https://doi.org/10.1007/s00165-019-00482-3
The Author(s) © 2019
Formal Aspects of Computing (2019) 31: 353–374

Formal Aspects of Computing

Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example

Cliff B. Jones[1] and Nisansala Yatapanage[1,2]

[1] School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
[2] School of Computer Science and Informatics, De Montfort University, Leicester, UK

Abstract. Decomposing the design (or documentation) of large systems is a practical necessity but finding *compositional* development methods for concurrent software is technically challenging. This paper includes the development of a difficult example in order to draw out lessons about such methods. The concurrent garbage collector development is interesting in several ways; in particular, the final step of its development appears to be just beyond what can be expressed by rely/guarantee relations. This prompts an exploration of the limitations of this well-known method. Although the rely/guarantee approach is used, most of the lessons are more general.

Keywords: Concurrency; Compositional methods; Rely-guarantee; Auxiliary/ghost variable

1. Introduction

The aim of this paper is to contribute to discussion about compositional development for concurrent programs. Much of the paper is taken up with the development, from an abstract specification, of a concurrent garbage collector but the important messages are by no means confined to the example and are identified as *lessons*.

The rely/guarantee approach (see Sect. 1.2 below) provides a compositional development method for many applications. The specific garbage collector algorithm is intricate in the sense that the $Collector$ and $Mutator$ routines were clearly thought out together. The final step of the development of the algorithm challenges the expressiveness of rely/guarantee conditions. Viewed positively, this makes it possible to explore the limits of the method and compare various possible extensions. Furthermore, the example points to a precise test for when auxiliary (or ghost) variables are needed and offers another application of the *possible values* notation (see Sect. 2.1).

Apart from the general lessons, the exploration of what is meant by "compositional" development should interest the reader.

Correspondence and offprint requests to: C. B. Jones, E-mail: cliff.jones@ncl.ac.uk

1.1. Compositional methods

To clarify the notion of "compositional" development of concurrent programs, it is worth beginning with some observations about the specification and design of sequential programs. A developer faced with a specification for S might make the design decision to decompose the task using two components that are to be executed sequentially ($S1$; $S2$); that top-level step can be justified by discharging a proof involving only the specifications of S, $S1$ and $S2$. Moreover, the developer of either of the sub-components need only be concerned with its specification—not that of its sibling nor that of its parent S. This not only facilitates separate development, it also increases the chance that any subsequent modifications are isolated within the boundary of one specified component.

Lesson I "Compositionality" is best understood by thinking about a development process in which, faced with a specified task (module), the developer proposes a decomposition (combinator), specifies sub-tasks and then proves the decomposition correct with respect to (only) the specifications. (The same process is then repeated on the sub-tasks.) Such specifications should genuinely insulate components from one another and from their context.

As far as is possible, the advantages of compositional development should be retained for concurrent programs. Because of the interference inherent in concurrency, compositionality is not easy to achieve and, clearly, (pre/)post conditions will not suffice. However, numerous examples exist to indicate that rely/guarantee conditions (see Sect. 1.2) facilitate the required separation where a designer chooses a decomposition of S into shared-variable sub-components that are to be executed concurrently ($S1 \mid\mid S2$).

1.2. Rely/guarantee thinking

The origin of the rely/guarantee (R/G) work goes back to [Jon81] and was published in [Jon83a, Jon83b]. The basic idea is simple: in addition to pre conditions (predicates over states) and post conditions (relations over initial and final states), information about interference during a specified operation is recorded. Rely conditions record the interference that the operation can tolerate and guarantee conditions record the interference that the operation can inflict on the environment. See Fig. 1. It is important to remember that pre and rely conditions are information to an implementer of a component –they define the contexts in which the final code must run– whereas guarantee and post conditions are obligations on the execution of the code. Some 20 theses have developed the original idea; for example: [Stø90, Xu92] address progress arguments, [Din00] moves in the direction of a refinement calculus form of R/G, [Pre01] provides an Isabelle-checked soundness proof of a slightly restricted form of R/G rules, [Col08] revisits the soundness of general R/G rules, [Pie09] addresses usability and [Vaf07, FFS07] explore ways to combine R/G thinking with Separation Logic. Furthermore, a number of *Separation Logic* (see below) papers also employ R/G reasoning (e.g. [BA10, BA13]) and [DFPV09, DYDG+10] from separation logic researchers build on R/G. Any reader who is unfamiliar with the R/G approach can find a brief introduction in [Jon96].[1]

The literature contains many diverse examples of R/G developments including:

- Susan Owicki's [Owi75] verifies a program that finds the minimum index i to an array A such that $A(i)$ satisfies a given predicate p; a development of such a program is tackled using R/G thinking in [HJC14]
- a staple of R/G presentations is a concurrent version of the *Sieve of Eratosthenes* introduced in [Hoa72]—see for example [JHC15]
- parallel "cleanup" operations in the *Fisher/Galler* algorithm for the so-called *union/find* problem are developed in [CJ00]
- a development of *Simpson's 4-slot algorithm* is given in [JP11]—an even nicer specification using "possible values" (see Sect. 2.1) is contained in [JH16]

The first two are examples in which the R/G conditions are symmetric in the sense that the concurrent sub-processes have the same specifications; the last two items and the concurrent garbage collector presented below are more interesting because the concurrent processes need different specifications.

[1] Fuller sets of references are contained in [HJC14, JHC15].

$$\overbrace{\underbrace{\sigma_0}_{pre} \quad \cdots \quad \underbrace{\sigma_i \ \sigma_{i+1}}_{rely} \quad \cdots \quad \overbrace{\sigma_j \ \sigma_{j+1}}^{guar} \quad \cdots \quad \sigma_f}^{post}$$

Fig. 1. R/G

Lesson II While acknowledging Lesson I, there does have to be some description of acceptable interference. By using relations to express interference, R/G conditions offer a plausible compositional approach to concurrency with a balance of expressiveness versus tractability—see Sects. 4 and 5.

The original way of writing R/G specifications displayed the predicates of a specification delimited by keywords; some subsequent papers (notably those concerned with showing the soundness of the Proof Obligations (POs)) present specifications as five-tuples. The reformulation in [HJC14, JHC15, HJ18] employs a refinement calculus format [Mor90, BvW98] in which it is natural to investigate algebraic properties of specifications. Since some of the predicates for the garbage collection example are rather long, the keyword style is adopted in this paper but algebraic properties (e.g. distributing rely and guarantee over sequential or parallel composition) are used as required.

1.3. Challenges

The extent to which compositionality depends on the expressivity of the specification notation is an issue and the "possible values" notation used in Sect. 2.1 provides an interesting discussion point. Much more telling is the contrast with methods which need the code of sibling processes to reason about interference. For example, the Owicki-Gries approach [Owi75, OG76] not only postpones a final interference freedom (*einmischungsfrie*) check until the code of all concurrent processes is to hand but it also follows that this expensive test has to be repeated when changes are made to any sub-component.

It is useful to distinguish progressively more challenging cases of interference and the impact that the difficulty has on reasoning about correctness:

1. The term "parallel" is often used for threads that share no variables: threads are in a sense entirely independent and only interact in that they overlap in time. Hoare [Hoa72] observes that, in this simple case, the conjunction of the post conditions of the individual threads provides an acceptable post condition for their combination.
2. Over-simplifying, Hoare's insight is a basis for concurrent separation logic (CSL). CSL [O'H07] and the many related logics are, however, aimed at –and capable of– reasoning about intricate heap-based programs. See also [Par10].
3. It is argued in [JY15] that careful use of abstraction can serve the purpose of reasoning about some forms of separation.
4. The interference in the Owicki example referred to in the preceding section is non-trivial because one thread affects a variable used to control repetition in the other thread. It would be possible to reason about the development of this example using "auxiliary" (aka "ghost") variables. The approach in [Owi75] actually goes further in that the code of the combined system is employed in the final *Einmischungsfrei* PO. Using the compositional R/G approach in [HJC14], however, the interference is adequately characterised by relations.
5. There are other examples in which relations alone do not appear to be enough. This is true of even the early stages of development of the concurrent garbage collector below. A notation for "possible values" [JP11, HBDJ13, JH16] obviates the need for auxiliary variables in some cases, see Sect. 2.1
6. The question of whether some examples *require* ghost variables is open and this discussion is resumed in Sect. 5. That their use is tempting in order to simplify reasoning about concurrent processes is attested to by the number of proofs that employ them.

2. Preliminary development

This section builds up to a specification of concurrent garbage collection that is then used as the basis for development in Sects. 3–6. The main focus is on the $Collector$ but, since this runs concurrently with some form of $Mutator$, some assumptions have to be recorded about the latter.

2.1. Abstract specification

It is useful to pin down the basic idea of inaccessible addresses (aka "garbage") before worrying about details of heap storage (see Sect. 2.2) and marking (Sect. 3). The set of addresses ($Addr$) is assumed to be some arbitrary but finite set; it is not to be equated with natural numbers since that would suggest that addresses could have arithmetic operators applied to them. (There is one point in Sect. 5 where it is important that the cardinality of $Addr$ is at least two—this is assumed where needed since it is pointless carrying such a low lower bound through all of the development.) Abstract states contain two sets of addresses: those that are in use ($busy$) and those that have been collected into a $free$ set. It is, of course, an essential property that the sets $busy/free$ are always disjoint. (VDM types are restricted by datatype invariants and the set Σ_0 only contains values that satisfy the invariant.)[2]

$$\begin{array}{lll} \Sigma_0 :: & busy & : Addr\textbf{-set} \\ & free & : Addr\textbf{-set} \end{array}$$

where

$$inv\text{-}\Sigma_0(mk\text{-}\Sigma_0(busy, free)) \quad \triangleq \quad busy \cap free = \{\,\}$$

There can however be elements of $Addr$ that are in neither set—such addresses are to be considered as "garbage" and the task of a garbage collector is to add such addresses to $free$.

Lesson III The use of abstract datatypes can clarify key concepts prior to discussion of implementation details. Implementations are then viewed as "reifications" that achieve the same effect as the abstraction. Formal POs are given, for example, in [Jon90]. This is commonplace for sequential programs but it has yet greater force for concurrent program development (where it is perhaps underemployed by many researchers). For example, in R/G examples such as [JP11], such abstractions make it possible to address interference and separation at early stages of design. Furthermore, in the example below, important issues such as invariant preservation on abstract states yield real insights into the abstract algorithm before they become clouded by details of the representation.

Effectively, the GC process is an infinite loop repeatedly executing the $Collector$ operation whose specification is on the left here:

$Collector$
ext wr $free$
  rd $busy$
pre true
rely $free' \subseteq free \wedge (busy' - busy) \subseteq free$
guar $free \subseteq free'$
post $(Addr - busy) \subseteq \bigcup \widehat{free}$

$Mutator$
ext wr $busy, free$
pre true
rely $guar\text{-}Collector \wedge busy' = busy$
guar $rely\text{-}Collector$
post true

[2] The use of VDM notation should present the reader with no difficulty since it has been widely used for decades and is the subject of an ISO standard; one useful reference is [Jon90]. Comments are however added for the less common notation. For example, X**-set** is a type which admits only finite sets (but since here $Addr$ is finite, power set could have been used). Record types are as in McCarthy's abstract syntax notation [McC66] but in VDM a record definition such as that for Σ_0 implicitly defines a constructor function ($mk\text{-}\Sigma_0$) and selectors ($\sigma.busy$, $\sigma.free$); furthermore, where the constructor appears in a parameter list, it names the values of the fields in an obvious way.

The predicate $guar\text{-}Collector$ reassures the designer of $Mutator$ that a chosen *free* cell will not disappear. Operation specifications (in VDM) define their *frames* which are the state components that an implementation can read (**rd**) or change (**wr**). For concurrent operations, these provide a convenient way of expressing what cannot change (e.g the $Collector$ is not allowed to change the $busy$ data that is at the heart of the $Mutator$ application). Thus the additional conjunct recorded in the rely condition of the $Mutator$ given on the right above is matched by the fact that the $Collector$ has only read access to $busy$ rather than being an explicit conjunct in the guarantee condition of the $Collector$ ($guar\text{-}Collector$). These abbreviations become important below where there are more state components but they must be explicitly expanded for any mechanical proof such as in Isabelle.

The rely condition assures the developer of $Collector$ that –although the $Mutator$ can consume *free* addresses– nothing in the environment of the $Collector$ can flag addresses as being free. A developer of $Collector$ also needs to know that the only place from where the environment acquires new addresses is the set *free*—in other words, the $Mutator$ cannot take garbage and use it directly in $busy$. Notice that the $post\text{-}Mutator$ indicates that it is free to do whatever is required by the application providing its steps respect $guar\text{-}Mutator$.

Given that the $Mutator$ can remove addresses from *free*, recording a post condition for $Collector$ is not quite trivial. In a sequential setting, it would be correct to write:

$$free' = (Addr - busy)$$

but the concurrent $Mutator$ might be removing addresses from the free set so the best that the $Collector$ can promise is to place all addresses that are originally garbage into the free set at some point during execution. This gives rise to the first use of the “possible values” notation in this paper. To cope with the fact that a concurrent $Mutator$ can acquire addresses from *free*, the correct statement is that all unreachable addresses should be members of a value of the variable *free* at some point in the execution of $Collector$. The notation discussed in [JP11, HBDJ13, JH16] for the set of possible values is $\widehat{free}$. Because the values of *free* are sets (of $Addr$), $post\text{-}Collector$ takes the distributed union of $\widehat{free}$.

The POs requiring that the guarantee conditions of each process imply the rely condition of the other process are, at this stage, trivial. Strictly, there is in VDM a “satisfiability” PO that requires that, for any valid initial state, it is possible to construct a final sate that satisfies the post condition. This is vacuously true for the $Mutator$ and –as is often the case– proving this for the $Collector$ is essentially done by constructing an implementation.

Lesson IV The “possible values” notation is a useful addition to the R/G style of specification. Since the values of variables that are visible to concurrent threads is a natural consequence of concurrency, some notation for possible values is likely to be of wider use.

2.2. The heap

This section introduces a model of the heap. The set of addresses that are busy can now be defined to be those that are reachable from a set of $roots$ by tracing all of the pointers in a heap (hp).

$$\begin{array}{lll} \Sigma_1 :: & roots & : Addr\textbf{-set} \\ & hp & : Heap \\ & free & : Addr\textbf{-set} \end{array}$$

where

$$inv\text{-}\Sigma_1(mk\text{-}\Sigma_1(roots, hp, free)) \quad \triangleq \quad free \cap reach(roots, hp) = \{\}$$

The invariant now defines the upper bound for garbage collection by saying that in no state should a reachable node appear in *free*. (It is assumed that $roots \neq \{\}$.) Because neither $Collector$ nor $Mutator$ has write access to $roots$, it remains constant (which is not recorded in the rely conditions). The $Heap$ links an address to the addresses to which it can be considered to point. This is modelled as a mapping from $Addr$ and an index to

$Addr$.[3] The set $Index$ is deliberately undefined—were it natural numbers, it would possible to think of mapping addresses to lists of addresses. The chosen model can be thought of as lists with gaps.

$$Heap = (Addr \times Index) \xrightarrow{m} Addr$$

The $reach$ function computes the relational image (with respect to its first argument) of the transitive closure (R^*) of the heap:

$$reach : Addr\textbf{-set} \times Heap \to Addr\textbf{-set}$$

$$reach(s, hp) \quad \triangleq \quad rel\text{-}image(child\text{-}rel(hp)^{\star}, s)$$

The following is a definition of the relational image operator (which is not part of standard VDM).

$$rel\text{-}image : (A \times B)\textbf{-set} \times A\textbf{-set} \to B\textbf{-set}$$

$$rel\text{-}image(r, s) \quad \triangleq \quad \{b \mid \exists a \in s \cdot (a, b) \in r\}$$

The $child\text{-}rel$ function extracts the relation over addresses from the heap (i.e. ignoring indexed positions).[4]

$$child\text{-}rel : Heap \to (Addr \times Addr)\textbf{-set}$$

$$child\text{-}rel(hp) \quad \triangleq \quad \{(a, b) \mid \exists a, b \in Addr \cdot b \in kids(a, hp)\}$$

The function $kids$ is obvious but notice that $kids(a, hp) = \{\,\}$ if there is no position i such that $(a, i) \in \textbf{dom}\, hp$,

$$kids : Addr \times Heap \to Addr\textbf{-set}$$

$$kids(a, hp) \quad \triangleq \quad \{hp(a, i) \mid \exists i \in Index \cdot (a, i) \in \textbf{dom}\, hp\}$$

A useful lemma states that starting from some set s, if there is an element a reachable from s that is not in s, then there must exist a path that contains an address not in s (but notice that $hp(b, j)$ might not be a).

Lemma 1

$$(\exists a \cdot a \in reach(s, hp) \wedge a \notin s) \;\Rightarrow\; \exists (b, j) \in \textbf{dom}\, hp \cdot b \in s \wedge hp(b, j) \notin s$$

Proof This can be proved by induction on the number of steps (over hp) from the set s to the $Addr$ a. □

The argument that this reification gives the same behaviour as in Sect. 2.1 is based on:

$$retr_0 : \Sigma_1 \to \Sigma_0$$

$$retr_0(mk\text{-}\Sigma_1(roots, hp, free)) \quad \triangleq \quad mk\text{-}\Sigma_0(reach(roots, hp), free)$$

[3] In VDM $D \xrightarrow{m} R$ is a type whose values are finite functions from D to R; the domain of a map m is written $\textbf{dom}\, m$; the only unusual operator used below is $s \mathbin{⩤} m$ whose value is all of the pairs in m whose first elements are not in the set s.

[4] Several alternative modelling decisions were considered. For example, viewing the $Heap$ as a relation would simplify notation but it was felt that the notions of one node pointing more than once to the same $Addr$ and the need to destroy links should be represented explicitly. Also it is tempting to make the $free$ pointer one of the $roots$ because it merges operations—this was not done because it is useful to distinguish the $Malloc$ and $Redirect$ operations (see Sect. 4.3 below).

VDM's data reification POs require that the representation is adequate in the sense that there exists an element of the more concrete type that corresponds (under the retrieve function) to any element of the abstract type. This is technically important because it makes it possible to argue that the concrete and abstract operations commute by quantifying over the concrete type.

Theorem 2 Adequacy

$$\forall \sigma_0 \in \Sigma_0 \cdot \exists \sigma_1 \in \Sigma_1 \cdot retr_0(\sigma_1) = \sigma_0$$

Proof It is easy to construct representative elements of $roots/hp$ corresponding to any $busy$ set. □

The specification of $Collector$ on Σ_1 is:

$Collector$
ext wr $free$
  rd $roots, hp$
pre true
rely $free' \subseteq free \wedge$
  $(reach(roots, hp') - reach(roots, hp)) \subseteq free$
guar $free \subseteq free'$
post $(Addr - reach(roots, hp)) \subseteq \bigcup \overparen{free}$

The invariant $inv\text{-}\Sigma_1$ establishes the upper bound of garbage collection; it is $post\text{-}Collector$ that fixes the lower bound.

The specification of the $Mutator$ is again the reflection of that for the $Collector$ (here, it is the constancy of hp that follows from the read access in the $Collector$.

The VDM POs for data reification require that each concrete operation commutes (under the retrieve function) with its abstract counterpart.

Theorem 3 The commutativity proof for the $Collector$ simply expands $retr_0$; commutativity for the $Mutator$ it is trivial because the its post condition is **true**.[5]

3. Marking

The intuition behind the garbage collection (GC) algorithm in [BA84] is to mark all addresses reachable over the relation defined by the $Heap$ from $roots$ then sweep any unmarked addresses into $free$. The states Σ_2 add the $marked$ field to Σ_1:

$$\begin{array}{lll} \Sigma_2 :: & roots & : Addr\text{-}\mathbf{set} \\ & hp & : Heap \\ & free & : Addr\text{-}\mathbf{set} \\ & marked & : Addr\text{-}\mathbf{set} \end{array}$$

where

$$\begin{array}{l} inv\text{-}\Sigma_2(mk\text{-}\Sigma_2(roots, hp, free, marked)) \quad \triangleq \\ \quad free \cap reach(roots, hp) = \{\,\} \wedge \\ \quad (roots \cup free) \subseteq marked \end{array}$$

[5] The advantage of a careful layering of abstractions is that most POs turns out to be relatively straightforward to discharge—see Sect. 7 for plans to check the proofs with Isabelle [NPW09].

In addition to bringing forward from $inv\text{-}\Sigma_1$ the need to keep used and *free* addresses disjoint, $inv\text{–}\Sigma_2$ requires that all addresses in $(roots \cup free)$ are always marked. The adequacy PO is trivial with a retrieve function that simply drops the *marked* field of Σ_2 but in this proof –and when proving satisfiability of the operations– the type limitation given by the invariant must be respected.

For R/G, the interesting issues arise where the garbage collection runs concurrently with a *Mutator* which can both take *free* addresses (linking them into the set reachable from *roots*) and give rise to garbage that is no longer accessible from *roots*. A fully concurrent garbage collector is covered below (see Sects. 4 and 5). This section introduces code that can be viewed as sequential in the sense that the *Mutator* would have to pause. Importantly this same code satisfies specifications for two concurrent situations that are both more challenging and interesting.

3.1. Sequential algorithm

Without interference, *rely-Collector* says that the none of the state variables are changed. The POs required to prove that code satisfies the restricted specification illustrate some points that are useful below; in particular, there are some interesting observations to be made about loop reasoning. The simpler setting also clarifies terminology (e.g. the upper bound for garbage follows from the lower bound for marking).

The *Collector* can be split into three phases. Providing the invariant is respected, the initial marking is not critical but, thinking of the *Collector* being run intermittently, it is reasonable to start by removing any stale marks.

$$Collector \triangleq (Unmark;\ \ Mark;\ \ Sweep)$$

The operation names are decorated below with subscripts to distinguish the sequential (here), atomic (Sect. 4) and truly concurrent (Sect. 5) versions; in this section, the subscript s marks them as sequential versions.

$Unmark_s$
ext wr *marked*
  rd *roots*, *free*
pre true
rely $marked = marked' \wedge free' = free$
guar true
post $marked' = (roots \cup free)$

$Mark_s$
ext wr *marked*
  rd *hp*
pre true
rely $marked' = marked \wedge hp' = hp$
guar true
post $marked' = reach(marked, hp)$

$Sweep_s$
ext wr *free*
  rd *marked*
pre $marked = reach((roots \cup free), hp)$
rely $marked' = marked \wedge free' = free$
guar true
post $free' = free \cup (Addr - marked)$

```
$Mark \triangleq$
    repeat
        $mc \leftarrow$ card $marked$;
        $Propagate$
    until card $marked = mc$

$Propagate \triangleq$
    $consid \leftarrow \{\}$;
    do while $consid \neq Addr$
        let $x \in (Addr - consid)$ in
        if $x \in marked$ then $Mark\text{-}kids(x)$
        else skip
        fi;
        $consid \leftarrow consid \cup \{x\}$
    od
```

Fig. 2. Code for *Mark*

The specification of $Sweep_s$ illustrates the point in Lesson III about important issues being addressed before they are obfuscated by details. The taxing part of the specification of $Sweep_s$ is the preservation of $inv\text{-}\Sigma_2$; in particular the separation of *free* from addresses reachable from *roots*. For this operation, the pre condition is vital so the satisfiability PO is spelled out in detail.

Lemma 4

$$\forall \sigma \in \Sigma_2 \cdot pre\text{-}Sweep_s(\sigma) \Rightarrow \exists \sigma' \in \Sigma_2 \cdot post\text{-}Sweep_a(\sigma, \sigma')$$

Proof Addresses added to $free'$ are disjoint from $reach((roots \cup free), hp)$ by the pre condition. □

In the sequential case, the composition PO only has to establish that the combination of the three sub-operations has the effect required of *Collector*; the fact that $pre\text{-}Sweep_s$ must always be true also has to be established (the other pre conditions are identically **true**).

Theorem 5 The POs for sequential composition are:

$$\forall \sigma, \sigma', \sigma'' \in \Sigma_2 \cdot post\text{-}Unmark_s(\sigma, \sigma') \wedge post\text{-}Mark_s(\sigma', \sigma'') \Rightarrow pre\text{-}Sweep_s(\sigma'')$$

$$\forall \sigma, \sigma', \sigma'', \sigma''' \in \Sigma_2 \cdot$$
$$post\text{-}Unmark_s(\sigma, \sigma') \wedge post\text{-}Mark_s(\sigma', \sigma'') \wedge post\text{-}Sweep_s(\sigma'', \sigma''') \Rightarrow post\text{-}Collector(\sigma, \sigma''')$$

Proof Without interference, the proofs are straightforward. For the second, substitute $\sigma'''.free$ for $\overleftarrow{free}$ in $post\text{-}Collector$. □

The main interest is in the marking phase. As shown in Fig. 2, the outer loop propagates a wave of marking over the *hp* relation; it iterates until no new addresses are marked.[6] The inner *Propagate* iterates over all addresses[7]: for each address that is itself marked, all of its children are marked.

In the case when the code runs without interference, R/G reasoning is not required: the specification of $Mark_s$ and proof that the code in Fig. 2 satisfies that specification are straightforward. (In fact, they are simplified cases of what follows in Sect. 4.) When the same code is placed in environments that admit interference, R/Gs and different POs are needed (see Sects. 4 and 5). The relation between the three sets of R/G conditions is itself enlightening.

The outer loop (cf. Fig. 2) propagates a wave of marking over the *hp* relation; the specification of the body of the outer loop is:

[6] Notation: **card** s gives the cardinality of set s.

[7] It would be more elegant to write:

$Propagate$: **for all** $x \in Addr$ **if** $x \in marked$ **then** $Mark\text{-}kids(x)$ **else skip fi**

but the set *consid* is useful to express some assertions below. Using the more implicit repetition construct would require a special PO.

$Propagate_s$
ext wr $marked$
  rd hp
pre true
rely $marked' = marked \land hp' = hp$
guar true
post $marked' = marked \cup \bigcup\{kids(a, hp) \mid a \in marked\}$

To establish the lower marking bound (i.e. must mark everything that is reachable from $roots$), a *to-end* induction is employed;[8] essentially the $to\text{-}end_s$ relation states that the remaining iterations of the loop will mark everything reachable from what is already marked:

$to\text{-}end_s : \Sigma_2 \times \Sigma_2 \to \mathbb{B}$

$$\begin{aligned} & to\text{-}end_s(mk\text{-}\Sigma_2(roots, hp, free, marked), mk\text{-}\Sigma_2(roots', hp', free', marked')) \quad \triangleq \\ & \quad roots' = roots \land hp' = hp \land free' = free \land \\ & \quad marked' = marked \cup reach(marked, hp) \end{aligned}$$

Lemma 6 Thus:

$$\begin{aligned} & \forall \sigma, \sigma', \sigma'' \in \Sigma_2 \cdot \\ & \quad post\text{-}Propagate_s(\sigma, \sigma') \land \mathbf{card}\, \sigma.marked < \mathbf{card}\, \sigma'.marked \land to\text{-}end_s(\sigma', \sigma'') \Rightarrow \\ & \qquad post\text{-}Mark_s(\sigma, \sigma'') \end{aligned}$$

Proof The proof follows by straightforward substitution. □

The body of the inner loop (cf. Fig. 2) has to satisfy:

$Mark\text{-}kids_s\ (x: Addr)$
ext wr $marked$
  rd hp
pre true
post $marked' = marked \cup kids(x, hp)$

Lemma 7 With:

$so\text{-}far_s : \Sigma_2 \times \Sigma_2 \to \mathbb{B}$

$$\begin{aligned} & so\text{-}far_s(mk\text{-}\Sigma_2(roots, hp, free, marked), mk\text{-}\Sigma_2(roots', hp', free', marked')) \quad \triangleq \\ & \quad roots' = roots \land hp' = hp \land free' = free \land \\ & \quad marked' = marked \cup \bigcup\{kids(a, hp) \mid a \in (marked \cap consid')\} \end{aligned}$$

$$\begin{aligned} & \forall \sigma, \sigma', \sigma'' \in \Sigma_2, x \in Addr \cdot \\ & \quad so\text{-}far_s(\sigma, \sigma') \land x \in (Addr - \sigma'.consid) \land post\text{-}Mark\text{-}kids_s(\sigma', x, \sigma'') \Rightarrow so\text{-}far_s(\sigma, \sigma'') \end{aligned}$$

Proof The proof is straightforward. □

Theorem 8 Finally:

$$\forall \sigma, \sigma' \in \Sigma_2 \cdot so\text{-}far_s(\sigma, \sigma') \land consid' = Addr \Rightarrow post\text{-}Propagate_s(\sigma, \sigma')$$

[8] There is an interesting point here. In many presentations of Floyd-Hoare axioms, post conditions are predicates of a single state. As soon as they are viewed (as in VDM) as relations, it becomes clear that the invariant relation can be composed on the left or the right of the post condition of the body of the loop. Left composition (as in *so-far*) corresponds most closely to standard loop invariants; right composition (as in *to-end*) is convenient where reasoning reflects the remaining computation. This is illustrated in [Jon90] with two versions of computing factorial where the version proved with *to-end* overwrites the initial value.

Proof The proof is immediate because $marked \subseteq Addr$. □

Lesson V Considering the sequential case is useful because the simpler conditions make it possible to note how the rely condition (nothing changes) and the guarantee condition (**true**) need to be changed to handle concurrency.

As becomes clear in the following sub-sections, the interesting facet of the development is that the code for the $Collector$ matches different sets of R/G conditions.

4. Concurrent GC with atomic interference

The complication in the concurrent case is that the $Mutator$ can redirect pointers and this interferes with the marking strategy of the $Collector$. If however the $Mutator$ marks the target address whenever it makes a change, the $Collector$ code of Fig. 2 can be shown to satisfy a revised specification.

The development is tackled in two stages: firstly, this section assumes a $Mutator$ that atomically both redirects a pointer and marks the new target; Sect. 5 shows that even separating the two steps still allows the $Collector$ code of Fig. 2 to achieve the lower bound of marking but the argument is more delicate and indicates a limitation of the expressive power of R/G relations. The argument to establish the upper bound for marking (and thus the lower bound of garbage collection) is separate and is given in Sect. 6.

If the $Mutator$ were able to update and mark atomically, specifications and proofs would be relatively straightforward; although this atomicity assumption is unrealistic, it is informative to compare the simpler R/Gs with those in Sect. 5. As proposed in Sect. 1, the argument is split into a justification of the parallel decomposition (Sect. 4.1) and the decompositions of the $Collector/Mutator$ sub-components; these are addressed in Sects. 4.2 and 4.3 respectively.

4.1. Parallel decomposition

In this section, the operation names have the subscript a to mark that they are linked to the atomicity assumption. Even given the atomicity assumption, an R/G specification of the collector is more complicated than that in Sect. 2.2:

$Collector_a$
ext wr $free, marked$
  rd $roots, hp$
pre true
rely $free' \subseteq free \wedge$
  $(reach(roots, hp') - reach(roots, hp)) \subseteq free \wedge$
  $marked \subseteq marked' \wedge$
  $\forall (a, i) \in \mathbf{dom}\, hp \cdot$
    $hp'(a, i) \neq hp(a, i) \wedge hp'(a, i) \in Addr \;\Rightarrow\; hp'(a, i) \in marked'$
guar $free \subseteq free'$
post $(Addr - reach(roots, hp)) \subseteq \bigcup \overleftarrow{free}$

The third conjunct of $rely\text{-}Collector_a$ tells the developer that the environment will never delete markings. The final conjunct of the rely condition is the key property that (for now) assumes that the environment (i.e. the $Mutator$) simultaneously marks any change it makes to the heap. This, in effect, provides an abstraction of the more complicated case in Sect. 5.

The upper bound of addresses to be collected is constrained by the second conjunct of $inv\text{-}\Sigma_2$. The lower bound for garbage collection requires setting an upper bound for marking addresses; this topic is postponed to Sect. 6.

Theorem 9 The corresponding specification of the $Mutator_a$ is again the reflection of that for $Collector_a$ (with the addition of a rely condition that hp is unchanged). Since the R/G PO for concurrent processes requires that each one's guarantee condition implies the rely condition of the other(s) the result is immediate.

4.2. Developing $Collector_a$

What remains to be done for the $Collector_a$ is to show that its development satisfies its specification (in isolation from that of the $Mutator_a$)—i.e. the decomposition of the the $Collector_a$ into three phases ($Unmark_a$; $Mark_a$; $Sweep_a$) satisfies the specification in Sect. 4.1.

The post condition for the sequential version of $Unmark_s$ in Sect. 3.1 constrains $marked'$ to be exactly equal to $roots \cup free$ but now interference must be considered. The rely condition indicates that the environment can mark addresses so whatever $Unmark_a$ removes from $marked$ could be reinserted. The possible values notation is again deployed so that $post\text{-}Unmark_a$ requires that, for every address outside those required by $inv\text{-}\Sigma_2$, a possible value of $marked$ exists which does not contain that address. So far, so good but this post condition alone would permit an implementation of $Unmark_a$ to first mark an address and then remove the marking; this erroneous behaviour is ruled out by $guar\text{-}Unmark_a$. The rely condition indicates that the $free$ set can also change but, since it can only reduce, this poses no problem. Relaxing the post condition is a classic example of Lesson VI.

Lesson VI A useful R/G development tactic is to split what is an equality in the specification of a sequential component into lower and upper bounds; one of these is often presented as a guarantee condition.

The rely and guarantee conditions of $Collector_a$ are distributed (with appropriate weakening of rely conditions or strengthening of guarantee conditions) over the three sub-components. More subtle is the form of distribution required for the possible values concept: only the third component ($Sweep_a$) has write access to the relevant variable $free$.

$Unmark_a$
ext wr $marked$
  rd $roots, free$
pre true
rely $free' \subseteq free$
guar $marked' \subseteq marked$
post $\forall a \in (Addr - (roots \cup free)) \cdot \exists m \in \widehat{marked} \cdot a \notin m$

The post condition for $Mark_a$ also has to cope with the interference absent from a sequential specification and this requires more thought. In the sequential case, $post\text{-}Mark_s$ can use a strict equality to require that all reachable nodes are added to $marked$ but here the equality is split into a lower and upper bound. The lower bound for marking is crucial to preserve the upper bound of garbage collection (see the first conjunct of $inv\text{-}\Sigma_2$). This lower bound is recorded in $post\text{-}Mark_a$. (The use of hp' is, of course, challenging but the post condition is stable [CJ07, WDP10] under the rely condition—lost links can't be traced—$free$ is the only souce of new addresses.) The "loss" (from the equality in the sequential case) of the other containment is compensated for by setting an upper bound for marking (see $no\text{-}mog$ in Sect. 6).

$Mark_a$
ext wr $marked$
  rd hp
pre true
rely $rely\text{-}Collector_a$
guar $marked \subseteq marked'$
post $reach(marked, hp') \subseteq marked'$

Similar observations to those for $Unmark_a$ relate to the specification of $Sweep_a$ which becomes:

$Sweep_a$
ext wr $free$
  rd $marked$
pre $reach((roots \cup free), hp) \subseteq marked$
rely $marked \subseteq marked' \wedge free' \subseteq free$
guar $free \subseteq free'$

$\textbf{post}\ marked \cap (free' - free) = \{\} \wedge$
$\quad \forall a \in (Addr - marked) \cdot \exists f \in \widehat{free} \cdot a \in f$

Since any operation also has to respect the state invariant, $Sweep_a$ has to mark the newly freed addresses.

Theorem 10 The sequential composition POs are similar to those in Theorem 5.

$$\forall \sigma, \sigma', \sigma'' \in \Sigma_2 \cdot post\text{-}Unmark_a(\sigma, \sigma') \wedge post\text{-}Mark_a(\sigma', \sigma'') \Rightarrow pre\text{-}Sweep_a(\sigma'')$$

$$\forall \sigma, \sigma', \sigma'', \sigma''' \in \Sigma_2 \cdot$$
$$post\text{-}Unmark_a(\sigma, \sigma') \wedge post\text{-}Mark_a(\sigma', \sigma'') \wedge post\text{-}Sweep_a(\sigma'', \sigma''') \Rightarrow post\text{-}Collector(\sigma, \sigma''')$$

Proof Even with (atomic) interference, these proofs are straightforward. □

Turning to the decomposition of $Mark_a$ to an iteration (see Fig. 2), in order to prove $post\text{-}Mark_a$, a specification is needed for $Propagate_a$ that copes with interference:

$Propagate_a$
$\textbf{ext wr}\ marked$
$\quad \textbf{rd}\ hp$
$\textbf{pre true}$
$\textbf{rely}\ rely\text{-}Collector_a$
$\textbf{guar}\ marked \subseteq marked'$
$\textbf{post}\ (\forall a \in marked \cdot kids(a, hp) \subseteq marked') \wedge$
$\quad (marked = marked' \Rightarrow reach(marked, hp') \subseteq marked')$

The first conjunct of the post condition indicates the progress required of the wave of marking but has to be weakened (in comparison to $post\text{-}Propagate_s$) to a containment because the $Mutator$ can add to $marked$. The second conjunct records the fact that, if no marks are added in a pass, all required marking has been done. This ensures that the outer loop terminates.

To prove the lower marking bound (i.e. must mark everything that is reachable from $roots$), an argument is again used that composes on the right a relation that expresses the rest of the computation: essentially the *to-end* relation states that the remaining iterations of the loop will mark everything reachable from what is already marked:

$to\text{-}end_a : \Sigma_2 \times \Sigma_2 \to \mathbb{B}$

$to\text{-}end_a(mk\text{-}\Sigma_2(roots, hp, free, marked), mk\text{-}\Sigma_2(roots', hp', free', marked')) \ \triangleq$
$\quad roots' = roots \wedge hp' = hp \wedge free' = free \wedge$
$\quad reach(marked, hp') \subseteq marked'$

Lemma 11 The PO is:

$$post\text{-}Propagate_a(\sigma, \sigma') \wedge \sigma'.marked \neq \sigma.marked \wedge to\text{-}end_a(\sigma', \sigma'') \Rightarrow post\text{-}Mark_a(\sigma, \sigma'')$$

Proof The proof is straightforward. □

The termination argument follows from there being a limit to the markable elements: a simple upper bound is **dom** hp but there is a tighter limit (cf. Sect. 6).

Pursuing the decomposition of $Propagate_a$ to a nested iteration (again, see Fig. 2) needs an adjusted specification of the inner operation:

$Mark\text{-}kids_a\ (x{:}\ Addr)$
$\textbf{ext wr}\ marked$
$\quad \textbf{rd}\ hp$
$\textbf{pre true}$
$\textbf{rely}\ rely\text{-}Collector_a$
$\textbf{guar}\ marked \subseteq marked'$
$\textbf{post}\ kids(x, hp) \subseteq marked'$

In this case, the proof is more conventional and a relation that expresses how far the marking has progressed is composed on the left:

$$so\text{-}far_a : \Sigma_2 \times \Sigma_2 \to \mathbb{B}$$

$$\begin{aligned}&so\text{-}far_a(mk\text{-}\Sigma_2(roots, hp, free, marked), mk\text{-}\Sigma_2(roots', hp', free', marked')) \quad \triangleq\\ &\quad roots' = roots \wedge hp' = hp \wedge free' = free \wedge\\ &\quad \forall a \in (marked \cap consid') \cdot kids(a, hp) \subseteq marked'\end{aligned}$$

Lemma 12 The relevant PO is:

$$\begin{aligned}&\forall \sigma, \sigma', \sigma'' \in \Sigma_2 \cdot\\ &\quad so\text{-}far_a(\sigma, \sigma') \wedge\\ &\quad x \in (Addr - \sigma'.consid) \wedge post\text{-}Mark\text{-}kids_a(\sigma', x, \sigma'') \wedge \sigma''.consid = \sigma'.consid \cup \{x\} \;\Rightarrow\\ &\qquad\qquad\qquad\qquad so\text{-}far_a(\sigma, \sigma'')\end{aligned}$$

whose discharge is obvious.

Theorem 13 The final obligation is to show:

$$so\text{-}far_a(\sigma, \sigma') \wedge \sigma'.consid = Addr \;\Rightarrow\; post\text{-}Propagate_a(\sigma, \sigma')$$

Proof The first conjunct of $post\text{-}Propagate_a$ is straightforward; the fact that (unless the marking process is complete) some marking must occur in this iteration of $Propagate_a$ follows from Lemma 1. □

4.3. Checking $Mutator_a$

The $Mutator$ is viewed as an infinite loop non-deterministically selecting one of $Redirect$, $Malloc$ and Zap as specified below. At this stage, these are viewed as atomic operations so no R/Gs are supplied here: their respective post conditions must be shown to imply $rely\text{-}Mark_a$:

$Redirect\ (a\colon Addr, i\colon Index, b\colon Addr)$
ext wr $hp, marked$
pre $\{a, b\} \subseteq reach(roots, hp)$
post $hp' = hp \dagger \{(a, i) \mapsto b\} \wedge marked' = marked \cup \{b\}$

Lemma 14 Since b was reachable, it follows trivially that:

$$post\text{-}Redirect(\sigma, \sigma') \;\Rightarrow\; guar\text{-}Mutator_a(\sigma, \sigma')$$

For this atomic case, the code could be written using multiple assignment and marking atomic execution with $< \cdots >$ as follows:

$$< hp(a, i), marked := \; b, marked \cup \{b\} >$$

In addition to making the free address b the new value of $hp'(a, i)$, the $Malloc$ operation cleans up the addresses reachable from the free chain element that is about to be used (notice also that b cannot be the same as a because of $inv\text{-}\Sigma_2$).

$Malloc\ (a\colon Addr, i\colon Index, b\colon Addr)$
ext wr $hp, free$
pre $a \in reach(roots, hp) \wedge b \in free$
post $hp' = \{(b, j) \mid j \in Index\} \triangleleft (hp \dagger \{(a, i) \mapsto b\}) \wedge free' = free - \{b\}$

$Malloc$ preserves the invariant because $inv\text{-}\Sigma_2$ insists that free addresses are always marked.

Lemma 15 It follows trivially that:

$$post\text{-}Malloc(\sigma, \sigma') \;\Rightarrow\; guar\text{-}Mutator_a(\sigma, \sigma')$$

Zap removes a given pointer from the heap:

$Zap\ (a\colon Addr, i\colon Index)$
ext wr hp
pre $a \in reach(roots, hp) \wedge (a, i) \in$ **dom** hp
post $hp' = \{(a, i)\} \lessdot hp$

Lemma 16 It again follows trivially that:

$$post\text{-}Zap(\sigma, \sigma') \;\Rightarrow\; guar\text{-}Mutator_a(\sigma, \sigma')$$

The old value of $hp(a, i)$ can become garbage if there are no other reachable pointers to the value.

5. Relaxing atomicity

The remaining challenge is to consider the impact of removing the unrealistic atomicity assumption about $Mutator_a$ in Sect. 4.2. Splitting the atomic assignment on the two shared variables $hp, marked$ in

$$< hp(a, i), marked := \; b, marked \cup \{b\} >$$

turns out to be delicate. The difficulty derives from the fact that the marking process is clearly designed so that the collector and mutator collaborate. This makes meaningful separation (see Lesson I) extremely challenging; some form of global argument is difficult to avoid. However, facing that challenge and looking at alternative extensions to R/G thinking is informative and minimising this global argument is interesting. (In this section, the subscript c on operations and their predicates marks the fact that they cover true concurrency.)

It is worth first disposing of a non-solution. One might think that performing the marking first would be safe but Scenario A provides a counter-example that shows that this would not work.

Scenario A *Suppose $Collector_c$ executes $Unmark_c$ immediately after $Redirect$ marks $hp(a, i)$ (but before it changes $hp(a, i)$ to point to, say, b). If the $Collector_c$ moves on to its $Mark_c$ phase and gets as far as a before $Mutator_c$ resumes, a can be added to $consid$ before the pending update $hp(a, i) \leftarrow b$ potentially introduces a link that fails to get the b-rooted structure marked. This could result in active heap data being collected as garbage.*

Having dismissed that ordering, the task is to show that the ordering:

$$< hp(a, i) \leftarrow b >;$$
$$< marked \leftarrow marked \cup \{b\} >$$

is in fact safe. The difficulty with justifying the split of the larger atomic statement can be understood by considering the following scenario.

Scenario B *$Redirect$ can, at the point that it changes $hp(a, i)$ to point to some address b, go to sleep before performing the marking on which the $Collector_a$ of Sect. 4.2 relies. There is in fact no danger because, even if b was not marked by $Redirect$, there must be another path to b (see $pre\text{-}Redirect$ in Sect. 4.3) and the $Collector_a$ should perform the marking when that path (say $hp(c, j)$) is encountered. Were it the case, however, that $hp(c, j)$ could be destroyed before $Collector_a$ gets to c, an incomplete marking would result that could cause live addresses to be collected as garbage. What saves the day is that the $Mutator_c$ cannot make another change without waking up and marking b. (This rules out multiple $Mutator$ threads.)*

For the general lessons that this example illustrates, the interesting conclusion is that there appears to be no way to maintain full compositionality (i.e. expressing everything that needs to be known about the mutator) with standard rely relations. The three step argument in Scenario B pinpoints the limitation of using two state relations in R/G reasoning. This section explores three alternative approaches for enhancing standard R/G thinking so as to be able to cope with the example in hand: Sect. 5.1 shows how an auxiliary variable can be used to overcome the limitation of R/G expressiveness—this serves as a reference point for the next two approaches; Sect. 5.2 discusses an alternative that suggests an extension to R/G; Sect. 5.3 outlines a way of avoiding a shared ghost variable but still, in some sense, uses a non-compositional argument.

5.1. Using an auxiliary variable

Section 4.2 effectively provides a useful abstraction in which many of the problems of concurrent execution of the *Mutator* and *Collector* are solved. The aim now must be to preserve as much as possible of that justification whilst coping with the possibility that the *Mutator* pauses at the worst possible point in its execution.

It might surprise readers who have heard the current authors inveigh against ghost variables that the development in this section does in fact use such a variable (see Lesson VII). There are several alternative choices that could be made for a specific ghost variable.[9] Here the state Σ_2 is extended with a variable *tbm* that can record an address as "to be marked". It streamlines the assertions below for this to be a set with at most one element (empty in the initial state). Essentially *tbm* records the delayed marking that is considered in Scenario B.

The extended state has an extra conjunct in its invariant that establishes the fact that, when *tbm* is set, there must be two paths to the address.

$$
\begin{array}{lll}
\Sigma_3 :: & roots & : Addr\textbf{-set} \\
& hp & : Heap \\
& free & : Addr\textbf{-set} \\
& marked & : Addr\textbf{-set} \\
& tbm & : Addr\textbf{-set}
\end{array}
$$

where

$$
\begin{array}{l}
inv\text{-}\Sigma_3(mk\text{-}\Sigma_3(roots, hp, free, marked, tbm)) \quad \triangleq \\
\quad inv\text{-}\Sigma_2(mk\text{-}\Sigma_2(roots, hp, free, marked)) \wedge \\
\quad \textbf{card}\, tbm \leq 1 \wedge \\
\quad \exists x \in tbm \cdot x \notin marked \;\; \Rightarrow \\
\qquad \exists \{(a, i), (c, j)\} \subseteq \textbf{dom}\, hp \cdot \\
\qquad \{a, c\} \subseteq reach(roots, hp) \wedge (a, i) \neq (c, j) \wedge hp(a, i) = x \wedge hp(c, j) = x
\end{array}
$$

Writing the first atomic step of *Redirect* as $< hp(a, i), tbm := \; b, \{b\} >$ falls foul of an end case where $Redirect(a, i, hp(a, i))$ does not in fact leave a second path. Rather than add an artificial pre condition, the effective **skip** case can be covered by adding the ghost variable to *Redirect* as follows:

$$
\begin{array}{l}
< \textbf{if}\; hp(a, i) \neq b \;\textbf{then}\; hp(a, i), tbm := \; b, \{b\} \;\textbf{fi} >; \\
< marked, tbm := \; marked \cup b, \{\,\} >
\end{array}
$$

Notice that the atomic brackets now only surround one shared variable in each case.

[9] An anonymous referee argued for using a set that contains all of the addresses that will be marked; Sect. 4.2 has in fact created precisely this as an abstraction. The choice here is to record the difference between what is actually marked and what will be marked.

With the non-atomic interference from the $Mutator$, the rely condition used in Sect. 4 is replaced by:

$$rely\text{-}Collector_c : \Sigma_3 \times \Sigma_3 \to \mathbb{B}$$

$$\begin{array}{l} rely\text{-}Collector_c(mk\text{-}\Sigma_3(roots, hp, free, marked, tbm), mk\text{-}\Sigma_3(roots', hp', free', marked', tbm')) \quad \triangleq \\ \quad free' \subseteq free \wedge \\ \quad (reach(roots, hp') - reach(roots, hp)) \subseteq free \wedge \\ \quad marked \subseteq marked' \wedge \\ \quad (\forall (a, i) \in \mathbf{dom}\, hp \cdot \\ \qquad hp'(a, i) \neq hp(a, i) \wedge hp'(a, i) \in Addr \;\; \Rightarrow \\ \qquad\qquad\qquad hp'(a, i) \in marked' \vee tbm' = \{hp'(a, i)\}) \wedge \\ \quad (tbm \neq \{\,\} \wedge tbm' \neq tbm \;\; \Rightarrow \;\; tbm \subseteq marked' \wedge tbm' = \{\,\}) \end{array}$$

The fourth conjunct of $rely\text{-}Collector_c$ records the fact that, if the $Mutator$ has paused before marking $hp'(a, i)$, then tbm' has a note of the address to be marked; the final conjunct ensures that tbm transitions back to the empty set at exactly the point in time when the delayed marking occurs.

Lemma 17 Looking at the non-atomic $Mutator$ argument, the only real challenge is $Redirect$:[10]

Proof The extra conjuncts in $inv\text{-}\Sigma_3$ are preserved by $rely\text{-}Collector_c$. □

Turning to the development of the collector code, this must be justified relying only on the revised $rely\text{-}Collector_c$. The only challenge here is the mark phase whose specification is:

$Mark_c$
ext wr $marked$
  rd $hp, roots, free, tbm$
pre true
rely $rely\text{-}Collector_c$
guar $marked \subseteq marked'$
post $reach(marked, hp') \subseteq marked'$

The code for $Mark_a$ is still that in Fig. 2—under interference, the post condition of $Propagate_c$ has to be further weakened (from Sect. 4.2) to reflect that, if there is an address in tbm, its reach might not yet be marked. Importantly, if the marking is not yet complete, there must have been some node marked in the current iteration:

$Propagate_c$
ext wr $marked$
  rd hp, tbm
pre true
rely $rely\text{-}Collector_c$
guar $marked \subseteq marked'$
post $(\forall a \in marked \cdot kids(a, hp) \subseteq marked' \cup tbm') \wedge$
  $(marked = marked' \;\; \Rightarrow \;\; reach(marked, hp') \subseteq marked')$

Notice that $post\text{-}Propagate_c$ implies there can be at most one address whose marking is problematic; this fact must be established using the final conjunct of the new $rely\text{-}Collector_c$.

The correctness of this loop is interesting—it follows the structure of that in Sect. 4.2 using a *to-end* relation and, in fact, the relation is still:

$$\begin{array}{l} to\text{-}end_c(\sigma, \sigma') \quad \triangleq \\ \quad roots' = roots \wedge hp' = hp \wedge free' = free \wedge \\ \quad reach(marked, hp') \subseteq marked' \end{array}$$

[10] When removing a pointer, no tbm is set—see $Zap(a, i)$ in Sect. 4.3; also no tbm is needed in the $Malloc$ case because $inv\text{-}\Sigma_3$ ensures that any $free$ address is marked.

Lemma 18 The PO is now:

$$post\text{-}Propagate_c(\sigma, \sigma') \wedge \sigma'.marked \subset \sigma.marked \wedge to\text{-}end_c(\sigma', \sigma'') \;\Rightarrow\; post\text{-}Mark_c(\sigma, \sigma'')$$

Proof In comparison with the PO in Sect. 4.2, the difficult case is where $tbm' = \{b\}$ (in the converse case the earlier proof suffices). What needs to be shown is that the stray address b will be marked: $inv\text{-}\Sigma_3$ ensures there is another path to b; this will be marked if there are further iterations of $Propagate$ and these are ensured by Lemma 1 which, combined with the second conjunct of $post\text{-}Propagate$, avoids premature termination. □

The code in Fig. 2 shows how $Propagate$ uses $Mark\text{-}kids_c$ in the inner loop.

$Mark\text{-}kids_c\ (x{:}\ Addr)$
ext wr $marked$
  rd hp, tbm
pre true
rely $rely\text{-}Collector_c$
guar $marked \subseteq marked'$
post $kids(x, hp') \subseteq marked' \cup tbm'$

Again, the POs are as for the atomic case, but with:

$$so\text{-}far_c : \Sigma_3 \times \Sigma_3 \to \mathbb{B}$$

$$\begin{aligned} & so\text{-}far_c(mk\text{-}\Sigma_3(roots, hp, free, marked, tbm), mk\text{-}\Sigma_3(roots', hp', free', marked', tbm')) \quad \triangleq \\ & \quad roots' = roots \wedge hp' = hp \wedge free' = free \wedge \\ & \quad \forall a \in (marked \cap consid') \cdot kids(a, hp) \subseteq marked' \cup tbm' \end{aligned}$$

As is normal, the ghost variable can be removed from the code and the conditional that avoids the **skip** in $Redirect$ can also be erased.

Lesson VII The use of "ghost" (aka "auxiliary") variables presents a danger to compositional development (cf. Lesson I). The case against is clear: in the extreme, ghost variables can be used to dictate the complete detail about the environment of a process. Few researchers would go to this extreme but minimising the use of ghost variables ought be an objective in compositional development.

Lesson VIII Auxiliary variables can undermine compositionality (cf. Lesson VII) because they eliminate the desired separation between sibling processes. Where they are claimed to be essential, it would be useful to have a test for this fact. The need for a "three-state" argument is such a test.

5.2. Exposing the order of steps of a process

This section shows that the auxiliary variable (tbm) of Sect. 5.1 can be avoided at the expense of saying more explicit things about the order of the steps in the mutator. As conceded below, this still limits the separation between the specifications of the collector and the mutator.

Scenario B makes clear that it is necessary to rule out there being another change to the heap between $mr\text{-}1/mr\text{-}2$

$mr\text{-}1$: $< hp(a, i) \leftarrow b >$;
$mr\text{-}2$: $< marked \leftarrow marked \cup \{b\} >$

In Sect. 5.1, there are actually two roles for tbm in the definition of $rely\text{-}Collector_c$ (Sect. 5.1): on the one hand, tbm provides a way to refer to the value of an unmarked $hp'(a, i)$; perhaps less obviously, the transitions between empty and non-empty values of tbm pinpoint the crucial point in the execution between $mr\text{-}1$ and $mr\text{-}2$. Lemma 17 uses tbm to identify the gap and the fact that there exists another path to an $hp'(a, i)$ in such a gap. This fact can be captured using the change in the value of $hp(a, i)$ as follows:

$$\begin{aligned} & \forall (a, i) \in hp \cdot \\ & \quad hp'(a, i) \neq hp(a, i) \wedge hp'(a, i) \in Addr \;\Rightarrow \\ & \qquad hp'(a, i) \in marked' \vee \\ & \qquad \exists (b, j) \in \mathbf{dom}\, hp' \cdot b \in reach(roots, hp) \wedge (b, j) \neq (a, i) \wedge hp'(b, j) = hp'(a, i) \end{aligned}$$

One difficulty with using this relation as a rely condition is that it is local in the sense that it would not hold if $Unmark$ runs. Fortunately it does hold over one incarnation of $Mark_c$ and such local rely conditions have been studied in [JH16]. A second issue is the need to pinpoint that no changes to hp can be made between mr-1/mr-2. The ability to locate assertions of this sort should be possible with RGITL [STER11].

Lesson IX Recording information about the order of steps in the environment is clearly non-compositional.

5.3. Abstracting interference with a predicate

The approaches in Sects. 5.1 and 5.2 rely on information from the mutator to help the designer of the collector to complete proofs. The idea outlined in this section[11] is that the developer of the mutator takes on an extra reasoning task. The crucial observation (cf. Scenario B) is that the ability to complete the marking always holds under interference from the mutator even if $Mutator_c$ stalls at the critical point. The clue as to why this is the case is the two-path property in Sect. 5.2.

A predicate can be defined that expresses the property that marking can be completed (i.e. it states that $Collector_c$ will always be able to mark all active $Addr$s). In essence, the $to\text{-}end_c$ relation of Sect. 5.1 is converted into an invariant.

Thus, in this approach, the designer of the mutator has to reason explicitly about the preservation of this property. In a sense, the designer of the mutator has to reason about the algorithm used in the collector. In contrast to the approach in Sect. 5.1, this avoids sharing tbm—a similar variable is used in $Mutator_c$ but it is strictly local.

Lesson X There are several approaches to reasoning about closely intertwined algorithms. Avoiding shared ghost variables is certainly desirable from a compositional point of view but creating a proof task for one process that relies on the design of its environment is also a reduction of separation.

6. Lower limit of GC

Sections 4 and 5 address (under different assumptions) the lower bound for marking and thus ensure that no active addresses are treated as garbage. Unless an upper bound for marking is established however, $Mark$ could mark every address and no garbage would be collected. The R/G technique of splitting, for example, a set equality into two containments often results in such a residual PO.

Addresses that were garbage in the initial state ($Addr - (reach(roots, hp) \cup free)$) should not be marked (thus any garbage will be collected at the latest after two passes of the collector). A predicate "no marked old garbage" can be used for the upper bound of marking:

$$no\text{-}mog : Addr\textbf{-set} \times Heap \times Addr\textbf{-set} \times Addr\textbf{-set} \to \mathbb{B}$$

$$no\text{-}mog(r, hf, m) \quad \triangleq \quad (Addr - (reach(r, h) \cup f)) \cap m = \{\}$$

The intuitive argument is simple: the $Collector$ and $Mutator$ only mark things reachable from $roots$ and, while the $Mutator$ can change the reachable graph, it only links to addresses (from $free$ or previously reachable from $roots$) that were never "garbage".

7. Related work and conclusions

An extensive book on garbage collection is [JHM16]. There also exist many papers on garbage collection algorithms, where the verification is usually performed at the code level, e.g. [GGH07] and [HL10], which both use the PVS theorem prover. In [TSBR08], a copying collector with no concurrency is verified using separation logic. The approach presented in [ZCD+17] also uses rely-guarantee techniques to verify a concurrent garbage collector,

[11] The full details of this approach are to be published in a separate paper by Yatapanage. There are several interesting technical points: the idea of localising rely conditions is again used together with a universally quantified set that can be instantiated to the $consid$ set of the collector.

but the verification is performed on an intermediate representation that is designed to be as close to the code level as possible. Another significant difference is that, unlike Ben-Ari's algorithm, the algorithm that they verify has strong coupling between the mutator and collector, requiring both to communicate with each other via shared variables to determine the other's current status. Such tight communication avoids some of the challenges that makes our example a useful study. It is interesting to note though that they still require the use of ghost variables. They have, however, implemented their proofs using Coq.

An Owicki-Gries proof of Ben-Ari's algorithm is given in [NE00]; while this examines multiple mutators, the method results in very large numbers of POs. The proof of Ben-Ari's algorithm in [vdS87], also using Owicki-Gries, reasons directly at the code level without using abstraction. The research on RGSim [Lia14] is strongly related to R/G thinking and [LFF14, §7.3] tackles a different GC algorithm that involves a "stop the world" phase.

Perhaps the closest approach to the development of the current paper is contained in [PPS10], which presents a refinement-based approach for deriving various GC algorithms from an abstract specification. This approach is very interesting and for future work it is worth exploring how the approach given here could be used to verify a similar family of algorithms. It would appear that the rely-guarantee method produces a more compositional proof, as the approach in [PPS10] requires more integrated reasoning about the actions of the Mutator and the Collector. Similarly, in [VYB06], a series of transformations is used to derive various concurrent garbage collection algorithms from an initial algorithm. The alternative of tackling the development using, as in [Jon96], the "fiction of atomicity" and "splitting atoms" does not appear to work on this example because the "atom" to be split is in the wrong process.

The objective in the current paper to achieve a compositional development has been only partially achieved. An unkind conclusion would be that this is because the authors chose to stay as close as possible to rely-guarantee conditions expressed as relations. But in so doing, both the inherent difficulty of the interconnection of the mutator and collector algorithms has been exposed and a clear set of alternative extensions to the R/G approach have been tabled. More experimentation should indicate the best way forward. Even if the alternative to use a shared ghost variable is taken, a clear test is offered to reduce the danger that such variables are used superfluously with the resulting diminution of separation between the concurrent processes.

It is hoped that the ten lessons are a transferable message of this paper even for approaches that do not use R/G thinking. The (garbage collection) example illustrates and hopefully clarifies the lessons for the reader. The current authors believe that examples are essential to drive such research.

Acknowledgements

The current journal paper is a major reworking of an earlier conference paper [JVY17] and we acknowledge the earlier enjoyable collaboration with our colleague Andrius Velykis before he moved to industry and then back to his homeland.

We have also benefitted from productive discussions with researchers including José Nuno Oliveira, Ian Hayes and attendees at the *Northern Concurrency Working Group*. In particular, Simon Doherty pointed out that GC is a nasty challenge for any compositional approach because the mutator/collector were clearly thought out together; while this is true, looking at an example at the fringe of R/G expressivity has informed the notion of compositional development. Leo Freitas is in the process of mechanising the proofs of the lemmas and theorems above and has made many useful comments. An anonymous referee also provided useful input that has hopefully led to clarifications.

The authors gratefully acknowledge funding for this research from EPSRC grants *Taming Concurrency* and *Strata*.

References

[BA84] Ben-Ari M (1984) Algorithms for on-the-fly garbage collection. ACM Trans Programm Lang Syst 6(3):333–344

[BA10] Bornat R, Amjad H (2010) Inter-process buffers in separation logic with rely-guarantee. Formal Asp Comput 22(6):735–772

[BA13] Bornat R, Amjad H (2013) Explanation of two non-blocking shared-variable communication algorithms. Formal Asp Comput 25(6):893–931

[BvW98] Back R-JR, von Wright J (1998) Refinement calculus: a systematic introduction. Springer, New York

[CJ00] Collette P, Jones CB (2000) Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language and interaction, chapter 10. MIT Press, pp 277–307

[CJ07] Coleman JW, Jones CB (2007) A structural proof of the soundness of rely/guarantee rules. J Log Comput 17(4):807–841

[Col08] Coleman JW (2008) Constructing a tractable reasoning framework upon a fine-grained structural operational semantics. PhD thesis, Newcastle University

[DFPV09] Dodds M, Feng X, Parkinson M, Vafeiadis V (2009) Deny-guarantee reasoning. In: Castagna G (ed) Programming languages and systems, volume 5502 of lecture notes in computer science. Springer, Berlin, pp 363–377

[Din00] Dingel J (2000) Systematic parallel programming. PhD thesis, Carnegie Mellon University, CMU-CS-99-172

[DYDG$^+$10] Dinsdale-Young T, Dodds M, Gardner P, Parkinson MJ, Vafeiadis V (2010) Concurrent abstract predicates. In: Proceedings of the 24th European conference on object-oriented programming, Berlin, Heidelberg, pp 504–528

[FFS07] Feng X, Ferreira R, Shao Z (2007) On the relationship between concurrent separation logic and assume-guarantee reasoning. In: ESOP: programming languages and systems. Springer, pp 173–188

[GGH07] Gao H, Groote JF, Hesselink WH (2007) Lock-free parallel and concurrent garbage collection by mark&sweep. Sci Comput Program 64(3):341–374

[HBDJ13] Hayes IJ, Burns A, Dongol B, Jones CB (2013) Comparing degrees of non-determinism in expression evaluation. Comput J 56(6):741–755

[HJ18] Hayes IJ, Jones CB (2018) A guide to rely/guarantee thinking. In: Bowen JP, Liu Z, Zhang Z (eds) Engineering trustworthy software systems, volume 11174 of LNCS. Springer, Cham, pp 1–38

[HJC14] Hayes IJ, Jones CB, Colvin RJ (July 2014) Laws and semantics for rely-guarantee refinement. Technical report CS-TR-1425, Newcastle University

[HL10] Hesselink WH, Lali MI (2010) Simple concurrent garbage collection almost without synchronization. Formal Methods Syst Des 36(2):148–166

[Hoa72] Hoare CAR (1972) Towards a theory of parallel programming. In: Operating system techniques. Academic Press, pp 61–71

[JH16] Jones CB, Hayes IJ (2016) Possible values: exploring a concept for concurrency. J Log Algebraic Methods Programm 85(5, Part 2):972–984

[JHC15] Jones CB, Hayes IJ, Colvin RJ (2015) Balancing expressiveness in formal approaches to concurrency. Formal Asp Comput 27(3):475–497

[JHM16] Jones R, Hosking A, Moss E (2016) The garbage collection handbook: the art of automatic memory management. Chapman and Hall

[Jon81] Jones CB (June 1981) Development methods for computer programs including a notion of interference. PhD thesis, Oxford University, June 1981. Available as: Oxford University Computing Laboratory (now Computer Science) Technical Monograph PRG-25

[Jon83a] Jones CB (1983) Specification and design of (parallel) programs. In: Proceedings of IFIP'83. North-Holland, pp 321–332

[Jon83b] Jones CB (1983) Tentative steps toward a development method for interfering programs. ACM ToPLaS 5(4):596–619

[Jon90] Jones CB (1990) Systematic software development using VDM, 2nd edn. Prentice Hall International

[Jon96] Jones CB (March 1996) Accommodating interference in the formal design of concurrent object-based programs. Formal Methods Syst Des 8(2):105–122

[JP11] Jones CB, Pierce KG (2011) Elucidating concurrent algorithms via layers of abstraction and reification. Formal Asp Comput 23(3):289–306

[JVY17] Jones CB, Velykis A, Yatapanage N (2017) General lessons from a rely/guarantee development. In: Larsen KG, Sokolsky O, Wang J (eds) Dependable software engineering: theories, tools, and applications, volume 10606 of LNCS. Springer, pp 3–24

[JY15] Jones CB, Yatapanage N (2015) Reasoning about separation using abstraction and reification. In: Calinescu R, Rumpe B (eds) Software engineering and formal methods, volume 9276 of LNCS. Springer, pp 3–19

[LFF14] Liang H, Feng X, Fu M (2014) A rely-guarantee-based simulation for compositional verification of concurrent program transformations. ACM Trans Programm Lang Syst 36(1):3:1–3:55

[Lia14] Liang H (2014) Refinement verification of concurrent programs and its applications. PhD thesis, USTC, China

[McC66] McCarthy J (1966) A formal description of a subset of ALGOL. In: Formal language description languages for computer programming. North-Holland, pp 1–12

[Mor90] Morgan C (1990) Programming from specifications. Prentice-Hall

[NE00] Nieto LP, Esparza J (2000) Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In: MFCS 2000, volume 1893 of LNCS. Springer, pp 619–628

[NPW09] Nipkow T, Paulson LC, Wenzel M (2009) Isabelle/HOL—a proof assistant for higher-order logic, volume 2283 of LNCS. Springer

[OG76] Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Inf 6(4):319–340

[O'H07] O'Hearn PW (May 2007) Resources, concurrency and local reasoning. Theor Comput Sci 375(1–3):271–307

[Owi75] Owicki S (1975) Axiomatic proof techniques for parallel programs. PhD thesis, Department of Computer Science, Cornell University

[Par10] Parkinson M (2010) The next 700 separation logics. In: Leavens G, O'Hearn P, Rajamani S (eds) Verified software: theories, tools, experiments, volume 6217 of LNCS. Springer, pp 169–182

[Pie09]	Pierce K (2009) Enhancing the useability of rely-guaranteee conditions for atomicity refinement. PhD thesis, Newcastle University
[PPS10]	Pavlovic D, Pepper P, Smith DR (2010) Formal derivation of concurrent garbage collectors. In: MPC 2010, volume 6120 of LNCS. Springer, pp 353–376
[Pre01]	Nieto LP (2001) Verification of parallel programs with the Owicki–Gries and Rely–Guarantee methods in Isabelle/HOL. PhD thesis, Institut für Informatic der Technischen Universitaet München
[STER11]	Schellhorn G, Tofan B, Ernst G, Reif W (2011) Interleaved programs and rely-guarantee reasoning with ITL. In: TIME, pap 99–106
[Stø90]	Stølen K (1990) Development of parallel programs on shared data-structures. PhD thesis, Manchester University, Available as UMCS-91-1-1
[TSBR08]	Torp-Smith N, Birkedal L, Reynolds JC (2008) Local reasoning about a copying garbage collector. ToPLaS 30:24:1–24:58
[Vaf07]	Vafeiadis V (2007) Modular fine-grained concurrency verification. PhD thesis, University of Cambridge
[vdS87]	van de Snepscheut JLA (1987) Algorithms for on-the-fly garbage collection revisited. Inf Process Lett 24(4):211–216
[VYB06]	Vechev MT, Yahav E, Bacon DF (2006) Correctness-preserving derivation of concurrent garbage collection algorithms. In: PLDI, pp 341–353
[WDP10]	Wickerson J, Dodds M, Parkinson MJ (2010) Explicit stabilisation for modular rely-guarantee reasoning. In: Gordon AD (ed) ESOP, volume 6012 of LNCS. Springer, pp 610–629
[Xu92]	Xu Q (1992) A theory of state-based parallel programming. PhD thesis, Oxford University
[ZCD$^+$17]	Zakowski Y, Cachera D, Demange D, Petri G, Pichardie D, Jagannathan S, Vitek J (2017) Verifying a concurrent garbage collector using a rely-guarantee methodology. In: Ayala-Rincón M, Muñoz CA (eds) Proceedings of interactive theorem proving—8th international conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, volume 10499 of lecture notes in computer science. Springer, pp 496–513

Received 2 August 2018
Accepted in revised form 14 March 2019 by Jim Woodcock
Published online 15 April 2019