \triangleq \begin{cases} z, & \text{if } \alpha.qu_C[j] = orca(\iota : z) \\ -1, & \text{if Reaches}_C(\alpha, \iota, j) \wedge \mathcal{O}(\iota) = \alpha \\ 0, & \text{otherwise} \end{cases}$$

Definition 3.20 (Queue Effect)**.**

$$\text{QueueEffect}_C(\alpha, \iota, n) \triangleq LRC_C(\iota) + \sum_{j=0}^{n} \text{Weight}_C(\alpha, \iota, j)$$

We now stray slightly from the definitions used in [Franco et al., 2018] by defining the predicate relevant, which we think makes $\mathbf{I}_7$ more succinct.

Definition 3.21 (Relevant)**.** For all $i \in \mathbb{N}$ and objects ι with $\mathcal{O}(\iota) = \alpha \neq \alpha'$

$$\begin{aligned}
\exists x, \overline{f}. \quad \mathcal{A}_C(\alpha, i.x, \overline{f}) &\longrightarrow \forall k \leq i.\ \text{Relevant}(\alpha, \iota, k) \\
\alpha.queue_C[i] = orca(\iota : z) &\longrightarrow \forall k < i.\ \text{Relevant}(\alpha, \iota, k) \\
\exists p. \quad \mathcal{A}_C(\alpha', p) = \iota &\longrightarrow \forall k \in \mathbb{N}.\ \text{Relevant}(\alpha, \iota, k)
\end{aligned}$$

Definition 3.22 ($\mathbf{I_7}$).

$\mathbf{I_7}$(a)

$$\forall n.\ \text{QueueEffect}_C(\alpha, \iota, n) \geq 0$$

$\mathbf{I_7}$(b)

$$\text{Relevant}(\alpha, \iota, n) \rightarrow\ \text{QueueEffect}_C(\alpha, \iota, n) > 0$$

3.7 Fine Grained Concurrency

If we break down the pseudocode listings into "atomic" statements we want to model execution in as fine-grained manner as possible. Is thus modelled as picking some actor non-deterministically and executing the next atomic statement. However, the above invariants do not hold under this fine grained execution. For example during message receival, $\mathbf{I_4}$ is broken when a message is popped off the queue but before the appropriate reference counts are mutated. $ORCA^0$ gives more precise definitions of AMC and OMC in order to combat this, we will list them here.

Definition 3.23 (Derived counters – preliminary for AMC and OMC)**.**

$$\text{LRC}_{\mathcal{C}}(\iota) \triangleq \mathcal{O}(\iota).\text{rc}_{\mathcal{C}}(\iota)$$

$$\text{FRC}_{\mathcal{C}}(\iota) \triangleq \sum_{\alpha \neq \mathcal{O}(\iota)} \alpha.\text{rc}_{\mathcal{C}}(\iota)$$

$$\text{OMC}_{\mathcal{C}}(\iota) \triangleq \sum_j \begin{cases} z & \text{if } \mathcal{O}(\iota).\text{qu}_{\mathcal{C}}[j] = \text{orca}(\iota : z) \\ 0 & \text{otherwise} \end{cases} + \text{OMC}^{\text{snd}}_{\mathcal{C}}(\iota) - \text{OMC}^{\text{rcv}}_{\mathcal{C}}(\iota)$$

$$\text{AMC}_{\mathcal{C}}(\iota) \triangleq \#\{\ (\alpha, k) \mid k > 0 \wedge \exists x.\overline{f}.\mathcal{A}_{\mathcal{C}}(\alpha, k.x.\overline{f}) = \iota\ \} + \text{AMC}^{\text{snd}}_{\mathcal{C}}(\iota) + \text{AMC}^{\text{rcv}}_{\mathcal{C}}(\iota)$$

where # denotes cardinality.

Definition 3.24 (Accessibility - Sending and Receiving). $\mathcal{A}_{\mathcal{C}}(\alpha, -1.x.\overline{f}) = (\iota, \kappa)$ iff
$\alpha.\text{st}_{\mathcal{C}} = \text{Receiving} \ \wedge\ 9 \leq \alpha.\text{st}_{\mathcal{C}} \leq 18 \ \wedge\ \mathcal{C}(\psi(x).\overline{f}) = \iota \ \wedge\ \alpha \vdash_{\mathcal{C}} b.x.\overline{f} : \kappa$
where $(b, \psi)$is the frame popped at line 8,
or
$\alpha.\text{st}_{\mathcal{C}} = \text{Sending} \ \wedge\ \alpha.\text{pc}_{\mathcal{C}} = 23 \ \wedge\ \mathcal{C}(\psi'(x).\overline{f}) = \iota \ \wedge\ \alpha' \vdash_{\mathcal{C}} b'.x.\overline{f} : \kappa$
where α' is the actor to receive the app-message, and
(b', ψ') is the frame to be sent in line 20.

Definition 3.25 (Auxiliary Counters for $\text{AMC}_{\mathcal{C}}(\_)$ and $\text{OMC}_{\mathcal{C}}(\_)$)**.**

$$\begin{aligned}
\text{AMC}^{\text{rcv}}_{\mathcal{C}}(\iota) \triangleq\ & \#\{\alpha \mid \alpha.\text{st}_{\mathcal{C}} = \text{RECEIVE} \ \wedge\ 9 \leq \alpha.\text{pc}_{\mathcal{C}} \ \wedge \\
& \iota \in \text{ws} \backslash \mathcal{C}urrAddrRcv_{\mathcal{C}}(\alpha)\} \\
\mathcal{C}urrAddrRcv_{\mathcal{C}}(\alpha) \triangleq\ & \begin{cases} \{\iota_{10}\} & \text{if } \alpha.\text{pc}_{\mathcal{C}} = 15 \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

In the above ws refers to the contents of the variable ws while the actor α is executing the pseudocode from Receiving, and ι_{10} refers to the contents of the variable ι arbitrarily chosen in line **??** of the code.
We define $\mathrm{AMC}^{\mathsf{snd}}_{\mathcal{C}}(\iota)$, $\mathrm{OMC}^{\mathsf{rcv}}_{\mathcal{C}}(\iota)$, and $\mathrm{OMC}^{\mathsf{snd}}_{\mathcal{C}}(\iota)$ similarly in Def. 3.26 .

Definition 3.26 ($\mathrm{AMC}_{\mathcal{C}}(\_)$ and $\mathrm{OMC}_{\mathcal{C}}(\_)$ – more cases)**.**

$$\mathrm{AMC}^{\mathsf{snd}}_{\mathcal{C}}(\iota) \triangleq \#\{\ \alpha \mid \alpha.\mathsf{st}_{\mathcal{C}} = \text{SEND} \ \wedge\ 12 \leq \alpha.\mathsf{pc}_{\mathcal{C}} \ \wedge \iota \in \mathsf{ws} \backslash \mathcal{C}urrAddrSnd_{\mathcal{C}}(\alpha)\ \}$$

$$\mathcal{C}urrAddrSnd_{\mathcal{C}}(\alpha) \triangleq \begin{cases} \{\iota_{12}\} & \text{if } \alpha.\mathsf{pc}_{\mathcal{C}} = 20 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{OMC}^{\mathsf{rcv}}_{\mathcal{C}}(\iota) \triangleq \textstyle\sum_{\alpha} \mathrm{OMC}^{\mathsf{rcv}}_{\alpha,\mathcal{C},}(\iota)$$

$$\mathrm{OMC}^{\mathsf{rcv}}_{\alpha,\mathcal{C}}(\iota) \triangleq \begin{cases} z & \text{if } \mathcal{O}(\iota) = \alpha, \text{and } \alpha \text{ is executing ReceiveORCA} \\ & \text{and } \alpha.\mathsf{pc}_{\mathcal{C}} = 6 \text{ and } \mathrm{top}(\alpha.\mathsf{qu}_{\mathcal{C}}) = \mathrm{ORCA}(\iota : z) \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{OMC}^{\mathsf{snd}}_{\mathcal{C}}(\iota) \triangleq \textstyle\sum_{\alpha} \mathrm{OMC}^{\mathsf{snd}}_{\alpha,\mathcal{C}}(\iota)$$

$$\mathrm{OMC}^{\mathsf{snd}}_{\alpha,\mathcal{C}}(\iota) \triangleq \begin{cases} 256 & \text{if } \alpha.\mathrm{state} = \text{SEND and ff}.\mathsf{pc}_{\mathcal{C}} = 19 \\ & \text{and } \iota \text{ is the address chosen in line 12} \\ \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) & \text{if } \alpha.sfstate = \text{COLLECT and } \alpha.\mathsf{pc}_{\mathcal{C}} = 24 \\ & \text{and } \iota \text{ is the address chosen in line 18} \\ 0 & \text{otherwise} \end{cases}$$

In the above ws refer to the contents of the variable ws while the actor α is executing the pseudocode from Receiving, and ι_{12} refers to the contents of the variable ι arbitrarily chosen in line 12 of the code.

3.8 Well Formed Executions

Finally we have a definition for $\mathbf{I}_8$ that is used to show correctness of the defined methods, with each having their own correctness criteria $\mathbf{I}_8{}^{Send}$ etc. This is given in full in Appendix A.

4 $ORCA^{\text{Ghost}}$

We now present $ORCA^{\text{Ghost}}$, a refinement of $ORCA^0$ with the goal to remove the special cases introduced to tackle fine grained concurrently.

To do this we rely on *ghost variables*, variables not present in any physical implementation but are considered during verification. We assume that we can mutate these variables instantaneously, and we can execute any regular pseudocode statement and an arbitrary sequence of statements that modify only ghost variables atomically. For this we use the notation:

```
[s1 | g1; g2; g3; ... ]
```

Where $s1$ is a statement in pseudocode, gi are statements mutating only ghost variables and the brackets indicate an atomic block.

In $ORCA^{\text{Ghost}}$, for each actor α, and each object ι we add an additional ghost reference count $\alpha.grc(\iota)$. With this, we redefine the local and foreign reference counts in terms of these ghost reference counts:

Definition 4.1.

$$LRC^{DS}(\iota) = \mathcal{O}(\iota).grc(\iota)$$
$$FRC^{DS}(\iota) = \sum_{\alpha \neq \mathcal{O}(\iota)} \alpha.grc(\iota)$$

Then we can use this in an updated $\mathbf{I}_4$

Definition 4.2 (${\mathbf{I}_4}^{DS}$, Reference Counting Equation)**.**

$${\mathbf{I}_4}^{DS} \triangleq \forall \iota.\ (LRC^{DS}(\iota) + OMC(\iota) = FRC^{DS}(\iota) + AMC(\iota))$$

In general we also 'lift' all other invariants $\mathbf{I}_n$ to ${\mathbf{I}_n}^{DS}$ where instead of referring to the reference counts they now state properties about the ghost reference counts.

- ${\mathbf{I}_2}^{DS}$ Now states that accessibility in a foreign actor implies the owner has a non-zero ghost reference count for the object.
- ${\mathbf{I}_3}^{DS}$ Now states that accessibility of a foreign object implies a non-zero ghost reference count for it.
- ${\mathbf{I}_5}^{DS}$ States that ghost reference counts are positive
- QueueEffect Now refers to ghost reference count
- ${\mathbf{I}_7}^{DS}$ Consuming messages will preserve inv 5
- ${\mathbf{I}_8}^{DS}$ Now uses ghost reference count in well formed definitions

As we give the updated pseudocode we will also define $\mathbf{I}_9$ which gives the exact relation between the ghost reference counts and the runtime's reference counts.

We will use the notation

```
if e ⟶ s
```

to denote the conditional execution of the statement `s` if the expression `e` holds, this is to make reasoning about specific lines easier. We add an additional state that an actor can be in `RECEIVE_ORCA` to distinguish between actors receiving application and ORCA messages.
The highlighted lines are those changed from $ORCA^0$.

4.1 Sending - $ORCA^{\text{Ghost}}$

```
1  α.st = EXECUTE && α.frame = (b, φ · φ′)
2  Sending<α>:
3  {
4    α.st := SEND
5
6    [ws := trace_frame( α, (b, φ′)) | gws := ws; ∀ι. α.rc0(ι):=α.rc(ι)]
7
8    for ι ∈ ws:
9      if (O(ι) = α):
10       α.rc(ι) += 1
11     else:
12       if α.rc(ι) > 1:
13         α.rc(ι) -= 1
14       else:
15         [O(ι).qu.push(orca(ι:256)) | α.grc(ι) := 257]
16         α.rc(ι) := 256
17     ws := ws \ {ι}
18
19   α.frame := (b, φ)
20
21   [α′.qu.push(app(b′, φ′)) |
22     for ι ∈ WS:
23       if O(ι) = α:
24         α.grc(ι) += 1
25       else:
26         α.grc(ι) -= 1]
27 }
```

Note that whenever we push messages we are simultaneously updating the ghost reference counts to counteract the effect on one side of the equation in $\mathbf{I}_4{}^{DS}$.

For some ι we define $\text{SendRC}(\alpha, \iota)$ by:

$$\text{SendRC}(\alpha, \iota) = \begin{cases} 257 & \text{for } \alpha = \mathcal{O}(\iota) \wedge \alpha.rc_0(\iota) = 1 \\ \alpha.rc_0(\iota) & \text{otherwise} \end{cases}$$

Here we use the notation ι_8 to refer to the variable ι in the pseudocode defined on line 8.

$$\begin{aligned} \mathbf{I}_9{}^{SEND} \triangleq\ & \alpha.st = \texttt{SEND} \ \rightarrow \\ & pc \leq 17 \longrightarrow \\ & \quad \forall \iota \neq \iota_8. \\ & \quad\quad \iota \notin (gws \setminus ws) \rightarrow \alpha.grc(\iota) = \alpha.rc(\iota) \\ & \quad\quad \iota \in (gws \setminus ws) \rightarrow \alpha.grc(\iota) = \text{SendRC}(\alpha, \iota) \\ & 8 \leq pc < 15 \longrightarrow \alpha.grc(\iota_8) = \alpha.rc_0(\iota_8) \\ & 15 \leq pc \leq 17 \longrightarrow \alpha.grc(\iota_8) = SendRC(\alpha, \iota_8) \end{aligned}$$

Where $pc = \alpha.pc_C$.

Where ι_8 is the program variable ι defined on line 8. For cases where ι_8 is not defined, we assume $\forall \iota.\ \iota \neq \iota_8$.

4.2 Receiving - $ORCA^{Ghost}$

```
α.st = IDLE && top(α.qu) = app(φ)
Receiving<α>:
{
  α.st := RECEIVE
  ws := trace_frame(α, φ)

  [pop(α.qu) |
    for ι ∈ WS:
      if 𝒪(ι) = α:
        α.grc(ι) -= 1
      else:
        α.grc(ι) += 1]

  for ι ∈ ws:
    if (𝒪(ι) = α) ⟶ α.rc(ι) -= 1
    if (𝒪(ι) ≠ α) ⟶ α.rc(ι) += 1
    ws := ws \ {ι}

  α.frame := φ

  α.st := EXECUTE

}
```

For some ι we define RecRC(α, ι) by:

$$\text{RecRC}(\alpha, \iota) = \begin{cases} \alpha.rc(\iota) + 1 & \text{for } \alpha = \mathcal{O}(\iota) \\ \alpha.rc(\iota) - 1 & \text{for } \alpha \neq \mathcal{O}(\iota) \end{cases}$$

$$\begin{aligned} \mathbf{I}_9{}^{REC} \triangleq\ & \alpha.st = \texttt{RECEIVE} \ \rightarrow \\ & pc < 7 \longrightarrow \forall \iota.\ \ \alpha.grc(\iota) = \alpha.rc(\iota) \\ & pc = 14 \longrightarrow \alpha.grc(\iota_{14}) = \text{RecRC}(\alpha, \iota_{14}) \\ & 14 < pc \leq 17 \longrightarrow \alpha.grc(\iota_{14}) = \alpha.rc(\iota_{14}) \\ & 7 \leq pc \leq 23 \longrightarrow \\ & \qquad \forall \iota \neq \iota_{14}.[\\ & \qquad \iota \notin ws \rightarrow \alpha.grc(\iota) = \alpha.rc(\iota) \\ & \qquad \wedge \iota \in ws \rightarrow \alpha.grc(\iota) = \text{RecRC}(\alpha, \iota) \\ & \qquad] \end{aligned}$$

Where $pc = \alpha.pc_C$.

4.3 Receiving orca message - $ORCA^{Ghost}$

```
α.st = IDLE && top(α.qu) = ORCA(ι : z)
Receiving<α>:
{
  α.st := RECEIVE_ORCA
  α.rc(ι) += z
  [pop(α.qu) | α.grc(ι) := α.rc(ι)]
```

```
  α.st := EXECUTE
}
```

$$\mathbf{I}_9{}^{REC\_ORCA} \triangleq\ \alpha.st = \texttt{RECEIVE\_ORCA} \ \rightarrow \\ \forall \iota \neq \iota.\ \ \alpha.grc(\iota) = \alpha.rc(\iota) \\ pc = 5 \longrightarrow \alpha.grc(\iota) + z = \alpha.rc(\iota)$$

Where $pc = \alpha.pc_C$.

4.4 Garbage Collection - $ORCA^{\text{Ghost}}$

```
α.st = IDLE && α.st = EXECUTE
GarbageCollection<α>:
{
  α.st := COLLECT
  ms := ∅

  // marking as unreachable
  for ι such that 𝒪(ι) = α || α.rc > 0:
    ms := ms[ι → U]

  // tracing and marking locally accessible as reachable
  for ι ∈ trace_this(α) ∪ trace_frame(α.frame):
    ms := ms[ι → R]

  // marking owned and globally accessible as reachable
  for ι such that 𝒪(ι) = α && α.rc > 0:
    ms := ms[ι → R]

  // collecting
  for ι such that ms(ι) = U:
    if 𝒪(ι) = α:
      𝒞.heap := 𝒞.heap[ι → ⊥]
      α.rc := α.rc[ι → ⊥]
    else:
      α.rc(ι) := 0
      [𝒪(ι).qu.push(orca(ι: −tmp)) | α.grc(ι) := 0]

  if α.frame = ∅:
    α.st := IDLE
  else:
    α.st := EXECUTE
}
```

$$\mathbf{I}_9{}^{GC} \triangleq\ \alpha.st = \texttt{COLLECT} \ \rightarrow \\ pc \neq 26 \longrightarrow \forall \iota\ \ \alpha.grc(\iota) = \alpha.rc(\iota) \\ pc = 26 \longrightarrow \\ \forall \iota \neq \iota_{20}.\ \ \alpha.grc(\iota) = \alpha.rc(\iota) \\ \alpha.rc(\iota_{20}) = 0$$

$$\mathbf{I}_9{}^{IDLE,EXEC} \triangleq \alpha.st = \texttt{IDLE} \vee \alpha.st = \texttt{EXECUTE} \rightarrow \forall \iota \;\; \alpha.grc(\iota) = \alpha.rc(\iota)$$

Where $pc = \alpha.pc_C$.

4.5 Correctness

We do not provide a full, line by line proof of the preservation of $\mathbf{I}_9$ because it is clear by comparing it to the pseudocode and it is not interesting.

Lemma 4.3. *$\boldsymbol{I}_3{}^{DS}$ is preserved*
$\boldsymbol{I}_5{}^{DS}$ is preserved
$\boldsymbol{I}_7{}^{DS}$ is preserved
$\boldsymbol{I}_8{}^{DS}$ is preserved

We argue that the proof of these invariants is not significantly different from those in $ORCA^0$.

Lemma 4.4. *$\boldsymbol{I}_4{}^{DS}$ is preserved*

Proof. This is far easier to prove than $\mathbf{I}_4$ in $ORCA^0$, we just note that whenever a message queue is altered in the code it is coupled with an operation on ghost reference counts that preserves $\mathbf{I}_4$. This accounts for all of the ghost reference count operations.
For example, take lines $7-12$ in `Receive`, when the message is popped off the queue it adjusts the AMC for the objects in the trace, but these are exactly the ghost reference counts we update. □

Proposition 4.5. *Any configuration that satisfies $\boldsymbol{I}_3 - \boldsymbol{I}_9$ also satisfies $\boldsymbol{I}_2{}^{DS}$*

Proof. We take the same approach as in the proof for $ORCA^0$. Take an arbitrary object ι and actors α, α_0, path p and message path mp such that $\alpha \neq \mathcal{O}(\iota) = \alpha_0$. We want to show $LRC_C^{DS}(\iota) > 0$:

Case 1: $\mathcal{A}_C(\alpha, lp) = \iota$, so by $\mathbf{I}_3{}^{DS}$ we have that the right hand side of $\mathbf{I}_4{}^{DS}$ is positive.

Case 2: p starts from a queue, i.e. $p = k.x.\overline{f}$ for some $k \geq 0$ and series of fields $\overline{f}$. By definition we have that $AMC_C(\iota) > 0$. By $\mathbf{I}_5$ we also have that the right hand side of $\mathbf{I}_4{}^{DS}$ is positive.

Case 3: p starts from a queue again, $p = -1.x.\overline{f}$. We can see this holds from the definition of $\mathbf{I}_8{}^{DS}$.

In the first two cases we have that the right hand side of $\mathbf{I}_4{}^{DS}$ is greater than zero, so the left hand side must be. Then $\mathbf{I}_7$ implies that $LRC_C(\iota)^{DS} > 0$

□

Lemma 4.6.

$$\forall \alpha, \iota.\ \alpha \neq \mathcal{O}(\iota) \longrightarrow \alpha.rc(\iota) > 0 \longrightarrow \alpha.grc(\iota) > 0$$

Proof. Case inspection of $\mathbf{I}_9$ □

Proposition 4.7. *If an actor α owns an object ι and is in garbage collection at configuration C then $\alpha.rc_C(\iota) = 0$ means that ι can be safely collected.*

Proof.

$$
\begin{array}{lll}
 & \alpha.rc_C(\iota) = 0 & \\
\implies & \alpha.grc_C(\iota) = 0 & (\text{In Garbage Collection}) \\
\implies & OMC_C(\iota) + LRC_C^{DS}(\iota) = 0 & (\mathbf{I}_7, \text{ Queue Effect }) \\
\implies & AMC_C(\iota) + FRC_C^{DS}(\iota) = 0 & (\mathbf{I}_4{}^{\text{DS}}) \\
\implies & FRC_C^{DS}(\iota) = 0 & (AMC_C \geq 0) \\
\implies & \forall \alpha' \neq \alpha.\ \alpha.grc_C(\iota) = 0 & (\text{Foreign reference counts non negative}) \\
\implies & \forall \alpha' \neq \alpha.\ \alpha.rc_C(\iota) = 0 & (\textit{Lemma } 4.6) \\
\implies & \iota \text{ is Globally Inaccessible} & (\mathbf{I}_3)
\end{array}
$$

□

We still have some of the "ugly" reasoning where we have to define the ghost reference counts for every possible line of execution, but we think this is easy to reason about then the original $ORCA^0$ model. In Section 8 we will improve upon this but for now we will continue and look at the Pony optimisations.

4.6 Evaluation

Comparing our definition to $ORCA^0$ we can see that it is more complex in several places. We introduced an additional invariant $\mathbf{I}_9$, introduced new notation with the atomic ghost statements, and we added complexity to the pseudocode. However, by doing so we were able to reduce the definitions of AMC and OMC to forms reminiscent of those before fine grained concurrency was considered.

In the next section we are going to build on this and we can do so because we move the cognitive overhead of fine grained concurrency onto the ghost variables. We can define more complex invariants in terms of the ghost variables and leave the distinction from the real variables in the proofs here.

5 $ORCA^{\text{Ghost+Val}}$

5.1 Immutability

Here we present an extension to $ORCA^{\text{Ghost}}$ to capture the optimisations made on immutable structure. We leverage the fact that immutability is both deep and persistent. By deep we mean that if some object ι is immutable and ι can reference some other object ι' then ι' is immutable.

Immutability is persistent in that there is no way to reclaim write access to an immutable object. If ι is immutable in some configuration C and $C \rightsquigarrow C'$ then ι is immutable in C' if it has not been collected.

We also know that if an immutable object ι has a reference to another object ι' then ι' must outlive ι. It is impossible for ι to be globally inaccessible before ι' because accessibility to ι *gives* accessibility to ι'. What we want to do therefore is give a reference count to ι that also acts as a reference count for ι' in a sound manner. This means that we can defer tracing of immutable objects until they are collected which allows more efficient sending and receiving.

Note that if ι had a mutable reference to ι' and ι is not immutable then a reference count for ι cannot be used for ι'. This is because the field in ι assigned to ι' can be changed, leaving no relation between the objects ι and ι'.

We say an object ι is *immutable* in a given configuration C, $imm_C(\iota)$ if and only if there does not exist some actor α, with path p (including fields and paths through messages on the queue) with write access to it and all child objects that ι can access through fields are also immutable.

Definition 5.1 (Immutability)**.**

$$\begin{aligned} imm_C(\iota) \triangleq\ & \nexists, \alpha, p. \quad \mathcal{A}_C(\alpha, p) = (\iota, \mathsf{write}) \\ & \wedge \forall f.\ \exists \iota'.\ (C(\iota, f) = \iota' \rightarrow imm_C(\iota')) \end{aligned}$$

This is satisfied by `val` types in Pony, so if we have reference to an object ι with `val` capability then we can assume $imm_C(\iota)$. In this model we assume that immutability is somehow known to actors in the runtime, later in Section 7 we will discuss how this can be realised.

We say an object ι *protects* another object ι' at a configuration C if in a garbage collection run, ι being reachable would prevent the collection of ι'. All objects protect themselves, but also ι protects ι' at C if ι is immutable at C and an actor α owns both ι and ι'. Further there must be a path from ι to ι' where all intermediate objects are also owned by α.

This would mean that ι' and all intermediate objects are immutable also, as by definition there is a path from an immutable object, ι to them, and from our assumptions about the host language all objects reachable from an immutable object are immutable.

Definition 5.2 (Protection)**.**

$$\begin{aligned} \iota.Protects_C(\iota') \triangleq\ & \iota = \iota' \\ & \vee\ (imm_C(\iota) \wedge \exists n \geq 0.\ \exists \overline{f} = f_1.f_2 \ldots f_n. \\ & \quad (C(\iota, \overline{f}) = \iota' \wedge \\ & \quad\ \forall i = 1, \ldots n.\ \mathcal{O}(\iota) = \mathcal{O}(C(\iota, f_1.f_2 \ldots f_i)))) \end{aligned}$$

We can then represent our statement earlier as a lemma. Because of deep immutability, protection is preserved as the program runs, as none of the objects can modify their fields. This also means that if an object ι protects another ι', then ι' will always live at least as long as ι.

Lemma 5.3.

$$\begin{aligned} \iota.Protects_C(\iota') \wedge \models & C \wedge C \rightsquigarrow C' \wedge \iota \in dom(C') \\ & \longrightarrow \iota' \in dom(C') \wedge \iota.Protects_{C'}(\iota') \end{aligned}$$

We can deduce this from the definition of imm_C, that there are no write references and the type safety of the host language, we assume objects can only be modified by actors with references with write capability.

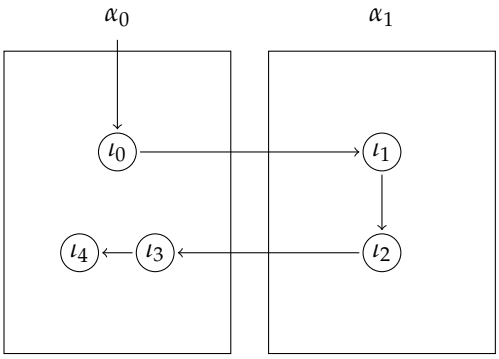


Figure 6: In this configuration, where all objects are immutable, all objects protect themselves but ι_0 does not protect anything else. We do have though that $\iota_1.Protects(\iota_2)$ and $\iota_3.Protects(\iota_4)$.

We form a definition for when an object ι is *collectable* in this model. We only collect an object ι owned by α if there is no local path from α to ι, its reference count from α is zero, and all objects that protect it are also collectable.

Definition 5.4 (Collectable)**.**

$$\begin{aligned} collectable_C(\iota) \triangleq \quad & \nexists lp.\ \mathcal{A}_C(\mathcal{O}(\iota), lp) = \iota \\ & \wedge\ \alpha.grc_C(\iota) = 0 \\ & \wedge\ (\nexists \iota'.\iota'.Protects(\iota) \wedge \alpha.grc_C(\iota') > 0) \end{aligned}$$

Note that we do not require that there is no path from α to ι'. This is because, if there was some path, then we could form a path from α to ι as ι' protects ι, but we know that this cannot exist! So by contradiction there cannot be a path to ι'.

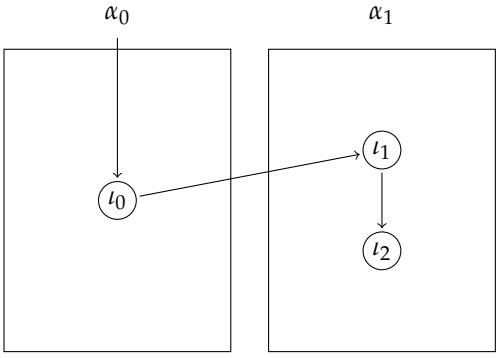


Figure 7: An example of a configuration with a reachable object that has a zero reference count but no objects are collectable.

Definition 5.5 (Subpath)**.** We say that a path p is a *subpath* of a path p', if either $p = p'$ or p is a prefix of p'.

$$p \sqsubseteq p' \triangleq \exists \overline{f}.\ p' = p.\overline{f}$$

Definition 5.6 (Strict Subpath)**.** A path p is a *strict subpath* of a path p' if $p \neq p'$ and p is a prefix of p'.

$$p \sqsubset p' \triangleq p \sqsubseteq p' \wedge p \neq p'$$

As with $ORCA^0$, we invoke a behaviour on an actor by adding the tuple (b, ϕ) to the actor's message queue. If we are passing an immutable object, in $ORCA^{\text{Ghost+Val}}$ we do not trace its fields and only alter the reference count of ϕ itself. If we are sending a mutable object ϕ' with a path to an immutable object, ι, we do not trace ι's children, and only alter the reference count of ι. We justify this as ι protects all its children, so they will not be affected in an updated version of collection.

Then, we define the following predicate to calculate the trace stopping at immutable objects.

Definition 5.7 (trace_ws)**.**

$$\begin{aligned} \mathit{trace\_ws}_C(\alpha, \phi) = \{\ \iota \mid\ & \exists x, \overline{f}, b, \psi.\ [\phi = (b, \psi) \wedge \\ & \mathcal{A}_C(\alpha, b.x.\overline{f}) = (\iota, \_) \wedge \\ & \forall p\ \exists \iota', \kappa.\ (p \sqsubset (b.x.\overline{f}) \rightarrow \\ & \quad (\mathcal{A}_C(x, p) = (\iota', \kappa) \rightarrow \kappa \neq \texttt{Val} \vee \kappa \neq \texttt{Tag}))]\} \end{aligned}$$

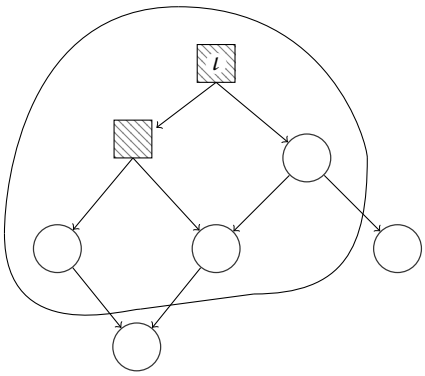


Figure 8: Showing the result of trace_ws on a frame containing only ι, squares represent mutable objects and circles immutable objects.

When we send and receive an object ι we update the reference counts in $\mathit{trace\_ws}_C(\iota)$.

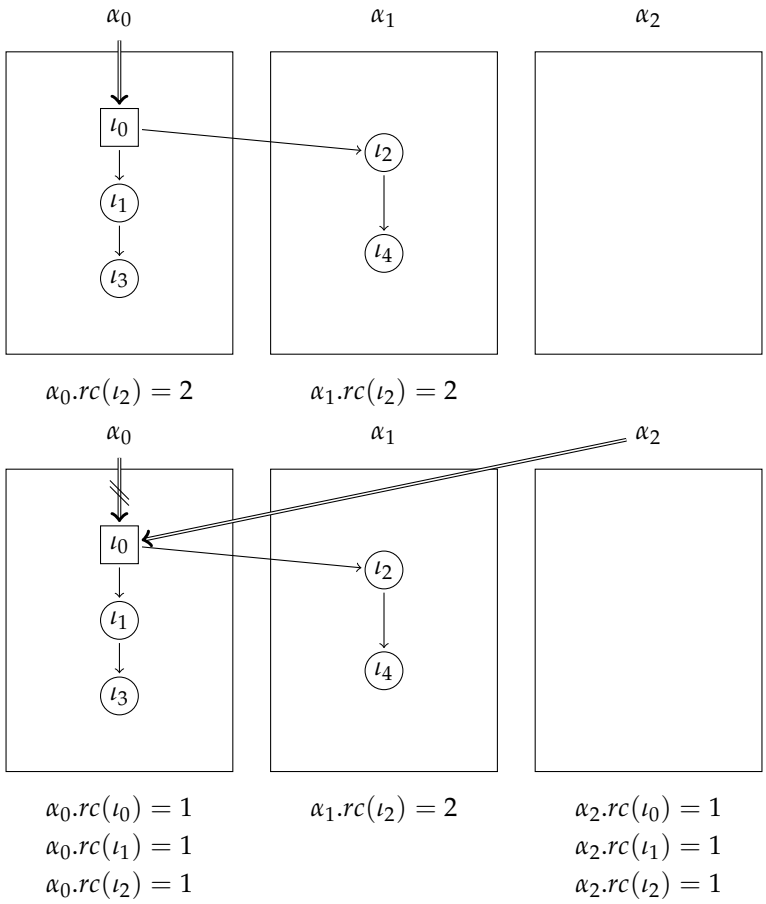


Figure 9: Here we represent an `Iso` reference with a double arrow. We have two configurations, before and after sending, α_0 sends a message with an `Iso` reference to ι_0, to α_2. ι_0 is mutable so its children are traced, but ι_1 and ι_2 are both immutable so their children are not traced. The traced objects have their reference counts altered.

When, during collection, we find an unreachable, immutable object ι we do not own, but have a positive reference count to, we calculate the full trace of the object. We trace the object to find children of the the object that we may not have a reference count for, but can still reach. These are objects that were being protected by ι and now may have lose their reference count in future collections or be freed. We increment our reference count for these objects and send an orca message to the owner telling them about this increase.

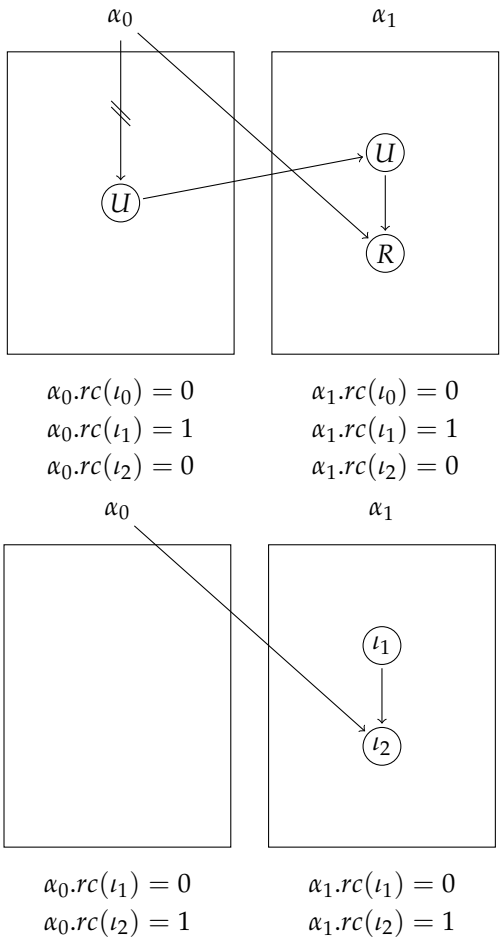


Figure 10: If we consider the layout from Figure 7, and then give α_0 a reference to ι_2 and drop its reference to ι_0, we get the above configuration. We then assume α_0 performs collection, the U and R show which objects are marked unreachable and reachable during the collection procedure. Below shows the result of the collection, ι_0 is freed as its reference count is zero and it is unreachable, ι_1 has its reference count reduced to 0 and ι_2 has its reference count increased by 1 as it is included in the trace of both ι_0 and ι_1. Both these changes are then communicated to α_1 via orca messages. Now when α_1 runs collection it will collect ι_1.

Remember if we are sending an object we own, then we increase its reference count and if we do not own it we decrement it. In the special case where we would decrement its count to zero, we instead notify the owner that we have added 256 references and set our reference count to 255, with the last reference accounted for by the message, this preserves $\mathbf{I}_4{}^{DS}$.

It is now possible to have a case where we send an immutable object that we have a reference count of zero, we will handle this in a similar way, we increment our reference count and send an orca message to the owner.

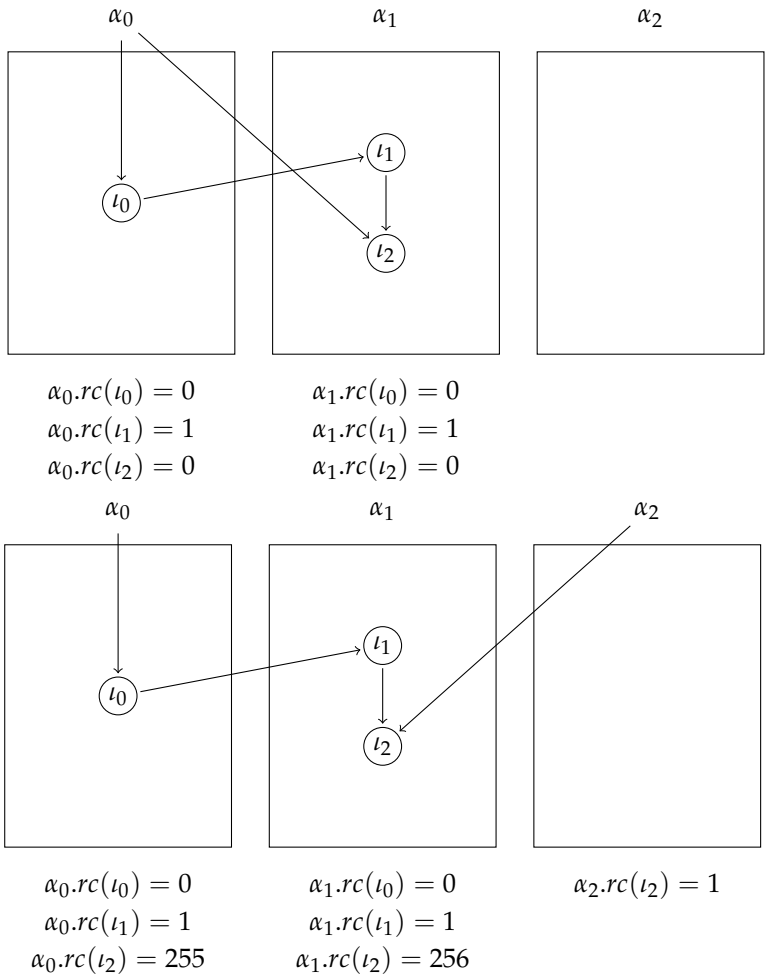


Figure 11: Two configurations before and after sending ι_2 to α_2. As α_0 has no reference count for ι_2 it increments its reference count to 255 and sends an orca message ι_1 to increase the number of references by 256. This is described in full in the pseudocode.

We introduce the more general *transitive protection* to formalise the property that if there is an immutable object ι that references an object ι' owned by a different actor, then ι' cannot be collected until ι has been collected. ι must outlive ι'. We want to form a property such that the following holds:

$$\iota.TransProtects_C(\iota') \wedge \models C \wedge C \rightsquigarrow C' \wedge \iota \in dom(C') \rightarrow \iota' \in dom(C') \wedge \iota.TransProtects_{C'}(\iota')$$

An object ι transitively protects another object ι' if there is some path from ι to ι' where every field access in the path goes either between objects that protect each other, or the reference goes between actors and there is a reference count greater than zero for the target object from the previous objects owner.

Definition 5.8 (Transitive Protection)**.**

$$\begin{aligned}\iota.TransProtects_C(\iota') \triangleq\ & \iota.protects_C(\iota') \\ & \vee\ (\exists \iota_{local}, \iota_{remote}, f.\ \mathcal{O}(\iota) = \mathcal{O}(\iota_{local}) \neq \mathcal{O}(\iota_{remote}) \\ & \quad \wedge \iota.protects_C(\iota_{local}) \\ & \quad \wedge C(\iota_{local}, f) = \iota_{remote} \\ & \quad \wedge \mathcal{O}(\iota_{local}).rc_C(\iota_{remote}) > 0 \\ & \quad \wedge \iota_{remote}.TransProtects_C(\iota)))\end{aligned}$$

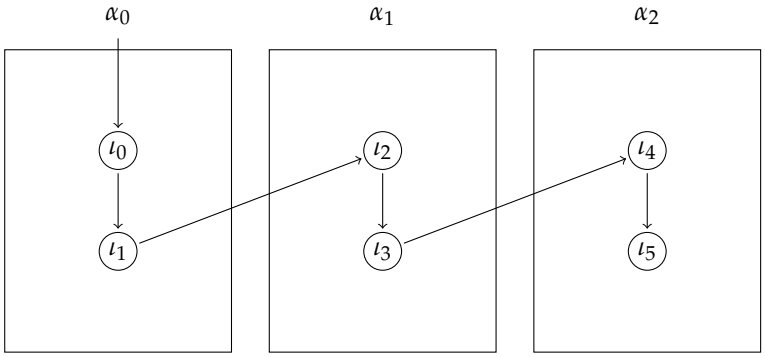


Figure 12: We can see that ι_0 protects ι_1 but not ι_2 or ι_3 as they have different owners. However, ι_0 transitively protects all other objects in the configuration. For example, $\iota_0.TransProtects_C(\iota_3)$. ι_2 protects ι_2 so $\iota_2.TransProtects_C(\iota_3)$ from the definition. Then ι_1 has a reference to an object ι_2, that transitively protects ι_3. The owner of ι_1 has a reference count greater than zero for this object and ι_0 protects ι_1. From these we get that $\iota_0.TransProtects_C(\iota_3)$.

5.2 Invariants

We generalise $\mathbf{I}_2$ to $\mathbf{I}_2{}^{\text{Val}}$, if an object ι is accessible via a message queue or by a non-owner, then the owner has a positive reference count for an object ι' that protects ι.

Definition 5.9 ($\mathbf{I}_2{}^{\text{Val}}$)**.** For $\alpha_0 = \mathcal{O}(\iota)$ with $\alpha \neq \alpha_0$

$$\begin{aligned}\mathbf{I}_2{}^{\text{Val}} \triangleq\ & \exists p', \iota'.\ (\mathcal{A}_C(\alpha, p) = \iota \wedge p' \sqsubseteq p) \vee (\mathcal{A}_C(\alpha_0, mp) = \iota \wedge p' \sqsubseteq mp) \\ & \longrightarrow \mathcal{A}_C(\alpha, p') = \iota' \wedge \iota'.Protects(\iota) \wedge \alpha_0.grc_C(\iota') > 0\end{aligned}$$

Note that we have for a fixed C, α and ι, $\mathbf{I}_2{}^{\text{Val}} \longrightarrow \mathbf{I}_2$

We make a similar extension to $\mathbf{I}_3$. If a non owner α can access an object ι through a field or its call stack, then we have either a positive reference count to some object that transitively protects ι.

Definition 5.10 ($\mathbf{I}_3{}^{\text{Val}}$)**.**

$$\mathbf{I}_3{}^{\text{Val}} \triangleq\ \mathcal{A}_C(\alpha, lp) = \iota \longrightarrow \exists \iota'.\ (\iota'.TransProtects_C(\iota) \wedge \alpha.grc_C(\iota') > 0)$$

Again we have for a fixed C and ι, $\mathbf{I}_3{}^{\text{Val}} \longrightarrow \mathbf{I}_3$

This definition takes into account what happens when we drop a reference to an immutable object but retain a reference to one of its children, we call this *subreferencing*. As an immutable object transitively protects its children, if there is some α, ι and ι' that satisfies $\mathbf{I}_3{}^{\text{Val}}$ but α does not have a path to ι' then we can assume that we had a reference to ι' and subreferenced it to ι.

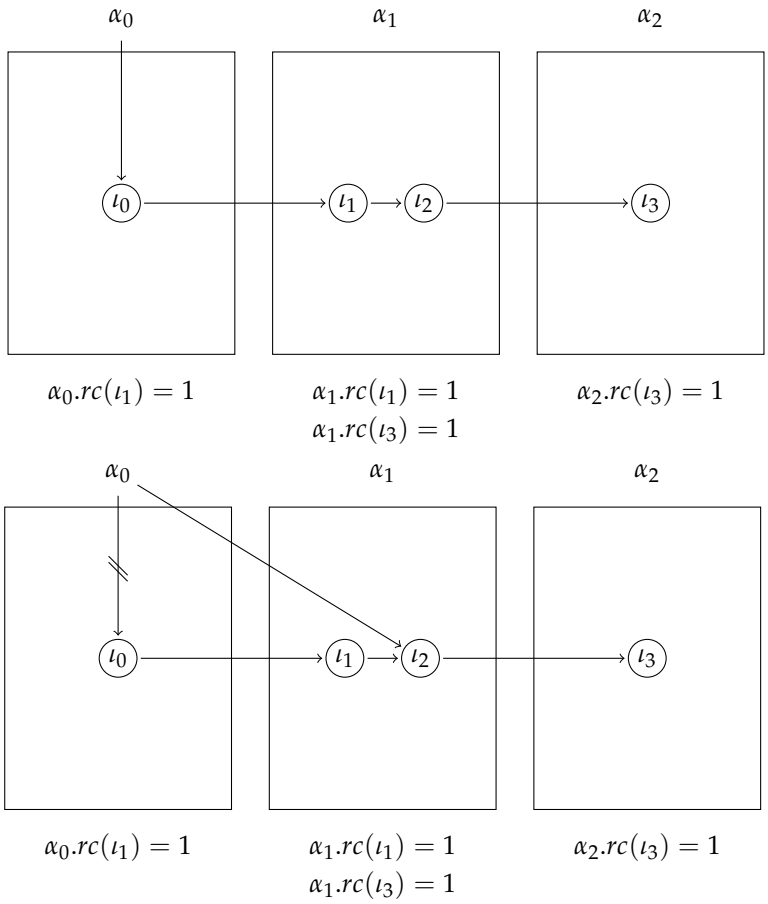


Figure 13: The above show α_0 subreferencing ι_0 to ι_2. In the first configuration we require that $\mathbf{I}_3{}^{\text{Val}}$ holds for all objects apart from ι_0 as they are all foreign to α_0 and α_0 has a path to all of them. This is then satisfied by the fact that α_0 has a positive reference count to ι_1 which protects all the required objects.
In the second configuration, α_0 only has paths to ι_2 and ι_3, but both of these are transitively protected by ι_1 which α_0 has retained a positive reference count for.

Because we require that for *all* foreign, reachable objects $\mathbf{I}_3{}^{\text{Val}}$ holds, we cannot have some an example as below.

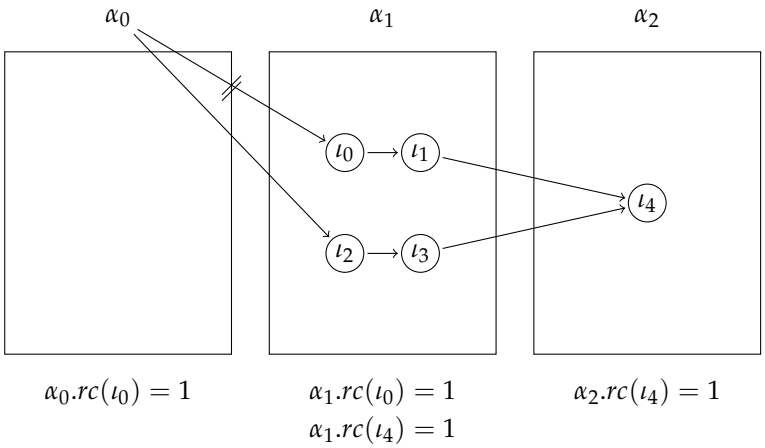


Figure 14: An example invalid configuration, even though $\mathbf{I}_3{}^{\text{Val}}$ holds for ι_4 it does not for ι_2 and ι_3.

Because we do not require a positive reference count to all foreign objects that we have a path to, if we subreference ι to give some ι' that we do not have a reference count to we need some process during garbage collection to fix this so that when the reference count for the, now unreachable, ι reduced to zero, $\mathbf{I}_3{}^{\text{Val}}$ is maintained for ι'. This is described in full later.

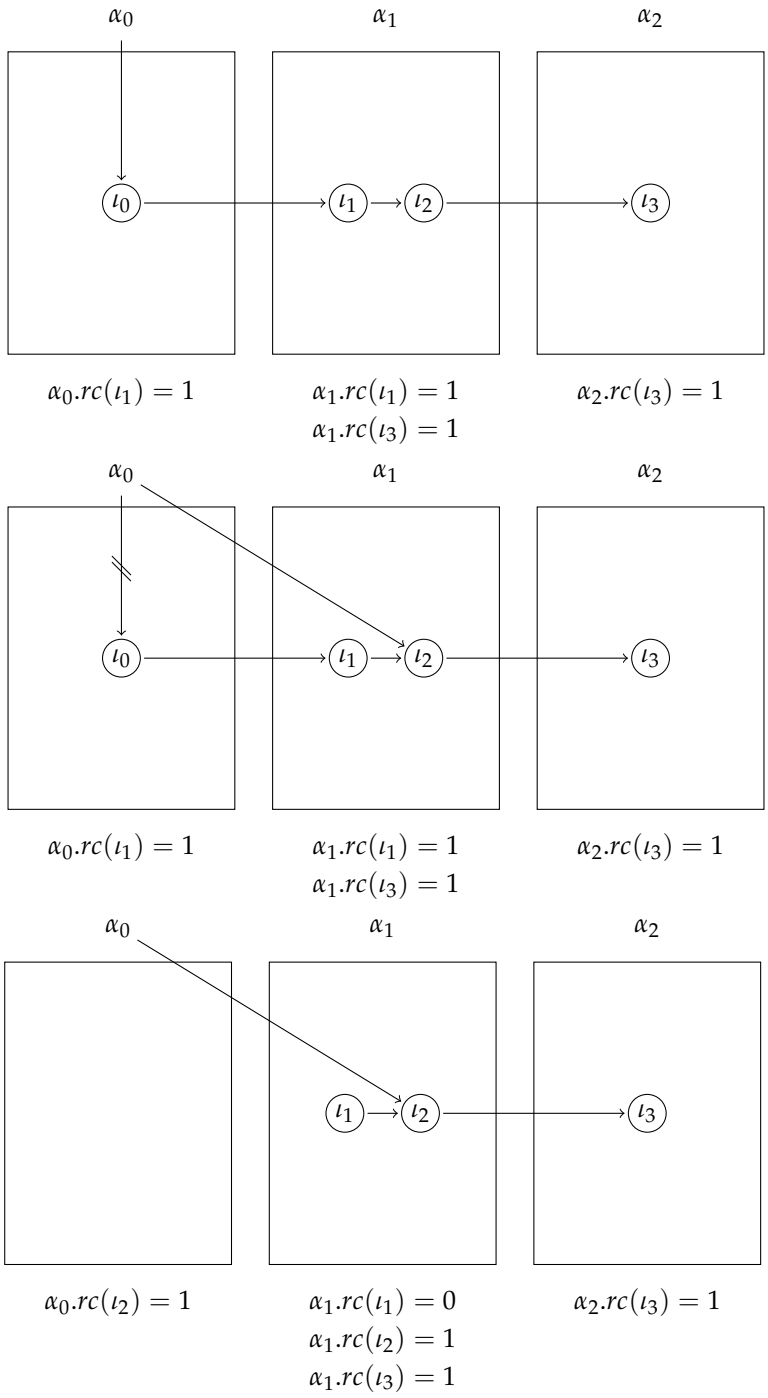


Figure 15: This shows Figure 13 with an additional configuration for after α_0 performs collection. During collection ι_0 is freed as it is unreachable from α_0 and with zero reference count, and ι_1 which is also unreachable, has its reference count decremented and this is propagated to its owner. Then a trace is made from ι_0 to find reachable objects, it finds ι_2 but does not recurse further. It then increments its reference count for ι_2 and sends the increment to ι_2's owner.

We also give a new definition for the application message count for some object to account for

not updating the reference counts on some objects. We use trace_ws to define this as this is what is used to construct the set of reference counts to update when we send and receive an object. So by doing this we make sure that $\mathbf{I}_4$ continues to hold.

Definition 5.11 (**$\mathbf{I}_4{}^{Val}$, Reference Counting Equation with Values**)**.**

$$\mathbf{I}_4{}^{Val} \triangleq \forall \iota.\ LRC^{DS}(\iota) + OMC(\iota) = FRC^{DS}(\iota) + AMC_C^{Val}(\iota)$$

$$AMC_C^{Val}(\iota) = \sum_{a \in C.actors} \sum_{\phi \in a.queue} \mathbb{1}_{trace\_ws_C(a,\phi)}(\iota)$$

Where $\mathbb{1}_S(x)$ is the indicator function on a set S

$$\mathbb{1}_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$$

Before introducing the pseudocode we introduce several *trace* predicates to be used.

Definition 5.12.

$trace_C(\iota) = \{\iota' | \exists fs.\ C(\iota, fs) = \iota'\}$
$trace\_local_C(\alpha, \iota) = \{\iota' | \mathcal{O}(\iota) = \alpha, \exists fs.\ C(\iota, fs) = \iota'\}$
$trace\_stop\_reach(ms, \iota) = \{\iota' | \exists fs.\ C(\iota, fs) = \iota', ms(\iota') = R, \nexists fs' \sqsubset fs.\ ms(C(\iota, fs')) = R\}$

5.3 Sending - $ORCA^{Ghost+Val}$

```
α.st = EXECUTE ∧ α.frame = (b, ψ · ψ′)
Sending<α>:
{
  α.st := SEND

  ws := trace_ws_C(α, (b, ψ′))

  for ι ∈ WS:
    if (O(ι) = α):
      α.rc(ι) += 1
    else:
      if α.rc(ι) > 1:
        α.rc(ι) -= 1
      else:
        // Note this now deals with the case for imm objects where rc = 0
        [O(ι).qu.push(orca(ι:256)) | α.grc(ι) += 256]
        α.rc(ι) += 255
    ws := ws \ {ι}

  α.frame := (b, ψ)

  [α′.qu.push(app(b′, ψ′)) |
    for ι ∈ WS:
      if O(ι) = α:
        α.grc(ι) += 1
      else:
        α.grc(ι) -= 1]
}
```

5.4 Receiving - $ORCA^{Ghost+Val}$

```
α.st = IDLE ∧ top(α.qu) = app(φ)
Receiving<α>:
{
  α.st := RECEIVE
  ws := trace_ws_C(α, φ)

  [pop(α.qu) |
    for ι ∈ WS:
      if O(ι) = α:
        α.grc(ι) -= 1
      else
        α.grc(ι) += 1]

  for ι ∈ ws:
    if (O(ι) = α) ⟶ α.rc(ι) -= 1
    if (O(ι) ≠ α) ⟶ α.rc(ι) += 1
    ws := ws \ {ι}

  α.frame := φ

  α.st := EXECUTE

}
```

5.5 Garbage Collection - $ORCA^{\text{Ghost+Val}}$

```
α.st = IDLE ∧ α.st = EXECUTE
GarbageCollection<α>:
{
  α.st := COLLECT
  ms := ∅

  // marking as unreachable
  for ι such that 𝒪(ι) = α ∨ α.rc > 0:
    ms := ms[ι → U]

  // tracing and marking locally accessible as reachable
  for ι ∈ trace_this(α) ∪ trace_frame(α.frame):
    ms := ms[ι → R]

  // marking owned and globally accessible as reachable
  for ι such that 𝒪(ι) = α && α.rc > 0:
    ms := ms[ι → R]
    if (imm_C(ι)) :
      for ι' ∈ trace_local_C(α, ι) :
        ms := ms[ι' → R]


  // collecting
  inc_set = ∅
  dec_set = ∅
  for ι such that ms(ι) = U:
    if 𝒪(ι) = α:
      C.heap := C.heap[ι → ⊥]
      α.rc := α.rc[ι → ⊥]
    else:
      dec_set = dec_set ∪{ι}
      if (imm(ι)):
        inc_set := inc_set ∪ trace_stop_reach_C(ms, ι)

  for ι ∈ inc_set:
      α.rc(ι) += 1
      [𝒪(ι).qu.push(orca(ι: +1)) | α.grc(ι) += 1]

  for ι ∈ dec_set:
      tmp := α.rc(ι)
      α.rc(ι) := 0
      [𝒪(ι).qu.push(orca(ι: -tmp)) | α.grc(ι) := 0]


  if α.frame = ∅:
    α.st := IDLE
  else:
    α.st := EXECUTE
}
```

We add logic for sending the increment messages first then the decrement messages to avoid the situation where after some decrement message is sent, the recipient actor performs collection and frees an object only to have the reference count incremented afterwards.

6 $ORCA^{\text{Ghost+Val}}$ Correctness

6.1 Preservation of $\mathbf{I}_4{}^{Val}$

An equivalent definition of $AMC_C(\iota)$ from $ORCA^{\text{Ghost}}$ and $ORCA^0$ is

$$AMC_C(\iota) = \sum_{a \in C.actors} \sum_{\phi \in a.queue} \mathbb{1}_{trace_C(a,\phi)}(\iota)$$

We when we introduce

$$AMC_C^{Val}(\iota) = \sum_{a \in C.actors} \sum_{\phi \in a.queue} \mathbb{1}_{trace\_ws_C(a,\phi)}(\iota)$$

We can see the only difference is the trace function. But this is the only difference between $\mathbf{I}_4{}^{Val}$ and $\mathbf{I}_4{}^{DS}$, we also know that the pseudocode in $ORCA^{\text{Ghost+Val}}$ changes *trace* to *trace_ws* in the send and receive methods. With this we argue that $\mathbf{I}_4{}^{Val}$ holds here for the same reasons that $\mathbf{I}_4$ held previously.

6.2 Preservation of $\mathbf{I}_3{}^{Val}$

Recall $\mathbf{I}_3{}^{Val}$ states that, for a non owner α of an object ι:

$$\mathcal{A}_C(\alpha, lp) = \iota \longrightarrow \exists \iota'.\ (\iota'.TransProtects_C(\iota) \wedge \alpha.grc_C(\iota') > 0)$$

We need to show that when a message is received all newly accessible objects must satisfy the above. If the message $\phi = (b, \psi)$ and $\iota = \mathcal{A}_C(\alpha, b.x.\overline{f})$ for some variable $x \in \psi$ and path $\overline{f}$. Take $\iota_0 = \mathcal{A}_C(\alpha, b.x)$, we consider two cases:

- $\iota \in trace\_ws_C(\iota_0)$

  We set $\iota' = \iota$ then $\alpha.grc_C(\iota) > 0$ holds by the same argument as $\mathbf{I}_2$ in earlier models.

- $\iota \notin trace\_ws_C(\iota_0)$

  Then there must exist some sequence of fields p such that $b.x.p \sqsubseteq b.x.\overline{f}$ and $\mathcal{A}_C(\alpha, b.x.p) = \iota'$ with $imm_C(\iota')$ and $\iota' \in trace\_ws_C(\iota_0)$. Now either $\iota'.Protects_C(\iota)$ or not. Suppose it does then $\iota'.TransProtects_C(\iota)$ and as $\iota' \in trace\_ws_C(\iota_0)$, $\alpha.grc_C(\iota') > 0$.

  Otherwise, the path from ι' to ι leaves $\mathcal{O}(\iota')$. So $\mathcal{O}(\iota') \neq \mathcal{O}(\iota)$ had a reference to ι, so as $\mathbf{I}_3{}^{Val}$ previously held $\mathcal{O}(\iota')$ must have some ι'' such that $\mathcal{O}(\iota').grc_{C_0}(\iota'') > 0$ and ι'' transitively protects ι. We can ensure that $\mathcal{O}(\iota').grc_C(\iota'') > 0$, i.e. the actor still has this positive reference count as ι' has a reference to it and ι' was sent and is in the immutable trace so we can assume it has a positive reference count at its owner, so ι'' has remained protected. This means that we have $\iota'.TransProtects(\iota)$ and $\alpha.grc_C(\iota') > 0$.

We also need to show that this is preserved through subreferencing and gc. Suppose there is some object ι reachable from a non owning actor α but with a zero reference count and protected by some ι' with positive reference count. Now suppose this ι' is unreachable so during collection will be added to the *dec_set* to have its reference count reduced to zero. The algorithm will now trace through ι' to find a set of reachable, accessible children. Either ι will be found, or some ι'' that transitively protects ι, this will be added to the *inc_set*. Then, before the reference count to ι' is removed, a reference count to ι'' (wlog) will be added and an orca message sent to its owner which will arrive *before the decrement message*.

6.3 Preservation of $\mathbf{I}_2{}^{Val}$

If an object ι is accessible either by a foreign actor α or by a message path to its owner α_0.

$$\mathbf{I}_2{}^{\text{Val}} \triangleq \exists \alpha', p', \iota'.\ (\alpha' = \alpha \wedge \mathcal{A}_C(\alpha, p) = \iota \wedge p' \sqsubseteq p) \vee (\alpha' = \alpha_0 \mathcal{A}_C(\alpha_0, mp) = \iota \wedge p' \sqsubseteq mp) \\ \longrightarrow \mathcal{A}_C(\alpha', p') = \iota' \wedge \iota'.Protects(\iota) \wedge \alpha_0.grc_C(\iota') > 0$$

We can break this down into the following cases:

- Case 1

  $\mathcal{A}_C(\alpha, lp) = \iota$, by $\mathbf{I}_3{}^{Val}$ we have that there is an object that transitively protects it with $\alpha.grc_C(\iota) > 0$. By the definition of transitive protection there must exists ι' with $\mathcal{O}(\iota') = \alpha_0$, $\iota'.Protects(\iota)$ such that there exists α' with $\alpha'.grc_C(\iota') > 0$. This implies by $\mathbf{I}_4{}^{Val}$ that $\alpha_0.grc_C(\iota') > 0$.

- Case 2

  p starts from a queue, $p = k.x.\overline{f}$ that is either accessible from α_0 or α. Either way take the actor as α' and let $\iota_0 = \mathcal{A}_C(\alpha, k.x)$ then if $\iota \in trace\_ws_C(\iota_0)$ then AMC is positive for ι. Otherwise, there is an object ι' with $b.x.p \sqsubset b.x.\overline{f}$, $\mathcal{A}_C(\alpha', b.x.p) = \iota'$ and $\iota' \in trace\_ws_C(\iota_0)$. Either ι was accessible from a foreign actor before in which case the positive AMC for ι' preserves the invariant, or ι' was constructed as an immutable object with a reference to ι, so $\mathcal{O}(\iota') = \mathcal{O}(\iota) = \alpha_0$ so $\iota'.Protects(\iota)$ and ι' has a positive reference count from α_0 by $\mathbf{I}_4{}^{Val}$ and the positive AMC.

- Case 3

  p starts from a queue currently being processed, $p = -1.x.\overline{f}$, then this holds by a similar argument and $\mathbf{I}_8$.

6.4 Completeness

We know that $ORCA^0$ is complete, that after a finite number of steps any globally unreachable object is collected and hope to show a similar result for $ORCA^{\text{Ghost+Val}}$. However, it is possible to construct an immutable cycle, which results in a cycle of transitive protections and so can never be collected. An example is detailed in Figure 16.

What we can do is prove that if an object is globally unreachable but cannot be collected then it must have a cycle in its object graph, so this is the only case where it can occur.

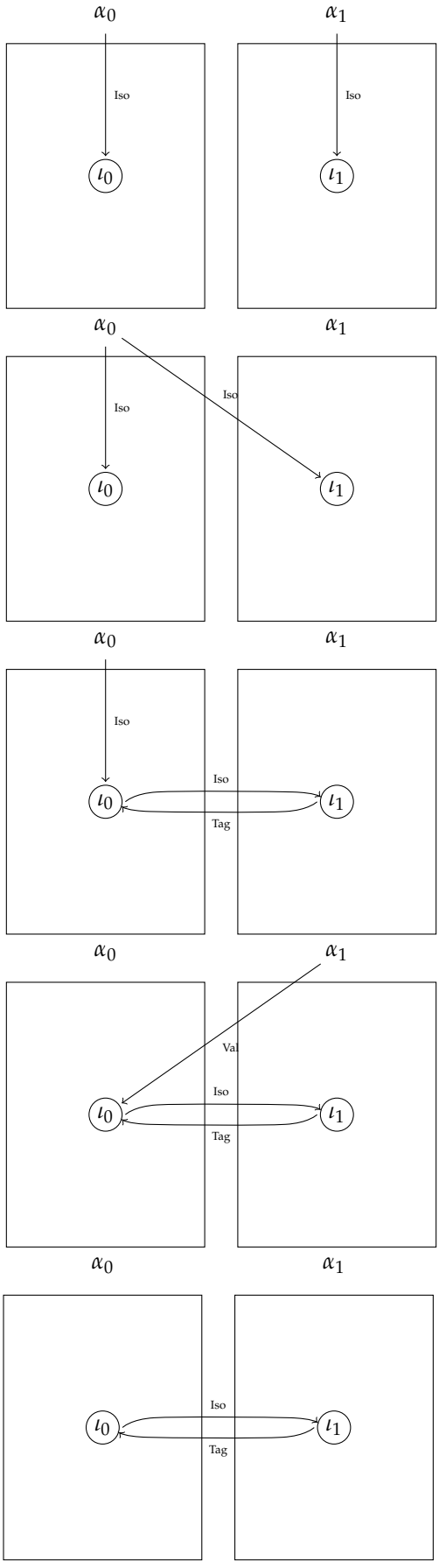


Figure 16: Constructing an immutable object cycle across actors in $ORCA^{\text{Ghost+Val}}$

```
use "collections"

// Pony code to produce an object with an immutable cycle across
// actors
actor Main
  var f : (A iso | None)
  var alice : Alice
  new create(env: Env) =>
    alice = Alice
    f = None
    alice.make(this)

  be make(x : A iso) =>
    var y : A iso = recover A end
    x.g = recover y end
    y.f = consume x
    var vl : A val = recover consume y end
    alice.take(consume vl)
    alice.make(this)


actor Alice

  var f : (A val | None)
  new create() =>
    f = None

  be make(x : Main) =>
    let y : A iso = recover
      A
    end
    x.make(consume y)

  be take(x : A val) =>
    f = consume x

class A
  var f : ( A iso | None )
  var g : ( A tag | None )
  new create() =>
    f = None
    g = None
```

Figure 17: Code listing in Pony for producing an immutable object cycle across two actors.

6.5 Cycle-Free Objects

We now show that for a configuration made up of cycle-free objects, $ORCA^{\text{Ghost+Val}}$ is complete.

Recall. *We defined that an object ι is collectable at C if there is no local path from the owner to the object, it has a local reference count of zero, and there is no other object that directly protects it with a positive reference count.*

$$\begin{aligned} collectable_C(\iota) \triangleq \quad & \nexists lp.\ \mathcal{A}_C(\alpha, lp) = \iota \\ & \wedge\ \alpha.rc_C(\iota) = 0 \\ & \wedge\ (\nexists \iota'.\iota'.Protects(\iota) \wedge \alpha.rc_C(\iota') > 0) \\ \text{where } \alpha = \mathcal{O}(\iota) & \end{aligned}$$

Definition 6.1. We say that an object ι is *Possibly Leaked* at a configuration C if it is not collectable, but there are no paths from actors or message queues to the object.

$$PossLeak_C(\iota) \triangleq\ \neg collectable(\iota) \wedge (\nexists \alpha, p.\ \mathcal{A}_C(\alpha, p) = \iota)$$

Theorem 6.2 (Completeness of Cycle-Free objects)**.** *Given an object ι in a configuration C such that $PossLeak_C(\iota)$ then there is a cycle in the object graph of ι.*

Proof. So *PossLeak* states that there are no paths from any actors, which means that there is definitely no path from the owner, so we have

$$PossLeak_C(\iota) \iff (\alpha.rc(\iota) \neq 0 \vee \exists \iota'.\iota'.Protects(\iota) \wedge \alpha.rc_C(\iota') > 0) \wedge (\nexists \alpha, p.\ \mathcal{A}_C(\alpha, p) = \iota)$$

Suppose there are no cycles in the object graph of ι (the set of objects accessible to, and accessible from ι), then the object graph forms a finite, directed acyclic graph (DAG).

By induction on the size of this graph we show that this will be collected in a finite number of steps.

Base Case: there are no objects that point to ι, so there are no objects protecting it, and after a finite number of steps its reference count will go to zero as it is unreachable from all actors that may have had foreign references.

Inductive Case: Suppose ι is part of an object graph of size n, if we consider it a poset under accessibility, that is $\iota_0 \sqsubseteq \iota_1$ if and only if ι_0 is accessible via some path from ι_1. Assuming the axiom of choice, we know from set theory that Zorn's lemma states that if every chain has an upper bound (all chains are finite here), then the poset has at least one maximal element.

In this case a maximal element is an object $\iota^{\top}$ that is not accessible by anything in the object graph. By assumption there are no actors that can reach any such element, so we can choose one arbitrarily and be sure that like in the base case, it will be collected in a finite number of steps.

Either we chose ι itself or we have reduced in finite steps, the graph down size $n - 1$ which by inductive assumption, shows ι will be collected in a finite number of steps. □

6.6 Evaluation

Being able to generalise invarients $\mathbf{I}_2$ and $\mathbf{I}_3$ allowed us to keep the proof structure which allows the arguments to be communicated far easier.

The main problem with this model and perhaps its greatest failing is with collection of cycles. Both the current implementation and our model cannot collect cycles of immutable objects across actors. We considered approaches for cycle detection and collection. We tried to construct an algorithm idiomatic with the design of ORCA relying only on message passing. An object graph would be identified and the owners sent a message asking them to make a judgement on whether they believed it to be a cycle. But because of concurrent scheduling there could be an active reference passed around avoiding any actors currently making a judgement.

An alternative approach that might be possible with the current implementation is to trace the entire object graph of an object and look for cycles the first time an object is seen as immutable. Then an immutable object structure can never form a cycle by adding individual objects.

We also considered a definition of protection where there was an "inferred" reference count for an object that relied on the reference counts of all objects that protect it. This would be done either with a sum or by taking a maximum. But this ended up in a model further from $ORCA^0$ or $ORCA^{\text{Ghost}}$ and had difficulty dealing with any cycles let alone those across objects.

7 Immutability

Up to this point we have unrealistically assumed that the immutability of an object is known to all actors. In this section we give several ways of embedding this information in the runtime.

We need to know the mutability of an object in three places, when we send an object, when we receive an object, and during garbage collection. In the first two cases, we have to consider the objects being sent and anything that can be referenced from them. From the assumptions of the host langauge, if we have an object ι sent or received, it can only ever have an `Iso`, `Val` or `Tag` capability. If ι is `Iso` then we know that there are no external writable references, to it or any of its children that ι has a read reference too. This distinction is important because it is possible for ι to have `Tag` references to mutable objects. However we know that if an object has a readable path from ι and does not have a writable path from ι then it is immutable.

If a variable is sent with `Val` capability, it can also have `Tag` children but we know that all children with a read capable path are immutable.

Finally if ι is sent with `Tag` then there are no accessible children, and we can make no judgements about the immutability of ι.

During collection there are two cases where we need to determine whether an object is immutable. Firstly when we own an object ι, with a positive reference count we if ι is immutable then we mark its trace as reachable. Also if we have an unreachable foreign object with a positive reference count, if this object is immutable we need to check to see if we have subreferenced it and need to alter the reference counts of any other objects.

A core problem is where we have been sent an immutable object and made a `Box` alias to it or one of its children. This reference is ambiguous because it could refer to a mutable object we also have a `Ref` reference to.

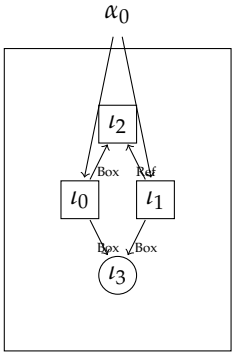


Figure 18: Here ι_2 is mutable as there is at least one write reference that can reach it whereas all references to ι_3 are read only.

7.1 Immutability in objects

We aim to construct local immutability functions, partial functions valid over some subset of objects to replace positions in the code where we test the immutability of an object using the imm_C predicate. The local functions must give answers consistent with the predicate for objects they defined for.

The cases we must deal with are for objects we are sending, objects we are receiving, owned objects with a positive reference count during collection, and unowned objects during collection. Note that until an object is sent it cannot fall into any of these categories.

The simplest approach is to embed in every object a boolean field denoting whether this object is immutable or not. This is set to mutable on object creation and only set when the object is sent

as immutable. Even if we send an object ι as `Iso` it is still possible for some of its children to be immutable. Because as discussed above, ι is `Iso` so all possible write-capable paths to a child ι' must have write capability from ι.

When we send or receive a frame we construct the set of mutable objects in that frame, objects with a write-capable path from the actor. We can use this to test if an object is immutable by checking inclusion.

Therefore we alter the *trace_ws* predicate to take a lambda expression for testing if an object is immutable $trace\_ws(\alpha, \phi, \lambda)$. This lambda is a local immutability function defined on objects reachable from the frame ϕ.

In the special case where the frame contains just one val object, the mutable set is empty.

Now for all other immutability checks we use $\iota.imm$ which is now well defined. This may be altered in further sends, but as we know from assumptions about the language immutability is persistent, so we can never reconstruct a mutable object.

7.2 Set of immutable references

Alternatively, we propose a scheme where we give each actor a set of *immutable roots*. This set denotes the objects and children who must be considered immutable in order to satisfy invariants ${\mathbf{I}_2}^{Val}$ and ${\mathbf{I}_3}^{Val}$.

Definition 7.1 (Well Formed Roots)**.** We say a set of objects X for an actor α at a configuration C is a set of *well formed roots* if the set obeys the following properties:

- All objects in the set are immutable
- If α has a path to some object not owned by α and has a zero reference count to the object then either that object or an object that protects it is in the immutable roots
- If some other actor $\alpha' \neq \alpha$ has a path to some object owned by α and α has a zero reference count to the object then either that object or an object that protects it is in the immutable roots of α.

$$
\begin{aligned}
&C \vDash_{WFR} \alpha, X \iff \\
&\qquad (a.) \quad \forall \iota.\ [\iota \in X \to imm_C(\iota)] \\
&\qquad (b.) \quad \forall \iota, p.\ [\mathcal{A}_C(\alpha, p) = \iota \wedge \mathcal{O}(\iota) \neq \alpha \wedge \alpha.grc_C(\iota) = 0 \longrightarrow \\
&\qquad\qquad \exists \iota', p'.\ (p' \sqsubset p \wedge \mathcal{A}_C(\alpha, p') = \iota' \wedge \iota'.TransProtects(\iota) \wedge \iota' \in X \wedge \alpha.grc_C(\iota') > 0)] \\
&\qquad (c.) \quad \forall \iota, \alpha', p.\ [\mathcal{A}_C(\alpha', p) = \iota \wedge \mathcal{O}(\iota) = \alpha \neq \alpha' \wedge \alpha.grc_C(\iota) = 0 \longrightarrow \\
&\qquad\qquad \exists \iota', p'.\ (p' \sqsubset p \wedge \mathcal{A}_C(\alpha', p') = \iota' \wedge \iota'.Protects(\iota) \wedge \iota' \in X \wedge \alpha.grc_C(\iota') > 0)]
\end{aligned}
$$

We introduce a new field for each actor $\alpha.ImmRoots$ that we will argue maintains this well formed roots property for α.

Definition 7.2 ($\mathbf{I}_{11}$)**.** The invariant $\mathbf{I}_{11}$ states that at configuration C the immutable roots sets of all actors satisfy the well formed roots property.

$$\mathbf{I}_{11} \triangleq\ \forall \alpha.\ C \vDash_{WFR} \alpha, \alpha.ImmRoots_C$$

We can think of the well formed roots condition as bounds on the size of the immutable roots set. At most it can contain all immutable objects in the configuration, we know from monotonicity of capabilities that if something is immutable it can never become mutable again. But the immutable roots must be at least the set containing enough objects to check whether an object being sent or collected is immutable or not.

The approach that we will take is to adjust the tracing on message send and receipt. As we stated earlier, conditions where immutability matters do not occur until a foreign actor has access to an object. We adjust the tracing to construct a set of objects to add to the immutable roots to preserve the well formed roots property. We have the problem of subreferencing however, what if we have an object in the immutable roots and drop our reference to it but retain a reference to a child? The approach we take is a lazy form of modification, where we adjust the garbage collection to perform an extra trace on dropped immutable roots members.

We will leverage the type system and use a trick to infer the immutable objects being sent. Assume we are sending an object ι with `Iso` capability. Now we know that there are no external read-capable or write-capable references to ι or any sub objects by the type system of the host language. We then know that anything not reachable with write capability from ι has no *write*

capable references globally, and therefore are immutable. So if we trace the write capable objects from ι, something we will always have to do, then the complement of that set in the complete trace is exactly the set of immutable objects.

Further, we know that the working set contains the write-capability set, so if we look at the difference of these two sets we compute a "boundary" between the mutable and immutable objects. These are the immutable objects that are given reference counts, and these are the objects added to the immutable roots set.

During collection, the two cases where we test for immutability we check for inclusion in the immutable roots. This does *not* accurately model immutability for the objects being checked. However, because the code executed conditionally on immutability always performs some kind of trace, and because immutable children always outlive their immutable parents, we can argue that if a child is every incorrectly classified as not being immutable when they are, then there is some parent that is classified that will trigger the same effect through the trace.

$$\begin{aligned} write_C(\alpha,\phi) &= \{\ \iota \mid \phi = (b,\psi), \exists x, \overline{f}.\ \ \mathcal{A}_C(\alpha, b.x.\overline{f}) = \iota\ \wedge \alpha \vDash_C \iota : write\} \\ read_C(\alpha,\phi) &= \{\ \iota \mid \phi = (b,\psi), \exists x, \overline{f}.\ \ \mathcal{A}_C(\alpha, b.x.\overline{f}) = \iota\ \wedge \alpha \vDash_C \iota : read\} \\ trace\_ws_C(\alpha,\phi) &= (write_C(\alpha,\phi), read_C(\alpha,\phi) \cap ws_C(\alpha,\phi), ws_C(\alpha,\phi)) \end{aligned}$$

$$write_C(\alpha,\phi) \subseteq read_C(\alpha,\phi)$$

This is clear as write capability implies read capability.

$$write_C(\alpha,\phi) \subseteq ws_C(\alpha,\phi)$$

This is true from the definition of the ws, we make sure to trace all mutable objects.

$$\begin{aligned} & write_C(\alpha,\phi) \subseteq read_C(\alpha,\phi) \\ & \longrightarrow \\ & write_C(\alpha,\phi) \cap ws_C(\alpha,\phi) \subseteq read_C(\alpha,\phi) \cap ws_C(\alpha,\phi) \\ & \longrightarrow \\ & write_C(\alpha,\phi) \subseteq read_C(\alpha,\phi) \cap ws_C(\alpha,\phi) \subseteq ws_C(\alpha,\phi) \end{aligned}$$

So *trace_ws* creates three increasing sets. We use $read_C(\alpha,\phi) \setminus write_C(\alpha,\phi)$ to approximate a set of immutable objects. Note that this does not include any objects with tag reference.

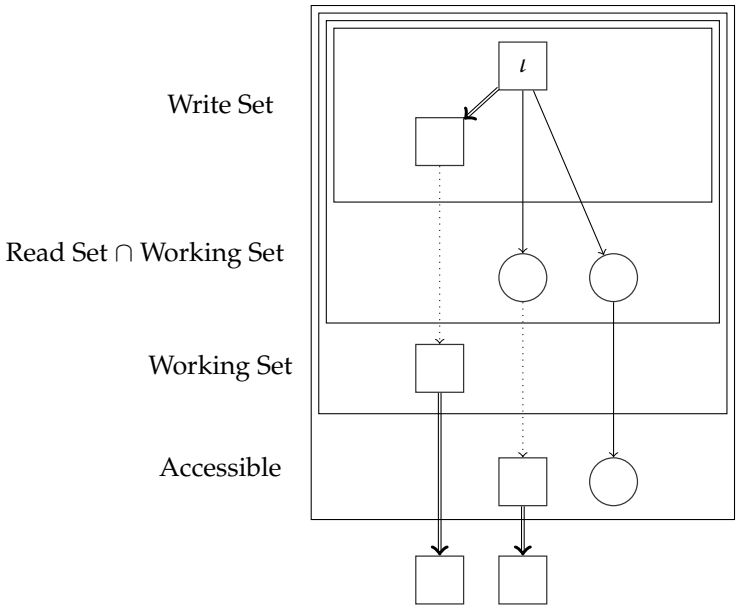


Figure 19: Again we use the double arrow to denote an `Iso` reference. This shows the trace of some object ι where dotted lines are tag references. This shows the nested subset relation between the write set, read set, working set and full trace.

We give the pseudocode for this approach below:

Send - *ORCA*$^{\text{Ghost+Val}}$ with Immutability

```
1  α.st = EXECUTE ∧ α.frame = (b, ψ · ψ′)
2  Sending<α>:
3  {
4    α.st := SEND
5
6    [ (write, read, ws, seen) := trace_ws(C, α, ψ) | WS := ws]
7    α.imm_roots := α.imm_roots ∪ (read \ write)
8
9    for ι ∈ ws:
10     if (O(ι) = α) ⟶ α.rc(ι) += 1
11     if (O(ι) ≠ α ∧ α.rc(ι) > 1) ⟶ α.rc(ι) -= 1
12     // Note this now deals with the case for imm objects where rc = 0
13     if (O(ι) ≠ α ∧ α.rc(ι) ≤ 1) ⟶ [O(ι).qu.push(orca(ι:256)) | α.grc(ι) += 256]
14     if (O(ι) ≠ α ∧ α.rc(ι) ≤ 1) ⟶ α.rc(ι) += 255
15     ws := ws \ {ι}
16
17
18   α.frame := (b, ψ)
19
20   [α′.qu.push(app(b′, ψ′)) |
21     for ι ∈ WS:
22       if O(ι) = α:
23         α.grc(ι) += 1
24       else:
25         α.grc(ι) -= 1]
26 }
```

Receive - $ORCA^{\text{Ghost+Val}}$ with Immutability

```
α.st = IDLE ∧ top(α.qu) = app(φ)
Receiving<α>:
{
  α.st := RECEIVE

  [ (write, read, ws, seen) := trace_ws(C, α, φ) | WS := ws]
  α.imm_roots := α.imm_roots ∪ (read \ write)

  [pop(α.qu) |
    for ι ∈ WS:
      if O(ι) = α:
        α.grc(ι) -= 1
      else
        α.grc(ι) += 1]

  for ι ∈ ws:
    if (O(ι) = α) ⟶ α.rc(ι) -= 1
    if (O(ι) ≠ α) ⟶ α.rc(ι) += 1
    ws := ws \ {ι}

  α.frame := φ

  α.st := EXECUTE

}
```

$$trace_C(\iota) = \{\iota' | \exists \overline{f}. \mathcal{A}_C(\iota, \overline{f}) = \iota\}$$
$$trace\_local_C(\alpha, \iota) = \{\iota' | \mathcal{O}(\iota) = \alpha, \exists fs.\ C(\iota, fs) = \iota'\}$$
$$trace\_stop\_reach(ms, \iota) = \{\iota' | \exists fs.\ C(\iota, fs) = \iota', ms(\iota') = R, \nexists fs' \sqsubset fs.\ ms(C(\iota, fs')) = R\}$$

Garbage Collection - $ORCA^{\text{Ghost+Val}}$ with Immutability

```
α.st = IDLE ∧ α.st = EXECUTE
GarbageCollection<α>:
{
  α.st := COLLECT
  ms := ∅

  // marking as unreachable
  for ι such that O(ι) = α ∨ α.rc > 0:
    ms := ms[ι → U]

  // tracing and marking locally accessible as reachable
  for ι ∈ trace_this(α) ∪ trace_frame(α.frame):
    ms := ms[ι → R]

  // marking owned and globally accessible as reachable
  for ι such that O(ι) = α && α.rc > 0:
    ms := ms[ι → R]
    if (ι ∈ α.imm_roots) :
```

```
19          for ι′ ∈ trace_local_C(α, ι) :
20              ms := ms[ι′ → R]
21
22
23      // collecting
24      inc_set = ∅
25      dec_set = ∅
26      add_imm_roots = ∅
27      rem_imm_roots = ∅
28      for ι such that ms(ι) = U:
29          if O(ι) = α:
30              C.heap := C.heap[ι → ⊥]
31              α.rc := α.rc[ι → ⊥]
32              if (ι ∈ α.imm_roots):
33                  add_imm_roots := add_imm_rootss ∪ trace_stop_reach(ms, ι)
34                  rem_imm_roots := rem_imm_roots ∪ {ι}
35          else:
36              dec_set = dec_set ∪{ι}
37              if (ι ∈ α.imm_roots):
38                  inc_set := inc_set ∪ trace_stop_reach(ms, ι)
39                  add_imm_roots := add_imm_roots ∪ trace_stop_reach(ms, ι)
40                  rem_imm_roots := rem_imm_roots ∪ {ι}
41
42      α.imm_roots := α.imm_roots ∪ add_imm_roots
43      for ι ∈ inc_set:
44          α.rc(ι) += 1
45          [O(ι).qu.push(orca(ι: +1)) | α.grc(ι) += 1]
46
47
48      α.imm_roots := α.imm_roots \ rem_imm_roots
49      for ι ∈ dec_set:
50          tmp := α.rc(ι)
51          α.rc(ι) := 0
52          [O(ι).qu.push(orca(ι: -tmp)) | α.grc(ι) := 0]
53
54
55      if α.frame = ∅:
56          α.st := IDLE
57      else:
58          α.st := EXECUTE
59  }
```

```
trace_ws :: Config →Actor →Path →[OID] →[OID] →[OID] →[OID]
            →([OID], [OID], [OID], [OID])
trace_ws cfg a p write read ws seen
        -- Already Seen
  = if | oid `elem` seen →(write, read, ws, seen)
       -- No capabilities (tag reference)
       | not (readPath cfg a p) →(write, read, oid:ws, oid:seen)
       -- Read capability
       | not (writePath cfg a p) →(write, recRead, recWs, recSeen)
       -- Read + Write capability
       | otherwise →(recWrite, recRead, recWs, recSeen)
  where
    obj@(Object oid fields) = lookupObj cfg a p
    fields' :: [FieldId]
    fields' = [ x | ((x , k), _) <- fields]

    f :: ([OID], [OID], [OID], [OID]) →FieldId →([OID], [OID], [OID], [OID])
    f (ws, rs, xs, ss) fid = trace_ws cfg a (p ++ [fid]) ws rs xs ss

    (write', read', ws', seen') = foldl f (write, read, ws, oid:seen) fields'
    recWrite = oid:write'
    recRead = oid:read'
    recWs = oid:ws'
    recSeen = seen'
```

Figure 20: An example recursive function implementation of trace_ws that efficiently generate the write, read and working sets for that trace of an individual path. It has an additional return set that is used to keep track of the addresses it has seen to prevent getting stuck in a loop in the object graph.

7.3 Correctness

We now show the correctness of the approach. The first thing we need to show is that elements we put into the immutable roots set are all immutable. That is:

Proposition 7.3. *If $read_{C,\alpha}$ and $write_{C,\alpha}$ are formed by performing a trace on an object ι from an actor α at a valid configuration C (where all invarients hold) then all the objects in $read_{C,\alpha} \setminus write_{C,\alpha}$ are immutable.*

$$\forall \alpha, \iota.\ \iota \in (read_{C,\alpha} \setminus write_{C,\alpha}) \longrightarrow imm_C(\iota)$$

Proof. As $\iota \in read$ we have that $\iota \in ws$. So we have that $\exists x, \overline{f}.\ \mathcal{A}_C(\alpha, b.x.\overline{f}) = \iota$. Without loss of generality, assume that this path $b.x.\overline{f}$ has the maximal capability of any path from α to ι.

So we also have some ι' and κ such that $\mathcal{A}_C(\alpha, b.x) = (\iota', \kappa)$.

We can only have that κ is either `Iso`, `Val` or `Tag` as those are the only capabilities that the language is allowed to send.

If $\kappa =$ `Tag` then $b.x.\overline{f} = b.x$ and $\iota = \iota'$ as no children from a `Tag` reference are accessible, but this is a contradiction as ι' then does not have read access so $\kappa \neq$ `Tag`.

If $\kappa =$ `Val` then we know from the host language that ι' and all its read-capability children have no write capable reference, so specifically we have $imm_C(\iota)$ as ι has a read-capable reference.

If $\kappa =$ `Iso` then for all its read-capable children, *all write capable references must be dominated by* ι' as a guarantee of the language. Specifically, we know ι has a read-capable reference, and we know there can be no foreign write capable references, and there is no write capable reference from ι', so ι must be immutable. □

Corollary 7.4. *In the sending and receiving pseudocode the line*

```
1 α.imm_roots := α.imm_roots ∪ (read \ write)
```

preserves the well formed roots property.

Now we wish to argue that the elements of $read_{C,\alpha} \setminus write_{C,\alpha}$ are sufficient to preserve the well formed roots property on receipt of a new message.

Proposition 7.5. *If some set X satisfies $C \vDash_{WFR} \alpha, X$ for some α in a configuration C with all invariants holding at C and $C \rightsquigarrow C'$ as α pushes the frame (b, ψ) onto the message queue of some α', then:*

$$C' \vDash_{WFR} \alpha, X \cup (read_{C,\alpha} \setminus write_{C,\alpha})$$

Proof. From corollary 7.4, we know that adding this term to the immutable roots preserves property (a.) that all objects in the set are immutable. We just need to prove property (c.) that all foreign-accessible owned objects with zero reference count from α are in the immutable roots of α.

As $C \vDash_{WFR} \alpha, X$ we assume that there is some ι, owned by α, made accessible to α' by the push of (b, ψ) with $\alpha.rc_{C'}(\iota) = 0$.

So $\iota \in trace(\psi)$ and is therefore reachable by some path $b.x.\overline{f}$. But $\iota \notin ws$ as otherwise it would have a positive reference count from α. So as it is not in the working set there must be some immutable ι' such that $\iota' \in ws$, $b.x.p \sqsubset b.x.\overline{f}$ and $\mathcal{A}_C(\alpha, b.x.p)$ and ι' has a reference to ι. This ι' must also have a positive reference count.

□

Proposition 7.6. *If some set X satisfies $X \vDash_C WFR_\alpha$ for some α in a configuration C with all invariants holding at C and $C \rightsquigarrow C'$ as α sets its frame to (b, ψ) it has received from its message queue, then:*

$$C' \vDash_{WFR} \alpha, X \cup (read_{C,\alpha} \setminus write_{C,\alpha})$$

Proof. Similar to Proposition 7.5 □

Proposition 7.7. *At some configuration C where all invariants hold and the garbage collection of α is being run, for all objects ι owned by α if ι is accessible from some foreign actor α' then it is marked as reachable.*

Proof. If $\alpha.grc_C(\iota) > 0$ then this is clear so assume $\alpha.grc_C = 0$. Then from the well formed roots property there exists some $\iota' \in \alpha.ImmRoots_C$ such that $\iota'.Protects(\iota)$ so $\iota \in trace(\iota')$.

Note because we have Protects we rather than TransProtects it is sufficient to trace locally!

□

Proposition 7.8. *During garbage collection we add the add_imm_roots to the immutable roots, this preserves the well formed roots property.*

Proof. As we know that all elements of the immutable roots are immutable and the *add_imm_roots* is constructed from objects in the trace of objects in the immutable roots, these objects are all immutable too. □

Proposition 7.9. *If some set X satisfies $X \vDash_C WFR_\alpha$ for some α in a configuration C at the beginning of garbage collection with all invariants holding at C and $C \rightsquigarrow^* C'$ as α performs garbage collection then:*

$$C' \vDash_{WFR} \alpha, (X \cup add\_imm\_roots) \setminus rem\_imm\_roots$$

Proof. From Proposition 7.8 the union is valid, we also assume that because we are executing garbage collection no local objects can become reachable by a foreign actor and no foreign objects can become reachable to this α.

Objects in the *rem_imm_roots* are foreign and unreachable from α, so we need to worry about the second condition, however the *add_imm_roots* is constructed in such a way that if some reachable object was transitively protected by an object in the *rem_imm_roots* there is now an object transitively protecting it in the add set. By the definition of *trace_stop_reach* if ι has a path from ι_0 and ι is marked as reachable in the marking set *ms*, then there is an object $\iota' \in trace\_stop\_reach(ms\iota_0)$ either with $\iota' = \iota$ or ι is accessible from ι'. Because both are children of an object in the immutable roots they are immutable so $\iota'.TransProtects(\iota)$ □

Therefore, we argue $\mathbf{I}_{11}$ is preserved by execution.

7.4 Evaluation

Unfortunately we were unable to perform a thorough analysis of the differences, particularly in performance of this model of immutability with that of the Pony runtime. Our given lazy determination of immutability is expensive during garbage collection but provides very cheap sends and receives. Whereas the Pony runtime uses a technique closer to the first we gave where reference counts are given a flag determining whether the object has been seen as immutable. This requires additional message sends for notifying the owner when the object is first seen as immutable.

If we had more time it would have been interesting to see the conditions where each algorithm succeeds, and perhaps gather benchmarks. We hypothesise that for message-heavy benchmarks our proposal would be more efficient, whereas for message-sparse benchmarks it would fall short.

8 *CALF* - Reference Counting Submodel

8.1 Motivation

We saw when describing $ORCA^0$ and $ORCA^{\text{Ghost}}$ there were two main problems. We had to carefully consider the our invariants under fine grained concurrency. We also had to deal with all aspects of the protocol at all points, the accessibility, the reference counts and the state of the workings sets.

In this section we build a submodel - *CALF* which only focuses on the reference counts. We also formalise the following argument: Given a configuration, we should be able to reason about the invariants *as if all send and receive events in all actors have finished*, and this should in some way be equivalent to reasoning about our current configuration.

Recall (Reference Counting Equation)**.**

$$LRC + OMC = FRC + AMC$$

If we can show that at every configuration, when the sending and receiving has been resolved, the reference counting equation holds then we could argue as follows.

If we have some actor with an owned object ι and a zero reference count and we know that we cannot have any orca messages, then because the reference counting equation holds when other actors are finished with their actions, there is no way that any of them could access ι.

8.2 Histories

In this section we aim to build a simpler model, *CALF* focused only on the reference counts and how they are altered during execution. We then define a relation between *CALF* and the larger model such that certain properties of the smaller model carry over. In particular we want to show that when we are performing garbage collection in the larger model with a local reference count of zero for some object then all other actors have zero reference count also to the object so it is safe to collect.

We will define a configuration, then three *protocols*: **send**, **receive**, and **drop**. These will be split up into multiple *events* each with corresponding execution rules which will represent atomic blocks that will be executed nondeterministically. For our model we will split each protocol into a begin event, several mid events and an end event.

A *history*, H is a list of events [Herlihy and Wing, 1990]. Each event will be of the form $e(p, \alpha, X)$ where p is the protocol, α is the actor and X is the set of objects it is acting upon. We can use this to *restrict* histories to an actor and a set of objects. We compute this by filtering the list of events based on equality of actors and intersection of objects. If $e(p, \alpha, X) \in H$ then $x \in H|\alpha', Y$ if an only if $\alpha = \alpha'$ and $X \cap Y \neq \emptyset$.

The ordering of the restricted history is always preserved.

Example 8.1. *An example of restricting the history H to $H|\alpha, \{1\}$:*

$$H = [begin(p1, \alpha, \{1, 2\}, mid(p2, \alpha', \{1\}, mid(p1, \alpha, \{2\}]$$
$$H|\alpha, \{1\} = [begin(p1, \alpha, \{1, 2\}]$$

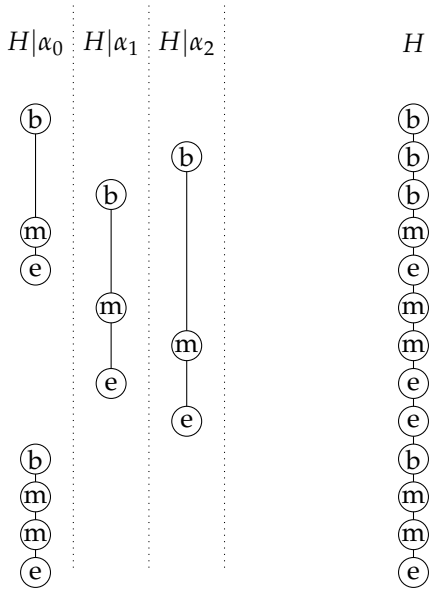


Figure 22: An example of a valid history H and its restriction to three actors.

Definition 8.2 (Story)**.** For our model we call a list of events a story if they are of the form

$$\begin{aligned} &begin(p, \alpha, X) \\ &mid(p, \alpha, \{\iota_1\}) \\ &mid(p, \alpha, \{\iota_2\}) \\ &\qquad \vdots \\ &mid(p, \alpha, \{\iota_n\}) \\ &end(p, \alpha, X) \end{aligned}$$

For some fixed protocol p, actor α and set of objects X where X is exactly the set $\{\iota_1, \ldots \iota_n\}$.

Definition 8.3 (Sequential)**.** A history H restricted to an actor α and a set of objects X, $H|\alpha, X$ is *sequential* if and only if it is a sequence of stories.

$$H|\alpha, X = [b, m \ldots, m, e, b, m, \ldots, m, e, \ldots]$$

Definition 8.4 (Valid History)**.** A history H is *valid* if for all actors α and all sets of objects X, $H|\alpha, X$ is sequential. This includes Ω the set of all objects, so we are stating that no actor can concurrently be executing two protocols.

We can also use Ω to restrict to just an actor, for which we will use the notation $H|\alpha$.

We define a configuration consisting of a countable number of actors and an ownership function from objects to actors. Each actor contains a reference count function from objects to an integer, a series of sets for ghost state while executing protocols and a message queue that maps objects to the increment or decrement the message contains.

The *SWS* is the **S**ending **W**orking **S**et, similarly the *RWS* is the **R**eceiving **W**orking **S**et. Finally *FS* is the shared, **F**inalise **S**et. We will reuse the *SWS* for drop events because they have the same signature and will be used in disjoint contexts.

Definition 8.5 (*CALF* Configuration)**.**

$$
\begin{array}{ll}
C = (a_0, a_1, a_2, \dots \quad \dots, O) & \in \mathcal{C} \\
a_i = (rc, SWS, RWS, FS, msgs) & \in \mathcal{A} \\
O = \lambda \iota.\ \alpha & \in \mathbb{N} \to \mathbb{N} \text{ is an ownership function} \\
rc & \in \mathbb{N} \to (\mathbb{N} \cup \{\bot\}) \\
RWS & \in \wp(\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\
SWS & \in \wp(\mathbb{N} \times \mathbb{N}) \\
FS & \in \wp(\mathbb{N} \times \mathbb{N}) \\
msgs & \in [(\mathbb{N} \to \mathbb{Z})]
\end{array}
$$

We write $C \rightsquigarrow_x C'$ if the execution of the rule corresponding to event x takes C to C'.

A list of these forms a history so we adopt the notation $C_0 \rightsquigarrow_H C_n$ for $C_0 \rightsquigarrow_{x_1} C_1 \cdots \rightsquigarrow_{x_n} C_n$ where $H = [x_1, \dots x_n]$.

Definition 8.6 (Closed Configuration)**.** We say that a configuration C is *closed* for an actor α and a set of objects X if for all actors, their reference count for all the objects is defined.

$$closed(C|\alpha, X) \triangleq\ \forall \alpha, \iota \in X.\ C.\alpha.rc(\iota) \neq \bot$$

Note that this is an abuse of notation, C is a configuration not a history, but they allude to the same thing. We will also be writing $O(\iota)$ for the owner instead of $C.O(\iota)$ because we can assume a fixed ownership map throughout, this will be explained further when we show how objects are related between an ORCA model and *CALF* .

Lemma 8.7.

$$closed(C|\alpha, X) \wedge closed(C|\alpha, Y) \iff closed(C|\alpha, (X \cup Y))$$

Proof. This follows clearly from the definition, two sets of objects have reference count $\bot$ if and only if the union of them does. □

8.3 Event Rules, Send

$$\frac{\begin{array}{c}\forall \iota \in X.\ rc'(\iota) = \bot \qquad \forall \iota \notin X.\ rc'(\iota) = C.\alpha.rc[\iota] \\ C.\alpha.SWS = \emptyset \qquad C.\alpha.RWS = \emptyset \qquad C.\alpha.FS = \emptyset \qquad \forall \iota \in X.\ O(\iota) \neq \alpha \longrightarrow C.\alpha.rc(\iota) > 0\end{array}}{C \rightsquigarrow_{begin(send,\alpha,X)} C\ \ [\alpha.SWS \mapsto \{(\iota, C.\alpha.rc(\iota)) | \iota \in X\},\ \alpha.rc \mapsto rc']}\ \text{begin(send, } \alpha\text{, X)}$$

$$\frac{C.\alpha.SWS = \{(\iota, 1)\} \uplus sws \qquad O(\iota) \neq \alpha}{C \rightsquigarrow_{mid(send,\alpha,\{\iota\})} C\big[O(\iota).msgs \mapsto C.O(\iota).msgs{+}{+}[f],\ \alpha.SWS \mapsto sws,\ \alpha.FS \mapsto C.\alpha.FS \cup \{(\iota, 256)\}\big]}\ \text{mid(send, } \alpha\text{, } \{\iota\})$$

Where $f(\iota) = -256$, $f(x) = 0$ for all $x \neq \iota$.

$$\frac{C.\alpha.SWS = \{(\iota, x)\} \uplus sws \qquad x \neq 1 \qquad O(\iota) \neq \alpha}{C \rightsquigarrow_{mid(send,\alpha,\{\iota\})} C[\alpha.SWS \mapsto sws,\ \alpha.FS \mapsto C.\alpha.FS \cup \{(\iota, x-1)\}]}\ \text{mid(send, } \alpha\text{, } \{\iota\})$$

$$\frac{\alpha.SWS = \{(\iota, x)\} \uplus sws \qquad O(\iota) = \alpha}{C \rightsquigarrow_{mid(send,\alpha,\{\iota\})} C[\alpha.SWS \mapsto sws',\ \alpha.FS \mapsto C.\alpha.FS \cup \{(\iota, x+1)\}]}\ \text{mid(send, } \alpha\text{, } \{\iota\})$$

$$\frac{\begin{array}{c}\forall (\iota, x) \in C.\alpha.FS.\ rc'(\iota) = x \qquad \forall \iota \notin X.\ rc'(\iota) = C.\alpha.rc(\iota) \\ C.\alpha.SWS = \emptyset \qquad X = \{\iota | (\iota, \_) \in C.\alpha.FS\}\end{array}}{C \rightsquigarrow_{end(send,\alpha,X)} C\ \ [\alpha.FS \mapsto \emptyset, \alpha'.msgs \mapsto C.\alpha'.msgs{+}{+}[\mathbb{1}_X],\ \alpha.rc \mapsto rc']}\ \text{end(send, } \alpha\text{, X)}$$

Where $\mathbb{1}_X$ is the characteristic function on X

8.4 Event Rules, Receive

$$\frac{\begin{array}{c}\forall \iota \in X.\ rc'(\iota) = \bot \qquad \forall \iota \notin X.\ rc'(\iota) = C.\alpha.rc[\iota] \\ C.\alpha.SWS = \emptyset \qquad C.\alpha.RWS = \emptyset \qquad C.\alpha.FS = \emptyset \qquad C.\alpha.msgs = Z : zs \qquad X = \{\iota | Z(\iota) \neq 0\}\end{array}}{C \rightsquigarrow_{begin(recv,\alpha,X)} C\ \ [\alpha.rc(\iota) \mapsto rc',\ \alpha.msgs \mapsto zs,\ \alpha.RWS \mapsto \{(\iota, Z(\iota), C.\alpha.rc(\iota)) | \iota \in X\}]}\ \text{begin(recv, } \alpha\text{, X)}$$

$$\frac{C.\alpha.RWS = \{(\iota, x, rc_0\} \uplus rws \qquad O(\iota) \neq \alpha}{C \rightsquigarrow_{mid(recv,\alpha,\{\iota\})} C[\alpha.RWS \mapsto rws,\ \alpha.FS \mapsto C.\alpha.FS \cup \{(\iota, rc_0 + x)\}]}\ \text{mid(recv, } \alpha\text{, } \{\iota\})$$

$$\frac{C.\alpha.RWS = \{(\iota, x, rc_0\} \uplus rws \qquad O(\iota) = \alpha}{C \rightsquigarrow_{mid(recv,\alpha,\{\iota\})} C[\alpha.RWS \mapsto rws,\ \alpha.FS \mapsto C.\alpha.FS \cup \{(\iota, rc_0 - x)\}]}\ \text{mid(recv, } \alpha\text{, } \{\iota\})$$

$$\frac{\begin{array}{c}\forall (\iota, x) \in C.\alpha.FS.\ rc'(\iota) = x \qquad \forall \iota \notin X.\ rc'(\iota) = C.\alpha.rc[\iota] \\ C.\alpha.RWS = \emptyset \qquad X = \{\iota | (\iota, \_) \in C.\alpha.FS\}\end{array}}{C \rightsquigarrow_{end(recv,\alpha,X)} C[\alpha.rc(\iota) \mapsto rc', \alpha.FS \mapsto \emptyset]}\ \text{end(recv, } \alpha\text{, X)}$$

8.5 Event Rules, Drop

$$\forall \iota \in X.\ rc'(\iota) = \bot \qquad \forall \iota \notin X.\ rc'(\iota) = C.\alpha.rc[\iota]$$

$$\frac{C.\alpha.SWS = \emptyset \qquad C.\alpha.RWS = \emptyset \qquad C.\alpha.FS = \emptyset}{C \rightsquigarrow_{begin(drop,\alpha,X)} C[\alpha.rc(\iota) \mapsto rc',\ \alpha.SWS \mapsto \{(\iota, \alpha.rc(\iota)) | \iota \in X\}]}\ \text{begin(drop, } \alpha\text{, X)}$$

$$\frac{C.\alpha.SWS = \{(\iota, x)\} \uplus sws \qquad O(\iota) \neq \alpha}{C \rightsquigarrow_{mid(drop,\alpha,\{\iota\})} C[\alpha.SWS \mapsto sws,\ \alpha.FS \mapsto C.\alpha.FS \cup \{(\iota, 0)\},\ O(\iota).msgs \mapsto C.O(\iota).msgs{+}{+}[f]]}\ \text{mid(drop, } \alpha\text{, } \{\iota\})$$

Where $f(\iota) = x$, $f(x) = 0$ for all $x \neq \iota$.

$$\frac{\begin{array}{l}\forall (\iota, x) \in C.\alpha.FS.\ rc'(\iota) = x \qquad \forall \iota \notin X.\ rc'(\iota) = C.\alpha.rc[\iota] \\ C.\alpha.SWS = \emptyset \qquad X = \{\iota | (\iota, \_) \in C.\alpha.FS\}\end{array}}{C \rightsquigarrow_{end(drop,\alpha,X)} C[\alpha.rc(\iota) \mapsto rc',\ \alpha.FS \mapsto \emptyset]}\ \text{end(drop, } \alpha\text{, X)}$$

8.6 Closures

A configuration may have several actors that are currently performing a story. We want to be able to reason about a hypothetical configuration where all these ongoing stories have been concluded. We call such a hypothetical configuration a *closure* of the configuration. Intuitively the closure of a configuration is the set of all possible endings, including all possible interleavings and nondeterministic possibilities.

Definition 8.8 (Follows Order)**.** On the set of configurations, $\mathcal{C}$, we define the partial order, $C \sqsubseteq^X_{Follows} C'$ as the least transitive closure such that the following holds:

$$C \sqsubseteq^X_{Follows} C$$
$$C \rightsquigarrow_{mid(p,\alpha,Y)} C' \wedge X \cap Y \neq \emptyset \longrightarrow C \sqsubseteq^X_{Follows} C'$$
$$C \rightsquigarrow_{end(p,\alpha,Y)} C' \wedge X \cap Y \neq \emptyset \longrightarrow C \sqsubseteq^X_{Follows} C'$$

Recall. *A chain $(S, \leq)$ is maximal if there is no other chain $(T, \leq)$ such that S is a proper subset of T.*

Lemma 8.9. *Let $S, \sqsubseteq^X_{Follows}$ be a maximal chain such that $C \in S$. If $C \vDash WFC$ then S has a maximal element.*

If we take a configuration C such that $C \nvDash WFC$ then it is possible for it to have an sequence without start events and thus an infinite chain, consider C such that all reference counts are $\bot$ and recall that there are an infinite number of actors in C.

Definition 8.10 (Closure)**.**

$$closure(C|X) = \left\{ max(Z) | Z \text{ is a maximal chain in } (\mathcal{C}, \sqsubseteq^X_{Follows}) \text{ such that } C \in Z \right\}$$

Lemma 8.11.

$$C \vDash WFC \wedge \lceil C \rceil \in closure(C|X) \longrightarrow closed(C|X)$$

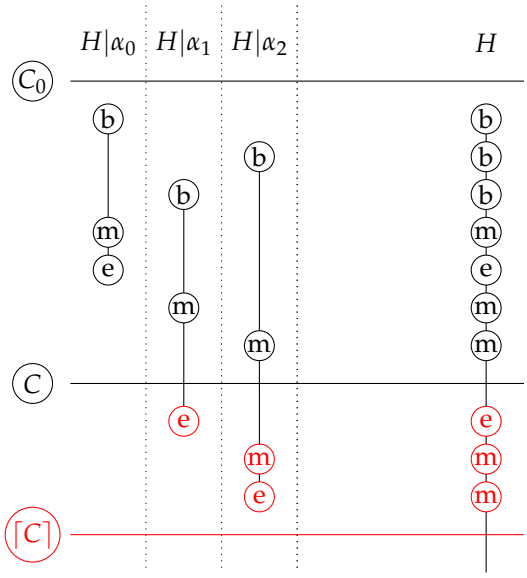


Figure 23: An example of a configuration C from evaluating C_0 under some history, and an element $\lceil C \rceil$ in the closure of C.

Proof. We know there exists some C_0 such that $C_0 \rightsquigarrow_H C \rightsquigarrow_{H'} \lceil C \rceil$ where H' only contains mid and end events. Suppose there is some $\iota \in X$ such that there exists an α with $\lceil C \rceil.\alpha.rc(\iota) = \bot$. Now we know that $Known(C_0)$ so there is some begin event $begin(p, \alpha, Y) \in H$ with $\iota \in Y$. So now there is at least an end event $end(p, \alpha, Y)$ that could have been performed in the history H' but was not which is a contradiction that $\lceil C \rceil$ is the max value of a maximum chain of executions. □

8.7 Invariants

We define predicates for the local and foreign reference counts in *CALF* , these look similar to those in full ORCA models but are only defined when all of their constituent reference counts are non bottom. We also define the *AMC* but this includes both what we would consider application messages in ORCA, between foreign actors say, but also orca messages for increments to the owner. This is because in this model there is no distinction between the two kinds of message.

Definition 8.12 (Reference Counters)**.**

$$\begin{aligned} LRC_C(\iota) &\triangleq O(\iota).rc(\iota) \\ FRC_C(\iota) &\triangleq \sum_{\alpha \neq O(\iota)} \alpha.rc(\iota) \\ AMC_C(\iota) &\triangleq \sum_{\alpha \in C} \sum_{Z \in \alpha.msgs} Z(\iota) \end{aligned}$$

We can then define what the reference counting equation should be in *CALF* , we will denote this $\mathbf{I}_1$.

Definition 8.13 ($\mathbf{I}_1$, Reference Counting Equation for *CALF*)**.**

$$\mathbf{I}_1(C, \iota) \triangleq LRC_C(\iota) = FRC_C(\iota) + AMC_C(\iota)$$

Note that being closed with respect to ι is a necessary condition for $\mathbf{I}_1$ to hold for that object ι because the counters LRC and FRC are only defined when all the constituent reference counts are non bottom.

We now define some straightforward properties of *CALF* configurations.

Definition 8.14 (Known)**.** $Known(C) \triangleq \forall \alpha, \iota. C.\alpha.rc(\iota) \neq \bot$

Definition 8.15 (Well Formed Messages)**.** $MessagesWellFormed(C) \triangleq \forall \alpha, \iota, n.\ ((C.\alpha.msgs[n])(\iota) < 0 \longrightarrow O(\iota) = \alpha)$

Definition 8.16 (Valid Foreign Reference Counts)**.** $ForeignValid(C) \triangleq \forall \alpha, \iota.\ (O(\iota) \neq \alpha \longrightarrow C.\alpha.rc(\iota) \geq 0 \vee C.\alpha.rc(\iota) = \bot)$

Definition 8.17 (Well Formed Configuration)**.** We say that C is a *well formed configuration* if it can be expressed as the result of executing a valid history from an initial condition C_0 satisfying various properties.

$$C \vDash WFC \iff \exists C_0.\ (\mathbf{I}_1(C_0) \wedge Known(C_0) \wedge ForeignValid(C_0) \wedge MessagesWellFormed(C_0) \wedge C_0 \rightsquigarrow_H C \wedge Valid(H))$$

It is clear that if a well formed configuration C executes to another configuration C' under a history H, $C \rightsquigarrow_H C'$. Then because C is well formed there is some C_0 such that $C_0 \rightsquigarrow_{H_0} C$ through the valid history H_0. So if the concatenation $H_0{+}{+}H$ is a valid history, then $C' \vDash WFC$.

Proposition 8.18. $C \vDash WFC \longrightarrow ForeignValid(C)$

Proof. Suppose for some α and ι such that ι is not owned by α, $C.\alpha.rc(\iota) < 0$. Then from the well formed condition there is some C_0 such that $C_0 \rightsquigarrow_H C$ for a valid history H and $ForeignValid(C_0)$ and $Known(C_0)$ so $C_0.\alpha.rc(\iota) \geq 0$. We know that $closed(C|\alpha, \iota)$, so there is a sequence of stories $H|\alpha, \iota$ that takes the reference count from positive to negative.

This cannot be a receive as by definition all message queue entries are non-negative, and we are not the owner so by the execution rules this can only increase the reference count. And this cannot be a send as we have a precondition guarding this. Contradiction. □

Lemma 8.19. $C \vDash WFC \longrightarrow MessagesWellFormed(C)$

Proof. We see that the only rule that allows us to append a negative value for an object onto a message queue is in $mid(send, \alpha, \{\iota\})$ and this sends the message to the owner of ι. □

Lemma 8.20. *If $C \rightsquigarrow_x C'$ and $x = e(p, \alpha, X)$. For all $\iota \notin X$, then all of the following hold:*

- $C.\alpha.rc(\iota) = C.\alpha.rc(\iota)$
- $LRC_C(\iota) = LRC_{C'}(\iota)$,
- $FRC_C(\iota) = FRC_{C'}(\iota)$
- $AMC_C(\iota) = AMC_{C'}(\iota)$

Proof. It is clear from inspecting the execution rules that they only mutate the reference counts of objects in the set X, which gives the first three results. The final is clear when you notice that all the alterations of message queues require all non-zero objects in the message to be elements of X. □

Proposition 8.21 (Completed Stories Preserve $\mathbf{I}_1$)**.**

$$C \vDash WFC \wedge \boldsymbol{I}_1(C, \iota) \longrightarrow \forall C'.\ (C \rightsquigarrow_S C' \longrightarrow \boldsymbol{I}_1(C', \iota))$$

Where S is a story $S = [begin(r, \alpha, X), mid(r, \alpha, \iota_1) \ldots mid(r, \alpha, \iota_n), end(r, \alpha, X)]$ *for some r.*

Proof. Either $\iota \notin X$ in which case ι is not in any effected message queues or actor reference counts, so the sums are unaffected.

So $C \rightsquigarrow_b C_b \rightsquigarrow_m \cdots \rightsquigarrow_m C_m \rightsquigarrow_m \cdots \rightsquigarrow_e C'$. (We have r, α and X fixed so we abbreviate here $b = begin(r, \alpha, X)$, $e = end(r, \alpha, X)$ and $m = mid(r, \alpha, \iota)$ and we will do this elsewhere.) We know from the definition of a story that for ι there is exactly one mid event that will effect it and we assume C_m is the event after that execution.

Or $\iota \in X$, first assume that the story is for **receive**.
We know $C.\alpha.msgs = Z : zs$ and that $(\iota, z_\iota) \in Z$. And let $rc_0 = C.\alpha.rc(\iota)$
Now $C_b.\alpha.rc(\iota) = \bot$ and $(\iota, Z(\iota), rc_0) \in C_b.\alpha.RWS$
If $O(\iota) = \alpha$ then $(\iota, rc_0 - z_\iota) \in C_m.FS$. Which means that $C'.\alpha.rc(\iota) = rc_0 - z_\iota$ and $C'.msgs = zs$. So $FRC_{C'} = FRC_C$ as we have not changed the values for any other α, but $LRC_{C'} = LRC_C - z_\iota$ and $AMC_{C'} = AMC_C - Z(\iota)$ as we have not modified any other actor's message queues and removed the top message from α. Altogether, this means that $\mathbf{I}_1$ is preserved into C'. If $O(\iota) \neq \alpha$ we have a similar argument where the values are summed and the FRC is updated.

Now we assume the story is for **send**.
Let $rc_0 = C.\alpha.rc(\iota)$ again, now $C_b.\alpha.rc(\iota) = \bot$ and $(\iota, rc_0) \in C_b.\alpha.SWS$. We split into ownership cases

- If $O(\iota) = \alpha$
  Then $(\iota, rc_0 + 1) \in C_m.FS$ so $C'.\alpha.rc(\iota) = rc_0 + 1$. So $LRC_{C'} = LRC_C + 1$ but we also have some other actor α' with an additional message in its message queue containing $(\iota, 1)$ with all other message queues preserved, so $AMC_{C'} = AMC_C + 1$. We also have that FRC is preserved as the only alterations to the reference counts are for α.
- If $O(\iota) \neq \alpha$ and $rc_0 > 1$
  Then $(\iota, rc_0 - 1) \in C_m.FS$ so $C'.\alpha.rc(\iota) = rc_0 - 1$. So $FRC_{C'} = FRC_C - 1$ similarly we have α' with an additional message in its message queue containing $(\iota, 1)$ so $AMC_{C'} = AMC_C + 1$.
- If $O(\iota) = \alpha$ and $rc_0 = 1$
  This is the most complex case where we send two messages, one to the owner and one to α'. $(\iota, 256) \in C_m.FS$ so $C'.\alpha.rc(\iota) = 256$ and $FRC_{C'} = FRC_C + 255$. But the messages we send set $AMC_{C'} = AMC_{C_m} + 1 = AMC_C - 255$.

We argue for **drop** in a similar way for **send** as it is similar but simpler.

□

Theorem 8.22.

$$C \vDash WFC \wedge closed(C|\iota) \Longrightarrow \boldsymbol{I}_1(\lceil C \rceil, \iota)$$

Proof. Fix ι arbitrarily, as C is well formed there is some configuration C_0 such that $C_0 \rightsquigarrow_H C$ via some valid history H.

As H is valid we know that for all actors α the restriction $H|\alpha, \{\iota\}$ is a sequence of stories. We will prove the result by induction on the total number of stories from all such restrictions.

Base Case, no stories: as C is closed for $\{\iota\}$ we know that $H|\{\iota\}$ has to be empty so holds trivially.

Inductive case, assume this holds for some n, to show for $n+1$. Now let b be the begin event in the history $b = begin(p, \alpha, Y)$. Then we have the execution $C_0 \rightsquigarrow_{x_1} \cdots \rightsquigarrow_{x_k} C_b \rightsquigarrow_b \rightsquigarrow \cdots \rightsquigarrow C$ defining $H_0 = \{x_1, \ldots x_k\}$. Now we take $\lceil C_b \rceil \in closure(C_b, \{\iota\})$. We know that there are n stories from the restricted histories $H_0|\alpha, \{\iota\}$ so by inductive assumption $\mathbf{I}_1(\lceil C_b \rceil, \iota)$. Now define $\lceil C \rceil$ such that $\lceil C_b \rceil \rightsquigarrow_S \lceil C \rceil$, we know that $\mathbf{I}_1(\lceil C \rceil, \iota)$ holds from the previous proposition.

Finally we argue $\mathbf{I}_1(\lceil C \rceil, \iota) \iff \mathbf{I}_1(C, \iota)$ which is true because C is equal to $\lceil C \rceil$ up to permutations of message queues which do not effect AMC.

□

We now define the queue effect in *CALF* and the derived $\mathbf{I}_2$ to show properties of the owner of an object's reference counts.

Definition 8.23 (Queue Effect)**.**

$$QueueEffect_C(\iota, n) \triangleq \begin{cases} \bot & \text{where } C.O(\iota).rc(\iota) = \bot \\ C.O(\iota).rc(\iota) - \sum_{i=0}^{n}(C.O(\iota).msgs[i])(\iota) & \text{otherwise} \end{cases}$$

Definition 8.24 ($\mathbf{I}_2$)**.**

$$\mathbf{I}_2(C, \iota) \triangleq \forall n \geq 0.\ QueueEffect_C(\iota, n) \geq 0$$

This invariant $\mathbf{I}_2$ is a generalisation of the idea that all reference counts are non-negative, if we take $n = 0$ then $QueueEffect_C(\iota, 0) > 0$ expands to a condition only on the reference count of the owner.

Similarly we define an invariant $\mathbf{I}_3$ stating that if in a configuration the owner of an object receives some number of messages to reach a reference count of zero for the object, then that reference count stays at zero throughout further receivals.

Definition 8.25 ($\mathbf{I}_3$)**.**

$$\mathbf{I}_3(C, \iota) \triangleq \forall n \geq m \geq 0.\ QueueEffect_C(\iota, m) = 0 \longrightarrow QueueEffect_C(\iota, n) = 0$$

8.8 Equivalence of Configurations

We now want to show that if we have a configuration then the elements in its closure have similar properties that we can use to reason about them.

Definition 8.26 (Equivalence of X-Closed Configurations)**.** The *equivalence of X-closed configurations* is the relation $\backsim_X$ for a set X

$$C \backsim_X C' \iff \exists C_0.\ C, C' \in closure(C_0|X)$$

Lemma 8.27. *For all sets of objects X, $\backsim_X$ is an equivalence relation on $X-$closed configurations.*

Proof. We show this obeys the equivalence relation axioms:

- Reflexivity: if C has $closed(C|X)$ then $C \in closure(C|X)$ so $C \backsim_X C$
- Symmetry: this is clear from the definition.
- Transitivity: if $C \backsim_X C'$ and $C' \backsim_X C''$ then there is some C_0 and C_1 such that $C' \in closure(C_0|X)$ and $C' \in closure(C_1|X)$ it must be the either $C_0 \rightsquigarrow^* C_1$ or $C_1 \rightsquigarrow^* C_0$ as they both evaluate to C' with only mids and ends. Therefore wlog we assume $C_0 \rightsquigarrow^* C_1$ therefore $C \in closure(C_0|X)$ and $C'' \in closure(C_0|X)$ so $C \backsim_X C''$.

□

Proposition 8.28.

$$C \backsim_X C' \longrightarrow \left(\forall \iota \in X. \begin{pmatrix} \boldsymbol{I}_{1C}(\iota) & \iff & \boldsymbol{I}_{1C'}(\iota) \\ \wedge\, \boldsymbol{I}_{2C}(\iota) & \iff & \boldsymbol{I}_{2C'}(\iota) \\ \wedge\, \boldsymbol{I}_{3C}(\iota) & \iff & \boldsymbol{I}_{3C'}(\iota) \end{pmatrix} \right)$$

Proof. We argue that C is the same as C' up to some permutation of message queues. All events that act on the sets of an actor are protocols of that actor, for instance sending does not directly alter the working set of another actor. The closures elements are formed by performing the same series of events but in different orders, so the only differences can be the message queue orderings. $\mathbf{I}_1$ and $\mathbf{I}_2$ follow directly from this.

For $\mathbf{I}_3$ we assume wlog $\mathbf{I}_3(C)$ and assume for contradiction $\neg\mathbf{I}_3(C'$ so there is some $\alpha, \iota, n < m$ such that the queue effect for ι at n is zero, but the queue effect at m is non-zero. We know that $\mathbf{I}_2$ holds so the queue effect must be positive, so must have been caused by an increment message. But the queue effect was at one point zero, we thus argue that causality must have been broken for this to happen. The increment message must have been processed before the messages that it allowed, which is of course, a contradiction. □

Proposition 8.29.

$$\boldsymbol{I}_1(C, \iota) \longrightarrow LRC_C(\iota) - AMC_C(\iota) \leq QueueEffect_C(\iota, n)$$

Where n is the length of the owner of ι's message queue.

Proof. We split the AMC into its constituent parts, the count from messages sent to the owner, and the count from messages sent elsewhere. We say that

$$AMC_C(\iota) = AMC_C^F(\iota) + AMC_C^L(\iota)$$

Where $AMC_C^F = \sum_{\alpha \neq O(\iota)} \ \sum_{Z \in \alpha.msgs} Z(\iota)$ and $AMC_C^L = \sum_{Z \in O(\iota).msgs} Z(\iota)$.

We know that $AMC_C^F(\iota) \geq 0$ as all messages to foreign actors are greater than zero, so $AMC_C^L(\iota) \leq AMC_C(\iota)$
Then $LRC_C(\iota) - AMC_C(\iota) \leq LRC_C(\iota) - AMC_C^L(\iota) = QueueEffect_C(\iota, n)$ by definition of queue effect, where n is the length of the message queue. □

Lemma 8.30 (Complete Stories Preserve Invariants)**.** *Suppose S is a story such that $S = [begin(p, \alpha, X) \ldots end(p, \alpha, X)]$ then*

$$\forall \iota \in X. \left(C \vDash WFC \wedge C \rightsquigarrow_S C' \wedge \begin{pmatrix} \boldsymbol{I}_1(C, \iota) \wedge \\ \boldsymbol{I}_2(C, \iota) \wedge \\ \boldsymbol{I}_3(C, \iota) \end{pmatrix} \longrightarrow \begin{pmatrix} \boldsymbol{I}_1(C', \iota) \wedge \\ \boldsymbol{I}_2(C', \iota) \wedge \\ \boldsymbol{I}_3(C', \iota) \end{pmatrix} \right)$$

Proof. Fix ι. We know from Proposition 8.21 that $\mathbf{I}_1(C', \iota)$ holds.

From $\mathbf{I}_2(C, \iota)$ we know that the queue effect is non-negative up to the length of the message queue n. We know for any foreign α

$$C.\alpha.rc(\iota) \leq FRC_C(\iota) = LRC_C(\iota) - AMC_C(\iota) \leq QueueEffect_C(\iota, n)$$

as only some of the messages are directed to the owner. Consider stories that impact the QueueEffect, in particular *drop* and *send* but these cannot produce an effect that exceeds α's own reference

count, so from the above inequality $\mathbf{I}_2(C', \iota)$ holds.
Similarly for $\mathbf{I}_3$

$$\begin{aligned} & 0 \leq FRC_C(\iota) \\ & = LRC_C(\iota) - AMC_C(\iota) \leq QueueEffect_C(\iota, n) = 0 \\ & \longrightarrow FRC_C(\iota) = 0 \end{aligned}$$

So there can be no *send* or *drop* events. □

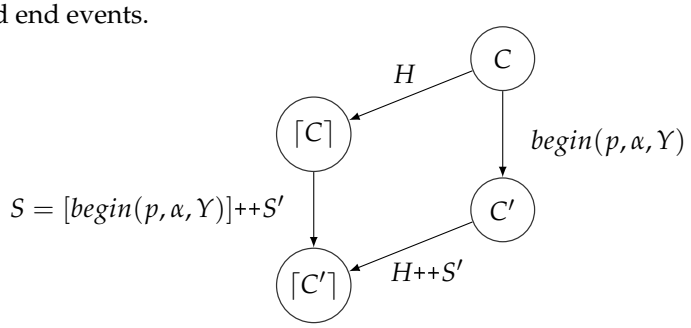
Proposition 8.31.

$$C \rightsquigarrow C' \wedge \forall \lceil C \rceil \in closure(C|X), \forall \iota \in X.\ (\boldsymbol{I}_1(\lceil C \rceil, \iota) \wedge \boldsymbol{I}_2(\lceil C \rceil, \iota) \wedge \boldsymbol{I}_3(\lceil C \rceil, \iota)) \\ \longrightarrow \forall \lceil C' \rceil \in closure(C'|X), \forall \iota \in X.\ (\boldsymbol{I}_1(\lceil C' \rceil, \iota) \wedge \boldsymbol{I}_2(\lceil C' \rceil, \iota) \wedge \boldsymbol{I}_3(\lceil C' \rceil, \iota))$$

Proof. This is clear when $C \rightsquigarrow_x C'$ and x is a mid or end event so assume $x = begin(p, \alpha, Y)$.

Take $\lceil C \rceil \in closure(C|X)$ so $C \rightsquigarrow_H \lceil C \rceil$ where $H = [h_1, \ldots, h_n]$ all mid and end events.

We argue that the following diagram holds, where S is a story formed from $begin(p, \alpha, Y)$ and some S' of mid and end events.



We claim that this holds because the effects of begin events are to set the reference counts of Y to $\perp$ and, in the case of receive, remove the head message. So doing this first or last should have no effect on the final configuration.

We know that all invariants are preserved into $\lceil C' \rceil$, now, for all $\lceil C'' \rceil \in closure(H, X)$ we clearly have $\lceil C' \rceil \backsim_X \lceil C'' \rceil$ so they hold for every element in the closure of C' with respect to X. □

Definition 8.32 (Complete Configuration)**.** $C \vDash \Diamond \iff C \vDash WFC \wedge \forall X.\ (\forall \lceil C \rceil \in closure(C, X) \longrightarrow \mathbf{I}_1(\lceil C \rceil) \wedge \mathbf{I}_2(\lceil C \rceil) \wedge \mathbf{I}_3(\lceil C \rceil))$

Corollary 8.33. *Completeness of a configuration is preserved under execution.*

Theorem 8.34. $C \vDash \Diamond \wedge LRC_C(\iota) = 0 \longrightarrow FRC_C(\iota) = 0$

Proof. Suppose not, then there is some $\alpha \neq O(\iota)$ such that $C.\alpha.rc(\iota) = \perp$ or $C.\alpha.rc(\iota) > 0$. Now take $\lceil C \rceil \in closure(C, \{\iota\})$ then we know that $LRC_{\lceil C \rceil}(\iota) = 0$. We now perform case analysis of the possible end events that could have occurred for α between C and $\lceil C \rceil$. As we have $C \vDash \Diamond$ we know all invariants holds for $\lceil C \rceil$

- Send - we know $FRC_{\lceil C \rceil}(\iota) \geq 0$ so $AMC_{\lceil C \rceil}(\iota) \leq 0$ as $FRC_{\lceil C \rceil}(\iota) + AMC_{\lceil C \rceil}(\iota) = LRC_{\lceil C \rceil}(\iota) = 0$. We also know that there is a message on some message queue, for this to sum to a negative value there must be a negative message on the owner's message queue, but from $\mathbf{I}_3$ we know there is none.
- Receive - Receiving for a foreign actor increments the reference count for objects. We know that $AMC_{\lceil C \rceil}(\iota) \geq 0$ from $\mathbf{I}_3$ (no messages on the owner's queue), and $FRC_{\lceil C \rceil}(\iota) \geq 0$. Either we get a contradiction as $\mathbf{I}_1$ does not hold, or receiving the message set the reference count to zero. In this case we know because $\lceil C \rceil$ is well-formed there is some C_0 such that $C_0 \rightsquigarrow^* \lceil C \rceil$ with all intermediate C_i well formed. But there must be one with a negative foreign reference count which is a contradiction.
- Drop - drop sends a message to the owner which directly contradicts $\mathbf{I}_3$.

So all possibilities end in a contradiction □

8.9 Evaluation

One of the challenges of designing the submodel is that when ORCA sends an application message, it may send many orca messages first.A This means that *CALF* had to be able to mimic this behaviour and led to the design of the story as it is presented. If we represented the three protocols as atomic blocks or just as beginning and end events we would not be able to capture all the possible behaviour that could occur. Linearisability of histories involves looking at the dependence of events on each other and trying to construct an equivalent history that is sequential. But with a design that allowed blocks to perform "side effects" such as send these messages reordering is no longer necessarily possible.

Initially in development we had invariants that were defined for a fixed initial condition and reasoned about what histories from that condition would preserve them. We started with a definition of a closed history instead of a closed configuration, giving rise to propositions like the following:

$$\mathbf{I}_C([]) \wedge \mathit{closed}(H) \longrightarrow \mathbf{I}_C(H)$$

Where $\mathbf{I}_C(H)$ is some statement about the configuration formed starting at C and applying the history H. $[]$ is the empty history.

This is closer to *CALF* 's origins in linearisability theory, considering histories as the main object of study rather than the configuration they form. However, when we had to relate the configurations to those of a larger model it became clear that this approach would be difficult.

We then re-oriented the ideas from studying the histories back towards configurations. The definition we gave for a closed configuration is equivalent to applying a closed history to an initially closed configuration. This can be seen in the definition of a well formed configuration used in *CALF* . From the histories perspective the closure of a history is more intuitive also, it is all the maximal concatenations of mid and end events.

Some of the reasoning about histories took the form building a Kripke Frame and viewing the histories through the lens of modal logic. Given an initial state each world would be a history and the relation between them would be a concatenation of an additional event. We will revisit this in Section 10.

We allow the reference counts to take on the bottom value $\perp$ to represent a lock of knowledge about their state. We split up the protocols send, receive and drop into "blocks" but during the execution of these we do not know exactly when the reference counts are updated. We can think of this slicing into blocks as the slicings of when "atomic" or synchronised values are updated, in this case message queues and some ghost values. Then during the execution of stories all the information about eventually mutated non-atomic values is lost. We think this works well as an abstraction.

9 Submodel Relation

9.1 $ORCA^1$

In order to map into *CALF* we must first make some minor adjustments to $ORCA^0$. We denote this modified model $ORCA^1$.

We have altered the ordering of some of the lines of pseudocode which will be presented below. Similar to $ORCA^{\text{Ghost}}$ we need to differentiate whether an actor is receiving an orca message or an application message so we add the actor state `RECEIVE_ORCA`.

9.2 Mapping Between $ORCA^0$ and *CALF*

We show that we can form a bisimulation between a *CALF* configuration and an $ORCA^1$ configuration. To avoid confusion we will use $\gamma \in \Gamma$ to refer to $ORCA^1$ configurations. Actors in Γ configurations will be labelled as β and addresses as σ.

We will need an injective mapping A to take actors in an $ORCA^1$ configuration to their corresponding *CALF* actor.

Definition 9.1.

$$A : ActorAddr \to \mathbb{N}$$

Mapping objects is more complex, in $ORCA^1$ we are able to create an arbitrary number of objects for any actor, but we do not have any methods for creating "new" objects in *CALF*.

We know from set theory that there are bijections between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$. Using one of these for the ownership function in a *CALF* configuration we are able to have an infinite number of objects for each actor.

We will use bijections θ between $ORCA^1$ addresses and *CALF* objects. Later we will require these to be correct up to the allocated objects in an $ORCA^1$ configuration. Then when the $ORCA^1$ configuration nondeterministically allocates an object we will choose a new θ' the agrees with θ for all pre-existing objects and assigns the new object the correct owner.

Definition 9.2.

$$\theta : ObjectAddr \to \mathbb{N}$$

9.3 Code - Event Mappings

We define a series of mappings for three different pseudocode function M^{Send}, M^{Recv}, M^{GC} that map the program counter to an event or a no-op.

```
1 α.st = IDLE && top(α.qu) = orcaa(σ : z)
2 Receiving<α>:
3 {
4   α.st := RECEIVE_ORCA
5   pop(α.qu)          ⟶ begin(Receive, A(α), {θ(σ)|σ ∈ trace_frame(α, φ))}
6   α.rc(σ) += z       ⟶ mid(Receive, A(α), θ(σ))
7   pass               ⟶ end(Receive, A(α), {θ(σ)|σ ∈ trace_frame(α, φ))}
8   α.st := EXECUTE
9 }
```

Figure 24: $M^{RecvOrca}$ - Mapping from $ORCA^1$ receive pseudocode to *CALF* events.

We define the *Fine Grained Correspondence* of the Receive Orca method between an $ORCA^1$ state γ and a *CALF* state C given an actor mapping A and an object bijection θ.

Note we use the notation X^c to be the compliment of the object set X that is, all objects not in X.

Definition 9.3 (Fine Grained Correspondence for Receive Orca)**.**

$$
\begin{aligned}
(FGC^{RecvOrca})^A_\theta(\beta,\gamma,C) \triangleq\ & \beta.st_\gamma = RECEIVE\_ORCA \rightarrow (\\
& (pc < 5 \lor pc > 7 \longrightarrow closed(C|A(\beta)))\\
& \land(5 \geq pc < 7 \longrightarrow closed(C|A(\beta), \{\theta(\sigma)\}^c))\\
& \land(pc = 6 \longrightarrow (\theta(\sigma), A(\beta).rc_\gamma(\sigma)) \in C.A(\beta).FS)\\
&)
\end{aligned}
$$

Where $pc = \beta.pc_\gamma$

```
1  β.st = IDLE && top(β.qu) = app(φ)
2  Receiving<β>:
3  {
4    β.st := RECEIVE
5    ws := trace_frame(β, φ)
6
7    pop(β.qu)                          ⟶ begin(Receive, A(α), {θ(σ)|σ ∈ trace_frame(α, φ))}
8
9    for σ ∈ ws:
10     if (O(σ) = α) ⟶ α.rc(σ) -= 1     ⟶ mid(Receive, A(α), θ(σ))
11     if (O(σ) ≠ α) ⟶ α.rc(σ) += 1     ⟶ mid(Receive, A(α), θ(σ))
12     ws := ws \ {σ}
13
14   β.frame := φ                       ⟶ end(Receive, A(α), {θ(σ)|σ ∈ trace_frame(α, φ))}
15
16   β.st := EXECUTE
17
18 }
```

Figure 25: M^{Recv} - Mapping from $ORCA^1$ receive pseudocode to *CALF* events.

Definition 9.4 (Fine Grained Correspondence for Receive)**.**

$$
\begin{aligned}
(FGC^{Recv})^A_\theta(\beta,\gamma,C) \triangleq\ & \beta.st_\gamma = RECEIVE \rightarrow (\\
& pc < 7 \lor pc > 16 && \longrightarrow closed(C|A(\beta))\\
& pc = 9 && \longrightarrow (\theta(\sigma_9), 1, \beta.rc_\gamma(\sigma_9)) \in C.A(\beta).RWS\\
& pc = 10 \land \mathcal{O}(\sigma) \neq \beta && \longrightarrow (\theta(\sigma_9), 1, \beta.rc_\gamma(\sigma_9)) \in C.A(\beta).RWS\\
& (pc = 10 \land \mathcal{O}(\sigma) = \beta) \lor pc = 11 && \longrightarrow (\theta(\sigma_9), \beta.rc_\gamma(\sigma_9)) \in C.A(\beta).FS\\
& 7 \geq pc < 14 && \longrightarrow \forall \sigma \neq \sigma_9.\ (\\
& && \quad (\sigma \in ws \longrightarrow (\theta(\sigma_9), 1, \beta.rc_\gamma(\sigma_9)) \in C.A(\beta).RWS)\\
& && \quad (\sigma \notin trace\_frame(\beta,\phi) \longrightarrow closed(C, A(\beta), \{\sigma\}))\\
& && \quad (\sigma \in trace\_frame(\beta,\phi) \setminus ws \longrightarrow\\
& && \qquad (\theta(\sigma), \beta.rc_\gamma(\sigma)) \in C.A(\beta).FS)\\
& && \quad)\\
&)
\end{aligned}
$$

Where $pc = \beta.pc_\gamma$

```
1  β.st = EXECUTE && β.frame = (b, φ · φ′)
2  Sending<β>:
3  {
4    β.st := SEND
5
6    ws := trace_frame( β, (b, φ′))   ——————→ begin(Send, A(α), {θ(σ)|σ ∈ trace_frame(α, φ))}
7
8    for σ ∈ WS:
9      if O(σ) = β:
10       β.rc(σ) += 1   ——————————————→ mid(Send, A(α), θ(σ))
11     else if β.rc(σ) > 1:
12       β.rc(σ) −= 1   ——————————————→ mid(Send, A(α), θ(σ))
13     else:
14       O(σ).qu.push(orca(σ:256)) ———→ mid(Send, A(α), θ(σ))
15       β.rc(σ) := 256
16     ws := ws \ {σ}
17
18   β.frame := (b, φ)
19
20   β′.qu.push(app(b′, φ′))   ———————→ end(Send, A(α), {θ(σ)|σ ∈ trace_frame(α, φ))}
21 }
```

Figure 26: M^{Send} - Mapping from $ORCA^1$ send pseudocode to *CALF* events.

Definition 9.5 (Fine Grained Correspondence for Send)**.**

$$
\begin{aligned}
(FGC^{Send})^A_\theta(\beta,\gamma,C) \triangleq\ & \beta.st_\gamma = SEND \rightarrow (\\
& pc < 6 \vee pc > 20 && \longrightarrow closed(C|A(\beta))\\
& pc = 8 \vee pc = 9 \vee pc = 11 && \longrightarrow (\theta(\sigma_8), \beta.rc_\gamma(\sigma_8)) \in C.A(\beta).SWS\\
& pc = 10 \vee pc = 12 \vee pc = 15 \vee pc = 16 && \longrightarrow (\theta(\sigma_8), \beta.rc_\gamma(\sigma_8)) \in C.A(\beta).FS\\
& pc = 14 && \longrightarrow (\theta(\sigma_8), 256) \in C.A(\beta).FS\\
& 6 \geq pc < 20 && \longrightarrow \forall \sigma \neq \sigma_8.\ (\\
& && \quad (\sigma \in ws \longrightarrow (\theta(\sigma_8), \beta.rc_\gamma(\sigma_8)) \in C.A(\beta).SWS)\\
& && \quad (\sigma \notin trace\_frame(\beta,\phi) \longrightarrow closed(C, A(\beta), \{\sigma\}))\\
& && \quad (\sigma \in trace\_frame(\beta,\phi) \setminus ws \longrightarrow\\
& && \qquad (\theta(\sigma), \beta.rc_\gamma(\sigma)) \in C.A(\beta).FS)\\
& && \quad)\\
&)
\end{aligned}
$$

Where $pc = \beta.pc_\gamma$

```
1  β.st = IDLE && β.st = EXECUTE
2  GarbageCollection<β>:
3  {
4    β.st := COLLECT
5    ms := ∅
6
7    // Marking as unreachable
8    for σ such that 𝒪(σ) = β || β.rc > 0:
9      ms := ms[σ → U]
10
11   // Tracing and marking locally accessible as reachable
12   for σ ∈ trace_this(β) ∪ trace_frame(β.frame):
13     ms := ms[σ → R]
14
15   // Marking owned and globally accessible as reachable
16   for σ such that 𝒪(σ) = β && β.rc > 0:
17     ms := ms[σ → R]
18   // Collection
19   pass // Begin loop ──────────────────────────► begin(Drop, A(β), {θ(σ)|ms[σ] = U ∧ 𝒪(σ) ≠ β})
20   for σ such that ms(σ) = U:
21     if 𝒪(σ) = β:
22       𝒞.heap := 𝒞.heap[σ → ⊥]
23       β.rc := β.rc[σ → ⊥]
24     else:
25       tmp := β.rc(σ)
26       β.rc(σ) := 0
27       𝒪(σ).qu.push(orca(σ: −tmp)) ──────────► mid(Drop, A(β), θ(σ))
28
29   pass // End of loop ──────────────────────► end(Drop, A(β), {θ(σ)|ms[σ] = U ∧ 𝒪(σ) ≠ β})
30
31   if β.frame = ∅:
32     β.st := IDLE
33   else:
34     β.st := EXECUTE
35 }
```

Figure 27: M^{GC} - Mapping from $ORCA^1$ send pseudocode to *CALF* events.

Definition 9.6 (Fine Grained Correspondence for Garbage Collection)**.**

$$\text{Let } \Sigma = \{\sigma | ms[\sigma] = U \wedge \mathcal{O}(\sigma) \neq \beta\}$$

$$
\begin{aligned}
(FGC^{GC})^{A}_{\theta}(\beta,\gamma,C) \triangleq\ & \beta.st_\gamma = COLLECT \rightarrow (\\
& pc < 19 \vee pc > 29 && \longrightarrow closed(C|A(\beta))\\
& && \\
& 20 \leq pc < 26 && \longrightarrow (\theta(\sigma_{20}), \beta.rc_\gamma(\sigma_{20})) \in C.A(\beta).SWS\\
& pc = 25 && \longrightarrow tmp = \beta.rc_\gamma(\sigma_{20})\\
& pc = 26 && \longrightarrow (\theta(\sigma_{20}), tmp) \in C.A(\beta).SWS\\
& pc = 27 && \longrightarrow (\theta(\sigma_{20}), 0) \in C.A(\beta).FS\\
& 19 \geq pc < 29 && \longrightarrow \forall \sigma \neq \sigma_{20}.\ (\\
& && \quad (\sigma \notin \Sigma \longrightarrow closed(C, A(\beta), \{\sigma\}))\\
& && \quad (\sigma \in \Sigma \wedge \beta.rc_\gamma(\sigma) \neq 0 \longrightarrow\\
& && \qquad (\theta(\sigma_9), \beta.rc_\gamma(\sigma_9)) \in C.A(\beta).SWS\\
& && \quad (\sigma \in \Sigma \wedge \beta.rc_\gamma(\sigma) = 0 \longrightarrow\\
& && \qquad (\theta(\sigma), 0) \in C.A(\beta).FS)\\
& && \quad)\\
&)
\end{aligned}
$$

Where $pc = \beta.pc_\gamma$

Definition 9.7 (Fine Grained Correspondence for Garbage Collection)**.**

$$
\begin{aligned}
FGC^{A}_{\theta}(\beta,\gamma,C) \triangleq\ & \\
& \beta.st_\gamma = RECEIVE && \longrightarrow (FGC^{Recv})^{A}_{\theta}(\beta,\gamma,C)\\
& \wedge \beta.st_\gamma = RECEIVE\_ORCA && \longrightarrow (FGC^{RecvOrca})^{A}_{\theta}(\beta,\gamma,C)\\
& \wedge \beta.st_\gamma = SEND && \longrightarrow (FGC^{Send})^{A}_{\theta}(\beta,\gamma,C)\\
& \wedge \beta.st_\gamma = COLLECT && \longrightarrow (FGC^{GC})^{A}_{\theta}(\beta,\gamma,C)\\
& \wedge \beta.st_\gamma = IDLE && \longrightarrow closed(C|A(\beta))\\
& \wedge \beta.st_\gamma = EXECUTE && \longrightarrow closed(C|A(\beta))
\end{aligned}
$$

9.4 Mapping Configurations Elements

We start by defining a mapping G^{msgs}_{θ} from an $ORCA^1$ message queue to a $CALF$ message queue using the object mapping θ.

Definition 9.8 (Message mapping)**.**

$$
\begin{aligned}
& G^{msgs}_{\theta}([]) = []\\
& G^{msgs}_{\theta}(app(\emptyset) : xs) = \lambda\iota.0 \ +\!\!+\ G^{msgs}_{\theta}(xs)\\
& G^{msgs}_{\theta}(app((\_, \psi)) : xs) = (\mathbb{1}_{dom(\psi)} \circ \theta^{-1}) \ +\!\!+\ G^{msgs}_{\theta}(xs)\\
& G^{msgs}_{\theta}(orca((\sigma : x)) : xs) = f_{\sigma,x} \ +\!\!+\ G^{msgs}_{\theta}(xs)\\
& \text{where } f_{\sigma,x}(y) = \begin{cases} x & y = \theta(\sigma) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

9.5 Relation

We define a relation between two configurations γ and C via some actor mapping A and object bijection θ. The relation states the following

- Based on A and θ the models agree on ownership.
- Based on the mapping of messages, the models agree on the message queues of all actors.
- There is a fine grained correspondence of all the actor states.
- Whenever the *CALF* model is closed for some actor and object, the corresponding reference counts in both models are equal, based on the mappings of actors and objects.

Definition 9.9 (*CALF* Correspondence Relation)**.**

$$\begin{aligned} R^A_\theta(\gamma, C) \iff & \\ & (\forall \sigma.\ \gamma.heap(\sigma) \neq \bot \longrightarrow O(\theta(\sigma)) = A(\mathcal{O}(\sigma))) \\ & \wedge \forall \beta.\ C.A(\beta).msgs = G^{msgs}_\theta(\beta.msgs_\gamma) \\ & \wedge \forall \beta.\ FGC^\gamma_\theta(\beta, \gamma, C) \\ & \wedge closed(C, X) \longrightarrow \forall \beta, \iota \in X.\ \beta.rc_\gamma(\theta^{-1}(\iota)) = C.A(\beta).rc(\iota) \end{aligned}$$

This correspondence should be preserved under execution of the larger model. So if γ is related to some *CALF* configuration C and $\gamma \rightsquigarrow \gamma'$ then there is some C' such that $C \rightsquigarrow C'$ and γ' is related to C'.

$$\begin{array}{ccc} & R^A_\theta(\gamma, C) & \\ \gamma & \longleftrightarrow & C \\ \Big(\rightsquigarrow_{ORCA^1}\Big) \downarrow & & \downarrow \Big(\rightsquigarrow_{CALF}\Big) \\ \gamma' & \longleftrightarrow & C' \\ & R^A_\theta(\gamma', C') & \end{array}$$

Proposition 9.10 (Preservation of Relation)**.**

$$\gamma \rightsquigarrow \gamma' \wedge R^A_\theta(\gamma, C) \longrightarrow \exists C', \theta'.\ (R^A_{\theta'}(\gamma, C) \wedge C \rightsquigarrow C' \wedge R^A_{\theta'}(\gamma', C'))$$

We argue that if there is a relation between γ and C and $\gamma \rightsquigarrow \gamma'$ by executing a line of pseudocode that maps to a *CALF* event e then $C \rightsquigarrow_e C'$ that corresponds to γ'. We have the more complex condition on the mappings to take into account non deterministic allocations and mapping them onto *CALF* objects.

Proof. Cases

1. We are allocating a new object σ' i.e. $\gamma.heap(\sigma') = \bot$:

   If γ and C are related with θ then we are looking for some θ' such that the following holds.

   $$(\forall \sigma.\ \gamma.heap(\sigma) \neq \bot \longrightarrow \theta(\sigma) = \theta'(\sigma)) \wedge O(\theta(\sigma')) = A(\mathcal{O}(\sigma'))$$

   That is, θ and θ' agree for all objects in γ and θ' maps the new object to some ι' such that ι' has the correct owner up to A.

   But there are an infinite number of θ' that satisfy the first condition, and there must be one that maps σ' to the owner specified in the second.

2. We are executing some behaviour or mutating some heap object:

   CALF fundamentally does not retain information about object states or relations between them, so these are no-ops in *CALF* .

   Specifically, these do not change any message queues or mutate any reference counts so will not affect the relation.

3. We are removing some object:

   This is similar to the above, removing an object has no meaning in *CALF* .

4. We are executing some ORCA method with a code mapping

   Receive Orca

   - Enter method: the only thing we need to check is that the fine grained condition is maintained. Before receiving we must have been in an idle states so the fine grained condition on γ and C states that C is closed with respect to $A(\beta)$. But this is all we require at the beginning of the method for the fine grained condition on γ' and C'. We know as there is no event mapping though that $C = C'$.
   - Line 5 (Pop): The begin receive event pops off the message queue. This means that the transition $C \rightsquigarrow C'$ and $\gamma \rightsquigarrow \gamma'$ act by removing a message from a message queue. We know from the message mapping that these two messages are related by θ and A, and the actor specified by γ's transition is precisely $A(\beta)$. From this we argue that the message queue mapping is preserved. Also for the fine grained condition the postcondition of this begin event on $A(\beta)$ places the reference count, object and message value into the RWS. For all other objects the reference counts in both C' and γ' are unaffected.This satisfies the fine grained condition for $\theta(\sigma)$.
   - Line 6 (RC increment): The reference count is updated in $ORCA^1$ and the value is taken into the finalise set in *CALF* , we can that this satisfies the fine grained condition.
   - Line 7 (Pass and end): The reference count is set to a non-bottom value in γ'. C' has the A and θ equivalent reference count set to the future set of C, which we know by the fine grained condition to be what is being set in γ'. So we restore $closed(C'|A(\beta))$ and all reference counts are equivalent for β.
   - Leave method: Similar to entering, we just need to check the fine grained condition. But we know that $C = C'$ and $closed(C, A(\beta))$.

   The entering and leaving of methods will follow similar arguments so we will omit them.

   Receive

   - Line 7 (Pop): We modify the message queues of γ and C in the same way on actors that are related by A. As $C \rightsquigarrow C'$ all the non zero entries in the message are entered into the RWS. Similarly as $\gamma \rightsquigarrow \gamma'$ all entries in the application message are put into the working set. These two sets objects are related by θ and the message mapping. Because of this we can argue that for all elements in the working set their θ-equivalent object is in the RWS, and all others are unaffected. Therefore we have the fine grained condition on γ' and C'.
   - Line 9 (Loop start): We check the fine grained condition for γ' and see a new $\sigma_9 \in ws$ is defined in γ', but the conditions we require are satisfied by all $\sigma \in ws$ in γ.
   - Line 10 (Update RC on owned): $C \rightsquigarrow C'$ by adding to the future set the previous reference count minus the received reference count. $\gamma \rightsquigarrow \gamma'$ by setting the reference count to its value minus one. We know from the fine grained condition that the received reference count in the RWS for C is one. Therefore this operation gives the new fine grained condition.

- Line 11 (Update RC on unowned): Similar to Line 10.
- Line 12 (Remove from working set): We check the fine grained condition for γ' and see that we require for σ_9 that $\theta(\sigma_9)$ is in the future set, but we have that from γ.
- Line 14 (Update frame): $C \rightsquigarrow C'$ by setting the reference counts to the value in the finalise set. We then know from the fine grained condition on C that all reference counts not in this set are non-bottom, so $closed(C'|A(\beta))$. Then as before, we argue this preserves the equivalence of reference counts up to the mappings A and θ.

Send

- Line 7 (trace working set): $\gamma \rightsquigarrow \gamma'$ by tracing the object σ and storing that set into the variable ws. $C \rightsquigarrow C'$ by considering the elements in that set under the action of θ. C' has the reference count for these corresponding objects in $A(\beta)$ set to $\perp$ and moves their values into the sending working set. But this corresponds with the fine grained condition.
- Line 8 (loop start): $C = C'$, we just check the fine grained condition as $\gamma \rightsquigarrow \gamma'$. All the constraints we have on σ_8 in γ' we have on all $\sigma \in ws$ for γ.
- Line 10 (Update RC on owned): $C \rightsquigarrow C'$ by adding to the future set the previous reference count minus one. $\gamma \rightsquigarrow \gamma'$ by setting the reference count to its value minus one. This maintains the fine grained condition.
- Line 12 (Update RC on unowned and RC not one): Similar to above.
- Line 14 (Push message): As $\gamma \rightsquigarrow \gamma'$ β sends an orca message to the owner of σ_8 to increase the reference count by 256. As $C \rightsquigarrow C'$ $A(\beta)$ puts a message onto $O(\theta(\sigma_8)$'s message queue for negative 256 for $\theta(\sigma_8)$. We know on C and γ that $O(\theta(\sigma_8)) = A(\mathcal{O}(\sigma_8))$ and so the equivalence of message queues is preserved.

  As well as this the finalise set of γ' is set to contain $(\theta(\sigma_8), 256)$ which fulfils the fine grained condition.
- Line 15 (Update RC on unowned and RC one): As $\gamma \rightsquigarrow \gamma'$ the reference count for σ_8 is set to 256, from the fine grained condition we know that for $A(\beta)$ $(\theta(\sigma), 256)$ is in the finalise set, and $C = C'$ so this just equalises those values.
- Line 16 (Remove from working set): From the fine grained condition σ_8 satisfies all the requirements for an object outside the working set. So the fine grained condition is preserved.
- Line 20 (Send): $\gamma \rightsquigarrow \gamma'$ by sending a message to some actor β' with the frame ϕ. $C \rightsquigarrow C'$ by sending a message to $A(\beta')$ with the θ-translated trace of ϕ. From the definition of G_θ^{msgs} we can see this preserves the message queue equivalence.

  Also from the fine grained condition on C we have $closed(C'|A(\beta)$ so the reference counts must match, but we also have that from the fine grained condition on C.

Garbage Collection All the cases are similar to ones seen in the other methods, we omit this case for brevity.

□

9.6 Pullback Invariants

With the relation R_θ^A the invarients on *CALF* induce *pullback invariants* on *ORCA*[1] . However these invariants only talk about the reference counts, we now need to relate these to accessibility.

Definition 9.11 (Pullback)**.**

$$Pullback(\gamma, C) \triangleq \exists A, \theta.\ R^A_\theta(\gamma, C) \wedge C \vDash \Diamond$$

Proposition 9.12 (Pullback Preserved)**.**

$$\gamma \rightsquigarrow \gamma' \wedge Pullback(\gamma, C) \longrightarrow \exists C'.\ (Pullback(\gamma', C') \wedge C \rightsquigarrow C') \tag{1}$$

Proof. We know from Proposition 9.10 that there exists some C', θ' such that $R^A_{\theta'}(\gamma', C')$. This configuration C' is also well formed as $C \rightsquigarrow C'$. We know from Proposition 8.31 that $C' \vDash \Diamond$ holds - but this is exactly $Pullback(\gamma')$. □

Theorem 9.13.

$$Pullback(\gamma, C) \wedge \beta.st_\gamma = COLLECT \wedge \mathcal{O}(\sigma) = \beta \wedge \beta.rc_\gamma(\sigma) = 0 \longrightarrow (\forall \beta'.\ \beta'.rc_\gamma(\sigma) = 0)$$

Proof. We know that there is some A, θ, C such that $R^A_\theta(\gamma, C)$. This gives us $(FGC^{GC})^A_\theta(\gamma, C)$ and it is clear that as σ is owned we have $closed(C|A(\beta), \{\theta(\sigma)\})$.

Furthermore we know from the correspondence that $\theta(\sigma)$ is local to $A(\beta)$ and that the reference counts $\beta.rc_\gamma(\sigma)$ and $C.A(\beta).rc(\theta(\sigma))$ are the same.

Now from theorems on *CALF* , we have $C \vDash \Diamond$ so $C \vDash WFC$. This means that for all actors α in C, $C.\alpha.rc(\theta(\sigma)) = 0$ so we have $closed(C|\{\theta(\sigma)\})$.

This means that the actors in γ that map onto the actors in C must all have zero reference count for σ. □

9.7 Access

So we managed to prove properties of the reference counts in $ORCA^0$, but we have not given them any meaning - we need to relate reference counts to accessibility.

This is great, but in our model we currently have no relationship between reference counts and accessibility, the above theorem does not really mean anything.

In order for to relate the reference counts to accessibility we introduce the following invariant similar to $\mathbf{I}_3$ in $ORCA^0$.

Definition 9.14 (Foreign Access)**.**

$$\mathbf{I}_{FA} \triangleq \mathcal{A}_C(\beta, lp) = \sigma \longrightarrow \beta.rc_C(\sigma) > 0$$

We assume that this can be proved in a similar way to $\mathbf{I}_3$ is proved in $ORCA^0$.

9.8 Evaluation

The fine grained relation conditions bear resemblance to the invariants about ghost variables we defined for $ORCA^{\text{Ghost}}$. It is unfortunate that we require such a condition but we need to be able to reason at every execution step the exact relation between related *CALF* configurations. However, specifying this single line-by-line invariant alongside our proofs about *CALF* gives rise to properties that required far more specification in $ORCA^0$.

Our original plan with this model was to define partial orders $\sqsubseteq_\gamma$ and $\sqsubseteq_C$ on $ORCA^1$ and *CALF* respectively. These would represent approximations on the states, so $\gamma \sqsubseteq_\gamma \gamma'$ would be interpreted as "γ is more specific than γ'". This in particular would be used to represent that when *CALF* reference counts are $\bot$, then the configuration has no information about what the value is. We then defined functions between configurations such that they form a *Galois Connection*. Then

properties on *CALF* would induce properties on $ORCA^1$. However, in models formed like this execution is expected to be denoted by an order-preserving function, but that is not the case here. When we apply begin events and end events we are explicitly manipulating the knowledge of the program which means that this approach is not possible.

Returning back to our final formulation, in order to prove $\mathbf{I}_{FA}$ it may be required to some of the the logic we were trying to remove by building this model which is a shame.

Overall though, we can see how this approach could be used to modularise the reasoning about a model, by building separate submodels for components and reasoning about these individually.We view this formulation as a success and as a proof of concept for using this approach for both reasoning with closures and for using submodels.

10 Closure Invariants

10.1 Modal Logic

In this section we show how we can think of closures of configurations in terms of modal logic and μ-calculus. Unfortunately we do not have any big theorems or results using this but we still think it is interesting enough to mention here.

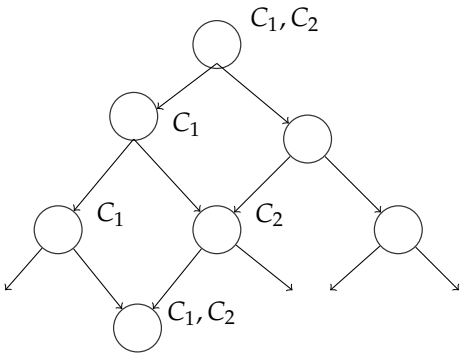


Figure 28: Showing a Kripke frame where C_ι denotes that the state is closed for the variable ι.

In order to prove the next results we define a modal logic through the construction of the following Kripke frame (W, R): We fix some set of protocols, rules and an initial state. Then W is the set of all valid histories. Then we define the relation R

Definition 10.1.

$$(H, H') \in R \iff H' = H \cdot e$$

where e is some event $begin(p, \alpha, X)$, $mid(p, \alpha, X)$ or $end(p, \alpha, X)$ and $(\cdot)$ is concatenation.

Then $\Box p$ states that, for a given world (some valid history) then p holds at every possible next execution state from that history.

If we fix C_0 as our starting configuration and have a system of rules such that with a finite history H if $C_0 \rightsquigarrow_H C$ then the closure of C is made up of $\lceil C \rceil$ such that $C \rightsquigarrow_{H'} \lceil C \rceil$ with a finite history H'. With this we can prove the following

Lemma 10.2. $(\forall H.\ C_0 \rightsquigarrow_H C \wedge \forall \lceil C \rceil \in closure(C|X).\ P(\lceil C \rceil)) \implies \nu x(P \vee \Box x)$ *is valid*

Here ν is the μ-calculus greatest fixed point operator, and the $\implies$ exists in some "meta logic" showing the first order formula implying the validity of the modal one.

This states that if some property holds at all configurations in closures of X then either execution continues forever execution eventually reaches a point where this holds no matter the nondeterministic choice.

Proof. Let H be an arbitrary history, suppose for contradiction there is some finite path through histories $H_0, \ldots H_n$ such that $H_n \notin closure(H_n|X)$, there is no infinite path from H_n and no finite path to a closed history. This is a contradiction if H_n is not closed, there are events $end(r_1), \ldots end(r_m)$ such that $H_n{+}{+}[end(r_1), \ldots end(r_m)]$ is closed with this and all intermediate histories valid from assumptions about the execution rules. □

From this we define a property as a *closure invariant*.

Definition 10.3 (Closure Invariant)**.**

$$\varkappa P \triangleq \nu x(P \vee \Box x)$$

Then if we have some property true at a world when some conjunction of closure invariants holds, then intuitively we can deduce this property of a system by reasoning only about the properties that hold at closures.

Theorem 10.4 (Closure Definable)**.**

$$ClosureDefinable(p) \triangleq p \iff (\varkappa Q_1) \wedge (\varkappa Q_2) \cdots \wedge (\varkappa Q_n)$$

We believe that this is an interesting field for future development and can be applied to other algorithms on actors and in concurrent and distributed contexts.

Conjecture. *There is some set of condition on protocols and events such that all modal formulae that the system models are closure definable.*

11 Conclusion

In conclusion we are pleased with the results that we produced from this project. The greatest challenge ended up being looking at the Pony runtime source code both to understand the current implementation and to when we tried to make changes. It took a long time of working through the codebase to understand how the immutable object optimisation was handled. In the end we focused on the theoretical aspects of the project, the source code may have played a role in this.

11.1 Dead Ends

During the process of the project we tried many ideas, inevitably leading to conceptual dead-ends. We have described in other sections the iterative process that allowed us to develop many of the definitions and results we have presented, but in order to reach this point we have thrown out many ideas.

11.1.1 Haskell Simulation

As part of the project we built a Haskell simulation of the runtime to test different collection algorithms. We believed this could have been used to produce quantitative benchmarks and comparisons between algorithms without making changes to the Pony runtime.

The simulation contained a base representation of actors with heaps, objects and reference counts and paths and a more advanced representation where we embedded the Pony capability system in the Haskell type system. Using *type families* we were able to model the capabilities and viewpointing system of Pony. If we wrote a program where an actor tries to read from a path, then the program would only compile if the path had the correct capability.

Unfortunately however, the simulator went unused.

11.1.2 Pony Algebra

Before *CALF* we tried defining another minimalistic model, the "Pony Algebra" was an attempt to reason about properties in a small actor context. This was inspired by the resource algebra defined in Iris [Jung, 2018]. The Pony Algebra contained a composition operator $\circ$ defined when two configurations were "compatible" with respect to object paths. For example two configurations with disjoint actors referencing the same object with an `Iso` capability would not be compatible.

Then the operator would form a larger configuration by fusing its two arguments. The idea was to form local properties that would hold under this fusion to become global properties.

$$\begin{array}{ccccc} C_0 & \circ & C_1 & = & C_2 \\ \downarrow & & \downarrow & & \downarrow \\ C_0' & \circ & C_1' & = & C_2' \end{array} \quad (\rightsquigarrow) \quad (\rightsquigarrow) \quad (\rightsquigarrow)$$

We would investigate properties that satisfy the following two conditions.

1. (Composition) $P(c) \wedge P(c') \iff P(c \circ c')$
2. (Execution Preservation) $c \rightsquigarrow c' \wedge P(c) \to P(c')$

We then proved that global inaccessibility was such a property, however beyond this the model seemed to have little use unfortunately.

11.1.3 Separation Logic

Reasoning about a concurrent garbage collector seems the perfect domain for the application of concurrent separation logic. In particular we looked at Iris [Jung, 2018] a framework proving properties of concurrent separation logic using Coq. Unfortunately we did not have the time to learn the tool to make useful progress.

11.2 Contributions

Regardless we feel the results we were able to gather from the project are valuable. We developed $ORCA^{\text{Ghost}}$ which we feel is an improvement over the previous model $ORCA^0$ due to the removal of technical definitions and simplification of invariants.

We managed to prove the soundness of the immutability optimisations, which should show that the overall approach taken in the Pony runtime is correct. We also showed that the only place where a memory leak can happen with this model is in an immutable object cycle across actors.

We managed to use reasoning about the closures of histories and configurations to both provide a clean submodel of ORCA and to show that the approach is possible. We believe that this technique is very promising and could be applied to other algorithms - concurrent, actor based or distributed.

Setting out on this project we aimed to prove the soundness and completeness of the ORCA system as used in the Pony runtime and we have mostly succeeded at this.

12 Further Work

We finish by giving some points of interest where the work we presented could be developed further:

- An implementation of $ORCA^{\text{Ghost+Val}}$. An implementation would allow us to benchmark the performance of the model against the current implementation. We theorised that in scenarios with more sending and receiving than garbage collection but a quantitative study would verify that. Also it may be that we value performance during sending and receiving over performance in garbage collection which would make this the prefered algorithm.
- Cycle detection for objects. A cycle detection algorithm could be formulated, verified and implemented. Adding this to $ORCA^{\text{Ghost+Val}}$ would give a complete algorithm.
- Mechanising the proofs with some proof assistant. As mentioned earlier we looked at Coq and Iris but were unable to make use of them within the project.
- Further development of reasoning on closures. Some interesting questions are
  - When can a system be fully specified by its closures?
  - How much information do you need to prove an arbitrary property on a story and history system?

References

[Akka, 2018] Akka (2018). Akka. https://akka.io/. [Online; accessed 20-June-2018].

[Chen et al., 2005] Chen, Y., Dios, R., Mili, A., Wu, L., and Wang, K. (2005). An empirical study of programming language trends. 22:72– 79.

[Clebsch et al., 2017] Clebsch, S., Franco, J., Drossopoulou, S., Yang, A. M., Wrigstad, T., and Vitek, J. (2017). Orca: Gc and type system co-design for actor languages. *Proc. ACM Program. Lang.*, 1(OOPSLA):72:1–72:28.

[Dijkstra et al., 1978] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. (1978). On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM 21*, 11:966–975.

[Erlang, 2018] Erlang (2018). Erlang. http://www.erlang.org/. [Online; accessed 24-January-2018].

[Franco et al., 2018] Franco, J., Clebsch, S., Drossopoulou, S., Vitek, J., and Wrigstad, T. (2018). Correctness of a concurrent object collector for actor languages. In Ahmed, A., editor, *Programming Languages and Systems*, pages 885–911, Cham. Springer International Publishing.

[Gosling et al., 2014] Gosling, J., Joy, B., Steele, G. L., Bracha, G., and Buckley, A. (2014). *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition.

[Greif, 1975] Greif, I. (1975). *Semantics of communicating parallel processes.* PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science.

[Herlihy and Wing, 1990] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

[Hewitt et al., 1973] Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Jung, 2018] Jung, R. (2018). *Iris from the ground up*. Accepted for publication in the Journal of Functional Programming (JFP), 2018.

[Kernighan. and Ritchie, 1988] Kernighan. and Ritchie (1988). *The C Programming Language*.

[Klabnik and Nichols, 2018] Klabnik, S. and Nichols, C. (2018). *The Rust Programming Language*.

[Liétar, 2017] Liétar, P. (2017). Formalizing generics for pony.

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

[Pony, 2018] Pony (2018). Pony language. http://ponylang.org. [Online; accessed 24-January-2018].

[Seacord, 2005] Seacord, R. C. (2005). *Secure coding in C and C++*.

[w. Appel,] w. Appel, A. *Modern Compiler Implementation in Java*. Cambridge University Press.

[Zorn, 1990] Zorn, B. (1990). Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 87–98, New York, NY, USA. ACM.

Appendices

A $ORCA^0$ I_8

Definition A.1 (Actor States, Working sets, and Marks)**.**

$$\begin{array}{rcl} \mathsf{st} \in \mathit{State} & ::= & \text{IDLE} \mid \text{EXECUTE} \mid \text{SEND} \mid \text{RECEIVE} \mid \text{COLLECT} \\ \mathsf{ws} \in \mathit{Workset} & = & \mathcal{P}(\mathit{Addr}) \\ \mathsf{ms} \in \mathit{Marks} & = & \mathit{Addr} \to \{\mathsf{R}, \mathsf{U}\} \\ \mathsf{pc} \in \mathit{PC} & = & [4..27] \end{array}$$

Definition A.2 (Tracing)**.** We define the functions
trace_this : $\mathit{Config} \times \mathit{ActorAddr} \to \mathcal{P}(\mathit{Addr})$
trace_frame : $\mathit{Config} \times \mathit{ActorAddr} \times \mathit{Frame} \to \mathcal{P}(\mathit{Addr})$
as follows

$$\begin{array}{l} \mathsf{trace\_this}_{\mathcal{C}}(\alpha) \triangleq \{\iota \mid \exists \overline{f}.\ \mathcal{A}_{\mathcal{C}}(\alpha, \mathsf{this}.\overline{f}) = \iota\} \\ \mathsf{trace\_frame}_{\mathcal{C}}(\alpha, \phi) \triangleq \{\iota \mid \exists x \in \mathit{dom}(\phi), \overline{f}.\ \mathcal{A}_{\mathcal{C}}(\alpha, x.\overline{f}) = \iota\} \end{array}$$

Definition A.3 (Well-formed marking state)**.** We define that an actor which is in state COLLECT is well formed, i.e. $\mathcal{C}, \alpha \vDash \text{COLLECT}$ if all the following criteria are satisfied:
$\mathcal{C}, \alpha \vDash \text{COLLECT} \Leftrightarrow$

initGC

$\alpha.\mathsf{st}_{\mathcal{C}} = \text{COLLECT}$

markU

$$\begin{array}{l} 6 < \alpha.\mathsf{pc}_{\mathcal{C}} < 12 \ \rightarrow \\ \quad (1)\ \ \mathit{dom}(\mathsf{ms}) \subseteq \mathcal{D}_{\mathcal{C}(\alpha)} \quad \wedge \\ \quad (2)\ \ \forall \iota.\ (\ \mathsf{ms}(\iota) = \mathsf{U} \longrightarrow [\ \mathcal{O}(\iota) = \alpha \ \vee\ \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) > 0\ ]\ ) \quad \wedge \\ \quad (3)\ \ \forall \iota \in \mathit{dom}(\mathsf{ms}).\, \mathsf{ms}(\iota) = \mathsf{U} \end{array}$$

trace

$$\begin{array}{l} 12 \leq \alpha.\mathsf{pc}_{\mathcal{C}} < 15 \ \rightarrow \\ \quad (1)\ \ \mathit{dom}(\mathsf{ms}) = \mathcal{D}_{\mathcal{C}(\alpha)} \quad \wedge \\ \quad (2)\ \ \forall \iota.\, [\, \mathsf{ms}(\iota) = \mathsf{R} \longrightarrow \exists lp.\mathcal{A}_{\mathcal{C}}(\alpha, lp) = \iota\,] \end{array}$$

markR

$$\begin{array}{l} 15 \leq \alpha.\mathsf{pc}_{\mathcal{C}} < 18 \ \rightarrow \\ \quad (1)\ \ \mathit{dom}(\mathsf{ms}) = \mathcal{D}_{\mathcal{C}(\alpha)} \quad \wedge \\ \quad (2)\ \ \forall \iota.\, [\, \exists lp.\mathcal{A}_{\mathcal{C}}(\alpha, lp) = \iota \longrightarrow \mathsf{ms}(\iota) = \mathsf{R}\,] \quad \wedge \\ \quad (3)\ \ \forall \iota.\, [\, \mathsf{ms}(\iota) = \mathsf{R} \longrightarrow (\, \exists lp.\mathcal{A}_{\mathcal{C}}(\alpha, lp) = \iota \ \vee\ \mathcal{O}(\iota) = \alpha \wedge \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) > 0\,)\,] \end{array}$$

cll

$$\begin{array}{l} 18 \leq \alpha.\mathsf{pc}_{\mathcal{C}} < 26 \ \rightarrow \\ \forall \iota.\ [\ \mathsf{ms}(\iota) = \mathsf{U} \longrightarrow \\ \qquad (1)\ \ [\ \forall lp.\ \mathcal{A}_{\mathcal{C}}(\alpha, lp) \neq \iota\ ] \quad \wedge \\ \qquad (2)\ \ [\ \mathcal{O}(\iota) \neq \alpha \longrightarrow \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) > 0\ ] \quad \wedge \\ \qquad (3)\ \ [\ \mathcal{O}(\iota) = \alpha \longrightarrow \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) = 0\ ] \quad]. \end{array}$$

For convenience, we define the following set:
$\mathcal{D}_{\mathcal{C}(\alpha)} \equiv \{\iota \mid \mathcal{O}(\iota) = \alpha \ \vee\ \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) > 0\}$

Definition A.4 (I_8)**. I8-exe** $\mathcal{C}, \alpha \models \text{EXECUTE}$ iff $\alpha.\mathsf{st}_{\mathcal{C}} = \text{EXECUTE}$

$\mathbf{I}_8$-idle $\mathcal{C}, \alpha \models$ IDLE iff $\alpha.\mathsf{st}_{\mathcal{C}} =$ IDLE and $\alpha.\mathsf{frame}_{\mathcal{C}} = \emptyset$

Definition A.5. $\mathcal{C}, \alpha \models$ RECEIVE iff

A $\alpha.\mathsf{st}_{\mathcal{C}} =$ RECEIVE

B $8 \leq \alpha.\mathsf{pc}_{\mathcal{C}} \;\rightarrow\; \mathsf{ws} \subseteq \mathrm{trace\_frame}_{\mathcal{C}}(\alpha, \phi)$

C $\forall \iota, x, \overline{f}.[\, \mathcal{A}_{\mathcal{C}}(\alpha, -1.x.\overline{f}) = \iota \;\longrightarrow\; \mathrm{LRC}_{\mathcal{C}}(\iota) > 0$

E $8 < \alpha.\mathsf{pc}_{\mathcal{C}} \;\longrightarrow$
$\forall \iota \in \mathsf{ws} \setminus \mathcal{C}urrAddrRcv_{\mathcal{C}}(\alpha).$
$[\, \mathcal{O}(\iota) = \alpha \;\rightarrow$
a $\forall n.\, QueueEffct_{\mathcal{C}}(\alpha, \iota, n) > 1$
b $Reaches_{\mathcal{C}}(\alpha, \iota, \alpha.\mathsf{qu}_{\mathcal{C}}[j]) \;\longrightarrow\; \forall k \leq j.\, QueueEffct_{\mathcal{C}}(\alpha, \iota, j) > 1$
c $\alpha.\mathsf{qu}_{\mathcal{C}}[j] = \mathsf{orca}(\iota : z) \;\longrightarrow\; \forall k < j.\, QueueEffct_{\mathcal{C}}(\alpha, \iota, j) > 1$
d $\exists p, \alpha'.\, \alpha' \neq \alpha \wedge \mathcal{A}_{\mathcal{C}}(\alpha', p) = \iota \;\longrightarrow\; QueueEffct_{\mathcal{C}}(\alpha, \iota, k) > 1 \;\;]$

Definition A.6. $\mathcal{C}, \alpha \models$ SEND iff

Q $\alpha.\mathsf{st}_{\mathcal{C}} =$ SEND

R $9 \leq \alpha.\mathsf{pc}_{\mathcal{C}} \;\rightarrow\; \mathsf{ws} \subseteq \mathrm{trace\_frame}_{\mathcal{C}}(\alpha, \phi)$

S $\forall \iota, x, \overline{f}.[\, \mathcal{A}_{\mathcal{C}}(\alpha, -1.x.\overline{f}) = \iota \;\longrightarrow\; \mathrm{LRC}_{\mathcal{C}}(\iota) > 0$