Ownership and Reference Counting based Garbage Collection in the Actor World

Sylvan Clebsch  Sebastian Blessing  Juliana Franco  Sophia Drossopoulou
Causality Ltd and Imperial College London

Abstract
We propose Pony-ORCA, a fully concurrent protocol for garbage collection in the actor paradigm. It allows actors to perform garbage collection concurrently with any number of other actors. It does not require any form of synchronization across actors except those introduced through the actor paradigm, i.e. message send and message receive.

Pony-ORCA is based on ideas from ownership and deferred, distributed, weighted reference counting. It adapts the messaging system of actors to keep the reference count consistent.

We introduce a formal model and describe the invariants on which it relies. We illustrate these invariants through an example and sketch how they are maintained. We show some benchmarks and argue that the protocol can be implemented efficiently.

1. Introduction
The actor paradigm [4] was proposed in 1973 by Carl Hewitt [23]. An actor is a computational entity that, in response to a message it receives, can: 1) send a finite number of (asynchronous) messages to other actors; 2) create a finite number of new actors; and 3) designate the code to be executed for the next message it receives. The code executed upon receipt of a message is called a behaviour. The actor paradigm has been adopted in a functional setting, e.g. in Erlang [7], and in the object-oriented paradigm [5, 10, 14, 34].

Implicit garbage collection is crucial for convenience of programming, however automatic garbage collection often proves to be a performance bottleneck, e.g. [12] reports application pauses of around 3 seconds. This has been subject of improvement in other garbage collection protocols, such as the Pauseless GC Algorithm [18] and the Continuously Concurrent Compacting Collector (C4) [35] from Azul Systems, which allow a non-stop-the-world, concurrent and parallel collection by using read barriers. In concurrent GC protocols, it is important to ensure the absence of race conditions—for instance, a race condition during a marking phase can cause an object to be marked for collection when there still exist references to it. This is often solved using synchronization mechanisms, as for instance in some of the JVM collectors [28], which can cause program performance loss.

In this paper, we propose Pony-ORCA ¹, a garbage collection protocol for actor-based object oriented programming languages, which is fully concurrent and where no synchronization mechanisms, such as read and write barriers, are needed, avoiding application latency spikes. We have implemented Pony-ORCA for the language Pony [6], but our protocol is applicable for any language which fulfills the criteria we give later. Our protocol is based on ownership and deferred distributed weighted reference counting.

Ownership types [15, 32] were proposed with the remit to delimit groups of related objects into different areas of the heap. They were used for garbage collection under the requirement that there are no incoming references to these areas [33]. Such a scheme works well in the concurrent setting, but the no-incoming references requirement is often far too strong.

Reference counting garbage collection puts no requirement on the heap structure; instead, it tracks, for each object, a count of the number of references to it held by other objects [8]. This approach has been further developed to detect cycles and to deal with the distributed setting [20, 30]. However, the approach has poor locality and thus, in the concurrent setting, it requires synchronization across the various threads [26].

We employ the locality found in actors, and the implicit synchronization afforded by the actor messaging system, to develop a fully concurrent garbage collection algorithm. Pony-ORCA allows the fully concurrent garbage collection of objects as well as actors. In particular:

• An actor may perform garbage collection concurrently with other actors while they are executing any kind of behaviour.
• An actor may decide whether to garbage collect an object solely based on its own local state, without consultation with, or inspecting the state of, any other actor.
• No synchronization between actors is required during garbage collection, other than potential message sends.
• An actor may garbage collect between its normal behaviours, i.e. it need not wait until its message queue is empty.
• Pony-ORCA can be applied to several programming languages, provided that they satisfy the following two requirements:
  • Actor behaviours are atomic.
  • Message delivery is causal—i.e. messages arrive before any messages they may have caused, if they have the same destination.

Our approach is based on implicit ownership, whereby an actor owns any object that it has allocated. Each actor is responsible for garbage collecting the objects it owns. The challenge is that an actor

¹ ORCA stands for Ownership Reference Counting for Actors and it is a reference to the killer whale.
may have no path to an object it owns, while other actors may still have paths to that object, or the object may appear in messages in some other actor’s message queue. It would be erroneous to garbage collect such an object.

In our approach, an actor maintains a reference count for all the objects it owns. The reference count is guaranteed to be non-zero for any object which is accessible from any message or any actor other than its owner. Thus, an actor can safely garbage collect any object which is locally unreachable, and whose reference count is zero.

There is the remaining challenge of ensuring that the reference count indeed is non-zero for objects which are accessible from actors other than their owner. This is maintained through a system whereby all the actors which may reach an object they do not own also maintain their (foreign) reference count, and whereby the sum of all foreign reference counts of an object corresponds to its owning actor’s (local) reference count for that object. This requirement needs to be modified to take into account the messages in the various queues. Therefore, in Pony-ORCA, when an actor receives or sends references to objects, it may need to inform the owning actor through protocol-specific messages.

Our paper is organized as follows: in Section 2 we present related work on garbage collection protocols; in Section 3 we briefly introduce the Pony language; in Section 4 we define the runtime configuration and what is a well-formed configuration, in the Pony-ORCA protocol; in Section 5 we argue why objects can be safely collected and how the consistency of the runtime configuration is maintained; we discuss our results in Section 6; and we finish the paper with conclusions in Section 7.

2. Garbage collection in the actor world

Previously, actor-model languages and libraries have used five different approaches to garbage collection. The first is to combine manual termination of actors with a standard tracing garbage collector. These are primarily JVM based implementations, such as Scala [22], Akka [5], Kilim [34], AmbientTalk [36], and SALSA 2.0 [37]. The second is to combine manual termination with a per-actor tracing collector, using copying of messages to move data into actor heaps, that also has some global data. These are primarily BEAM based implementations, such as Erlang [7] and Elixir [21]. The third is to transform the actor graph into an algebraic data type, in effect removing the actor’s message passing semantics. This is done in ActorFoundry [3]. The fourth is to use reference listing and snapshots to collect actors, as done in SALSA 1.0 [38]. The fifth is to use a message-based actor collection protocol that can collect actors and detect termination fully concurrently, without a stop-the-world step [17].

In this work, we extend message-based actor collection [17], applying it to passive object collection when objects are shared by reference amongst actors.

Our work draws heavily on existing distributed garbage collection algorithms, particularly on distributed reference counting [19, 20, 26, 29–31]. A key difference is that Pony-ORCA does not have reference count cycles amongst objects, since only actors hold reference counts for objects and those counts are independent of the number of paths to an object in an actor’s reachable heap. Thus, we do not require cycle detection [8, 19, 20, 26, 29] or reference listing [31]. This approach also eliminates heap change related reference count changes, which in effect gives highly deferred reference counting [9].

Pony-ORCA is also influenced by the Emerald garbage collector [27], particularly in the design goals of comprehensiveness, concurrency, expediency, efficiency, and correctness. In addition, independently collected actor heaps are similar to Emerald’s local collector, with our message protocol effectively replacing the global collector.

3. The language Pony and Causal Message Delivery

The language Pony supports actors (active objects), and objects (passive objects). Objects and actors have fields and synchronous methods; in addition, actors have asynchronous methods, called behaviours. Actors may receive asynchronous messages which contain any number of parameters. These may be addresses or literals (e.g. integers). The messages are stored in a queue. Whenever an actor is enabled, it removes the top message from its queue and executes the body of the corresponding behaviour. Actors and objects may call synchronous methods on objects and asynchronous behaviours on other actors. Pony also contains further features: traits, algebraic data types, generics etc., which, however, will not be discussed here. A formal semantics of the Pony subset discussed here appears in [6].

Causal message delivery requires that whenever a message, \( \text{msg}_1 \), is a direct or indirect cause of another message, \( \text{msg}_2 \), and the destination of the two messages is the same, \( \text{msg}_1 \) will be delivered before \( \text{msg}_2 \). A message \( \text{msg} \) is a cause of a further message \( \text{msg}' \), if a) an actor sends \( \text{msg}' \) after receiving \( \text{msg} \), or b) an actor sends \( \text{msg}' \) after sending \( \text{msg} \), or c) there exists an intermediate message \( \text{msg}'' \) such that \( \text{msg} \) is a cause of \( \text{msg}'' \), and \( \text{msg}' \) is a cause of \( \text{msg}'' \). Therefore, the causal relationship is asymmetric, acyclic, and transitive.

For example, if actor \( A \) sends message \( \text{msg}_1 \) to actor \( B \), and then sends message \( \text{msg}_2 \) to actor \( C \), and actor \( C \) sends message \( \text{msg}_3 \) to actor \( B \) after receiving \( \text{msg}_2 \), then \( \text{msg}_1 \) is a cause of \( \text{msg}_2 \), and also \( \text{msg}_2 \) is a cause of \( \text{msg}_1 \), and by transitivity \( \text{msg}_2 \) is a cause of \( \text{msg}_3 \). Therefore, causal message delivery requires that actor \( B \) will receive \( \text{msg}_3 \) before receiving \( \text{msg}_2 \).

Causal message delivery is not required in the original formulation of the actor paradigm [23], which mandates that message delivery is guaranteed, but need not be ordered. The motivation for this is to make the actor-model as general as possible. For the same reason, the original formulation does not require buffered queues. However, in [17] it is shown how causal messaging can be implemented efficiently in the concurrent setting, and in [11] we have implemented it in the distributed setting.

Crucially, causal message delivery has been of paramount importance in the development of the actor collection protocol [17], and of various distributed protocols developed in [11]. We plan to discuss efficient implementation of causal messaging in further work.

4. The Pony-ORCA Garbage Collection Protocol

Pony-ORCA is based on a reference counting scheme, whereby each actor keeps a reference count for some other objects or actors. An actor can decide whether to garbage collect an object it owns solely on the basis of whether the object is reachable from the owning actor fields and whether the object’s (local) reference count is zero. For this to work, the owner’s reference count for an object to a true reflection of the global configuration, namely the owner’s (local) reference count after receiving all the pending ORCA protocol specific messages must be the same as the sum of the (foreign) reference counts in the non-owning actors together with pending Pony-level application messages. In order to maintain this invariant, whenever objects are sent, received, or become unreachable, the reference count will need to be modified and/or ORCA-specific messages sent to the owners of such objects, thus resulting in the updating of local reference counts of these objects. Note, that an actor does not need to keep a reference count for all
other objects or actors. We will discuss the size of the counter tables in Section 4.2.

In this section we describe our protocol in more detail, and develop a formal model. In section 4.1 we show diagrammatically some actors, heaps and queues, and discuss which objects are globally unreachable. In section 4.2 we show Pony-ORCA specific data structures. In section 4.3 we define what it means for the owner’s reference count to be a true reflection of the global configuration along with further necessary well-formedness conditions.

4.1 Actors and reachable objects

In Pony, actors can create new objects, send messages to other actors (with references to actors and objects, not necessarily allocated by the sending actor) and receive messages. As we said earlier, it is possible for an actor to own an object without having a reference to it, while non-owning actors do have references to it—for instance, an actor may create an object, send it to other actors and then drop the reference to that object. Given this, the protocol needs to ensure that an object, even though is not reachable from its owner, is not garbage collected if there exists another actor or a message in one of the actor’s queues with a reference to it.

Example 1. Consider the Ownership and References diagram from Figure 1. We have actors \(a_1, a_2\) and \(a_3\) and show them in rounded double boxes. We have objects \(ω_1,ω_2,..,ω_8\) and we show ownership through square boxes, e.g. \(a_3\) owns \(ω_7\). We show references through arrows, e.g. \(ω_3\) references \(ω_1\). Notice that object \(ω_3\) is not reachable from \(a_2\), its owner, but it is reachable from \(a_3\).

4.2 Runtime configuration modelling Pony-ORCA

We now model the data structures used in Pony-ORCA, as well as the Pony runtime entities relevant to the soundness of our protocol. In Pony-ORCA each actor contains a (local) reference count for any object it owns, as well as a (foreign) reference count for any other actor or object it does not own. We represent ownership through the mapping \(Owner\), and we unify local and foreign reference counts to one mapping, \(RC\).

We consider sets of addresses, \(S_{all}\), and distinguish between object addresses, \(S_{obj}\), and actor addresses, \(S_{act}\). Every actor or object has an owner, which is an actor. We require that the owner of an actor is the actor itself. We indicate addresses through \(i, j\), etc., actor addresses through \(a, a'\), etc., and object addresses through \(ω, ω'\), etc.

Definition 1 (Addresses and Owners). We require enumerable sets \(S_{all}\), \(S_{obj}\), and \(S_{act}\), and a function

\[Owner : S_{all} \rightarrow S_{act}\]

such that

\[S_{all} = S_{obj} \cup S_{act}\]

\[i \in S_{all}\]

\[a \in S_{act}\]

\[ω \in S_{obj}\]

and

\[∀a \in S_{act}. Owner(a) = a\]

A runtime configuration consists of a per-actor heap, a per-actor queue of messages, and a per-actor counter table. In order to argue soundness we need to model the heap. We do not need to distinguish the class of objects, or the contents of their fields. All we need to model is the set of addresses which are reachable from a given address. In the current paper we over-approximate this information, and our model only represents the set of addresses reachable from a given actor. In fact, we expect that a valid heap for an actor contains a superset of the addresses reachable from that actor, or from objects in that actor’s heap, but we do not model reachability. We will give a finer grained model in further work.

The message queue is a sequence of messages, where the order matters. Messages are either Pony-level messages, or ORCA-specific messages. Our protocol is not concerned with the exact Pony-level messages sent, but it is concerned with the addresses these may contain. Pony-level messages are \(APP(i^*)\). The ORCA-specific messages \(DEC(i, k)\) and \(INC(i, k)\) require the actor to change its reference count to \(i\) accordingly. The counter table gives the reference count for a given address, reflecting references from other actors or messages in queues.

Definition 2 (Runtime Configurations).

\[K \in \text{RunTimeCfg} = S_{act} \rightarrow (P(S_{all}) \times Msg^* \times (S_{all} \rightarrow Z))\]

\[Msg ::= \text{APP}(i^*) \mid \text{INC}(i, k) \mid \text{DEC}(i, k)\]

\[k \in \mathbb{N}\]

\[q \in Msg^*\]

In the remainder we use the following abbreviations

- \(Heap_K(a) = K(a)_1\)
- \(Queue_K(a) = K(a)_2\)
- \(Message_K(a, j) = K(a)_2(j)\)
- \(RC_K(a, i) = K(a)_3(i)\)

And we require the following well-formedness conditions:

WF1. \(RC_K(a, i) \geq 0\)

WF2. \(ω \in Heap_K(α) \implies Owner(ω) \in Heap_K(α)\)

In our model, the counter table is a total function from actor-address pairs to integers. We do that for the sake of simplicity. The implementation is more efficient than that: an actor keeps reference count entries only for objects it owns which are reachable from other actors, or objects/actors which it can reach and which it does not own.

Example 2. In Figure 2 we show an abstract representation of heaps, message queues and reference counts. We will discuss heaps and reference counts in the next section. We have message queues \(Q_{α1}\) and \(Q_{α2}\). In \(Q_{α1}\) all the queues are empty. In \(Q_{α2}\) the actor at \(α2\) has a Pony-level application message \(APP(ω_0)\), the actor at \(α2\) has a message \(APP(ω_7, ω_3, ω_4)\) and the queue of the third actor is empty. Message identifiers are not used in Pony-ORCA; only addresses are considered.

If we consider the diagram from Figure 1 together with the queues from \(Q_{α1}\) then the objects \(ω_1, ω_0, ω_3, ω_4\) and actor \(α3\) are globally unreachable. However, if we consider the queues from \(Q_{α2}\), then \(ω_3\) is the only globally unreachable reference.

figure 1. Ownership and References diagram.
Heaps

$H_1$:  
$\text{Heap}_1(\alpha_1) = \{ \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6, \omega_7, \omega_8 \}$  
$\text{Heap}_1(\alpha_2) = \{ \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3, \omega_5, \omega_7, \omega_8 \}$  
$\text{Heap}_1(\alpha_3) = \{ \alpha_2, \alpha_3, \omega_5, \omega_6 \}$

$H_2$:  
$\text{Heap}_2(\alpha_1) = \{ \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3 \}$  
$\text{Heap}_2(\alpha_2) = \{ \alpha_2 \}$

Queues

$Qs_1$:  
$\text{Qs}_1(\alpha_1) \approx \emptyset$  
$\text{Qs}_1(\alpha_2) \approx \emptyset$  
$\text{Qs}_1(\alpha_3) \approx \emptyset$

$Qs_2$:  
$\text{Qs}_2(\alpha_1) \approx \text{APP}(\omega_6)$  
$\text{Qs}_2(\alpha_2) \approx \text{APP}(\omega_7, \omega_6, \omega_7)$  
$\text{Qs}_2(\alpha_3) \approx \emptyset$

Counter Tables

$CT_1$:  
\[
\begin{array}{|c|c|c|}
\hline
\alpha_1 & \alpha_2 & \alpha_3 \\
\hline
5 & 5 & 0 \\
1 & 2 & 1 \\
0 & 0 & 0 \\
30 & 50 & 0 \\
3 & 3 & 0 \\
10 & 10 & 0 \\
0 & 0 & 0 \\
10 & 40 & 30 \\
0 & 0 & 0 \\
10 & 10 & 0 \\
10 & 10 & 0 \\
\hline
\end{array}
\]

$CT_2$:  
\[
\begin{array}{|c|c|c|}
\hline
\alpha_1 & \alpha_2 & \alpha_3 \\
\hline
6 & 5 & 0 \\
1 & 4 & 1 \\
0 & 0 & 2 \\
50 & 50 & 0 \\
3 & 3 & 0 \\
10 & 10 & 0 \\
0 & 0 & 0 \\
10 & 40 & 28 \\
0 & 0 & 2 \\
12 & 10 & 1 \\
10 & 12 & 1 \\
\hline
\end{array}
\]

Figure 2. Heaps, message queues and counter tables.

Example 3. If we consider the ownership and references diagram from Figure 1, then $\text{Heap}_1$, defined in Figure 2 is valid. Similarly, $\text{Heap}_2$ from Figure 2 is also valid. $\text{Heap}_1$ represents a possible heap before garbage collection, while $\text{Heap}_2$ represents a possible heap after all actors have performed all possible garbage collection steps.

On the other hand, $\text{Heap}_3$ defined below is invalid because even though $\omega_6$ is in the heap of $\alpha_3$, and $\omega_5$ is reachable from $\omega_6$, the heap of $\alpha_3$ does not contain $\omega_5$.

Example 4. In Figure 2 we show different reference count tables, $CT_1$ and $CT_2$. We thus have possible configurations:

$K_1 = (\text{Heap}_1, Qs_1, CT_1)$  
$K_2 = (\text{Heap}_1, Qs_2, CT_1)$  
$K_3 = (\text{Heap}_1, Qs_2, CT_2)$

4.3 Well-formed Configurations

In this section we define when a configuration is well-formed. Given that the actor only uses the values in its counter table to decide when to garbage collect an object, the counter for that specific object should be consistent with the reference count of other actors with references to it and the messages in all the queues of a configuration containing a reference to it. In other words, a configuration is well-formed, if it satisfies the conditions introduced in the previous section, an in addition:

WF3. If there is a message in some queue containing an address $\iota$, then the local reference count of $\iota$ is greater than zero.

WF4. If an actor $\alpha$ can reach an address which it does not own, then both the owner’s (local) reference count and $\alpha$’s (foreign) reference count for that object are greater than zero.

WF5. The sum of the local reference count and the increment-decrement count for an address is always the same as the sum of the total foreign reference count and total application message count for that address.

WF6. For any prefix of any actor $\alpha$’s queue, if we add to the local reference count for $\iota$ the sum of weights of increment and decrement messages for $\iota$, and subtract the number of application messages that contain $\iota$, the result is greater or equal to zero.

Given this, we define five derived counts. We define now the first four derived counts and we leave the fifth one for later. For any address $\iota$ and a global configuration, $K$, we define:

1. The local reference count of an address $\iota$, $LRC_K(\iota)$, gives the reference count for $\iota$ in the counter table of its owner.
2. The foreign reference count of an address, $FRC_K(\iota)$, is the sum of the reference counts for $\iota$ from “outside”, that is, from all actors different from the owner.
3. The increment-decrement count of an address, $IDC_K(\iota)$, is the sum of weighted references to $\iota$ in the current in-flight IDC, DEC messages in the queue of the owning actor.
4. The application message count of an address, $AMC_K(\iota)$, is the number of Pony-level messages which contain $\iota$, addresses owned by $\iota$ or addresses which are reachable from $\iota$.

In more detail, the first four counts are defined as follows.
DEFINITION 3. For a configuration $K$, and address $i$, we define the functions

$$
\begin{align*}
FRC_i, LRC_i, AMC &: \text{RunTimeCfg} \times S_{\text{all}} \rightarrow \mathbb{Z} \\
IDC &: \text{RunTimeCfg} \times S_{\text{all}} \rightarrow \mathbb{Z}
\end{align*}
$$

as follows:

1. $LRC_i(i) = RC_i(\text{Owner}(i), i)$
2. $FRC_i(t) = \sum_{\alpha \neq \text{Owner}(i)} RC_i(\alpha, t)$
3. $IDC_i(i) = \sum_j \text{Weight}(i, \text{Message}_j(\text{Owner}(i), j))$
4. $AMC_i(i) = \# \{ (\alpha, j) \mid \text{Message}_j(\alpha, j) = \text{APP}(\text{args}) \land i \in \text{APP}(\text{args}) \}$

We now define the weight of a message, $\text{Weight}(i, \text{msg})$, and the notion of an address to be contained in a message, $i \in \text{msg}$.

DEFINITION 4. The weight of a message, regarding an address $i$, is given by the function

$$
\text{Weight} : \text{Addr} \times \text{Msg} \rightarrow \mathbb{Z}
$$

$$
\text{Weight}(i, \text{msg}) = \begin{cases} 
  k & \text{if msg} = \text{INC}(i, k) \\
  -k & \text{if msg} = \text{DEC}(i, k) \\
  0 & \text{otherwise}
\end{cases}
$$

DEFINITION 5. An address $i$ is contained in a Pony-level message if and only if it is reachable from one of the arguments, or is the owner of an object reachable from one of the arguments.

$$
i \in \text{msg} \iff \exists \alpha' \in \text{msg}, \alpha'' \in \text{msg}' \land \alpha' \land \alpha'' \land (i = \alpha'' \lor i = \text{Owner}(\alpha'))
$$

Note that every address is reachable from itself.

EXAMPLE 5. In the heap from Figure 1, we have that

- $\alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3 \in \text{APP}(\omega_3)$,
- $\alpha_2, \omega_5 \in \text{APP}(\omega_5)$,
- $\alpha_2, \alpha_3, \omega_5, \omega_6 \in \text{APP}(\omega_6)$,
- $\alpha_1, \alpha_2, \omega_7, \omega_8 \in \text{APP}(\omega_7)$.

EXAMPLE 6. Consider configuration $K_1$ and in particular its object $\omega_1$, then:

$$
\begin{align*}
FRC_{K_1}(\omega_1) &= 50  & LRC_{K_1}(\omega_1) &= 50 \\
IDC_{K_1}(\omega_1) &= 0  & AMC_{K_1}(\omega_1) &= 0
\end{align*}
$$

On the other hand, in $K_2$, and considering in particular objects $\omega_3, \omega_6, \omega_7$ and $\omega_8$, we have that:

$$
\begin{align*}
AMC_{K_2}(\omega_3) &= AMC_{K_2}(\omega_6) = 2  & AMC_{K_2}(\omega_7) &= AMC_{K_2}(\omega_8) = 1 \\
AMC_{K_2}(\alpha_1) &= 1  & AMC_{K_2}(\alpha_2) &= 2  & AMC_{K_2}(\alpha_3) &= 2 \\
\text{and since we have no INC/DEC messages, } \forall i. IDC_{K_2}(i) &= 0.
\end{align*}
$$

We now define the fifth derived count, which we call pending changes count, in short $PCC$. It sums the weights of INC and DEC messages for $i$ in some queue and subtracts the number of pending application messages containing $i$ in that queue. This last count summed with the local reference count of $i$ will be the new local reference count for $i$ in a configuration where all the messages of $q$ have been consumed by $\alpha$.

DEFINITION 6. The pending changes count of an address $i$, in an actor $\alpha$, $PCC_{\alpha}(i, \alpha, q)$, is defined as follows:

$$
PCC : \text{RunTimeCfg} \times S_{\text{all}} \times S_{\text{act}} \times \text{Msg} \rightarrow \mathbb{Z}
$$

$$
PCC_{\alpha}(i, \alpha, q) = \sum_j \text{Weight}(i, q(j)) - \\
\begin{cases} 
  0 & \text{if Owner}(i) \neq \alpha \\
  \# \{ j \mid i \in q(j) \} & \text{otherwise}
\end{cases}
$$

EXAMPLE 7. Consider objects $\omega$ and $\omega'$ owned by the actor $\alpha_1$ and queues, $q_1 = \text{INC}((\omega', 1), q_2 = \text{APP}(\omega, \omega')$, and $q_1 : q_2$, then have $PCC(\omega', \alpha, q_1) = 1$ and $PCC(\omega, \alpha, q_1) = 0$. Also, $PCC(\omega', \alpha, q_2) = PCC(\omega, \alpha, q_2) = -1$. Therefore, $PCC(\omega', \alpha, q) = 0$ but $PCC(\omega, \alpha, q) = -1$.

We are now able to define when a configuration is well-formed.

DEFINITION 7. A configuration $K$ is well formed iff $\forall \alpha, i. \forall q_1, q_2 \in \text{Msg}$

$$
\begin{align*}
\text{WF3} &. i \subseteq \text{Message}_K(\alpha, j) \implies LRC_K(i) > 0 \\
\text{WF4} &. i \in \text{Heap}_K(\alpha) \land \alpha \neq \text{Owner}(i) \implies RLC_K(\alpha) > 0 \land LRC_K(i) > 0 \\
\text{WF5} &. LRC_K(i) + IDC_K(i) = FRC_K(i) + AMC_K(i) \\
\text{WF6} &. q_1 : q_2 = \text{Queue}_K(\alpha) \implies LRC_K(i) + PCC_K(i, \alpha, q_1) \geq 0
\end{align*}
$$

DEFINITION 8. An address $i$ is contained in a message $\text{msg}$ if $\text{msg}$ is an Pony-level message that contains $i$ or if it is an increment or decrement message for $i$.

$$
i \in \text{msg} \iff i \in \text{msg} \land \text{msg} = \text{INC}(\alpha, \beta) \lor \text{msg} = \text{DEC}(\alpha, \beta)
$$

The condition $\text{WF6}$ ensures that whenever consuming a message makes the actor decrease its count for some owned address (as we will see later), this count will not become negative.

EXAMPLE 8. It is easy to check that $K_1$ is well-formed. On the other hand, $K_2$ is not a well-formed configuration as $\text{WF3}$ and $\text{WF5}$ do not hold. $\text{WF3}$ does not hold because $\text{Queue}_{K_2}(\alpha_2) = \text{APP}(\omega_6)$ even though $RC_{K_2}(\alpha_3, \omega_6) = 0$. With respect to $\text{WF5}$, the AMC for the addresses $\alpha_1, \alpha_2, \alpha_3, \omega_5, \omega_6, \omega_7, \omega_8$ are no longer 0. For instance:

$$
\begin{align*}
LRC_{K_2}(\omega_5) + IDC_{K_2}(\omega_5) &= 40 + 0 \\
FRC_{K_2}(\omega_5) + AMC_{K_2}(\omega_5) &= (10 + 30) + 2
\end{align*}
$$

$K_3$ is a well-formed configuration. It is based on $K_2$, but the reference count is given by $\text{CT}_3$.

5. Pony-ORCA: “killing” them safely

Pony-ORCA allows an actor to collect its own objects without checking information from other actors or from any queue. That is, an actor does not need to check any other heap, nor does it need to examine any queue, including its own, in order to determine whether an object is collectable or not. In this section we argue why it is safe to collect objects under these conditions, and what actions the protocol needs to perform in order to preserve well-formedness.

Pony-ORCA runs on top of the Pony type system [6], which guarantees that there will be no race conditions even though it does not use any locking or synchronisation mechanism other than the messaging system.

In Section 5.1 we show why it is sound to collect any actor or object whose local reference count is 0. In Section 5.2 we describe what actions need to be taken when the configuration changes, i.e. upon message send, message receipt, or addresses becoming unreachable. We show a garbage collection scenario in Section 5.3. In Section 5.4 we discuss the role of causality. And in Section 5.5 we discuss the absence of race conditions in the system.
5.1 Application to Garbage collection

Here we will argue why it is sound to garbage collect an actor or object when its local reference count is 0. We will use the well-formedness conditions from sections 4.2 and 4.3 to show that under those circumstances the object or the actor is globally unreachable.

Definition 9 (Globally unreachable objects and actors). An address $i$ is globally unreachable if and only if it does not appear in any actor’s heap (i.e. $\forall \alpha \in \text{Heap}_\alpha(i)$) and does not appear in any Pony level message (i.e. $\forall j \in \text{Message}_K(\alpha, j)$).

5.1.1 Collectable objects

Definition 10. An object $\omega$ is collectable by an actor $\alpha$, iff

1. $\alpha$ owns $\omega$, i.e., $\alpha = \text{Owner}(\omega)$.
2. $\alpha$ has no path leading to $\omega$, i.e., $\omega \notin \text{Heap}_\alpha(\alpha)$.
3. $\alpha$’s (local) reference count for $\omega$ is 0, i.e., $RC_K(\alpha, \omega) = 0$.

These three requirements are local and therefore the actor can garbage collect without need to consult other actors or examine any queues, including its own.

Moreover, these three requirements guarantee that garbage collection is safe. We now argue informally why in a well-formed configuration any collectable object is globally unreachable.

1. From C1, C3 and Definition 3 we know that the local reference count for $\omega$ is 0, i.e., $LRC_K(\omega) = 0$.
2. From C1, C3 and WF3 we know that the counts for increment/decrement and application messages are 0, i.e., $IDC_K(\omega) = 0$ and $AMC_K(\omega) = 0$.
3. From 1, 2 and WF5 we know that the foreign reference count of $\omega$ is 0, i.e., $FRC_K(\omega) = 0$.
4. From C2 we obtain that $\omega$ is not reachable from $\alpha$.
5. From 3 and WF4 we obtain that $\omega$ is not reachable from any further actor $\alpha' \neq \alpha$.
6. From 2, we obtain that $\omega$ is not in any in-flight message.

Therefore, $\omega$ is globally unreachable.

5.1.2 Collectable actors

Definition 11. An actor $\alpha$ is collectable, iff

1. Its local reference count for itself is 0, i.e., $RC_K(\alpha, \alpha) = 0$.
2. Its queue is empty, i.e., $\text{Queue}_K(\alpha) = 0$.

The argument that a collectable actor is globally unreachable is similar to that for objects with the additional consideration of WF2. In addition, we can expand the protocol so as to collect cycles of blocked actors whose local reference count is greater than 0 [16].

5.2 Maintaining well-formedness

Language level computations, such as application message sends and receives, and dropping of references, affect the validity of the well-formedness conditions from section 4.3. Therefore, we need to take corrective actions. Here we outline what these actions are.

5.2.1 Sending a Pony level message

Consider that an actor $\alpha$ sends a message $\text{APP}(args)$. This action does not modify the heap, therefore WF1, WF2 and WF4 are preserved. However it does affect the values of $AMC$, and thus affects validity of WF5. Therefore, for all $\iota \in \text{APP}(args)$, the actor $\alpha$ performs the following updates:

1. If $\alpha = \text{Owner}(\iota)$ then it will increase $RC(\alpha, \iota)$ by 1.
2. If $\alpha \neq \text{Owner}(\iota)$ then it will decrease $RC(\alpha, \iota)$ by 1.

If decreasing the counter were to set the value of $RC(\alpha, \iota)$ to 0 or below then, before sending the Pony message, an $\text{INC}(\iota, k + 1)$ message is sent to $\text{Owner}(\iota)$ and $RC(\alpha, \iota)$ is set to $k$, for some $k > 0$. Such an increment message may be sent even if $RC > 1$, to allow future sends of $\iota$ without requiring additional increment messages.

These actions restore WF5 and do not affect validity of WF3 and WF6.

5.2.2 Receiving a Pony level message

The actions taken to preserve the well-formedness of a configuration are similar to those for sending a message containing $\iota$, taking into consideration that when a message is received, it is removed from the queue. We consider an actor $\alpha$ that receives a message $\text{APP}(args)$. For all addresses $\iota$ such that $\iota \in \text{APP}(args)$ the receiving actor performs the following actions, which are essentially the opposite to those in Section 5.2.1:

1. If $\alpha = \text{Owner}(\iota)$ then it will decrease $RC(\alpha, \iota)$ by 1.
2. If $\alpha \neq \text{Owner}(\iota)$ then it will increase $RC(\alpha, \iota)$ by 1.

The address $\iota$ is added to the heap of $\alpha$. Validity of WF4 is trivially preserved in all four cases. Moreover, WF6 from the previous configuration guarantees preservation of WF1 in the new configuration.

5.2.3 Receiving an ORCA specific message

When an actor receives a message $\text{INC}(\iota, k)$ or $\text{DEC}(\iota, k)$ then, by construction, it is the owner of $\iota$. Therefore,

1. When $\alpha$ receives $\text{INC}(\iota, k)$, it increments $RC(\alpha, \iota)$ by $k$.
2. When $\alpha$ receives $\text{DEC}(\iota, k)$, it decrements $RC(\alpha, \iota)$ by $k$.

Condition WF6 guarantees that WF1 and WF6 are preserved. The other conditions are not affected.

5.2.4 Collecting in actor’s heap

An actor garbage collects its objects between the execution of its behaviours, that is, when its stack of execution is empty. We now show the object collection algorithm step-by-step:

1. All owned objects are marked unreachable.
2. All unowned objects with a foreign reference count greater than zero are marked as unreachable.
3. Tracing occurs from the actor’s fields only, marking objects reachable, whether they are owned or not.
4. All owned objects with a local reference count greater than zero are marked as reachable.
5. Owned objects that are marked as unreachable are collected.
6. Decrement messages are sent for unowned objects that are unreachable, and their $RC$ is set to 0.

Soundness of step 5 has been discussed in Section 5.1. Step 6 trivially preserves WF1 and WF2. Moreover, when the actor sends the decrement message, $\text{DEC}(\iota, k)$, where $\iota$ is the address to be collected and $k$ is the former reference count of the address $\iota$, both $FRC(\iota)$ and $LRC_K(\alpha)$ will decrease by $k$ and WF5 is preserved.

5.3 A Garbage Collection Scenario

We will elucidate how the garbage collector works by means of a scenario where $\alpha_3$ runs garbage collection on configuration $K_3$. Then $\alpha_2$ runs garbage collection, followed by $\alpha_1$, which again is followed by $\alpha_2$. In the end, we will have collected all the globally inaccessible actors or objects, and we will be left with a heap as in $\text{Heap}_2$. We explain this now in more detail:
Causality is sufficient in Pony-ORCA to maintain WF4 of ORCA unless, of course, the heaps or queues were to change. If the queue entry for \( k \) is 0, then \( k \) stays unmodified. The actor will set its RC entry for \( k \), \( \omega \), and its heap will have the contents as in Table 3.

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \omega )</th>
<th>( \omega )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \omega )</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>( \omega )</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

3rd Step: actor \( \alpha_3 \) performs garbage collection. The references \( \omega_4 \), \( \omega_5 \), \( \omega_6 \), and \( \omega_7 \) are locally unreachable. Of these addresses, the actor owns \( \omega_4 \) and \( \omega_5 \), and their RC entry is 0, both objects will be collected. We now have \( Heap_3 = \{ \omega_1, \omega_3, \omega_4, \omega_5, \omega_6, \omega_7 \} \).

When receiving a message, increment messages are only generated when the sending actor has an RC of 1 for a sent actor or object it does not own; otherwise, the sending actor simply increases its RC for objects it owns and decreases its RC for other actors and objects. When sending a message, increment messages are only generated when there is no stack. This means there is no requirement for a stack map or a stack crawler.

6. Discussions

The message overhead of our approach is low. Reference counts (RC) are not tied to heap references but rather to messages, which means no ORCA specific messages are required for heap mutation. When receiving a message, no ORCA specific messages are generated, because the receiving actor can simply decrease its RC for objects it owns and increase its RC for other actors and objects. When sending a message, increment messages are only generated when there is no stack. This means there is no requirement for a stack map or a stack crawler.

Because the disappearance of a reference from an actor’s reachable heap is detected only during garbage collection, decrement messages are generated lazily. These decrement messages are also combined, as for increment messages, such that at most one decrement message is generated for any given actor. This reduces the ORCA specific message bounds from \( O(unreachable) \) to \( O(Owner(unreachable)) \).

7. Conclusion

Causal garbage collection allows us to never require an acknowledgement for an ORCA specific message. This means that there is no ORCA specific message related latency due to requiring a round trip.

Our approach only runs a garbage collection pass on an actor when that actor is not executing a behaviour. As a result, GC only occurs when there is no stack. This means there is no requirement for a stack map or a stack crawler.

We do not require any form of read or write barrier [38]. This is achieved by combining a data-race free type system with handling all RC changes when sending and receiving messages.

While this does mean that data structures are traced when they are sent and received, it eliminates the need to treat objects with a local reference count greater than zero as GC roots. Instead, such objects are simply marked as reachable and need not to be traced any further. This is a key element of the system: it protects such objects from collection by the owning actor when some other actor is in the process of mutating them. This allows heaps to be garbage collected entirely independently, as mutation in another actor does not affect the read capabilities of each of the fields in that class, we ensure that tracing does not interfere with any other actor’s activity.
not affect GC. We have not modelled this in our current paper, as we do not model the tracing of structures, but will describe this in more detail in further work.

The combination of having no stack when garbage collecting and garbage collecting each actor’s heap independently means safe-points [25] are not required. Any actor can GC its own heap without waiting for a safepoint to be reached.

Independent actor heap collection functions as both a concurrent GC mechanism (actors can GC concurrently) and an incremental GC mechanism (actors can GC separately). This allows optimisations to both the tracing algorithm and the memory allocator. We use a mark-and-don’t-sweep collector [24] that keeps mark bits, rather than pointers, in the heap data structure. By moving the mark bit out of the object, object contents are never written to during GC. Moreover, unreachable objects are not marked or swept. These two approaches together minimise cache pollution, eliminate page misses on unreachable objects, and reduce the trace phase to $O(\text{reachable})$ rather than $O(\text{reachable} + \text{unreachable})$.

This approach also results in a memory allocator that works like a bump allocator [24]. No best/first-fit search is required, and free list maintenance is handled by page (or group of pages) rather than by object. Memory allocation cost is amortised to the cost of a single find-first-set bit operation.

These optimisations are made possible because independent actor heap collection guarantees mutation does not affect either tracing or allocation semantics.

This protocol was implemented in the Pony compiler which was benchmarked against other actor-model languages with the CAF [13] benchmark suite [1] and against MPI with HPC Challenge LINPACK GUPS [2]. We now report the results of these experiments taken from [6]. Benchmarking was done on a 12-core 2.3 GHz Opteron 6338P with 64 GB of memory across 2 NUMA nodes. The results shown are the average of 100 runs.

The first benchmark, shown in Figure 3 shows the performance of creating large numbers of actors. In particular, the performance of garbage collecting the actors [17] and passive objects. It shows better results than the other existing systems (except CAF which is not garbage collected).

In Figure 4, we can see the performance of a highly contended mailbox, where additional cores tend to degrade performance. The third experiment, illustrated in Figure 5, shows performance of a mixed case, where a heavy message load is combined with brute force factorisation of large integers. Finally, Figure 6 shows a benchmark that is not tailored for actors: we take the GUPS benchmark that is not tailored for actors: we take the GUPS benchmark suite [1] and against MPI with HPC Challenge LINPACK GUPS [2]. We now report the results of these experiments taken from [6]. Benchmarking was done on a 12-core 2.3 GHz Opteron 6338P with 64 GB of memory across 2 NUMA nodes. The results shown are the average of 100 runs.

Figure 3. Actor creation performance

Figure 4. Mailbox performance

Figure 5. Mixed case performance

mark from high-performance computing, which tests random access memory subsystem performance, and demonstrate that Pony’s implementation is significantly faster than the highly optimised MPI implementation.

7. Conclusions and Further Work

Pony is a concurrent and distributed object-oriented, actor-based programming language which supports (passive) objects. One of the features of Pony is its message-based garbage collection mechanism. This allows the collection of dead actors, as presented in [17]. We have presented the garbage collection for passive objects in Pony. We have formally defined what constitutes a runtime configuration, with respect to the data structures that the garbage collector maintains, and in particular what constitutes a well-formed configuration which allows the deallocation of passive objects. Moreover, we informally describe how to keep a runtime configuration consistent when certain operations are executed.

We plan to give a full formal model, including a model of the heap and reachability, and proof of soundness. We want to investigate in how far we can weaken the requirements for causality. Finally, we want to adapt ORCA to cater for applications to further language features such as futures or promises.
References