# Correctness of a Concurrent Object Collector for Actor Languages

Juliana Franco[1(✉)], Sylvan Clebsch[2], Sophia Drossopoulou[1], Jan Vitek[3,4], and Tobias Wrigstad[5]

[1] Imperial College London, London, UK
j.vicente-franco@imperial.ac.uk
[2] Microsoft Research Cambridge, Cambridge, UK
[3] Northeastern University, Boston, USA
[4] CVUT, Prague, Czech Republic
[5] Uppsala University, Uppsala, Sweden

**Abstract.** ORCA is a garbage collection protocol for actor-based programs. Multiple actors may mutate the heap while the collector is running without any dedicated synchronisation. ORCA is applicable to any actor language whose type system prevents data races and which supports causal message delivery. We present a model of ORCA which is parametric to the host language and its type system. We describe the interplay between the host language and the collector. We give invariants preserved by ORCA, and prove its soundness and completeness.

## 1 Introduction

Actor-based systems are massively parallel programs in which individual actors communicate by exchanging messages. In such systems it is essential to be able to manage data automatically with as little synchronisation as possible. In previous work [9,12], we introduced the ORCA protocol for garbage collection in actor-based systems. ORCA is language-agnostic, and it allows for concurrent collection of objects in actor-based programs with no additional locking or synchronisation, no copying on message passing and no stop-the-world steps. ORCA can be implemented in any actor-based system or language that has a type system which prevents data races and that supports causal message delivery. There are currently two instantiations of ORCA, one is for Pony [8,11] and the other for Encore [5]. We hypothesise that ORCA could be applied to other actor-based systems that use static types to enforce isolation [7,21,28,36]. For libraries, such as Akka, which provide actor-like facilities, pluggable type systems could be used to enforce isolation [20].

This paper develops a formal model of ORCA. More specifically, the paper contributions are:

1. Identification of the requirements that the host language must statically guarantee;


A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 885–911, 2018.
https://doi.org/10.1007/978-3-319-89884-1_31

2. Description and model of ORCA at a language-agnostic level;
3. Identification of invariants that ensure global consistency without synchronisation;
4. Proofs of *soundness*, *i.e.* live objects will not be collected, and proofs of *completeness*, *i.e.* all garbage will be identified as such.

A formal model facilitates the understanding of how ORCA can be applied to different languages. It also allows us to explore extensions such as shared mutable state across actors [40], reduction of tracing of immutable references [12], or incorporation of borrowing [4]. Alternative implementations of ORCA that rely on deep copying (*e.g.*, to reduce type system complexity) across actors on different machines can also be explored through our formalism.

Developing a formal model of ORCA presents challenges:

*Can the model be parametric in the host language?* We achieved parametricity by concentrating on the effects rather than the mechanisms of the language. We do not model language features, instead, we model actor behaviour through non-deterministic choice between heap mutation and object creation. All other actions, such as method call, conditionals, loops etc., are irrelevant.

*Can the model be parametric in the host type system?* We achieved parametricity by concentrating on the guarantees rather than the mechanism afforded by the type system. We do not define judgments, but instead, assume the existence of judgements which determines whether a path is readable or writeable from a given actor. Through an (uninterpreted) precondition to any heap mutation, we require that no aliasing lets an object writeable from an actor be readable/writeable from any other actor.

*How to relax atomicity?* ORCA relies on a global invariant that relates the number of references to any data object and the number of messages with a path to that object. This invariant only holds if actors execute atomically. Since we desire actors to run in parallel, we developed a more subtle, and weaker, definition of the invariant.

The full proofs and omitted definitions are available in appendix [16].

## 2 Host Language Requirements

ORCA makes some assumptions about its host language, we describe them here.

### 2.1 Actors and Objects

Actors are active entities with a thread of control, while objects are data structures. Both actors and objects may have fields and methods. Method calls on objects are synchronous, whereas method calls on actors amount to asynchronous message sends—they all called *behaviours*. Messages are stored in a FIFO queue. When idle, an actor processes the top message from its queue. At any given point of time an actor may be either idle, executing a behaviour, or collecting garbage.

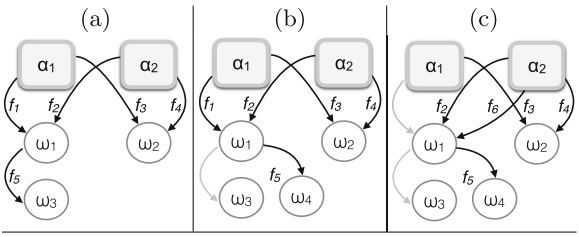



**Fig. 1.** Actors and objects. Full arrows are references, grey arrows are overwritten references: references that no longer exist.

| Actor | Path | Capapability | Actor | Path | Capability |
|---|---|---|---|---|---|
| | this.$f_1$ | write | | this.$f_2$ | tag |
| | this.$f_1$.$f_5$ | write | | this.$f_2$.$f_5$ | $\bot$ |
| $\alpha_1$ | this.$f_3$ | read | $\alpha_2$ | this.$f_4$ | read |
| | | | | this.$f_6$ | write |
| | | | | this.$f_6$.$f_5$ | write |

**Fig. 2.** Capabilities. Heap mutation may modify what object is reachable through a path, but not the path's capability.

Figure 1 shows actors $\alpha_1$ and $\alpha_2$, objects $\omega_1$ to $\omega_4$. In [16] we show how to create this object graph in Pony. In Fig. 1(a), actor $\alpha_1$ points to object $\omega_1$ through field $f_1$ to $\omega_2$ through field $f_3$, and object $\omega_1$ points to $\omega_3$ through field $f_5$. In Fig. 1(b), actor $\alpha_1$ creates $\omega_4$ and assigns it to this.$f_1$.$f_5$. In Fig. 1(c), $\alpha_1$ has given up its reference to $\omega_1$ and sent it to $act_2$ which stored it in field $f_6$. Note that the process of sending sent not only $\omega_1$ but also implicitily $\omega_4$.

### 2.2 Mutation, Transfer and Accessibility

Message passing is the only way to share objects. This falls out of the capability system. If an actor shares an object with another actor, then either it gives up the object or neither actor has a write capability to that object. For example, after $\alpha_1$ sends $\omega_1$ to $\alpha_2$, it cannot mutate $\omega_1$. As a consequence, heap mutation only decreases accessibility, while message sends can transfer accessibility from sender to receiver. When sending immutable data the sender does not need to transfer accessibility. However, when it sends a mutable object it cannot keep the ability to read or to write the object. Thus, upon message send of a mutable object, the actor must consume, or destroy, its reference to that object.

### 2.3 Capabilities and Accessibility

ORCA assumes that a host language's type system assigns *access rights* to paths. A path is a sequence of field names. We call these access rights *capabilities*.

We expect the following three capabilities; read, write, tag. The first two allow reading and writing an object's fields respectively. The tag capability only allows

identity comparison and sending the object in a message. The type system must ensure that actors have no read-write races. This is natural for actor languages [5, 7,11,21].

Figure 2 shows capabilities assigned to the paths in Fig. 1: $\alpha_1.f_1.f_5$ has capability write, thus $\alpha_1$ can read and write to the object reachable from that path. Note that capapabilities assigned to paths are immutable, while the contents of those paths may change. For example, in Fig. 1(a), $\alpha_1$ can write to $\omega_3$ through path $f_1.f_5$, while in Fig. 1(b) it can write to $\omega_4$ through the same path. In Fig. 1(a) and (b), $\alpha_2$ can use the address of $\omega_1$ but cannot read or write it, due to the tag capability, and therefore cannot access $\omega_3$ (in Fig. 1(a)) nor $\omega_4$ (in Fig. 1(b)). However, in Fig. 1(c) the situation reverses: $\alpha_2$, which received $\omega_1$ with write capability is now able to reach it through field $f_6$, and therefore $\omega_4$. Notice that the existence of a path from an actor to an object does not imply that the object is accessible to the actor: In Fig. 1(a), there is a path from $\alpha_2$ to $\omega_3$, but $\alpha_2$ cannot access $\omega_3$. Capabilities protect against data races by ensuring that if an object can be mutated by an actor, then no other actor can access its fields.

### 2.4 Causality

ORCA uses messages to deliver protocol-related information, it thus requires causal delivery. Messages must be delivered after any and all messages that caused them. Causality is the smallest transitive relation, such that if a message $m'$ is sent by some actor after it received or sent $m$, then $m$ is a cause of $m'$. Causal delivery entails that $m'$ be delivered after $m$.

For example, if actor $\alpha_1$ sends $m_1$ to actor $\alpha_2$, then sends $m_2$ to actor $\alpha_3$, and $\alpha_3$ receives $m_2$ and sends $m_3$ to $\alpha_2$, then $m_1$ is a cause of $m_2$, and $m_2$ is a cause of $m_3$. Causal delivery requires that $\alpha_2$ receive $m_1$ before receiving $m_3$. No requirements are made on the order of delivery to different actors.

## 3 Overview of ORCA

We introduce ORCA and discuss how to localise the necessary information to guarantee safe deallocation of objects in the presence of sharing. Every actor has a local heap in which it allocates objects. An actor *owns* the objects it has allocated, and ownership is fixed for an object's life-time, but actors are free to reference objects that they do not own. Actors are obligated to collect their own objects once these are no longer needed. While collecting, an actor must be able to determine whether an object can be deallocated using only local information. This allows all other actors to make progress at any point.

### 3.1 Mutation and Collection

ORCA relies on capabilities for actors to reference objects owned by other actors and to support concurrent mutation to parts of the heap that are not being concurrently collected. Capabilities avoid the need for barriers.

**I₁** An object accessible with write capability from an actor is not accessible with read or write capability from any other actor.

This invariant ensures an actor, while executing garbage collection, can safely trace any object to which it has read or write access without the need to protect against concurrent mutation from other actors.

### 3.2 Local Collection

An actor can collect its objects based on local information without consulting other actors. For this to be safe, the actor must know that an owned, locally inaccessible, object is also globally inaccessible (*i.e.*, inaccessible from any other actors or messages)[1]. Shared objects are reference counted by their owner to ensure:

**I₂** An object accessible from a message queue or from a non-owning actor has reference count larger than zero in the owning actor.

Thus, a locally inaccessible object with a reference count of 0 can be collected.

### 3.3 Messages and Collection

**I₁** and **I₂** are sufficient to ensure that local collection is safe. Maintaining **I₂** is not trivial as accessibility is affected by message sends. Moreover, it is possible for an actor to share a read object with another actor through a message. What if that actor drops its reference to the object? The object's owner should be informed so it can decrease its reference count. What happens when an actor receives an object in a message? The object's owner should be infomed, so that it can increase its reference count. To reduce message traffic, ORCA uses *distributed, weighted, deferred* reference counts. Each actor maintains reference counts that tracks the sharing of its objects. It also maintains counts for "foreign objects", tracking references to objects owned by other actors. This reference count for non-owning actors is what allows sending/receiving objects without having to inform their owner while maintaining **I₂**. For any object or actor $\iota$, we denote with $\mathrm{LRC}(\iota)$ the reference count for $\iota$ in $\iota$'s owner, and with $\mathrm{FRC}(\iota)$ we denote the sum of the reference counts for $\iota$ in all other actors. The counts do not reflect the number of references, rather the existence of references:

**I₃** If a non-owning actor can access an object through a path from its fields or call stack, its reference count for this object is greater than 0.

An object is globally accessible if it is accessible from any actor or from a message in some queue. Messages include reference increment or decrement messages—these are ORCA-level messages and they are not visible to applications. We introduce two logical counters: $\mathrm{AMC}(\iota)$ to account for the number of application

[1] For example, in Fig. 1(c) $\omega_4$ in is locally inaccessible, but globally accessible.

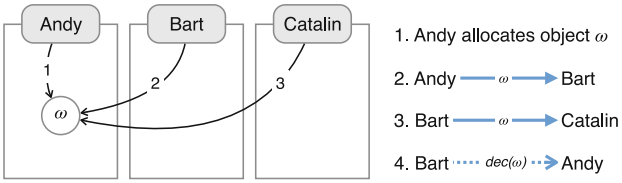



**Fig. 3.** Black arrows are references, numbered in creation order. Blue solid arrows are application messages and blue dashed arrows ORCA-level message. (Color figure online)

messages with paths to $\iota$, and $\mathrm{OMC}(\iota)$ to account for ORCA-level messages with reference count increment and decrement requests. These counters are not present at run-time, but they will be handy for reasoning about ORCA. The owner's view of an object is described by the LRC and the OMC, while the foreign view is described by the FRC and the AMC. These two views must agree:

$$\mathbf{I_4} \qquad \forall\, \iota.\ \ \mathrm{LRC}(\iota) + \mathrm{OMC}(\iota) = \mathrm{AMC}(\iota) + \mathrm{FRC}(\iota)$$

$\mathbf{I_2}$, $\mathbf{I_3}$ and $\mathbf{I_4}$ imply that a locally inaccessible object with $\mathrm{LRC} = 0$ can be reclaimed.

### 3.4 Example

Consider actors Andy, Bart and Catalin, and steps from Fig. 3.

*Initial State.* Let $\omega$ be a newly allocated object. As it is only accessible to its owning actor, Andy, there is no entry for it in any RC.

*Sharing $\omega$.* When Andy shares $\omega$ with Bart, $\omega$ is placed on Bart's message queue, meaning that $\mathrm{AMC}(\omega) = 1$. This is reflected by setting $\mathrm{RC}_{\mathsf{Andy}}(\omega)$ to 1. This preserves $\mathbf{I_4}$ and the other invariants. When Bart takes the message with $\omega$ from his queue, $\mathrm{AMC}(\omega)$ becomes zero, and Bart sets his foreign reference count for $\omega$ to 1, that is, $\mathrm{RC}_{\mathsf{Bart}}(\omega) = 1$. When Bart shares $\omega$ with Catalin, we get $\mathrm{AMC}(\omega) = 1$. To preserve $\mathbf{I_4}$, Bart could set $\mathrm{RC}_{\mathsf{Bart}}(\omega)$ to 0, but this would break $\mathbf{I_3}$. Instead, Bart sends an ORCA-level message to Andy, asking him to increment his (local) reference count by some $n$, and sets his own $\mathrm{RC}_{\mathsf{Bart}}(\omega)$ to $n$.[2] This preserves $\mathbf{I_4}$ and the other invariants. When Catalin receives the message later on, she will behave similarly to Bart in step 2, and set $\mathrm{RC}_{\mathsf{Catalin}}(\omega) = 1$.

The general rule is that when an actor sends one of its objects, it increments the corresponding (local) RC by 1 (reflecting the increasing number of foreign references) but when it sends a non-owned object, it decrements the corresponding (foreign) RC (reflecting a transfer of some of its stake in the object). Special care needs to be taken when the sender's RC is 1.

[2] This step can be understood as if Bart "borrowed" $n$ units from Andy, added $n - 1$ to his own RC, and gave 1 to the AMC, to reach Catalin eventually.

Further note that if Andy, the owner of $\omega$, received $\omega$, he would decrease his counter for $\omega$ rather than increase it, as his reference count denotes foreign references to $\omega$. When an actor receives one of its owned objects, it *decrements* the corresponding (local) RC by 1 but when it receives a non-owned object, it *increments* the corresponding (foreign) RC by 1.

*Dropping References to $\omega$.* Subsequent to sharing $\omega$ with Catalin, Bart performs GC, and traces his heap without reaching $\omega$ (maybe because it did not store $\omega$ in a field). This means that Bart has given up his stake in $\omega$. This is reflected by sending a message to Andy to decrease his RC for $\omega$ by $n$, and setting Bart's RC for $\omega$ to 0. Andy's local count of the foreign references to $\omega$ are decreased piecemeal like this, until LRC($\omega$) reaches zero. At this point, tracing Andy's local heap can determine if $\omega$ should be collected.

*Further Aspects.* We briefly outline further aspects which play a role in ORCA.

**Concurrency.** Actors execute concurrently. For example, sharing of $\omega$ by Bart and Catalin can happen in parallel. As long as Bart and Catalin have foreign references to $\omega$, they may separately, and in parallel cause manipulation of the global number of references to $\omega$. These manipulations will be captured locally at each site through FRC, and through increment and decrement messages to Andy (OMC).

**Causality.** Increment and decrement messages may arrive in any order. Andy's queue will serialise them, *i.e.* concurrent asynchronous reference count manipulations will be ordered and executed sequentially. Causality is key here, as it prevents ORCA-level messages to be overtaken by application messages which cause RCs to be decremented; thus causality keeps counters non-negative.

**Composite Objects.** Objects message must be traced to find the transitive closure of accessible data. For example, when passing $\omega_1$ in a message in Fig. 1(c), objects accessible through it, *e.g.*, $\omega_4$ will be traced. This is mandated by $\mathbf{I_3}$ and $\mathbf{I_4}$.

Finally, we reflect on the nature of reference counts: they are *distributed*, in the sense that an object's owner and every actor referencing it keep separate counts; *weighted*, in that they do not reflect the number of aliases; and *deferred*, in that they are not manipulated immediately on alias creation or destruction, and that non-local increments/decrements are handled asynchronously.

## 4 The ORCA Protocol

We assume enumerable, disjoint sets *ActorAddr* and *ObjAddr*, for addresses of actors and objects. The union of the two is the set of addresses including null. We require a mapping $\mathcal{C}lass$ that gives the name of the class of each actor in a given configuration, and a mapping $\mathcal{O}$ that returns the owner of an address

$$\begin{aligned} Addr &= ActorAddr \uplus ObjAddr \uplus \{\mathsf{null}\} \\ \mathcal{C}lass&: \ Config \times ActorAddr \to ClassId \\ &\mathcal{O} : Addr \to ActorAddr \end{aligned}$$

such that the owner of an actor is the actor itself, *i.e.*, $\forall \alpha \in ActorAddr.\ \mathcal{O}(\alpha) = \alpha$.

Definition 1 describes run-time configurations, $\mathcal{C}$. They consist of a heap, $\chi$, which maps addresses and field identifiers to addresses,[3] and an actor map, $as$, from actor addresses to actors. Actors consist of a frame, a queue, a reference count table, a state, a working set, marks, and a program counter. Frames are either empty, or consist of the identifier for the currently executing behaviour, and a mapping from variables to addresses. Queues are sequences of messages. A message is either an *application message* of the form $\mathsf{app}(\phi)$ denoting a high-level language message with the frame $\phi$, or an ORCA message, of the form $\mathsf{orca}(\iota : z)$, denoting an in-flight request for a reference count change for $\iota$ by $z$. The state distinguishes whether the actor is idle, or executing some behaviour, or performing garbage collection. We discuss states, working sets, marks, and program counters in Sect. 4.3. We use naming conventions: $\alpha \in ActorAddr$; $\omega \in ObjAddr$; $\iota \in Addr$; $z \in \mathbb{Z}$; $n \in \mathbb{N}$; $b \in BId$; $x \in VarId$; $A \in ClassId$; and $\iota s$ for a sequence of addresses $\iota_1 ... \iota_n$. We write $\mathcal{C}.\mathsf{heap}$ for $\mathcal{C}$'s heap; and $\alpha.\mathsf{qu}_{\mathcal{C}}$, or $\alpha.\mathsf{rc}_{\mathcal{C}}$, or $\alpha.\mathsf{frame}_{\mathcal{C}}$, or $\alpha.\mathsf{st}_{\mathcal{C}}$ for the queue, reference count table, frame or state of actor $\alpha$ in configuration $\mathcal{C}$, respectively.

**Definition 1 (Runtime entities and notation)**

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

*State, Workset, Marks, and PC described in Definition 7.*

**Example:** Figure 4 shows $\mathcal{C}_0$, our running example for a runtime configuration. It has three actors: $\alpha_1$–$\alpha_3$, represented by light grey boxes, and eight objects, $\omega_1$–$\omega_8$, represented by circles. We show ownership by placing the objects in square boxes, *e.g.* $\mathcal{O}(\omega_7) = \alpha_1$. We show references through arrows, *e.g.* $\omega_6$ references $\omega_8$ through field $f_7$, that is, $\mathcal{C}_0.\mathsf{heap}(\omega_6, f_7) = \omega_8$. The frame of $\alpha_2$ contains behaviour identifier $b'$, and maps $x'$ to $\omega_8$. All other frames are empty. The message queue of $\alpha_1$ contains an application message for behaviour $b$ and argument $\omega_5$ for $x$, the queue of $\alpha_2$ is empty, and the queue of $\alpha_3$ an ORCA message for $\omega_7$. The bottom part shows reference count tables: $\alpha_1.\mathsf{rc}_{\mathcal{C}_0}(\alpha_1) = 21$,

[3] Note that we omitted the class of objects. As our model is parametric with the type system, we can abstract from classes, and simplify our model.

and $\alpha_1.\mathsf{rc}_{\mathcal{C}_0}(\omega_7) = 50$. Entries of owned addresses are shaded. Since $\alpha_2$ owns $\alpha_2$ and $\omega_2$, the entries for $\alpha_2.\mathsf{rc}_{\mathcal{C}_0}(\alpha_2)$ and $\alpha_2.\mathsf{rc}_{\mathcal{C}_0}(\omega_2)$ are shaded. Note that $\alpha_1$ has a non-zero entry for $\omega_7$, even though there is no path from $\alpha_1$ to $\omega_7$. There is no entry for $\omega_1$; no such entry is needed, because no actor except for its owner has a path to it. The 0 values indicate potentially non-existent entries in the corresponding tables; for example, the reference count table for actor $\alpha_3$ needs only to contain entries for $\alpha_1$, $\alpha_3$, $\omega_3$, and $\omega_4$. Ownership does not restrict access to an address: *e.g.* actor $\alpha_1$ does not own object $\omega_3$, yet may access it through the path $\mathsf{this}.f_1.f_2.f_3$, may read its field through $\mathsf{this}.f_1.f_2.f_3.f_4$, and may mutate it, *e.g.* by $\mathsf{this}.f_1.f_2.f_3 = \mathsf{this}.f_1$.

Lookup of fields in a configuration is defined in the obvious way, *i.e.*

**Definition 2.** *$\mathcal{C}(\iota.f) \equiv \mathcal{C}.\mathsf{heap}(\iota, f)$, and $\mathcal{C}(\iota.\overline{f}.f') \equiv \mathcal{C}.\mathsf{heap}(\mathcal{C}(\iota.\overline{f}, f'))$*

### 4.1 Capabilities and Accessibility

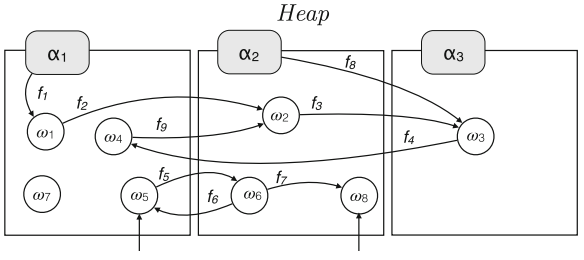
ORCA considers three capabilities:

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

where read allows reading, write allows reading and writing, and tag forbids both read and write, but allows the use of an object's address. To describe the capability at which objects are visible from actors we use the concepts of *static* and *dynamic paths*.

*Static paths* consist of the keyword this (indicating a path starting at the current actor), or the name of a behaviour, $b$, and a variable, $x$, (indicating a path starting at local variable $x$ from a frame of $b$), followed by any number of fields, $f$.

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




*Frames*

$\alpha_1.\mathsf{frame} = \emptyset$

$\alpha_2.\mathsf{frame} = (b', x' \mapsto \omega_8)$

$\alpha_3.\mathsf{frame} = \emptyset$

*Reference Count Tables*

| | $\alpha_1$ | $\alpha_2$ | $\alpha_3$ |
|---|---|---|---|
| $\alpha_1.\mathsf{rc}$: | 21 | 1 | 10 |
| $\alpha_2.\mathsf{rc}$: | 0 | 2 | 20 |
| $\alpha_3.\mathsf{rc}$: | 20 | 0 | 30 |

| $\omega_2$ | $\omega_3$ | $\omega_4$ | $\omega_5$ | $\omega_6$ | $\omega_7$ | $\omega_8$ |
|---|---|---|---|---|---|---|
| 45 | 60 | 10 | 2 | 0 | 50 | 0 |
| 45 | 100 | 0 | 1 | 1 | 0 | 1 |
| 0 | 160 | 10 | 0 | 0 | 0 | 0 |

*Queues*

$\alpha_1.\mathsf{qu} = \mathsf{app}(b, x \mapsto \omega_5)$

$\alpha_2.\mathsf{qu} = \emptyset$

$\alpha_3.\mathsf{qu} = \emptyset :: \mathsf{orca}(\omega_7, -50)$

**Fig. 4.** Configuration $\mathcal{C}_0$. $\omega_1$ is absent in the ref. counts, it has not been shared.

The host language must assign these capabilities to static paths. Thus, we assume it provides a static judgement of the form

$$A \vdash sp : \kappa \qquad \text{where } A \in \mathit{ClassId}$$

meaning that a static path $sp$ has capability *capability* when "seen" from a class $A$. We highlight static judgments, *i.e.*, those provided by the type system in blue.

We expect the type system to guarantee that read and write access rights are "deep", meaning that all paths to a read capability must go through other read or write capabilities (**A1**), and all paths to a write capability must go through write capabilities (**A2**).

**Axiom 1** *For class identifier $A$, static path $sp$, field $f$, capability $\kappa$, we assume:*

***A1*** $A \vdash sp.f : \kappa \longrightarrow \exists \kappa' \neq \mathsf{tag}.\ A \vdash sp : \kappa'.$

***A2*** $A \vdash sp.f : \mathsf{write} \longrightarrow A \vdash sp : \mathsf{write}.$

Such requirements are satisfied by many type systems with read-only references or immutability (*e.g.* [7,11,18,23,29,33,37,41]). An implication of **A1** and **A2** is that capabilities degrade with growing paths, *i.e.*, the prefix of a path has more rights than its extensions. More precisely: $A \vdash sp : \kappa$ and $A \vdash sp.f : \kappa'$ imply that $\kappa \leq \kappa'$, where we define $\mathsf{write} < \mathsf{read} < \mathsf{tag}$, and $\kappa \leq \kappa'$ *iff* $\kappa = \kappa'$ or $\kappa < \kappa'$.

**Example:** Table 1 shows capabilities for some paths from Fig. 4. Thus, $A_1 \vdash \mathsf{this}.f_1 : \mathsf{write}$, and $A_2 \vdash b'.x' : \mathsf{write}$, and $A_2 \vdash \mathsf{this}.f_8 : \mathsf{tag}$. The latter, together with **A1** gives that $A_2 \nvdash \mathsf{this}.f_8.f : \kappa$ for all $\kappa$ and $f$.

As we shall see later, the existence of a path does not imply that the path may be navigated. For example, $\mathcal{C}_0(\alpha_2.f_8.f_4) = \omega_4$, but actor $\alpha_2$ cannot access $\omega_4$ because of $A_2 \vdash \mathsf{this}.f_8 : \mathsf{tag}$.

Moreover, it is possible for a path to have a capability, while not being defined. For example, Table 1 shows $A_1 \vdash \mathsf{this}.f_1.f_2 : \mathsf{write}$ and it would be possible to have $\mathcal{C}_i(\alpha_1.f_1) = \mathsf{null}$, for some configuration $\mathcal{C}_i$ that derives from $\mathcal{C}_0$.

**Table 1.** Capabilities for paths, where $A_1 = \mathcal{C}lass(\alpha_1)$ and $A_2 = \mathcal{C}lass(\alpha_2)$.

| ClassId | Path | Capability |
|---|---|---|
| $A_1$ | $\mathsf{this}.f_1$ | write |
| | $\mathsf{this}.f_1.f_2$ | write |
| | $\mathsf{this}.f_1.f_2.f_3$ | write |
| | $\mathsf{this}.f_1.f_2.f_3.f_4$ | tag |
| | $b.x$ | write |
| | $b.x.f_5$ | write |
| | $b.x.f_5.f_7$ | tag |
| | $b.x.f_5.f_6$ | write |

| ClassId | Path | Capability |
|---|---|---|
| $A_2$ | $\mathsf{this}.f_8$ | tag |
| | $b'.x'$ | write |

*Dynamic paths* (in short paths $p$) start at the actor's fields, or frame, or at some pending message in an actor's queue (the latter cannot be navigated yet, but will be able to be navigated later on when the message is taken off the queue). Dynamic paths may be local paths ($lp$) or message paths. Local paths consist of this or a variable $x$ followed by any number of fields $f$. In such paths, this is the current actor, and $x$ is a local variable from the current frame. Message paths consist of $k.x$ followed by a sequence of fields. If $k \geq 0$, then $k.x$ indicates the local variable $x$ from the $k$-th message from the queue; $k = -1$ indicates variables from either (a) a message that has been popped from the queue, but whose frame has not yet been pushed onto the stack, or (b) a message whose frame has been created but not yet been pushed onto the queue. Thus, $k = -1$ indicates that either (a) a frame will be pushed onto the stack, during message receiving, or (b) a message will be pushed onto the queue during message sending.

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

We define accessibility as the lookup of a path provided that the capability for this path is defined. The *partial* function $\mathcal{A}$ returns a pair: the address accessible from actor $\alpha$ following path $p$, and the capability of $\alpha$ on $p$. A path of the form $p.\mathsf{owner}$ returns the owner of the object accessible though $p$ and capability tag.

**Definition 3 (accessibility).** *The partial function*
$\mathcal{A} : Config \times ActorAddr \times Path \rightarrow (Addr \times Capability)$
*is defined as*

$$\begin{array}{lll}
\mathcal{A}_{\mathcal{C}}(\alpha, \mathsf{this}.\overline{f}) = (\iota, \kappa) & \textit{iff} & \mathcal{C}(\alpha.\overline{f}) = \iota \ \wedge \ \mathcal{C}lass(\alpha) \vdash \mathsf{this}.\overline{f} : \kappa \\
\mathcal{A}_{\mathcal{C}}(\alpha, x.\overline{f}) = (\iota, \kappa) & \textit{iff} & \exists b.\psi.\ [\ \alpha.\mathsf{frame}_{\mathcal{C}} = (b, \psi) \ \wedge \ \mathcal{C}(\psi(x).\overline{f}) = \iota \\
 & & \quad \wedge \ \mathcal{C}lass(\alpha) \vdash b.x.\overline{f} : \kappa\ ] \\
\mathcal{A}_{\mathcal{C}}(\alpha, k.x.\overline{f}) = (\iota, \kappa) & \textit{iff} & k \geq 0 \ \wedge \ \exists b.\psi.\ [\ \alpha.\mathsf{qu}_{\mathcal{C}}[k] = app(b, \psi) \ \wedge \\
 & & \quad \mathcal{C}(\psi(x).\overline{f}) = \iota \ \wedge \ \mathcal{C}lass(\alpha) \vdash b.x.\overline{f} : \kappa\ ] \\
\mathcal{A}_{\mathcal{C}}(\alpha, -1.x.\overline{f}) = (\iota, \kappa) & \textit{iff} & \alpha \textit{ is executing Sending or Receiving, and ...} \\
 & & \textit{continued in Definition 9.} \\
\mathcal{A}_{\mathcal{C}}(\alpha, p.\mathsf{owner}) = (\alpha', \mathsf{tag}) & \textit{iff} & \exists \iota.[\mathcal{A}_{\mathcal{C}}(\alpha, p) = (\iota, _) \wedge \mathcal{O}(\iota) = \alpha'\,]
\end{array}$$

We use $\mathcal{A}_{\mathcal{C}}(\alpha, p) = \iota$ as shorthand for $\exists \kappa.\, \mathcal{A}_{\mathcal{C}}(\alpha, p) = (\iota, \kappa)$. The second and third case above ensure that the capability of a message path is the same as when the message has been taken off the queue and placed on the frame.

**Example:** We obtain that $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, \mathsf{this}.f_1.f_2.f_3) = (\omega_3, \mathsf{write})$, from the fact that Fig. 4 says that $\mathcal{C}_0(\alpha_1.f_1.f_2.f_3) = \omega_3$ and from the fact that Table 1 says that $A_1 \vdash \mathsf{this}.f_1.f_2.f_3 : \mathsf{write}$. Similarly, $\mathcal{A}_{\mathcal{C}_0}(\alpha_2, \mathsf{this}.f_8) = (\omega_3, \mathsf{tag})$, and $\mathcal{A}_{\mathcal{C}_0}(\alpha_2, x') = (\omega_8, \mathsf{write})$, and $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, 0.x.f_5.f_7) = (\omega_8, \mathsf{tag})$.

Both $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, \mathsf{this}.f_1.f_2.f_3)$, and $\mathcal{A}_{\mathcal{C}_0}(\alpha_2, \mathsf{this}.f_8)$ describe paths from actors' fields, while $\mathcal{A}_{\mathcal{C}_0}(\alpha_2, x')$ describes a path from the actor's frame, and finally $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, 0.x.f_5.f_7)$ is a path from the message queue.

Accessibility describes what may be read or written to: $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, \mathsf{this}.f_1.f_2.f_3) = (\omega_3, \mathsf{write})$, therefore actor $\alpha_1$ may mutate object $\omega_3$. However, this mutation is not

visible by $\alpha_2$, even though $\mathcal{C}_0(\alpha_2.f_8) = \omega_3$, because $\mathcal{A}_{\mathcal{C}_0}(\alpha_2, \mathsf{this}.f_8) = (\omega_3, \mathsf{tag})$, which means that actor $\alpha_2$ has only opaque access to $\omega_3$.

Accessibility plays a role in collection: If the reference $f_3$ were to be dropped it would be safe to collect $\omega_4$; even though there exists a path from $\alpha_2$ to $\omega_4$; object $\omega_4$ is not accessible to $\alpha_2$: the path $\mathsf{this}.f_8.f_4$ leads to $\omega_4$ but will never be navigated ($\mathcal{A}_{\mathcal{C}_0}(\alpha_2, \mathsf{this}.f_8.f_4)$ is undefined). Also, $\mathcal{A}_{\mathcal{C}}(\alpha_2, \mathsf{this}.f_8.\mathsf{owner}) = (\alpha_3, \mathsf{tag})$; thus, as long as $\omega_4$ is accessible from some actor, *e.g.* through $\mathcal{C}(\alpha_2.f_8) = \omega_4$, actor $\alpha_3$ will not be collected.

Because the class of an actor as well as the capability attached to a static path are constant throughout program execution, the capabilities of paths starting from an actor's fields or from the same frame are also constant.

**Lemma 1.** *For actor $\alpha$, fields $\overline{f}$, behaviour $b$, variable $x$, fields $\overline{f}$, capabilities $\kappa$, $\kappa'$, configurations $\mathcal{C}$ and $\mathcal{C}'$, such that $\mathcal{C}$ reduces to $\mathcal{C}'$ in one or more steps:*

- $\mathcal{A}_{\mathcal{C}}(\alpha, \mathsf{this}.\overline{f}) = (\iota, \kappa) \quad \wedge \quad \mathcal{A}_{\mathcal{C}'}(\alpha, \mathsf{this}.\overline{f}) = (\iota', \kappa') \quad \longrightarrow \quad \kappa = \kappa'$
- $\mathcal{A}_{\mathcal{C}}(\alpha, x.\overline{f}) = (\iota, \kappa) \quad \wedge \quad \mathcal{A}_{\mathcal{C}'}(\alpha, x.\overline{f}) = (\iota', \kappa') \quad \wedge$
  $\alpha.\mathsf{frame}_{\mathcal{C}} = (b, \_) \quad \wedge \quad \alpha.\mathsf{frame}_{\mathcal{C}'} = (b, \_) \quad \longrightarrow \quad \kappa = \kappa'$

### 4.2 Well-Formed Configurations

We characterise data-race free configurations ($\vDash \mathcal{C} \diamondsuit$):

**Definition 4 (Data-race freedom).** $\vDash \mathcal{C} \diamondsuit$ ***iff***
$\forall \alpha, \alpha', p, p', \kappa, \kappa'.$
$\alpha \neq \alpha' \quad \wedge \quad \mathcal{A}_{\mathcal{C}}(\alpha, p) = (\iota, \kappa) \quad \wedge \quad \mathcal{A}_{\mathcal{C}}(\alpha', p') = (\iota, \kappa')$
$\longrightarrow$
$\kappa \sim \kappa'$
*where we define*
$\kappa \sim \kappa'$ *iff* $[\ (\kappa = \mathsf{write} \longrightarrow \kappa' = \mathsf{tag}) \ \wedge \ (\kappa' = \mathsf{write} \longrightarrow \kappa = \mathsf{tag})\ ]$

This definition captures invariant $\mathbf{I_1}$. The remaining invariants depend on the four derived counters introduced in Sect. 3. Here we define LRC and FRC, and give a preliminary definition of AMC and OMC.

**Definition 5 (Derived counters—preliminary for** AMC **and***ss* OMC**)**

$$\mathrm{LRC}_{\mathcal{C}}(\iota) \equiv \mathcal{O}(\iota).\mathsf{rc}_{\mathcal{C}}(\iota)$$

$$\mathrm{FRC}_{\mathcal{C}}(\iota) \equiv \sum_{\alpha \neq \mathcal{O}(\iota)} \alpha.\mathsf{rc}_{\mathcal{C}}(\iota)$$

$$\mathrm{OMC}_{\mathcal{C}}(\iota) \equiv \sum_j \begin{cases} z & \text{if } \mathcal{O}(\iota).\mathsf{qu}_{\mathcal{C}}[j] = \mathsf{orca}(\iota : z) \\ 0 & \text{otherwise} \end{cases} + \ldots \textit{c.f. Definition 12}$$

$$\mathrm{AMC}_{\mathcal{C}}(\iota) \equiv \#\{\ (\alpha, k) \mid k > 0 \wedge \exists x.\overline{f}.\mathcal{A}_{\mathcal{C}}(\alpha, k.x.\overline{f}) = \iota\ \} + \ldots \textit{c.f. Definition 12}$$

*where # denotes cardinality.*

For the time being, we will be reading this preliminary definition as if ... stood for 0. This works under the assumption the procedures are atomic. However Sect. 5.3, when we consider fine-grained concurrency, will refine the definition of AMC and OMC so as to also consider whether an actor is currently in the process of sending or receiving a message from which the address is accessible. For the time being, we continue with the preliminary reading.

**Example:** Assuming that in $\mathcal{C}_0$ none of the actors is sending or receiving, we have $\mathrm{LRC}_{\mathcal{C}_0}(\omega_3) = 160$, and $\mathrm{FRC}_{\mathcal{C}_0}(\omega_3) = 160$, and $\mathrm{OMC}_{\mathcal{C}_0}(\omega_3) = 0$, and $\mathrm{AMC}_{\mathcal{C}_0}(\omega_3) = 0$. Moreover, $\mathrm{AMC}_{\mathcal{C}_0}(\omega_6) = \mathrm{AMC}_{\mathcal{C}_0}(\alpha_2) = 1$: neither $\omega_6$ nor $\alpha_2$ are arguments in application messages, but they are indirectly reachable through the first message on $\alpha_1$'s queue.

A well-formed configuration requires: $\mathbf{I_1}$–$\mathbf{I_4}$: introduced in Sect. 3; $\mathbf{I_5}$: the RC's are non-negative; $\mathbf{I_6}$: accessible paths are not dangling; $\mathbf{I_7}$: processing message queues will not turn RC's negative; $\mathbf{I_8}$: actors' contents is in accordance with their state. The latter two will be described in Definition 14.

**Definition 6 (Well-formed configurations—preliminary).** $\vDash \mathcal{C}$, iff *for all* $\alpha$, $\alpha_o$, $\iota$, $\iota'$, $p$, $lp$, *and* $mp$, *such that* $\alpha_o = \mathcal{O}(\iota) \neq \alpha$:

$\mathbf{I_1}$ $\vDash \mathcal{C} \diamondsuit$
$\mathbf{I_2}$ $[\ \mathcal{A}_{\mathcal{C}}(\alpha, p) = \iota \ \vee \mathcal{A}_{\mathcal{C}}(\alpha_o, mp) = \iota\ ] \longrightarrow \mathrm{LRC}_{\mathcal{C}}(\iota) > 0$
$\mathbf{I_3}$ $\mathcal{A}_{\mathcal{C}}(\alpha, lp) = \iota \longrightarrow \alpha.\mathsf{rc}_{\mathcal{C}}(\iota) > 0$
$\mathbf{I_4}$ $\mathrm{LRC}_{\mathcal{C}}(\iota) + \mathrm{OMC}_{\mathcal{C}}(\iota) = \mathrm{FRC}_{\mathcal{C}}(\iota) + \mathrm{AMC}_{\mathcal{C}}(\iota)$
$\mathbf{I_5}$ $\alpha.\mathsf{rc}_{\mathcal{C}}(\iota') \geq 0$
$\mathbf{I_6}$ $\mathcal{A}_{\mathcal{C}}(\alpha, p) = \iota \longrightarrow \mathcal{C}.\mathsf{heap}(\iota) \neq \bot$
$\mathbf{I_7}$, $\mathbf{I_8}$ *description in Definition 14.*

For ease of notation, we take $\mathbf{I_5}$ to mean that if $\alpha.\mathsf{rc}_{\mathcal{C}}(\iota')$ is defined, then it is positive. And we take any undefined entry of $\alpha.\mathsf{rc}_{\mathcal{C}}(\iota)$ to be 0.

### 4.3 Actor States

We now complete the definition of runtime entities (Definition 1), and describe the states of an actor, the worksets, the marks, and program counters. (Definition 7). We distinguish the following states: idle (IDLE), collecting (COLLECT), receiving (RECEIVE), sending a message (SEND), or executing the synchronous part of a behaviour (EXECUTE). We discuss these states in more detail next.

Except for the idle state, IDLE, all states use auxiliary data structures: *worksets*, denoted by $\mathsf{ws}$, which stores a set of addresses; *marks* maps, denoted by $\mathsf{ms}$, from addresses to $\mathsf{R}$ (reachable) or $\mathsf{U}$ (unreachable), and program counters. Frames are relevant when in states EXECUTE, or SEND, and otherwise are assumed to be empty. Worksets are used to store all addresses traced from a message or from the actor itself, and are relevant when in states SEND, or RECEIVE, or COLLECT, and otherwise are empty. Marks are used to calculate reachability and are used in state COLLECT, and are ignored otherwise. The program counters record the instruction an actor will execute next; they range between $\mathsf{4}$ and $\mathsf{27}$ and are ghost state, *i.e.* only used in the proofs.

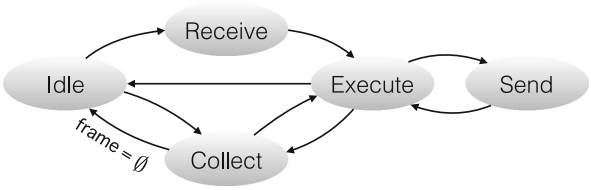



**Fig. 5.** State transitions diagram for an actor.

**Definition 7 (Actor States, Working sets, and Marks)**

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

We write $\alpha.\mathsf{st}_{\mathcal{C}}$, or $\alpha.\mathsf{ws}_{\mathcal{C}}$, or $\alpha.\mathsf{ms}_{\mathcal{C}}$, or $\alpha.\mathsf{pc}_{\mathcal{C}}$ for the state, working set, marks, or the program counter of $\alpha$ in $\mathcal{C}$, respectively.

Actors may transition between states. The state transitions are depicted in Fig. 5. For example, an actor in the idle state (IDLE) may receive an orca message (remaining in the same state), receive an app message (moving to the RECEIVE state), or start garbage collection (moving to the COLLECT state).

In the following sections we describe the actions an actor may perform. Following the style of [17,26,27] we describe actors' actions through pseudo-code procedures, which have the form:

procedure_name$\langle\alpha\rangle$:
$\xrightarrow{}$ condition
{ instructions }

We let $\alpha$ denote the executing actor, and the left-hand side of the arrow describes the condition that must be satisfied in order to execute the instructions on the arrow's right-hand side. Any actor may execute concurrently with other actors. To simplify notation, we assume an implicit, globally accessible configuration $\mathcal{C}$. Thus, instruction $\alpha$.state:=EXECUTE is short for updating the state of $\alpha$ in $\mathcal{C}$ to be EXECUTE. We elide configurations when obvious, *e.g.* $\alpha.\mathsf{frame} = \phi$ is short for requiring that in $\mathcal{C}$ the frame of $\alpha$ is $\phi$, but we mention them when necessary—*e.g.* $\vDash \mathcal{C}[\iota_1, f \mapsto \iota_2] \diamondsuit$ expresses that the configuration that results from updating field $f$ in $\iota_1$ is data-race free.

*Tracing Function.* Both garbage collection, and application message sending/receiving need to find all objects accessible from the current actor and/or from the message arguments. We define two functions: trace_this finds all addresses which are accessible from the current actor, and trace_frame finds all addresses which are accessible through a stack frame (but not from the current actor, this).

```
1  GarbageCollection⟨α⟩:
2    α.st = IDLE ∨ α.st = EXECUTE
3  →
4  {
5    α.st := COLLECT
6    α.ms := ∅
7
8    // marking as unreachable
9    forall ι with α = 𝒪(ι) ∨ α.rc(ι)>0 do α.ms := α.ms[ι ↦ U]
10
11   // tracing and marking locally accessible as reachable
12   forall ι ∈ trace_this(α) ∪ trace_frame(α.frame) do α.ms := α.ms[ι ↦ R]
13
14   // marking owned and globally accessible as reachable
15   forall ι with α = 𝒪(ι) ∧ α.rc(ι)>0 do α.ms := α.ms[ι ↦ R]
16
17   // collecting
18   forall ι with α.ms(ι) = U do
19     if 𝒪(ι) = α then
20       𝒞.heap := 𝒞.heap[ι ↦ ⊥]
21       α.rc := α.rc[ι ↦ ⊥]
22     else
23       𝒪(ι).qu.push(orca(ι:−α.rc(ι)))
24       α.rc := α.rc[ι ↦ ⊥]
25
26   if α.frame=∅ then α.st := IDLE else α.st := EXECUTE
27 }
```

**Fig. 6.** Pseudo-code for garbage collection.

**Definition 8 (Tracing).** *We define the functions*

$\mathsf{trace\_this} : \mathit{Config} \times \mathit{ActorAddr} \to \mathcal{P}(\mathit{Addr})$

$\mathsf{trace\_frame} : \mathit{Config} \times \mathit{ActorAddr} \times \mathit{Frame} \to \mathcal{P}(\mathit{Addr})$

*as follows*

$$\mathsf{trace\_this}_{\mathcal{C}}(\alpha) \equiv \{\iota \mid \exists \overline{f}.\ \mathcal{A}_{\mathcal{C}}(\alpha, \mathsf{this}.\overline{f}) = \iota\}$$

$$\mathsf{trace\_frame}_{\mathcal{C}}(\alpha, \phi) \equiv \{\iota \mid \exists x \in \mathit{dom}(\phi), \overline{f}.\ \mathcal{A}_{\mathcal{C}}(\alpha, x.\overline{f}) = \iota\}$$

### 4.4 Garbage Collection

We describe garbage collection in Fig. 6. An idle, or an executing actor (precondition on line 2) may start collecting at any time. Then, it sets its state to COLLECT (line 5), and initialises the marks, ms, to empty (line 6).

The main idea of ORCA collection is that the requirement for global unreachability of owned objects can be weakened to the local requirement to local unreachability and a $LRC = 0$. Therefore, the actor marks all owned objects, and all addresses with a RC $> 0$ as U (line 9). After that, it traces the actor's fields, and also the actor's frame if it happens not to be empty (as we shall see later, idle actors have empty frames) and marks all accessible addresses as R (line 12). Then, the actor marks all owned objects with RC $> 0$ as R (line 15). Thus we expect that: **(*)** *Any* $\iota$ with $\mathsf{ms}(\iota) = \mathsf{U}$ *is locally unreachable, and if owned by the current actor, then its* LRC *is 0.* For each address with $\mathsf{ms}(\iota) = \mathsf{U}$, if the actor

owns $\iota$, then it collects it (line 20)—this is sound because of $\mathbf{I_2}$, $\mathbf{I_3}$, $\mathbf{I_4}$ and (*). If the actor does not own $\iota$, then it asks $\iota$'s owner to decrement its reference count by the current actor's reference count, and deletes its own reference count to it (thus becoming 0) (line 24)—this preserves $\mathbf{I_2}$, $\mathbf{I_3}$ and $\mathbf{I_4}$.

There is no need for special provision for cycles across actor boundaries. Rather, the corresponding objects will be collected by each actor separately, when it is the particular actor's turn to perform GC.

**Example:** Look at the cycle $\omega_5$–$\omega_6$, and assume that the message $\mathsf{app}(b, \omega_5)$ had finished execution without any heap mutation, and that $\alpha_1.\mathsf{rc}_{\mathcal{C}}(\omega_5) = \alpha_1.\mathsf{rc}_{\mathcal{C}}(\omega_6) = 1 = \alpha_2.\mathsf{rc}_{\mathcal{C}}(\omega_5) = \alpha_2.\mathsf{rc}_{\mathcal{C}}(\omega_6)$—this will be the outcome of the example in Sect. 4.5. Now, the objects $\omega_5$ and $\omega_6$ are globally unreachable. Assume that $\alpha_1$ performs GC: it will *not* be able to collect any of these objects, but it will send a $\mathsf{orca}(\omega_6 : -1)$ to $\alpha_2$. Some time later, $\alpha_2$ will pop this message, and some time later it will enter a GC cycle: it will collect $\omega_6$, and send a $\mathsf{orca}(\omega_5 : -1)$ to $\alpha_1$. When, later on, $\alpha_1$ pops this message, and later enters a GC cycle, it will collect $\omega_5$.

At the end of the GC cycle, the actor sets is state back to what it was before (line 26). If the frame is empty, then the actor had been IDLE, otherwise it had been in state EXECUTE.

### 4.5 Receiving and Sending Messages

Through message send or receive, actors share addresses with other actors. This changes accessibility. Therefore, action is needed to re-establish $\mathbf{I_3}$ and $\mathbf{I_4}$ for all the objects accessible from the message's arguments.

*Receiving application messages* is described by Receiving in Fig. 7. It requires that the actor $\alpha$ is in the IDLE state and has an application message on top of its queue. The actor sets its state to RECEIVE (line 5), traces from the message arguments and stores all accessible addresses into ws (line 7). Since accessibility is not affected by other actors' actions, *c.f., last paragraph in Sect.* 4.6 it is legitimate to consider the calculation of trace_frame as one single step. It then pops the message from its queue (line 8), and thus the AMC for all the addresses in ws will decrease by 1. To preserve $\mathbf{I_4}$, for each $\iota$ in its ws, the actor:

- if it is $\iota$'s owner, then it *decrements* its reference count for $\iota$ by 1, thus decreasing $\mathrm{LRC}_{\mathcal{C}}(\iota)$ (line 12).
- if it is *not* $\iota$'s owner, then it *increments* its reference count for $\iota$ by 1, thus increasing $\mathrm{FRC}_{\mathcal{C}}(\iota)$ (line 14).

After that, the actor sets its frame to that from the message (line 17), and goes to the EXECUTE state (line 18).

**Example:** Actor $\alpha_1$ has an application message in its queue. Assuming that it is IDLE, it may execute Receiving: It will trace $\omega_5$ and as a result store

```
1  Receiving⟨α⟩:
2    α.st = IDLE ∧ α.qu.top() = app(φ)
3  →
4  {
5    α.st := RECEIVE
6
7    α.ws := trace_frame(α, φ)
8    pop(α.qu)
9
10   foreach ι ∈ α.ws do
11     if α = 𝒪(ι) then
12       α.rc(ι) −= 1
13     else
14       α.rc(ι) += 1;
15     α.ws := α.ws \ {ι}
16
17   α.frame := φ
18   α.st := EXECUTE
19 }
```

```
1  ReceiveORCA⟨α⟩:
2    α.state = IDLE ∧ α.qu.top() = ORCA(ι : z)
3  →
4  {
5    α.rc(ι) += z
6    α.qu.pop()
7  }
```

**Fig. 7.** Receiving application and ORCA messages.

$\{\omega_5, \omega_6, \omega_8, \alpha_1, \alpha_2\}$ in its ws. It will then decrement its reference count for $\omega_5$ and $\alpha_1$ (the owned addresses) and increment it for the others. It will then pop the message from its queue, create the appropriate frame, and go to state EXECUTE.

*Receiving ORCA messages* is described in Fig. 7. An actor in the IDLE state with an ORCA message at the top, pops the message from its queue, and adds the value $z$ to the reference count for $\iota$, and stays in the IDLE state.

*Sending application messages* is described in Fig. 8. The actor must be in the EXECUTE state for some behaviour $b$ and must have local variables which can be split into $\psi$ and $\psi'$—the latter will form part of the message to be sent. As the AMC for all the addresses reachable through the message increases by 1, in order to preserve $\mathbf{I_4}$ for each address $\iota$ in ws, the actor:

- increments its reference count for $\iota$ by 1, if it owns it (line 14);
- decrements its reference count for $\iota$ if it does not own it (line 16). But special care is needed if the actor's (foreign) reference count for $\iota$ is 1, because then a simple decrement would break $\mathbf{I_5}$. Instead, the actor set its reference count for $\iota$ by 256 (line 18) and sends an ORCA message to $\iota$'s owner with 256 as argument.

After this, it removes $\psi'$ from its frame (line 22), pushes the message app$(b', \psi')$ onto $\alpha'$'s queue, and transitions to the EXECUTE state.

```
1  Sending⟨α⟩:
2   α.st = EXECUTE   ∧   α.frame = (b, ψ · ψ′)   ∧
3   ∀x ∈ dom(ψ), x′ ∈ dom(ψ′).∀κ, κ′.∀f̄, f̄′. [
4         [   [ A_C(α, x.f̄) = (ι, κ) ∧ A_C(α, x′.f̄′) = (ι, κ′)  ⟶  κ′ ∼ κ ]  ∧
5               [ Class(α) ⊢ b.x′.f̄′ : κ′ ⟷ Class(α′) ⊢ b′.x′.f̄′ : κ′ ]   ]
6  →
7  {
8    α.st := SEND
9
10   α.ws := trace_frame(α, (b, ψ′))
11
12   foreach ι ∈ α.ws do
13     if α = O(ι) then
14       α.rc(ι) += 1
15     elseif α.rc(ι) > 1 then
16       α.rc(ι) −= 1
17     else
18       α.rc(ι) := 256
19       O(ι).qu.push(orca(ι : 256))
20     α.ws := α.ws\{ι}
21
22   α.frame := (b, ψ)
23   α′.qu.push(app(b′, ψ′))
24
25   α.st := EXECUTE
26 }
```

**Fig. 8.** Pseudo-code for message sending.

We now discuss the preconditions. These ensure that sending the message $\mathsf{app}(b, \psi')$ will not introduce data races: Line 4 ensures that there are no data races between paths starting at $\psi$ and paths starting at $\psi'$, while Line 5 ensures that the sender, $\alpha$, and the receiver, $\alpha'$ see all the paths sent, *i.e.* those starting from $(b', \psi')$, at the same capability. We express our expectation that the source language compiler produces code only if it satisfies this property by adding this static requirement as a precondition. These static requirements imply that after the message has been sent, there will be no races between paths starting at the sender's frame and those starting at the last message in the receiver's queue. In more detail, after the sender's frame has been reduced to $(b, \psi)$, and $app(b', \psi')$ has been added to the receiver's queue (at location $k$), we will have a new configuration $\mathcal{C}'{=}\mathcal{C}[\alpha, \mathsf{frame} \mapsto (b, \psi)][\alpha', \mathsf{queue} \mapsto \alpha'.\mathsf{queue}_{\mathcal{C}} :: (b', \psi')]$. In this new configuration lines 4 and 5 ensure that $\mathcal{A}_{\mathcal{C}'}(\alpha, x.\overline{f}) = (\iota, \kappa) \wedge \mathcal{A}_{\mathcal{C}'}(\alpha', k.x'.\overline{f'}) = (\iota, \kappa') \implies \kappa' \sim \kappa$, which means that if there were no data races in $\mathcal{C}$, there will be no data races in $\mathcal{C}'$ either. Formally: $\vDash \mathcal{C} \diamondsuit \implies \vDash \mathcal{C}' \diamondsuit$.

We can now complete Definition 3 for the receiving and the sending cases, to take into account paths that do not exist yet, but which will exist when the message receipt or message sending has been completed.

**Definition 9 (accessibility—receiving and sending).** *Completing Definition 3:* $\mathcal{A}_{\mathcal{C}}(\alpha, -1.x.\overline{f}) = (\iota, \kappa)$ *iff*

$\alpha.\mathsf{st}_{\mathcal{C}} = \mathit{Receiving} \ \wedge \ 9 \leq \alpha.\mathsf{pc}_{\mathcal{C}} < 18 \ \wedge \ \mathcal{C}(\psi(x).\overline{f}) = \iota \ \wedge \ \mathcal{C}lass(\alpha) \vdash b.x.\overline{f} : \kappa$
*where* $(b, \psi)$*is the frame popped at line 8,*

*or*

$\alpha.\mathsf{st}_{\mathcal{C}} = \mathit{Sending} \ \wedge \ \alpha.\mathsf{pc}_{\mathcal{C}} = 23 \ \wedge \ \mathcal{C}(\psi'(x).\overline{f}) = \iota \ \wedge \ \mathcal{C}lass(\alpha') \vdash b'.x.\overline{f} : \kappa$
*where* $\alpha'$ *is the actor to receive the app-message, and* $(b', \psi')$ *is the frame to be sent in line 23.*

**Example:** When actor $\alpha_1$ executes Receiving, and its program counter is between 9 and 18, then $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, -1.x.f_5) = (\omega_6, \mathsf{write})$, even though $x$ is not yet on the stack frame. As soon as the frame is pushed on the stack, and we reach program counter 20, then t $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, -1.x.f_5)$ is undefined, but $\mathcal{A}_{\mathcal{C}_0}(\alpha_1, x.f_5) = (\omega_6, \mathsf{write})$.

### 4.6 Actor Behaviour

As our model is parametric with the host language, we do not aim to describe any of the actions performed while executing behaviours, such as synchronous method calls and pushing frames onto stacks, conditionnals, loops etc. Instead, we concentrate on how behaviour execution may affect GC; this happens only when the heap is mutated either by object creation or by mutation of objects' fields (since this affects accessibility). In particular, our model does not accommodate for recursive calls; we claim that the result from the current model would easily be extended to a model with recursion in synchronous behaviour, but would require a considerable notation overhead.

Figure 9 shows the actions of an actor $\alpha$ while in the EXECUTE state, *i.e.* while it executes behaviours synchronously. The description is nondeterministic: the procedures GoIdle, or Create, or MutateHeap, may execute when the corresponding preconditions hold. Thus, we do not describe the execution of a given program, rather we describe all possible executions for any program. In GoIdle, the actor $\alpha$ simply passes from the execution state to the idle state; the only condition is that its state is EXECUTE (line 2). It deletes the frame, and sets the actor's state to IDLE (line 4). Create creates a new object, initialises its fields to null, and stores its address into local variable $x$.

The most interesting procedure is field assignment, MutateHeap. line 8 modifies the object at address $\iota_1$, reachable through local path $lp1$, and stores in its field $f$ the address $\iota_2$ which was reachable through local path $lp2$. We require that the type system makes the following two guarantees: line 2, second conjunct, requires that $lp1$ should be writable, while line 3 requires that $lp2$ should be accessible. Line 4 and line 5 requite that capabilities of objects do not increase through heap mutation: any address that is accessible with a capability $\kappa$ after the field update was accessible with the same or more permissive capability $\kappa'$ before the field update. This requirment guarantees preservation of data race freedom, *i.e.* that $\vDash \mathcal{C} \diamondsuit$ implies $\vDash \mathcal{C}[\iota_1, f \ \mapsto \ \iota_2] \diamondsuit$.

```
1  GoIdle⟨α⟩:
2    α.st = EXECUTE
3  →
4  { α.frame := ∅; α.st := IDLE; }
5
6  Create⟨α⟩:
7    α.st = EXECUTE   ∧   fresh ω ∧ 𝒪(ω) = α
8  →
9  {
10   heap :=
11     heap[ω ↦ (f1 ↦ null, ..., fn ↦ null)]
12   α.frame := α.frame[x ↦ ω]
13 }
```

```
1  MutateHeap⟨α⟩:
2     α.st = EXECUTE ∧  𝒜_𝒞(α, lp1) = (ι1, write)
3       ∧  𝒜_𝒞(α, lp2) = ι2
4       ∧  ∀ι, κ, lp [ 𝒜_{𝒞[ι1, f↦ι2]}(α, lp) = (ι, κ)   ⟶
5            (∃κ', lp'  𝒜_𝒞(α, lp') = (ι, κ') ∧ κ' ≤ κ ])
6  →
7  {
8       heap := heap[ι1, f ↦ ι2]
9  }
```

**Fig. 9.** Pseudo-code for synchronous operations.

*Heap Mutation Does not Affect Accessibility in Other Actors.* Heap mutation either creates new objects, which will not be accessible to other actors, or modifies objects to which the current actor has write access. By $\vDash \mathcal{C} \diamondsuit$ all other actors have only tag access to the modified object. Therefore, because of *capabilities' degradation with growing paths (as in* **A1** *and* **A2***)*, no other actor will be able to access objects reachable through paths that go through the modified object.

## 5 Soundness and Completeness

In this section we show soundness and completeness of ORCA.

### 5.1 $\mathbf{I_1}$ and $\mathbf{I_2}$ Support Safe Local GC

As we said earlier, $\mathbf{I_1}$ and $\mathbf{I_2}$ support safe local GC. Namely, $\mathbf{I_1}$ guarantees that as long as GC only traces objects to which the actor has read or write access, there will be no data races with other actors' behaviour or GC. And $\mathbf{I_2}$ guarantees that collection can take place based on local information only:

**Definition 10.** *For a configuration $\mathcal{C}$, and object address $\omega$ we say that*

- $\omega$ *is* globally inaccessible *in* $\mathcal{C}$, *iff* $\forall \alpha, p.\, \mathcal{A}_{\mathcal{C}}(\alpha, p) \neq \omega$
- $\omega$ *is* collectable, *iff* $\mathrm{LRC}_{\mathcal{C}}(\omega) = 0$, *and* $\forall lp.\ \mathcal{A}_{\mathcal{C}}(\mathcal{O}(\omega), lp) \neq \omega$.

**Lemma 2.** *If* $\mathbf{I_2}$ *holds, then every collectable object is globally inaccessible.*

### 5.2 Completeness

In [16] we show that globally inaccessible objects remain so, and that for any globally inaccessible object there exists a sequence of steps which will collect it.

**Theorem 1 (Inaccessibility is monotonic).** *For any configurations $\mathcal{C}$, and $\mathcal{C}'$, if $\mathcal{C}'$ is the outcome of the execution of any single line of code from any of the procedures from Figs. 6, 7, 8 and 9, and $\omega$ is globally inaccessible in $\mathcal{C}$, then $\omega$ is globally inaccessible in $\mathcal{C}'$.*

**Theorem 2 (Completeness of ORCA).** *For any configuration $\mathcal{C}$, and object address $\omega$ which is globally inaccessible in $\mathcal{C}$, there exists a finite sequence of steps which lead to $\mathcal{C}'$ in which $\omega \notin dom(\mathcal{C}')$.*

### 5.3 Dealing with Fine-Grained Concurrency

So far, we have discussed actions under an assumption atomicity. However, ORCA needs to work under fine-grained concurrency, whereby several actors may be executing concurrently, each of them executing a behaviour, or sending or receiving a message, or collecting garbage. With fine-grained concurrency, and with the preliminary definitions of AMC and OMC, the invariants are no longer preserved. In fact, they need never hold!

**Example:** Consider Fig. 4, and assume that actor $\alpha_1$ was executing Receiving. Then, at line 7 and before popping the message off the queue, we have $\mathrm{LRC}(\omega_5) = 2$, $\mathrm{FRC}(\omega_5) = 1$, $\mathrm{AMC}^p(\omega_5) = 1$, where $\mathrm{AMC}^p(\_)$ stands for the preliminary definition of AMC; thus $\mathbf{I_4}$ holds. After popping and before updating the RC for $\omega_5$, *i.e.* between lines 9 and 11, we have $\mathrm{AMC}^p(\omega_5) = 0$—thus $\mathbf{I_4}$ is broken. At first sight, this might not seem a big problem, because the update of RC at line 12 will set $\mathrm{LRC}(\omega_5) = 1$, and thus restore $\mathbf{I_4}$. However, if there was another message containing $\omega_5$ in $\alpha_2$'s queue, and consider a snapshot where $\alpha_2$ had just finished line 8 and $\alpha_1$ had just finished line 12, then the update of $\alpha_1$'s RC will *not* restore $\mathbf{I_4}$.

The reason for this problem is, that with the preliminary definition $\mathrm{AMC}^p(\_)$, upon popping at line 8, the AMC is decremented in one atomic step for all objects accessible from the message, while the RC is updated later on (at line 12 or line 14), and one object at a time. In other words, the updates to AMC and LRC are not in sync. Instead, we give the full definition of AMC so, that AMC is in sync LRC; namely it is not affected by popping the message, and is reduced one object at a time once we reach program counter line 15. Similarly, because updating the RC's takes place in a separate step from the removal of the ORCA-message from its queue, we refine the definition of OMC:

**Definition 11 (Auxiliary Counters for AMC, and OMC)**

$$\mathrm{AMC}^{rcv}_{\mathcal{C}}(\iota) \equiv \#\{\alpha \mid \alpha.\mathsf{st}_{\mathcal{C}} = \mathit{RECEIVE} \wedge 9 \leq \alpha.\mathsf{pc}_{\mathcal{C}} \wedge \iota \in \alpha.\mathsf{ws} \backslash \mathcal{C}\mathit{urrAddrRcv}_{\mathcal{C}}(\alpha)\}$$

$$\mathcal{C}\mathit{urrAddrRcv}_{\mathcal{C}}(\alpha) \equiv \begin{cases} \{\iota_{10}\} & \text{if } \alpha.\mathsf{pc}_{\mathcal{C}} = 15 \\ \emptyset & \text{otherwise} \end{cases}$$

*In the above* $\alpha.\mathsf{ws}$ *refers to the contents of the variable* ws *while the actor* $\alpha$ *is executing the pseudocode from Receiving, and* $\iota_{10}$ *refers to the contents of the variable* $\iota$ *arbitrarily chosen in line 10 of the code.*

*We define* $\mathrm{AMC}^{snd}_{\mathcal{C}}(\iota)$, $\mathrm{OMC}^{rcv}_{\mathcal{C}}(\iota)$, *and* $\mathrm{OMC}^{snd}_{\mathcal{C}}(\iota)$ *similarly in [16].*

The counters $\mathrm{AMC}^{\mathsf{rcv}}$ and $\mathrm{AMC}^{\mathsf{snd}}$ are zero except for actors which are in the process of receiving or sending application messages. Also, the counters $\mathrm{OMC}^{\mathsf{rcv}}$ and $\mathrm{AMC}^{\mathsf{snd}}$ are zero except for actors which are in the process of receiving or sending ORCA-messages. All these counters are always $\geq 0$. We can now complete the definition of AMC and OMC:

**Definition 12 (AMC and OMC – full definition)**

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

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

*where # denotes cardinality.*

**Example:** Let us again consider that $\alpha_1$ was executing Receiving. Then, at line 10 we have $\mathsf{ws} = \{\iota_5, \iota_6\}$ and $\mathrm{AMC}(\omega_5) = 1 = \mathrm{AMC}(\omega_6)$. Assume at the first iteration, at line 10 we chose $\iota_5$, then right before reaching line 15 we have $\mathrm{AMC}(\omega_5) = 0$ and $\mathrm{AMC}(\omega_6) = 1$. At the second iteration, at line 10 we will chose $\iota_6$, and then right before reaching 15 we have $\mathrm{AMC}(\omega_6) = 0$.

### 5.4 Soundness

To complete the definition of well-formed configurations, we need to define what it means for an actor or a queue to be well-formed.

**Well-Formed Queues - $\mathbf{I_7}$.** The owner's reference count for any live address (*i.e.* any address reachable from a message path, or foreign actor, or in an ORCA message) should be greater than 0 at the current configuration, as well as, at all configurations which arise from receiving pending, but no new, messages from the owner's queue. Thus, in order to ensure that ORCA decrement messages do not make the local reference count negative, $\mathbf{I_7}$ requires that the effect of any prefix of the message queue leaves the reference count for any object positive. To formulate $\mathbf{I_7}$ we use the concept of $\mathit{QueueEffect}_{\mathcal{C}}(\alpha, \iota, n)$, which describes the contents of $LRC$ after the actor $\alpha$ has consumed and reacted to the first $n$ messages in its queue—*i.e.* is about "looking into the future". Thus, for actor $\alpha$, address $\iota$, and number $n$ we define the effect of the $n$-prefix of the queue on the reference count as follows:

$$\mathit{QueueEffect}_{\mathcal{C}}(\alpha, \iota, n) \ \equiv\ \mathrm{LRC}_{\mathcal{C}}(\iota) - z + \textstyle\sum_{j=0}^{n} \mathit{Weight}_{\mathcal{C}}(\alpha, \iota, j)$$

where $z = k$, if $\alpha$ is in the process of executing ReceiveORCA, and $\alpha.\mathrm{pc}_{\mathcal{C}} = 6$, and $\alpha.\mathsf{qu.top} = \mathsf{orca}\ (\iota : k)$, and otherwise $z = 0$.

And where,

$$Weight_{\mathcal{C}}(\alpha,\iota,j) \equiv \begin{cases} z' & \text{if } \alpha.\mathsf{qu}_{\mathcal{C}}[j] = \mathsf{orca}(\iota : z') \\ -1 & \text{if } \exists x.\exists \overline{f}.\ \mathcal{A}_{\mathcal{C}}(\alpha, k.x.\overline{f}) = \iota \wedge\ \mathcal{O}(\iota) = \alpha \\ 0 & \text{otherwise} \end{cases}$$

$\mathbf{I_7}$ makes the following four guarantees: **[a]** The effect of any prefix of the message queue leaves the *LRC* non-negative. **[b]** If $\iota$ is accessible from the $j$-th message in its owner's queue, then the *LRC* for $\iota$ will remain $>0$ during execution of the current message queue up to, and including, the $j$-th message. **[c]** If $\iota$ is accessible from an ORCA-message, then the *LRC* will remain $>0$ during execution of the current message queue, up to and excluding execution of the ORCA-message itself. **[d]** If $\iota$ is globally accessible (*i.e.* reachable from a local path or from a message in a non-owning actor) then $LRC(\iota)$ is currently $>0$, and will remain so after during popping of all the entries in the current queue.

**Definition 13 ($\mathbf{I_7}$).** $\models_{Queues} \mathcal{C}$, *iff for all* $j \in \mathbb{N}$, *for all addresses* $\iota$, *actors* $\alpha$, $\alpha'$, *where* $\mathcal{O}(\iota) = \alpha \neq \alpha'$, *the following conditions hold:*

**a** $\forall n.\ QueueEffect_{\mathcal{C}}(\alpha,\iota,n) \geq 0$
**b** $\exists x.\ \exists \overline{f}.\ \mathcal{A}_{\mathcal{C}}(\alpha, j.x.\overline{f}) = \iota \longrightarrow \forall k \leq j.\ QueueEffect_{\mathcal{C}}(\alpha,\iota,k) > 0.$
**c** $\alpha.\mathsf{qu}_{\mathcal{C}}[j] = \mathsf{orca}(\iota : z) \longrightarrow \forall k < j.\ QueueEffect_{\mathcal{C}}(\alpha,\iota,k) > 0.$
**d** $\exists p.\mathcal{A}_{\mathcal{C}}(\alpha', p) = \iota \longrightarrow \forall k \in \mathbb{N}.\ QueueEffect_{\mathcal{C}}(\alpha,\iota,k) > 0.$

For example, in a configuration with $LRC(\iota) = 2$, and a queue with $\mathsf{orca}(\iota : -2) :: \mathsf{orca}(\iota : -1) :: \mathsf{orca}(\iota : 256)$ is illegal by $\mathbf{I_7}$.**[a]**. Similarly, in a configuration with $LRC(\iota) = 2$, and a queue with $\mathsf{orca}(\iota : -2) :: \mathsf{orca}(\iota : 256)$, the owning actor could collect $\iota$ before popping the message $\mathsf{orca}(\iota : 256)$ from its queue. Such a configuration is also deemed illegal by $\mathbf{I_7}$.**[c]**.

**$\mathbf{I_8}$-Well-Formed Actor.** In [16] we define well-formedness of an actor $\alpha$ through the judgement $\mathcal{C}, \alpha \vdash \mathsf{st}$. This judgement depends on $\alpha$'s current state st, and requires, among other things, that the contents of the local variables ws, ms are consistent with the contents of the pc and RC. Remember also, that because Receiving and Sending modify the ws or send ORCA-messages before updating the frame or sending the application message, in the definition of AMC and OMC we took into account the internal state of actors executing such procedures.

**Well-Formed Configuration.** The following completes Definition 6 from Sect. 4.2.

**Definition 14 (Well-formed configurations—full).** *A configuration* $\mathcal{C}$ *is well-formed,* $\vDash \mathcal{C}$, iff $\mathbf{I_1}$–$\mathbf{I_6}$ *(Definition 6) for* $\mathcal{C}$, *if its queues are well-formed (*$\models_{Queues} \mathcal{C}$, $\mathbf{I_7}$*), as well as, all its actors (*$\mathcal{C}, \alpha \vdash \alpha.\mathsf{st}_{\mathcal{C}}$, $\mathbf{I_8}$*).*

In [16] we consider the execution of each line in the codes from Sect. 4, and prove:

**Theorem 3 (Soundness of ORCA).** *For any configurations* $\mathcal{C}$ *and* $\mathcal{C}'$*: If* $\vDash \mathcal{C}$, *and* $\mathcal{C}'$ *is the outcome of the execution of any single line of code from any of the procedures from Figs. 6, 7, 8 and 9, then* $\vDash \mathcal{C}'$.

This theorem together with $\mathbf{I_6}$ implies that ORCA never leaves accessible paths dangling. Note that the theorem is stated so as to be applicable for a fine interleaving of the execution. Even though we expressed ORCA through procedures, in our proof we cater for an execution where one line of any of these procedures is executed interleaved with any other procedures in the other actors.

## 6 Related Work

The challenges faced when developing and debugging concurrent garbage collectors have motivated the development of formal models and proofs of correctness [6,13,19,30,35]. However, most work considers a global heap where mutator and collector threads *race* for objects and relies on synchronisation mechanisms (or atomic reduction steps), such as read or write barriers, in contrast to ORCA which considers many local heaps, no atomicity or synchronization, and relies on the properties of the type system. McCreight et al. [25] introduced a framework to reason about and build certified garbage collectors, verifying independently both mutator and collector threads. Their work focuses mainly on garbage collectors similar to those that run on Java programs, such as STW mark-and-sweep, STW copying and incremental copying. Vechev et al. [39] specified concurrent mark-and-sweep collectors with write barriers for synchronisation. The authors also present a parametric garbage collector from which other collectors can be derived. Hawblitzel and Petrank [22] mechanized proofs of two real-world collectors (copying and mark-and-sweep) and their respective allocators. The assembly code was instrumented with pre- and post-conditions, invariants and assertions, which were then verified using Z3 and Boogie. Ugawa et al. [38] extended a copying, on-the-fly, concurrent garbage collector to process reference types. The authors model-checked their algorithm using a model that limited the number of objects and threads. Gamie et al. [17] machine-checked a state-of-the-art, on-the-fly, concurrent, mark-and-sweep garbage collector [32]. They modelled one collector thread and many mutator threads. ORCA does not limit the number of actors running concurrently.

Local heaps have been used in the context of garbage collection to reduce the amount of synchronisation required before [1–3,13,15,24,31,34], where different threads have their own heap and share a global heap. However, only two of these have been proved correct. Doligez and Gonthier [13] proved a collector [14] which splits the heap into many local heaps and one global heap, and uses mark-and-sweep for individual collection of local heaps. The algorithm imposes restrictions on the object graph, that is, a thread cannot access objects in other threads' local heaps. ORCA allows for references across heaps. Raghunathan et al. [34] proved correct a hierarchical model of local heaps for functional programming languages. The work restricted objects graphs and prevented mutation.

As for collectors that rely on message passing, Moreau et al. [26] revisited the Birrell's reference listing algorithm, which also uses message passing to update reference counts in a distributed system, and presented its formalisation and proofs or soundness and completeness. Moreover, Clebsch and Drossopoulou [10] proved correct MAC, a concurrent collector for actors.

## 7 Conclusions

We have shown the soundness and completeness of the ORCA actor memory reclamation protocol. The ORCA model is not tied to a particular programming language and is parametric in the host language. Instead it relies on a number of invariants and properties which can be met by a combination of language and static checks. The central property that is required is the absence of data races on objects shared between actors.

We developed a formal model of ORCA and identified requirements for the host language, its type system, or associated tooling. We described ORCA at a language-agnostic level and identified eight invariants that capture how global consistency is obtained in the absence of synchronisation. We proved that ORCA will not prematurely collect objects (soundness) and that all garbage will be identified as such (completeness).

**Acknowledgements.** We are deeply grateful to Tim Wood for extensive discussions and suggestions about effective communication of our ideas. We thank Rakhilya Mekhtieva for her contributions to the formal proofs, Sebastian Blessing and Andy McNeil for their contributions to the implementation, as well as the anonymous reviewers for their insightful comments. This work was initially funded by Causality Ltd, and has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412) and the FP7 project UPSCALE, the Swedish Research council through the grant Structured Aliasing and the UPMARC Linneaus Centre of Excellence, the EPSRC (grant EP/K011715/1), the NSF (award 1544542) and ONR (award 503353).

## References

1. Armstrong, J.: A history of Erlang. In: HOPL III (2007)
2. Auerbach, J., Bacon, D.F., Guerraoui, R., Spring, J.H., Vitek, J.: Flexible task graphs: a unified restricted thread programming model for Java. In: LCTES (2008)
3. Auhagen, S., Bergstrom, L., Fluet, M., Reppy, J.: Garbage collection for multicore NUMA machines. In: MSPC (2011). https://doi.org/10.1145/1988915.1988929
4. Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: a generalisation of uniqueness and read-only. In: ECOOP (2001). https://doi.org/10.1007/3-540-45337-7_2
5. Brandauer, S., et al.: Parallel objects for multicores: a glimpse at the parallel language Encore. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 1–56. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_1
6. Cheng, P.S.D.: Scalable real-time paralllel garbage collection for symmetric multiprocessors. Ph.D. thesis, Carnegie Mellon University (2001)
7. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal ownership for active objects. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 139–154. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_11
8. Clebsch, S.: Pony: co-designing a type system and a runtime. Ph.D. thesis, Imperial College London (2018, to be published)

9. Clebsch, S., Blessing, S., Franco, J., Drossopoulou, S.: Ownership and reference counting based garbage collection in the actor world. In: ICOOOLPS (2015)
10. Clebsch, S., Drossopoulou, S.: Fully concurrent garbage collection of actors on many-core machines. In: OOPSLA (2013). https://doi.org/10.1145/2544173.2509557
11. Clebsch, S., Drossopoulou, S., Blessing, S., McNeil, A.: Deny capabilities for safe, fast actors. In: AGERE! (2015). https://doi.org/10.1145/2824815.2824816
12. Clebsch, S., Franco, J., Drossopoulou, S., Yang, A., Wrigstad, T., Vitek, J.: Orca: GC and type system co-design for actor languages. In: OOPSLA (2017). https://doi.org/10.1145/3133896
13. Doligez, D., Gonthier, G.: Portable, unobtrusive garbage collection for multiprocessor systems. In: POPL (1994). https://doi.org/10.1145/174675.174673
14. Doligez, D., Leroy, X.: A concurrent, generational garbage collector for a multithreaded implementation of ML. In: POPL (1993). https://doi.org/10.1145/158511.158611
15. Domani, T., Goldshtein, G., Kolodner, E.K., Lewis, E., Petrank, E., Sheinwald, D.: Thread-local heaps for Java. In: ISMM (2002). https://doi.org/10.1145/512429.512439
16. Franco, J., Clebsch, S., Drossopoulou, S., Vitek, J., Wrigstad, T.: Soundness of a concurrent collector for actors (extended version). Technical report, Imperial College London (2018). https://www.doc.ic.ac.uk/research/technicalreports/2018/
17. Gamie, P., Hosking, A., Engelhard, K.: Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In: PLDI (2015). https://doi.org/10.1145/2737924.2738006
18. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: OOPSLA (2012). https://doi.org/10.1145/2384616.2384619
19. Gries, D.: An exercise in proving parallel programs correct. Commun. ACM **20**(12), 921–930 (1977). https://doi.org/10.1145/359897.359903
20. Haller, P., Loiko, A.: LaCasa: lightweight affinity and object capabilities in Scala. In: OOPSLA (2016). https://doi.org/10.1145/2983990.2984042
21. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_17
22. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. In: POPL (2009). https://doi.org/10.1145/1480881.1480935
23. Kniesel, G., Theisen, D.: JAC-access right based encapsulation for Java. Softw. Pract. Exp. **31**(6), 555–576 (2001)
24. Marlow, S., Peyton Jones, S.: Multicore garbage collection with local heaps. In: ISMM (2011). https://doi.org/10.1145/1993478.1993482
25. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: PLDI (2007). https://doi.org/10.1145/1250734.1250788
26. Moreau, L., Dickman, P., Jones, R.: Birrell's distributed reference listing revisited. ACM Trans. Program. Lang. Syst. (TOPLAS) **27**(6), 1344–1395 (2005). https://doi.org/10.1145/1108970.1108976
27. Moreau, L., Duprat, J.: A construction of distributed reference counting. Acta Informatica **37**(8), 563–595 (2001). https://doi.org/10.1007/PL00013315
28. Östlund, J.: Language constructs for safe parallel programming on multi-cores. Ph.D. thesis, Department of Information Technology, Uppsala University (2016)

29. Östlund, J., Wrigstad, T., Clarke, D., Åkerblom, B.: Ownership, uniqueness, and immutability. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 178–197. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69824-1_11
30. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**(4), 319–340 (1976)
31. Pizlo, F., Hosking, A.L., Vitek, J.: Hierarchical Real-Time Garbage Collection (2007). https://doi.org/10.1145/1254766.1254784
32. Pizlo, F., Ziarek, L., Maj, P., Hosking, A.L., Blanton, E., Vitek, J.: Schism: fragmentation-tolerant real-time garbage collection. In: PLDI (2010). https://doi.org/10.1145/1806596.1806615
33. Potanin, A., Östlund, J., Zibin, Y., Ernst, M.D.: Immutability. In: Clarke, D., Noble, J., Wrigstad, T. (eds.) Aliasing in Object-Oriented Programming. Types, Analysis and Verification. LNCS, vol. 7850, pp. 233–269. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36946-9_9
34. Raghunathan, R., Muller, S.K., Acar, U.A., Blelloch, G.: Hierarchical memory management for parallel programs. In: ICFP (2016). https://doi.org/10.1145/2951913.2951935
35. Ramesh, S., Mehndiratta, S.: The liveness property of on-the-fly garbage collector - a proof. Inf. Process. Lett. **17**(4), 189–195 (1983)
36. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
37. Tschantz, M.S., Ernst, M.D.: Javari: adding reference immutability to Java. In: OOPSLA (2005). https://doi.org/10.1145/1094811.1094828
38. Ugawa, T., Jones, R.E., Ritson, C.G.: Reference object processing in on-the-fly garbage collection. In: ISMM (2014). https://doi.org/10.1145/2602988.2602991
39. Vechev, M.T., Yahav, E., Bacon, D.F.: Correctness-preserving derivation of concurrent garbage collection algorithms. In: PLDI (2006). https://doi.org/10.1145/1133981.1134022
40. Yang, A.M., Wrigstad, T.: Type-assisted automatic garbage collection for lock-free data structures. In: ISMM (2017). https://doi.org/10.1145/3092255.3092274
41. Zibin, Y., Potanin, A., Li, P., Ali, M., Ernst, M.D.: Ownership and immutability in generic Java. In: OOPSLA (2010). https://doi.org/10.1145/1932682.1869509