Verifying a Concurrent Garbage Collector using a Rely-Guarantee Methodology[⋆]

Yannick Zakowski[1], David Cachera[1], Delphine Demange[2], Gustavo Petri[3], David Pichardie[1], Suresh Jagannathan[4], Jan Vitek[5]

[1] ENS Rennes – IRISA – Inria [2] Université Rennes 1 – IRISA – Inria
[3] IRIF – Université Paris Diderot [4] Purdue University [5] Northeastern University

Abstract Concurrent garbage collection algorithms are an emblematic challenge in the area of concurrent program verification. In this paper, we address this problem by proposing a mechanized proof methodology based on the popular Rely-Guarantee (RG) proof technique. We design a specific compiler intermediate representation (IR) with strong type guarantees, dedicated support for abstract concurrent data structures, and high-level iterators on runtime internals. In addition, we define an RG program logic supporting an incremental proof methodology where annotations and invariants can be progressively enriched.
We formalize the IR, the proof system, and prove the soundness of the methodology in the Coq proof assistant. Equipped with this IR, we prove a fully concurrent garbage collector where mutators never have to wait for the collector.

1 Introduction

Modern programming languages like ML, Java, and C# rely on garbage collection (GC) for the automatic reclamation of memory no longer used by the application. The GC is considered to be one of the most subtle parts of modern runtime systems, carefully engineered to minimize runtime overheads of the applications it supports. A family of garbage collection algorithms, named *on-the-fly* garbage collectors [2], allows the detection of garbage and its reclamation to occur concurrently with an application's threads. Such algorithms are notably difficult to implement, test, and prove, and constitute a significant challenge for mechanized verification. Many on-the-fly algorithms are inherently racy, and some algorithms never require application threads (called *mutators*) to wait for the *collector* thread, which detects and frees unused memory. This paper focuses on an emblematic algorithm in this landscape [5,3,4], where no locks are required – *i.e.* it is *lock-free*.

This challenge has been identified and addressed in various settings [8,9,11,12]. This paper provides an independent proof, and it explores a different proof method in the design space. First, the backbone of the formalization is a new compiler intermediate representation, named RtIR, which we use to implement the garbage collector. Our experience implementing on-the-fly garbage collectors [20] indicates that the choice of programming abstractions is of paramount importance in reasoning and optimizing this

⋆ This material is based upon work supported by grants ANR 14-CE28-0004, NSF CCF-1318227, CCF-1544542, SHF-1518844 and ONR N00014-15-1-2332.

kind of algorithm. This concern necessitates a representation that makes the expression and proof of invariants tractable. Moreover, in this work, we strive to make our proof well suited to the context of a larger project, described in [1,14], aiming at the formal verification of a compiler for concurrent, managed languages. Our intermediate representation has special support for the implementation of efficient runtime mechanisms: (i) strong type guarantees, (ii) abstract concurrent data structures, (iii) high-level iterators for reflective inspection of objects, used to implement low-level services, *e.g.* ensuring the garbage collector visits every live object (iv) native support for threads, and (v) native support for the root management of a concurrent garbage collector (each thread must be able to iterate over the set of memory references it can access directly).

Another important characteristic of our approach is the dedicated rely-guarantee program logic that accompanies our intermediate representation. While previous approaches [8,9,12] attack the proof by means of an abstract state transition system requiring a monolithic global invariant to be established, we follow the well established rely-guarantee [15] (RG) methodology. RG is a major technique for proving the correctness of concurrent programs that provides explicit thread-modular reasoning. In this setting, interferences between threads are described using binary relations: *relies* and *guarantees*. Each thread is proved correct under the assumption it is interleaved with threads fulfilling a *rely* relation. The effect of the thread itself on the shared memory must respect its *guarantee* relation. This guarantee must also be coherent with respect to the relies that the other threads assume. Being able to reason in a thread modular way is key to realize a tractable correctness proof because it avoids the need to explicitly consider all possible interleavings. We prove the soundness of our RG logic, and develop a set of tactics that reduce the proof effort required to discharge the invariants.

Finally, we report on an original *incremental* proof technique that we put in place to carry out this massive endeavour. Starting from the full GC implementation, we progressively annotate the program in order to prove stronger and stronger invariants. At each level, dedicated specification annotations and tactics allow us to refine and reuse what has been proved at the previous levels.

Using the Coq proof assistant, we achieved the following formalizations: (i) the syntax, semantics and the soundness of an RG program logic for our intermediate representation, (ii) a number of tactics and structural lemmas to facilitate the so-called *stability proofs* required by the RG methodology, (iii) a realistic implementation of Domani *et al.*'s GC algorithm [5] in our intermediate representation and (iv) an RG proof ensuring the correctness of the GC: the collector never frees references accessible by the running threads. Our formal development is available online [7].

2 The RtIR Intermediate Representation

2.1 Syntax

Figure 1 shows the syntax of RtIR (RunTime IR). It provides two kinds of variables: *global* or *shared* variables that can be accessed by all threads, and *local variables* used for thread-local computations. Expressions (e) are built from constants and local variables with the usual arithmetic and boolean operators. Commands include standard in-

$$
\begin{array}{lllll}
X, Y \in \texttt{gvar} & x, y \in \texttt{lvar} & t, m, C \in \texttt{tid} & f \in \texttt{fid} & rn \in \texttt{list fid} \\
\texttt{cmd} \ni c := \texttt{skip} & \mid \texttt{assume}\ e & \mid x = e & & \\
\quad\mid\ c_1\ ;\ c_2 & \mid c_1 \oplus c_2 & \mid \texttt{loop}(c) & \mid \texttt{atomic}\ c & \\
\quad\mid\ x = \texttt{alloc}(rn) & \mid \texttt{free}(x) & \mid \texttt{isFree?}(x) & & \\
\quad\mid\ x = Y & \mid X = e & \mid x = y.f & \mid x.f = e & \\
\quad\mid\ x.\texttt{push}(y) & \mid x = y.\texttt{empty?}() & \mid x = y.\texttt{top}() & \mid x.\texttt{pop}() & \mid X = y.\texttt{copy}() \\
\quad\mid\ \texttt{foreach}\ (x\ \mathit{in}\ \texttt{l})\ \texttt{do}\ c\ \texttt{od} & & \mid \texttt{foreachField}\ (f\ \texttt{of}\ x)\ \texttt{do}\ c\ \texttt{od} & & \\
\quad\mid\ \texttt{foreachObject}\ x\ \texttt{do}\ c\ \texttt{od} & & \mid \texttt{foreachRoot}\ (x\ \texttt{of}\ t)\ \texttt{do}\ c\ \texttt{od} & &
\end{array}
$$

Figure 1: Simplified Syntax of RtIR. Proof annotations elided.

structions, such as `skip`, `assume` e, local variable update $x = e$, and classic combinators: sequencing, non-deterministic choice ($c_1 \oplus c_2$), and loops. The usual conditional (`if` e `then` c_1 `else` c_2) can be defined as (`assume` $e; c_1$) $\oplus$ (`assume` $!e; c_2$), where we write $!e$ for the boolean negation of e. While loops and repeat-until loops can be encoded similarly. RtIR also provides atomic blocks (`atomic` c). In our GC, we use atomic blocks only to add ghost-code – code only used for the proof, not taking part in the computation – and to model linearizable data structures. These atomic constructs can be refined into low-level, fine-grained implementations using techniques like [14,26].

Instruction `alloc`(rn) allocates a new object in the heap by extracting a fresh reference from the freelist – a pool of unused references – and initializing all of its fields in the record name rn to their default value. Conversely, `free` puts a reference back into the freelist. Instruction `isFree`? looks up the freelist to test whether a reference is in it. We use these memory management primitives to implement the GC.

In RtIR, basic instructions related to shared-memory accesses are fine-grained, *i.e.* they perform exactly one global operation (either read or write). These include loads and stores to global variables and field loads and updates. This allows us, when conducting the proofs, to consider each possible interleaving of memory operations arising from different threads, while keeping the semantics reasonably simple. Apart from these basic memory accesses, RtIR provides abstract concurrent queues which implement the *mark buffers* of [5], accessible through standard operations $y = x.$`top`(), $x.$`pop`(), $x.$`push`(y), $x = y.$`empty`?(). The use of these buffers, necessary for the implementation of the GC, will be made clear in Section 4. While we could implement these data structures directly in RtIR, we argue that to carry out the proof of the GC, it is better to reason about them at a higher level, and hence to assume that they behave atomically. Implementing these data structures in a correct and linearizable [13] fashion is an orthogonal problem, that we address separately [26]. Mark buffers also provide an operation $X = y.$`copy`(), to perform a deep copy, only used in ghost code.

A salient ingredient of RtIR is its native support for *iterators*, allowing to easily express many bookkeeping tasks of the GC. The iterator `foreach` (x *in* `l`) `do` c `od`, where the variable x can be free in command c, iterates c through all elements x of the static list `l`. Some more sophisticated bookkeeping tasks include the visiting of all the fields of a given object, the marking of each of the *roots* – references bound to local variables – of mutators, or the visiting of every object in the heap (performed during the *sweeping* phase). In those cases, the lists of elements to be iterated upon is not known statically, so we provide dedicated iterators. The iterator `foreachField` (f `of` x) `do` c `od` iterates c on all the fields f of the object stored in x. Command `foreachRoot` (r `of` t) `do` c `od` iterates over the roots of mutator thread t, while `foreachObject` x `do` c `od` iterates over all objects. We stress the fact that iterators have a fine-grained behavior: the body command c executes in a small-step fashion.

2.2 Operational semantics

The operational semantics of RtIR is mostly standard. We provide two kinds of operational semantics: (i) a *big-step* semantics, used to define the semantic validity of Hoare-like tuples for basic instructions (see Section 3), as well as commands in atomic blocks; (ii) a small-step interleaving semantics used to prove our final soundness results. We only present here the description of execution states, and refer the interested reader to the Coq development [7] for the formal semantics.

Typing information The semantics of RtIR is enriched with typing information. Basic types in `typ` include `TNum` for numeric constants, `TRef` for references to regular objects (see below), and `TRefSet` for non-null references to abstract mark-buffers. Local variables, global variables, and field identifiers are declared to have exactly one of these types, respectively accessible through functions `lvar_typ`, `gvar_typ` and `fid_typ`. RtIR manipulates two kinds of values: numeric values in the Coq type `Z` and references in `ref`. Types are mapped to values with the function `value` of type `typ → Type`.

```
typ ≜ { TNum, TRef, TRefSet }
lvar ≜ varId × typ
gvar ≜ varId × typ
fid ≜ fieldId × typ
```

```
Definition value (t:typ):Type :=
  match t with
  | TNum ⇒ Z
  | TRef | TRefSet ⇒ ref end.
```

Execution states Local (resp. global) environments map local (resp. global) variables to values of their declared type. Environments are hence dependent functions of type:

```
Definition lenv := ∀ x:lvar, value (lvar_typ x).
Definition genv := ∀ X:gvar, value (gvar_typ X).
```

A thread-local state is defined by a local environment and a command to execute. A global state includes a global environment `ge` and a heap `hp` – a partial map from references to `objects`. We consider two distinct kinds of objects: regular objects, mapping fields to values, and abstract mark-buffers.

```
Definition thread_state := (cmd * lenv).
Record gstate := {  ge: genv;               freelist: ref → bool;
                    hp: ref → option object; roots: tid → ref → nat }.
```

Global states also include two components essential to the implementation of a GC: `roots` and a `freelist`. The freelist is indeed a shared data structure, while roots are considered to be thread-local – mutators are responsible for handling their own roots with thread-local counters. Here, we model roots as part of the global state only to ease proof annotations – our final theorem is an invariant of the program global state.

Finally, execution states include the states of all threads and a global state.

```
Definition state := ((tid → option thread_state) * gstate).
```

Well-typedness invariants A number of invariants are guaranteed by typing: (i) each variable in the local or global environment contains a value of the appropriate type, (ii) any reference of type `TRef` is either null, in the domain of the heap, or in the freelist, and (iii) each abstract mark-buffer is accessible from a unique global variable, indexed by a thread identifier. This mechanism enforces separation of mark-buffers by typing.

3 RtIR Proof System

On top of RtIR, we design a program logic, based on a variation of rely-guarantee (RG). In a nutshell, RG [15] extends Hoare-logic to handle concurrency in a thread modular fashion. In addition to the standard Hoare-tuples, side conditions ensure that program annotations take into account the possible interferences of other threads. When thinking about a particular thread's code, we shall refer to the actions of the other concurrent threads as its *context*. This context is formally encoded as a *rely* relation stating its possible execution steps. Thus, each annotation in the code of a thread must be proved to be *stable w.r.t.* its rely condition, meaning that its validity is not affected by possible state changes induced by any number of rely steps. We follow a similar approach to encode guarantees (cf. Section 1). In fact, throughout our development we only ever need to define guarantees, and we synthesize the relies of other threads from guarantees.

High-level design choices of proof rules In our approach, we firstly annotate a program, as is usually done on paper, and then prove the annotated program using syntax-directed proof rules. We thus extend the syntax of commands to include *annotations*. Syntax-directed proof rules were capital for proof automation.

The proof system decouples sequential and concurrent reasoning. Its first layer is a Hoare-like system, with no use of relies or guarantees. A second layer handles interference: proof obligations about relies, guarantees and stability checks of annotations.

Finally, to avoid polluting programs with routine annotations, typically the global invariants, the first layer of the system *assumes* that such invariants hold, and the second layer requires to separately prove their invariance as a stability check.

Annotations We use a shallow embedding into Coq, with annotations of type either `pred≜gstate→lenv→Prop`, or `gpred≜gstate→Prop` when they deal with the global state only. Typically, the global invariant of the GC is of type `gpred`. We also define the usual logical connectives on `pred` and `gpred` with the expected meaning. Conjunction is written `A⩕B` and implication is written `A⟶B`. Annotations of type `gpred` are automatically cast into `pred` when needed.

The syntax presented in Section 2 is extended to take annotations into account. While elementary commands that do not utilize the global state do not need to be extended, basic commands accessing memory (*e.g.* field loads and updates, global loads and stores, and mark-buffer operations) have to take an extra argument of type `pred`, representing the pre-condition of the command. This is also the case for loops, annotated with a loop-invariant, and atomic blocks, whose body may affect the global state. The semantics of RtIR completely ignores annotations which are only relevant for proofs.

In the sequel, we use the informal notation `P`@c for a command c annotated with `P`.

Sequential Layer We start by defining the following predicate, `I ⊨ t: ⟨P⟩ c ⟨Q⟩` that corresponds to the validity of a sequential Hoare tuple, with respect to the big-step operational semantics of commands. This semantic judgment asserts that, for thread `t`, if command `c` runs in a state satisfying precondition `P`, and if the execution terminates, the final state must satisfy post-condition `Q` under the assumption that the global predicate `I` is an invariant. Proving that `I` is indeed invariant is done separately.

First-layer logic judgments for commands are of the form `I ⊢ t: ⟨P⟩ c ⟨Q⟩`. For basic commands which do not require annotations and simple command compositions (sequence, non-deterministic choice and loops), proof rules follow the traditional weakest-precondition style. This can be seen in the following rules:

$$\frac{}{\texttt{I} \vdash \texttt{t: } \langle \texttt{P} \rangle\, \texttt{skip}\, \langle \texttt{P} \rangle} \qquad \frac{\texttt{I} \vdash \texttt{t: } \langle \texttt{P} \rangle\, c_1\, \langle \texttt{R} \rangle \quad \texttt{I} \vdash \texttt{t: } \langle \texttt{R} \rangle\, c_2\, \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t: } \langle \texttt{P} \rangle\, c_1 ; c_2\, \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vdash \texttt{t: } \langle \texttt{P1} \rangle\, c_1 \langle \texttt{Q} \rangle \quad \texttt{I} \vdash \texttt{t: } \langle \texttt{P2} \rangle\, c_2 \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t: } \langle \texttt{P1} \,\mathbin{\wedge\!\!\wedge}\, \texttt{P2} \rangle\, c_1 \oplus c_2\, \langle \texttt{Q} \rangle}$$

On the other hand, commands that require annotations directly embed the semantic judgment `I ⊨ t: ⟨P⟩ c ⟨Q⟩` as a proof obligation. For instance:

$$\frac{\texttt{I} \vDash \texttt{t: } \langle \texttt{P} \rangle\ \texttt{P}@X = e\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t: } \langle \texttt{P} \rangle\ \texttt{P}@X = e\ \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vDash \texttt{t: } \langle \texttt{P} \rangle\, c\, \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t: } \langle \texttt{P} \rangle\ \texttt{P@atomic}\ c\ \langle \texttt{Q} \rangle}$$

Interference Layer This layer takes into account threads interference with a given command, handling the validity of guarantees and the stability of program annotations *w.r.t.* the context. This can be seen in the definition of a valid RG tuple:

```
Record RGt (t:tid) (R:rg) (G:list rg) (I:gpred) (P Q:pred) (c:cmd) := {
  RGt_hoare:     I ⊢ t: ⟨P⟩ c ⟨Q⟩
; RGt_stable:    stable I P R ∧ stable I Q R ∧ AllStable I c R
; RGt_guarantee: AllRespectGuarantee t I c G }.
```

Here, the type `rg≜gstate→gstate→Prop` defines relies and guarantees as binary relations between global states. In our development, we build them from annotated commands. For a command `P@c`, the associated `rg` is defined by running the (big-step) operational semantics of `c` from a pre-state satisfying `P` to a post-state (in Section 5, we explain how our proof methodology benefits from this definition).

Predicate `stable` defines the stability of a `pred` *w.r.t.* a rely, given some invariant:

```
Definition stable (I:gpred) (H:pred) (R:rg) : Prop := ∀ gs1 gs2 l,
  I gs1 ∧ H gs1 l ∧ R gs1 gs2 ∧ I gs2 → H gs2 l.
```

The predicate `AllStable` builds the conjunction of the stability conditions for all assertions syntactically appearing therein. We omit its formal definition here.

The validity of the guarantee of a command (predicate `AllRespectGuarantee`) follows the same principle, this time accumulating proof obligations that all elementary effects of the command are reflected by an elementary guarantee in the list `G`.

Program RG specification The RG specification of a program `p` is defined as a record considering guarantees `G` and pre- and post-conditions `P` and `Q` for all threads. Formally:

```
Record RGt_prog (G:tid → rg) (I:gpred) (P Q:tid → pred) (p:program) := {
  RGp_t:∀ t ∈ (threads p), RGt t (Rely G t) (G t) I (P t) (Q t) (cmd t p)
; RGp_I:∀ t, stable TTrue I (G t) }.
```

Obligation `RGp_t` requires that each thread's command is proved valid. It is worth noting that only guarantees need to be considered: for each thread, we build its rely from other threads' guarantees (`Rely G t`). This significantly reduces redundancies in specifications. Second, obligation `RGp_I` requires that `I` is invariant. We encode this as a stability condition under the union of all threads' guarantees, assuming the trivial invariant `TTrue ≜ (fun _ _ ⇒ True)`. Indeed, as all threads' code satisfy their guarantees, this is enough to prove that the global invariant `I` is preserved by any number of program steps.

Reasoning about Iterators As expected, the case of iterators is more involved. We illustrate their treatment on `foreach`. Though more technically involved, others iterators are similar. Recall that `foreach` iterates on a list of data of type `A`, morally representing a loop. Hence, its proof involves a loop invariant, predicated over the visited elements of the list. Predicates annotating `foreach` are thus indexed by a list of visited elements. And, as the loop body may include annotations about visited elements, we also index it by a list of visited elements and a current element. Summing up, the syntax of `foreach`, extended with annotations is `P@foreach (`x *in* `l) do c od` where annotation `P` has type `list A → pred`, and `c` has type `list A → A → cmd`. The associated proof rule is:

$$\frac{\begin{array}{l}\forall\ \texttt{a seen, prefix (seen++[a]) l} \rightarrow \\ \qquad\quad \texttt{I} \vdash \texttt{t:}\ \langle\texttt{P seen}\rangle\ \texttt{(c seen a)}\ \langle\texttt{P (seen++[a])}\rangle \\ \texttt{P l} \mathrel{⩕} \texttt{I} \longrightarrow \texttt{Q}\end{array}}{\texttt{I} \vdash \texttt{t:}\ \langle\texttt{P nil}\rangle\ \texttt{P@foreach}\ (x\ \mathit{in}\ \texttt{l})\ \texttt{do c od}\ \langle\texttt{Q}\rangle}$$

The first premise amounts to proving a valid tuple whose pre- and post-conditions are adjusted to the list of already visited elements. The second premise requires precondition `P` applied to the whole list of elements to entail the post-condition of the iterator itself. We define a more general rule in Coq, to get an induction principle usable to prove the soundness of the logic.

Soundness of the logic Soundness states that invariant `I` holds in every state reachable from a well-formed initial state – which must satisfy `I` by construction – through the small-step semantics mentionned in Section 2. Formally:

```
Hypothesis init_wf : ∀ tsi gsi, init_state p (tsi,gsi) →
      RGt_prog G I P p Q                           (* program RG spec *)
   ∧ (∀ t c le, tsi(t) = Some(c, le) → P t gsi le) (* pre-conds. hold *)
   ∧ I gsi.                                        (* I holds initially *)
Theorem soundness : ∀ ts gs, reachable init_state p (ts,gs) → I gs.
```

The proof of this theorem relies on an auxiliary proof system, proved equivalent to the one presented earlier. The auxiliary system reuses the same basic components, but proof rules now require to prove everything in situ: the invariant, the pre- and post-conditions, the stability of annotations, and the validity of guarantees. For instance, compare the rule for instruction $X = e$ in the previous system (left) with the proof rule of the auxiliary system (right):

$$\frac{\texttt{I} \vDash \texttt{t:}\ \langle\texttt{P}\rangle\ \texttt{P@}X = e\ \langle\texttt{Q}\rangle}{\texttt{I} \vdash \texttt{t:}\ \langle\texttt{P}\rangle\ \texttt{P@}X = e\ \langle\texttt{Q}\rangle} \qquad \frac{\begin{array}{c}\texttt{TTrue} \vDash \texttt{t:}\ \langle\texttt{P} \mathrel{⩕} \texttt{I}\rangle\ \texttt{P@}X = e\ \langle\texttt{Q} \mathrel{⩕} \texttt{I}\rangle \\ \texttt{stable TTrue (P} \mathrel{⩕} \texttt{I) G} \quad \texttt{stable TTrue (Q} \mathrel{⩕} \texttt{I) G} \\ \texttt{RespectGuarantee t I G (P@}X = e\texttt{)}\end{array}}{\texttt{R, G, I} \vdash \texttt{t:}\ \langle\texttt{P}\rangle\ \texttt{P@}X = e\ \langle\texttt{Q}\rangle}$$

This auxiliary system is very close to the classic RG [15,25]. Its verbosity makes it easier to reason about the soundness proof.

The soundness proof itself consists in a subject-reduction lemma *w.r.t.* the following property: in the current execution state, every possible thread currently running is in fact running a piece of code that conforms to `RGt` (the pre-conditions map `P` is hence updated at each step), and the global invariant `I` holds. Invariance of `I` follows from the fact that, in each rule of the auxiliary system, the invariant is part of the pre- and post-conditions, which are stable against any step of the rely and the guarantee of the stepping thread.

4 The Concurrent Garbage Collector

We now describe our implementation in RtIR of the concurrent GC, and its associated correctness theorem. The algorithm is based on [5], a variant of the well known concurrent *mark-and-sweep* algorithm due to Doligez, Leroy and Gonthier [4,3].

Main Theorem Intuitively, we want to show that the collector thread never reclaims memory that could potentially be used by mutators. To do this, we program the collector and the mutators in RtIR, and prove that their parallel composition preserves an invariant on global execution states, using the soundness theorem of our program logic.

The particularity of mutators is that they participate to the bookkeeping required for the collection to be correct. In practice, bookkeeping code is injected in client code by the compiler. Here, we consider a *Most General Client* (MGC) representing a collector thread composed with an arbitrary number of mutators with identifiers in `Mut`, each running relevant injected pieces of code.[6]

$$\begin{aligned} \mathtt{mutator} &\triangleq \mathsf{loop}(\ \mathtt{update}(\mathtt{x},\mathtt{f},\mathtt{v}) \\ &\qquad\qquad \oplus \mathtt{load}(\mathtt{x},\mathtt{f}) \oplus \mathtt{alloc}() \\ &\qquad\qquad \oplus \mathtt{cooperate}() \oplus \mathtt{changeRoots}()) \\ \mathtt{mgc} &\triangleq \mathtt{collector} \parallel \mathtt{mutator} \parallel \ldots \parallel \mathtt{mutator} \end{aligned}$$

Recall that the special global variable `freelist` a pool of unused references. Hence, upon allocation, a reference is fetched from the freelist. Symmetrically, to reclaim an unused object, the collector puts back its reference into the freelist.

Our main invariant establishes that in a given state `gs`, any reference `r` reachable from any mutator `m` is not in the freelist, and hence has not been collected.

```
Definition I_correct: gpred :=
  fun gs ⇒ ∀ m r, In m Mut ∧ Reachable_from m gs r → ¬ in_freelist gs r.
```

We can now formulate our main theorem. It uses the predicate `reachable_mgc` stating that a global state `gs` can be reached, from a predefined initial state, by the code of the `mgc` shown above.

```
Theorem gc_sound: ∀ gs, reachable_mgc gs → I_correct gs.
```

[6] We present a simplified pseudo-code of the MGC, with variable `x`, field `f`, and value `v` assumed non-deterministically chosen from the thread environment. The actual definition in Coq is an operational characterization of this thread system.

```
// collector ::=
while (true) do
  atomic  // ghost
    stage[C] = CLEAR
    phantom_flipped = 0
  atomic  // linearizable[4]
    foreachObject o do
      if !(isFree?(o)) then
        o.color = WHITE
    od
    phantom_flipped = 1
  handshake()  // SYNCH1
  handshake()  // SYNCH2
  stage[C] = TRACING
  handshake()  // ASYNCH
  trace()
  stage[C] = SWEEPING
  sweep()
  stage[C] = RESTING
od
```

Listing 1: Collector

```
// handshake() ::=
phantom_hdsk = 1
phase[C] = phase[C] + 1 mod 3
foreach (m in Mut) do
  repeat skip
  until phase[m]==phase[C]
od
phantom_hdsk = 0
```

Listing 2: Handshake

```
// tid m : cooperate ::=
if phase[m] != phase[C] then
  if phase[C] == ASYNCH then
    foreachRoot (r of m) do
      markGrey(buffer[m], r)
    od
  phase[m] = phase[C]
```

Listing 3: Cooperate

```
// tid m : update(x,f,v) ::=
if (phase[m] != ASYNCH
     stage[C] == TRACING) then
  old = x.f
  markGrey(buffer[m],old)
  markGrey(buffer[m],v)
x.f = v
```

Listing 4: Write Barrier

```
// markGrey(buffer,x) ::=
if (x != NULL
    && x.color != BLACK) then
     buffer.push(x)
```

Listing 5: MarkGrey

The initial state we consider is obtained by a startup phase of the runtime, that carefully initializes intrinsic features of the runtime, and establishes key invariants.

Evidently, this theorem would be impossible to prove without the aid of other intermediate invariants. In the sequel we explain the important aspects of the implementation, and a few salient auxiliary invariants. Describing the algorithm and our code in full details is out of the scope of this paper. We refer the reader to the explanations in [5] and to the formal proof [7] for details.

High-level Principles of the Algorithm Our GC is of the *mark and sweep* family: the heap is traversed, marking objects that are presumably alive, *i.e.* reachable from mutators local variables, henceforth called *roots*. Once the marking procedure finishes, the sweeping procedure reclaims objects detected as not reachable by putting them back in the freelist.

The marking conventions to denote the reachability of objects follows the tricolor convention [2]. Color `WHITE` is used for objects not yet visited. `GREY` is used for visited, hence presumably live objects, whose children (through fields accesses) have not yet been visited. `BLACK` is used for visited objects whose children have all been visited. In our implementation, colors `WHITE` and `BLACK` are implemented with numerical constants. We explain the encoding of `GREY` later. The heap traversal (marking) procedure is called *tracing*, and completes once no `GREY` objects remain.

Extra care is required to cope with the concurrent execution of mutators: they could modify the object graph at any point, and thus invalidate the properties of the coloring. In particular, mutators are responsible for publishing their own roots by marking them as `GREY` before tracing begins. This is the goal of the `cooperate` procedure. Similarly, object field update should not break color-related reachability invariants during tracing. This is the goal of the so-called *write-barriers*, implemented by the `update` procedure. Finally, the right color should be assigned to newly allocated objects. For space reasons,

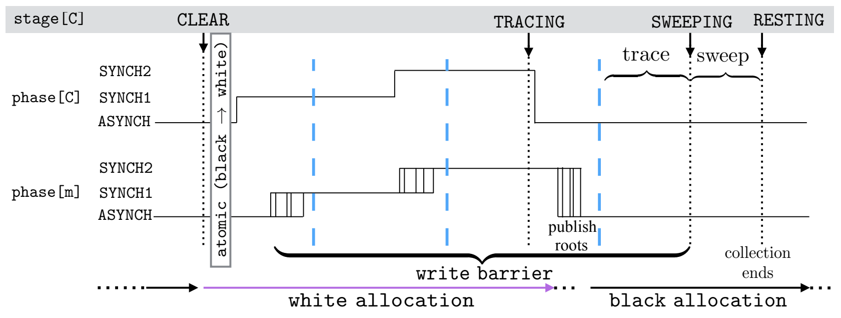

Figure 2: Timeline of a collection cycle. All mutators are coalesced into the bottom line, and the collector is shown in the top line. Dotted lines represent the GC start of a new stage, and dashed lines represent the end of a phase change (handshake).

we elude the details of the `alloc` procedure that we have implemented, and refer to our formalization [7] and the descriptions in [5] for the details.

All these subtle procedures, run by the collector and the mutators, are orchestrated using the global variable `stage[C]`, which encodes for the various stages of the collection cycle (including the tracing and sweeping), and the global variables `phase[m]` – one for each mutator – and `phase[C]` – one for the collector – to coordinate mutators with the collector. A diagrammatic representation of a collection cycle is shown in Figure 2, gathering all previously mentioned ingredients. We will refer to it below in more detail.

RtIR **Implementation and Main Invariants** Let us now describe the implementation. Code snippets in RtIR use a simplified syntax from the one we presented above for space and readability reasons.

Stage and Phase Protocol. The code of the collector is presented in Listing 1. For the moment, we concentrate only on the calls to the `handshake()` procedure (Listing 2), and its counterpart `cooperate()` (Listing 3) executed by the mutators. A collection cycle is structured using four stages: `CLEAR`, `TRACING`, `SWEEPING` and `RESTING`. The current stage is written by the collector to a global variable `stage[C]`. This global variable allows mutators to coordinate with the collector at a coarse level. At a finer level, a handshake mechanism is required, and the status of each thread, the mutators and the collector, is tracked with a `phase` variable, with values ranging over `ASYNCH`, `SYNCH1` or `SYNCH2`. Each phase is encoded with a dedicated integer between 0 and 2. Instead of presenting a detailed description to justify these phases, let us point out that the original algorithm of [4] used only two phases, which was later discovered to be incorrect. A new phase was added to correct it in [3].

We concentrate now on the horizontal lines of Figure 2, showing the evolution of `phase[C]`, as well as the aggregated representation of all the `phase[m]` variables of mutators. Each phase starts by the collector modifying the `phase[C]` variable (second line of Listing 2). Mutators query it (first line of Listing 3), to acknowledge possible changes, in which case mutators *respond* by updating their own `phase[m]` variable (the last line of Listing 3). When the collector acknowledges that all mutators have up-

```
// trace() ::=
all_empty = false
while (!all_empty) do
  atomic // ghost code
    foreach (m in Mut) do
      phantom_buffer[m].copy(buffer[m])
    od
  all_empty = true
  foreach (m in Mut) do
    is_empty = buffer[m].isEmpty()
    while (!is_empty) do
      all_empty = false
      x = buffer[m].top()
      if (x.color == WHITE) then
          buffer[C].push(x)
          buffer[m].pop()
      else buffer[m].pop()
      is_empty = buffer[m].isEmpty()
    od
  od

  while (!buffer[C].isEmpty()) do
    all_empty = false
    ob = buffer[C].top()
    if (ob.color == WHITE) then
      foreachField (f of ob) do
        if (ob.f!=NULL
            && ob.f.color==WHITE) then
          buffer[C].push(ob.f)
      od
      ob.color = BLACK
    buffer[C].pop()
  od
od


// sweep() ::=
foreachObject o do
  if (!isFree?(o) && o.color == WHITE) then
    free(o)
od
```

Listing 6: Trace and Sweep (Collector)

dated `phase[m]`, the phase transition is completed (dashed line in Figure 2). Importantly, `phase[C]` and `phase[m]` are subject to race conditions. We also point out that threads do never stop their execution while executing `cooperate`.

An important invariant relating the phases of the collector and the mutators is that any mutator's phase is at most one step behind the collector's phase.

```
Definition I_phases : gpred := fun gs ⇒
∀ m, In m Mut → phase[C]gs = phase[m]gs ∨ phase[C]gs=(phase[m]gs+1) mod 3.
```

Buffers and `GREY`. Objects are marked `GREY` with the `markGrey` procedure (Listing 5) when mutators publish their roots (Listing 3) and during the write barriers (Listing 4). Each mutator owns a `buffer[m]` abstract data structure, in which it adds references to be traced. Hence, `buffer[m]` serves as an interface between mutators and the collector to mark objects as `GREY`. In other words, an object is considered `GREY` if it is present in any `buffer` and its color field is `WHITE`. In this sense, `GREY` is a convention rather than a constant like `BLACK` or `WHITE`.

Write barriers. Their code is shown in Listing 4. The barrier will conditionally either directly update the field `f` (fast-path) or `markGrey` two objects (slow-path).[7] Notice that the slow-path of the write barrier is only executed when the collector is ready to start tracing, and not after it starts sweeping (see Figure 2). The code of write barriers is intrinsically racy since the client code itself might contain races at the field; moreover, the accesses to the `buffer` data structures are not protected by synchronization between mutators and the collector. Finally, we emphasize that the order in which the `markGrey` operations are performed in the write barrier is critical to the GC correctness.

Trace This is the most challenging code to verify, and its verification by means of program logics would be remarkably hard without some of the design choices of RtIR, and our proof methodology.

[7] The write barrier in [5] avoids marking `old` in some cases. We drop this optimization.

The `trace` procedure (Listing 6) traverses the object graph starting from `GREY` objects. More precisely, the collector visits each of the mutators `buffer[m]` in the `foreach` loop at Line 9, transferring their contents into its own `buffer[C]`. If the collector sees empty buffers for all mutators, tracing ends. Otherwise, it traverses the graph starting from objects in `buffer[C]`, and marking `BLACK` objects whose children have been seen.

Regarding the complexity of the code, we emphasize that it contains three nested loops, a number of `foreach` constructs, and heavily uses the buffer abstract data structures. Moreover, it exhibits races in all threads (through write barriers and buffer operations) since it traverses the object graph, while mutators concurrently modify it.

An important invariant establishes that during the tracing phase, any `WHITE` object that is alive must be reachable from a `GREY` object, signaling that it still has to be visited. Since another invariant, `I_black_to_white`, states that any path from a `BLACK` object to a `WHITE` object goes through a `GREY` object, this translates to the property that all objects reachable from the roots are either `BLACK`, or reachable from a `GREY` one.

```
Definition I_trace_grey_reach_white : gpred := fun gs ⇒ ∀ m r,
stage[C]gs≠CLEAR ∧ In m Mut ∧ phase[m]gs=ASYNCH ∧ Reachable_from m gs r→
 Black gs r ∨ (∃ r0, Grey Mut gs r0 ∧ reachable gs r0 r).
```

When this code terminates, we are able to prove that: (i) there are no more `GREY` objects, (ii) all objects reachable from the mutators roots are `BLACK`, and consequently (iii) there are no `WHITE` objects reachable from any of the mutators roots.

Property (i), namely that all buffers are *simultaneously* empty at the end of tracing (Listing 6, Line 35), is particularly difficult to prove, given the write barriers executed concurrently by mutators. We prove that this property is established at Line 4 of the last iteration of the enclosing while loop. We proceed as follows. We first prove that, at Line 4, `buffer[C]` is always empty. As for mutators' buffers, we use ghost variables `phantom_buffer[m]` to take their snapshot at Line 4. Mutators can only push on their buffers, so, in a given iteration of the enclosing while loop, if a mutator buffer is empty, so was its ghost counterpart during the same iteration. In the last iteration of the while loop, all buffers are witnessed empty, one at a time. But this implies that all phantom buckets are simultaneously empty at Line 8. This, in turn, implies that all buffers are, this time *simultaneously*, empty at Line 4. This property remains true until Line 35: it is both stable under mutators' guarantees, and preserved by the while loop. Indeed, if all buffers are empty (there are no `GREY` objects), the above invariant `I_trace_grey_reach_white` implies that both the old and new objects that `markGrey` could push on a buffer are in fact `BLACK`, and thus not pushed on any buffer (Listing 5). As a consequence, no reference is pushed on the collector's buffer (Line 15).

Sweep The sweep phase (Listing 6) recycles all the objects that remain `WHITE` after `TRACING`. This is the only place where instruction `free` is ever used. Note that this code is also non-blocking. A key property, whose proof we have sketched above, is that during sweeping, no `GREY` objects remains. Formally,

```
Definition I_sweep_no_grey : gpred := fun gs ⇒
  (stage[C]gs = SWEEPING ∨ stage[C]gs = RESTING) → ∀ r, ¬ Grey Mut gs r.
```

This invariant, with `I_trace_grey_reach_white` above, implies that no `WHITE` object is reachable from any thread-local variable.

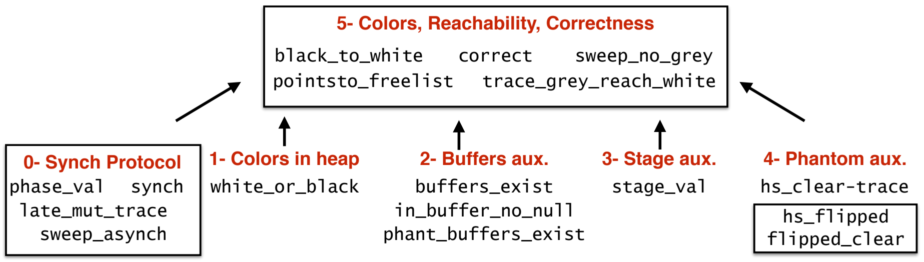

Figure 3: Main Invariants of the GC. Numbers are timestamps in the incremental proof methodology. Dependencies are shown with boxes (inter-dependency) and arrows.

5 Proof methodology

Mechanizing such a sizable proof raises methodological concerns. While the proof system of Section 3 separates proof concerns between sequential reasoning and stability checks, we deal here with the intrinsic complexity of the proof and its scalability.

First, stating upfront the right set of invariants, guarantees, and assertions is unrealistic for such a proof. To tackle this issue, we group invariants related to distinct aspects, *e.g.* the phase protocol or coloring invariants. To reflect this structure in our proof, and avoid constant refactoring of proof scripts, we design an incremental workflow.

Second, we must deal with the quantity of proof obligations. For the GC code, proof obligation `RGp_I` involves 18 invariants, which must be proved stable under 17 guarantees, thus requiring 306 stability proof obligations. On top of this, proof obligation `RGt_stable` adds more than 60 annotated lines of code, each bearing several predicates, that must be proved stable under significant subsets of the 17 guarantees. This becomes quickly intractable without a disciplined methodology and automation.

5.1 Workflow

Figure 3 shows the major invariants of the GC, organized in groups. In each boxed group, invariants are inter-dependent, while arrows indicate a dependency of the target group on the source group.

RG proofs are thread-modular, but RG does not solve the interdependency problem: invariants, guarantees and code annotations are all eventually connected to form the end-result. To maximize proof reuse, we use a simple mechanism: invariants `I` and guarantees `G` are indexed by a natural number – morally a timestamp of their introduction into the development (Figure 3). When introducing a new increment to an invariant, all invariants with a lower timestamp are not modified. Nor are their proofs, resulting in an incremental, non-destructive methodology. More concretely, at each level:

1. we enrich the invariant, refine the guarantees and code annotations;
2. we prove the new stability proof obligations, for which we can reuse prior stability proofs, and we use automation to discharge as many obligations as possible;
3. we adapt sequential Hoare proofs, and prove that enriched guarantees are still valid.

This workflow proved robust during our development, allowing for an incremental and manageable proof effort. We detail below the first two items of this methodology.

5.2 Incremental proofs

Let us focus on obligation `RGp_I` from Section 3, which requires establishing the invariant stability under all threads' guarantees. Let us fix a thread and index both the invariant and guarantee by `n`. The obligation is thus `(stable TTrue (I n) (G n))`. Let us now see how we establish `(stable TTrue (I n+1) (G n+1))` by using the already proved `(stable TTrue (I n) (G n))` obligation.

Monotonicity of `I` *and* `G`. We build `(I n+1)` as a conjunction of prior established invariant `(I n)`, and the increment at the current level: `(I n+1) ≜ (I n) ⩕ (Ic n+1)`. Hence, we have that ∀n, `(I n+1) ⟶ (I n)`.

Recall that in our proof system, guarantees are expressed through the effect of a command, under certain hypotheses on the pre-state. At each level, the command will not change – it is effectively executed by the code. Levels are rather used to refine the hypotheses on the pre-state. Therefore, guarantees are monotonic in the sense that ∀n, `(G n+1) ⊆ (G n)`: they are made more precise as the level index increases.

Reuse of Proof of Prior Invariants. We start by proving that prior invariant `(I n)` is stable under refined guarantee `(G n+1)`, *i.e.* `(stable TTrue (I n) (G n+1))`. To do so, we reuse our previous proofs at level `n` and conclude with the following lemma using guarantee monotonicity – below, we abuse notations and use _ as a valid Coq identifier.

```
Lemma stable_refineG: ∀ _ I G1 G2, G2 ⊆ G1 ∧ stable _ I G1 → stable _ I G2.
```

New Invariant Stability. It remains to prove the stability of increment `(Ic n+1)` under refined guarantee `(G n+1)`. In simple cases, `(stable TTrue (Ic n+1) (G n+1))` is provable independently from prior invariants. In this case, we combine the stabilities of `(Ic n+1)` and `(I n)` into the one of `(I n+1)` with lemma `stable_and`:

```
Lemma stable_and: ∀ _ I1 I2 G,
  stable _ I1 G ∧ stable _ I2 G → stable _ (I1 ⩕ I2) G.
```

However, the situation is often more involved, requiring prior invariants to prove the stability of `(Ic n+1)`. Formally, we have `(stable (I n) (Ic n+1) (G n+1))`. We can then combine the stability of `(I n)` and `(Ic n+1)` under `(G n+1)` using this lemma:

```
Lemma stable_with: ∀ _ I1 I2 G,
  stable _ I1 G ∧ stable I1 I2 G → stable _ (I1 ⩕ I2) G.
```

5.3 Proof Scalability

To tackle the blowup of stability checks alluded to earlier, we built a toolkit of structural stability lemmas, and develop some tactic-based partial automation. This allowed us to discharge automatically 186 obligations among the 306 obligations induced by `RGp_I`. The remaining obligations are also partially reduced by the automation.

Structural lemmas. Structural lemmas serve three purposes. First, they are critical to enable the incremental methodology delineated above. Second, they allow for complex stability proof obligations to be simplified: both annotations, invariants, and interferences can be structurally split up. Thus, intrinsically complex arguments are isolated from trivial ones, that are automatically discharged. Finally, to reuse as much proofs as possible, we rely on a custom notion of stability under extra-hypotheses:

```
Definition stable_hyps (I: gpred) (H P: pred) (R: rg): Prop := ∀ gs1 gs2 l,
 I gs1 ∧ H gs1 l ∧ P gs1 l ∧ R gs1 gs2 ∧ I gs2 ∧ H gs2 l → P gs2 l.
```

Typically, this notion allows to leverage stability results from previous levels, notably through the following lemmas:

```
Lemma stable_weakI: ∀ I1 I2 P G, I2 ⊆ I1 → stable I1 P G → stable I2 P G.
Lemma stable_weakH : ∀ I (H P: pred) R,
    stable I H R → stable_hyps I H P R → stable I (H ⩕ P) R.
```

By decomposing annotations and relaxing interferences, we can factor out the proof of stability of annotations that reappear in the code.

Automation. We developed a set of tactics that simplify stability goals into elementary ones before attempting to solve them. This leads to clearer goals and more tractable proof contexts. The tactics combine our structural lemmas with two additional ideas: systematic inversion on guarantee actions – defined operationally using commands – and rewriting in predicates.

6 Related Work

Concurrent GC. The literature on garbage collection is vast. We refer the reader to [16] for a comprehensive and up-to-date presentation of garbage collection techniques. We use [5] as a starting point. It is a state-of-the-art non-blocking concurrent GC based on the earlier DLG algorithm [4,3]. Many of the invariants we prove are inspired by those of [3].

Mechanized GC proofs. Many prior efforts have tackled the verification of sequential GCs [18,11]. Unfortunately, the addition of concurrency renders these approaches inadequate. Insofar our work could be subsequently integrated into a verified run-time, it is possible to reuse some methodological aspects of [19], such as the structuring in a multi-layer refinement of the garbage collection specification.

The first mechanized proof of a concurrent GC was presented by Gonthier [9]. Unlike ours, Gonthier's proof rests on an abstract encoding of the algorithm. Our development sidesteps this additional modelling step by proving the implementation in RtIR. A similar remark can be made of the approaches in [10,8], which formalize GCs in the PVS and Isabelle/HOL provers respectively.

Liang et al. [17] provide a proof of a mostly-concurrent GC based on the RGSim methodology. While the meta-theory of the logic is mechanized, the proof of the GC itself is not.

Mechanized concurrent program logics. In [21] an RG logic for a simple imperative concurrent language is formalized and proved sound in Isabelle/HOL. In contrast, our program logic is customized for runtime system implementations, and therefore supports local and global environments, references, iterators, etc. Also, the proof rules of [21] mix sequential reasoning with side conditions about stability and guarantee checks. We decouple these aspects and avoid redundancies by extracting relies from the guarantees of the context.

Other approaches to the mechanized verification of concurrent code are [6,24,17,23] to mention but a few. These works are mostly concerned with concurrent data structure correctness, whereas we are concerned with the implementation of a runtime system.

7 Conclusion

This paper presents the mechanized proof of an emblematic challenge in program verification: an on-the-fly concurrent garbage collector. Overcoming this challenge requires a number of methodological advances. We follow a programming language-based approach: a well-chosen intermediate representation, a companion program logic, and a dedicated proof workflow. RtIR strikes a balance between low-level features for the expression of efficient concurrent code, and high-level features which remove the burden of dealing with low-level details in the proofs. Our program logic is inspired by Rely-Guarantee, a milestone in concurrency proof techniques, but one that has heretofore not been used for the mechanized verification of garbage collectors. Our incremental proof workflow, combined with specific and efficient tool support via Coq tactics, is efficient and flexible enough for such a verification challenge.

There are two major avenues for future work. The first is pragmatic, and concerns the embedding of our work in a verified compiler tool chain. Using our theorem about the most-general client, we can build a refinement proof between an IR with implicit memory management and RtIR. We then need to have a fully executable version of the GC. This would require cleaning up ghost code, coding iterators as low-level macros, and implementing abstract concurrent data structures natively supported by RtIR. The two first tasks are essentially administrative. The third task is more challenging, requiring us to formally prove an atomicity refinement result for linearizable, fine-grained data-structures. To that end, we have developed the meta-theory in [26].

The second is methodological. Our proof is the first GC proof to be mechanized using Rely-Guarantee, but it does not take advantage of other tools like Separation Logic [22]. Methods combining RG and Separation Logic exist [25]. It remains to be seen how (or if) these techniques could improve our current proof.

Acknowledgement. We thank the anonymous reviewers and Peter Gammie for their thorough comments and suggestions on how to improve the final version of the paper. We also thank Vincent Laporte for his work earlier in this project, and his help on implementing parts of the garbage collector presented here.

References

1. Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: a buffered memory model for java. In *POPL '13*, pages 329–342, 2013.

2. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.
3. D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proc. of POPL 1994*, pages 70–83, 1994.
4. D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of POPL 1993*, pages 113–123, 1993.
5. T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanover. Implementing an on-the-fly garbage collector for Java. In *Proc. of ISMM 2000*, pages 155–166, 2000.
6. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proc. of POPL 2009*, pages 2–15, 2009.
7. Y. Zakowski et al. Verifying a concurrent garbage collector using an RG methodology, 2017. http://www.irisa.fr/celtique/ext/cgc/.
8. P. Gammie, A. L. Hosking, and K. Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In *Proc. of PLDI 2015*, pages 99–109, 2015.
9. G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Proc . of CAV'96*, pages 462–465, 1996.
10. K. Havelund. Mechanical verification of a garbage collector. In *Proc. of IPPS/SPDP'99*, pages 1258–1283, 1999.
11. C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *Proc of POPL 2009*, pages 441–453, 2009.
12. C Hawblitzel, E. Petrank, S. Qadeer, and S Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proc. of CAV 2015*, 2015.
13. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
14. S. Jagannathan, V. Laporte, G. Petri, D. Pichardie, and J. Vitek. Atomicity refinement for verified compilation. *ACM Trans. Program. Lang. Syst.*, 36(2):6:1–6:30, 2014.
15. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
16. R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.
17. H. Liang, X. Feng, and M. Fu. Rely-Guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Trans. Program. Lang. Syst.*, 2014.
18. A. McCreight, T. Chevalier, and A. P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proc. of ICFP 2010*, pages 273–284, 2010.
19. M. O. Myreen. Reusable Verification of a Copying Collector. In *VSTTE '10*, 2010.
20. F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proc. of PLDI*, 2010.
21. L. Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In *Proc. of ESOP*, 2003.
22. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS 2002*, pages 55–74, 2002.
23. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proc. of PLDI'15*, pages 77–87. ACM, 2015.
24. V. Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276, 2011.
25. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proc. of CONCUR 2007*, pages 256–271, 2007.
26. Y. Zakowski, D. Cachera, D. Demange, and D. Pichardie. Compilation of linearizable data structures - a mechanised RG logic for semantic refinement, 2017. Technical Report, available at https://hal.archives-ouvertes.fr/hal-01538128.