SCHOOL OF
COMPUTING

Title: Investigating the limits of rely/guarantee conditions based on a concurrent garbage collector example

Names: Cliff B. Jones and Nisansala Yatapanage

TECHNICAL REPORT SERIES

No. TR – 1521 - 29th June 2018

TECHNICAL REPORT SERIES

No. CS-TR- 1521 Date 27th June 2018

Title: Investigating the limits of rely/guarantee conditions based on a concurrent garbage collector example

Authors: Cliff B. Jones and Nisansala Yatapanage

Abstract: Decomposing the design (or documentation) of large systems is a practical necessity; 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 conditions.
This facilitates an exploration of the limitations of this well-known method. Although the rely/guarantee approach is used, the lessons are more general.

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building,
Newcastle upon Tyne, NE4 5TG, England.

NEWCASTLE UNIVERSITY - Bibliography

School of Computing. Technical Report Series. CS-TR- 1521

Title - Investigating the limits of rely/guarantee conditions based on a concurrent garbage collector example

Authors : Cliff B. Jones and Nisansala Yatapanage

Abstract :

Decomposing the design (or documentation) of large systems is a practical necessity; 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 conditions.
This facilitates an exploration of the limitations of this well-known method.
Although the rely/guarantee approach is used, the lessons are more general.

About the authors

Cliff Jones is Professor of Computing Science at Newcastle University. He is best known for his research into "formal methods" for the design and verification of computer systems; under this heading, current topics of research include concurrency, support systems and logics. He is also currently applying research on formal methods to wider issues of dependability. Running up to 2007 his major research involvement was the five university IRC on "Dependability of Computer-Based Systems" of which he was overall Project Director - this was followed by a *Platform Grant* "Trustworthy Ambient Systems" (TrAmS) (Cliff was PI - funding from EPSRC) and is now CI on TrAmS-2. He also coordinates the three work packages on methodology in the DEPLOY project (on which he is CI) and is PI on an EPSRC-funded AI4FM project. As well as his academic career, Cliff has spent over twenty years in industry (which might explain why "applicability" is an issue in most of his research). His fifteen years in IBM saw, among other things, the creation - with colleagues in the Vienna Lab- of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his Oxford doctoral thesis in two years (and enjoyed the family atmosphere of Wolfson College). From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which -among other projects- was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural" (Formal Method) Support

Systems theorem proving assistant).During his time at Manchester, Cliff had a 5-year *Senior Fellowship* from the research council and later spent a sabbatical at Cambridge for the whole of the Newton Institute event on "Semantics" (and there appreciated the hospitality of a Visiting Fellowship at Gonville & Caius College. Much of his research at this time focused on formal (compositional) development methods for concurrent systems.
In 1996 he moved to Harlequin, directing some fifty developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999.

Nisansala Yatapanage completed her PhD in Griffith University, Australia, in 2012, on the topic of slicing of Behavior Tree specifications for model checking. This included the development of a novel form of branching bisimulation known as Next-preserving Branching Bisimulation, which has the unique property of preserving the Next temporal logic operator while still allowing stuttering steps to be removed. Nisansala has worked on research projects in software specification and verification since 2004 in both Griffith University and The University of Queensland (UQ), centering on the Behavior Tree specification language and model checking. From 2004 to 2007 she worked on the Dependability in Complex Computer-based Systems project, as part of the ARC Centre for Complex Systems, where she developed a translator from the Behavior Tree language to the input languages of model checkers, in order to automate Failure Modes and Effects Analysis. After completion of her PhD, she applied this technique to actual case studies as part of a project at UQ. Nisansala is now a Research Associate on the Taming Concurrency project at Newcastle University, UK.

5-year *Senior Fellowship* from the research council and later spent a sabbatical at Cambridge for the whole of the Newton Institute event on "Semantics" (and there appreciated the hospitality of a Visiting Fellowship at Gonville & Caius College. Much of his research at this time focused on formal (compositional) development methods for concurrent systems.
In 1996 he moved to Harlequin, directing some fifty developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999.

Suggested keywords

Concurrent programs; garbage collection; rely/guarantee; ghost variables;

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

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

[1] School of Computing Science, Newcastle University, United Kingdom
[2] School of Computer Science and Informatics, De Montfort University, United Kingdom

Abstract. Decomposing the design (or documentation) of large systems is a practical necessity; 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 conditions. This facilitates an exploration of the limitations of this well-known method. Although the rely/guarantee approach is used, the lessons are more general.

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 its 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 Section 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 Section 2.1).

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

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

* This technical report is a preliminary version of a journal submission

sequentially ($S1; S2$); that top-level step can be justified by discharging a proof obligation 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.

As far as is possible, the advantages of compositional development should be retained for concurrent programs.

Lesson I *The notion of "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).*

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 Section 1.2) facilitate the required separation where a designer chooses a decomposition of S into shared-variable sub-components that are to be executed in parallel ($S1 \parallel S2$).

1.2 Rely/Guarantee thinking

The origin of the rely/guarantee (R/G) work goes back to [Jon81]. Some 20 theses have developed the original idea including [Stø90,Xu92] that look at progress arguments, [Din00] that moves in the direction of a refinement calculus form of R/G, [Pre01] that provides an Isabelle-checked soundness proof of a slightly restricted form of R/G rules, [Col08] that revisits soundness of general R/G rules, [Pie09] that 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 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 much more 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 such as distribution are used as required.

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

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

– 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 for the *Fisher/Galler* Algorithm for the *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 Section 1.3) is contained in [JH16]

The first two contain 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.

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 Sections 4 and 5.*

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 below 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 (*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 the sense 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 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* check. 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 Section 2.1
6. The question of whether some examples require ghost variables is open and the discussion is resumed in Section 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 Sections 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 Section 2.2) and marking (Section 3).

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 proof obligations are given, for example, in [Jon90].*

Lesson III is commonplace for sequential programs but it actually has yet greater force for concurrent program development (where it is perhaps underemployed by many researchers). For example, it is argued in [JY15] that careful use of abstraction can serve the purpose of reasoning about separation. Furthermore, in R/G examples such as [JP11], such abstractions also make it possible to address interference and separation at early stages of design.

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.

Abstract states contain two sets of addresses: those that are in use (*busy*) and those that have been collected into a *free* set.[2]

Σ_0 :: *busy* : *Addr*-**set**
free : *Addr*-**set**

[2] The use of VDM notation should present the reader with no difficulty: it has been widely used for decades and is the subject of an ISO standard; one useful reference is [Jon90].

where

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

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.) 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*.

Effectively, the GC process is an infinite loop repeatedly executing the *Collector* operation whose specification is:

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

The predicate *guar-Collector* reassures the designer of *Mutator* that a chosen *free* cell will not disappear. The read/write "frames" in a VDM specification provide a shorthand for access and interference: thus *Collector* actually has an implied guarantee condition that it cannot change the value of *busy*.

The rely condition warns the developer of *Collector* that the *Mutator* can consume *free* addresses. Given this fact, recording a post condition for *Collector* is not quite trivial. In a sequential setting, it would be correct to write:

$$\mathit{free}' = (\mathit{Addr} - \mathit{busy})$$

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 in time. Here is the first use of the "possible values" notation in this paper. In a sequential formulation, *post-Collector* would set the lower bound for garbage collection by requiring that any addresses not reachable (in the initial heap) from *roots* would be in the final *free* set. To cope with the fact that a concurrent *Mutator* can acquire addresses from *free*, the correct statement is that all unreachable addresses should appear in some value of *free*. The notation discussed in [JP11,HBDJ13,JH16] for the set of possible values that can be observed by a component is $\overbrace{\mathit{free}}$.

Lesson IV *The "possible values" notation is a useful addition to –at least– the R/G style of specification.*

Theorem 1. *The POs requiring that the guarantee conditions of each process imply the rely condition of the other process are, at this stage, finessed by making:*

$$\mathit{guar\text{-}Collector} \iff \mathit{rely\text{-}Mutator}$$
$$\mathit{guar\text{-}Mutator} \iff \mathit{rely\text{-}Collector}$$

2.2 The heap

This section introduces a model of the heap. The set of addresses that are busy is defined to be those that are reachable from a set of roots by tracing all of the pointers in a heap. Because neither *Collector* nor *Mutator* has write access to *roots*, it remains constant (which is not recorded in the rely conditions).

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

where

$$\begin{array}{l} \mathit{inv}\text{-}\Sigma_1(\mathit{mk}\text{-}\Sigma_1(\mathit{roots},\mathit{hp},\mathit{free})) \triangleq \\ \quad \mathbf{dom}\,\mathit{hp} = \mathit{Addr} \wedge \\ \quad \mathit{free} \cap \mathit{reach}(\mathit{roots},\mathit{hp}) = \{\} \wedge \qquad \text{upper bound for GC} \\ \quad \forall a \in \mathit{free} \cdot \mathit{hp}(a) = \{[\,]\} \end{array}$$

$$\mathit{Heap} = \mathit{Addr} \xrightarrow{m} \mathit{Node}$$

$$\mathit{Node} = \left[\mathit{Addr}\right]^*$$

When addresses are deleted from nodes, their position is set to the **nil** value. To smooth the use of this model of *Heap*, $\mathit{hp}(a,i)$ is written for $\mathit{hp}(a)(i)$ and $(a,i) \in \mathbf{dom}\,\mathit{hp}$ has the obvious meaning.[3]

The second conjunct of the invariant defines the upper bound of garbage collection (i.e. no addresses reachable from *roots* should appear in *free*); the final conjunct requires that free addresses map to empty nodes.

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

$$\mathit{reach} : \mathit{Addr}\text{-}\mathbf{set} \times \mathit{Heap} \to \mathit{Addr}\text{-}\mathbf{set}$$

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

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

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

$$\mathit{rel}\text{-}\mathit{image}(r,s) \triangleq \{b \mid (a,b) \in r \wedge a \in s\}$$

[3] Several alternative modelling decisions were considered. For example, it is tempting to make the *free* pointer one of the *roots* because it merges the operations — this was not done because it is useful to distinguish the *Malloc* and *Redirect* operations (see Section 4.3 below). Also 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.

The *child-rel* function extracts the relation over addresses from the heap (i.e. ignoring pointer positions); it drops any **nil** values.

$$\textit{child-rel} : \textit{Heap} \to (\textit{Addr} \times \textit{Addr})\text{-}\mathbf{set}$$
$$\textit{child-rel}(hp) \triangleq \{(a,b) \mid a \in \mathbf{dom}\, hp \wedge b \in (\textit{Addr} \cap \mathbf{elems}\, hp(a))\}$$

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 *Node* which contains an address not in s (but notice that $hp(b,j)$ might not be a).

Lemma 1. *A useful lemma is:*

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

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 is based on:

$$\textit{retr}_0 : \Sigma_1 \to \Sigma_0$$
$$\textit{retr}_0(\textit{mk-}\Sigma_1(\textit{roots}, hp, \textit{free})) \triangleq \textit{mk-}\Sigma_0(\textit{reach}(\textit{roots}, hp), \textit{free})$$

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*

$$\forall \sigma_0 \in \Sigma_0 \cdot \exists \sigma_1 \in \Sigma_1 \cdot \textit{retr}_0(\sigma_1) = \sigma_0$$

Proof. It is straightforward to find a representative element

Collector
ext wr *free*
  rd *roots*, *hp*
pre true
rely $\textit{free}' \subseteq \textit{free}$
guar $\textit{free} \subseteq \textit{free}'$
post $(\textit{Addr} - \textit{reach}(\textit{roots}, hp)) \subseteq \bigcup \widehat{\textit{free}}$ lower bound for GC

Strictly, the fact that the *Collector* (in particular, its *Sweep* component) does not have write access to *hp* means that it cannot clean up the pointers in *free* as required by the final conjunct of $inv\text{-}\Sigma_1$. Changing the guarantee conditions is uninformative but *rely-Mutator* below does show the more precise predicate.

Mutator
ext wr $hp, free$
  rd *roots*
pre true
rely $free' \lhd hp' = free' \lhd hp \wedge$
  $free \subseteq free'$
guar $free' \subseteq free$
post true

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

Theorem 3. *The commutativity proofs are trivial.*[4]

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* (and maintain the invariant that addresses in $(roots \cup free)$ are always marked) then sweep any unmarked addresses into *free*.

Σ_2 :: *roots* : *Addr*-**set**
  hp : *Heap*
  free : *Addr*-**set**
  marked : *Addr*-**set**

where

$inv\text{-}\Sigma_2(mk\text{-}\Sigma_2(roots, hp, free, marked)) \triangleq$
  $\mathbf{dom}\, hp = Addr \wedge$
  $free \cap reach(roots, hp) = \{\} \wedge$ upper bound for GC
  $(roots \cup free) \subseteq marked \wedge$
  $\forall a \in free \cdot hp(a) = \{[\,]\}$

The real issue is where the garbage collection runs concurrently with a *Mutator* which can acquire *free* addresses and give rise to garbage that is no longer accessible from *roots*. A fully concurrent garbage collector is covered below (see Sections 4 and 5).

This section introduces code that can be viewed as sequential in the sense that the *Mutator* would have to pause; interestingly this same code satisfies specifications for two more challenging concurrent situations.

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

3.1 Sequential algorithm

In order to clarify issues about POs in general and loop (proofs) in particular, verification of the **non-interference case** is considered first; i.e. rely conditions for the *Collector* saying the *heap* is unchanged. This helps sort out some issues in a simple setting (and the code carries over to the concurrent algorithm with, however, more complicated specifications and justifications). It also clarifies terminology (viz. the upper bound for garbage requires a lower bound for marking). Effectively, the specification of *Collector* is a simplification of that in Section 4 with all rely conditions saying no change to any variable; in particular, *marked* is effectively a local variable.

The *Collector* can be split into three phases. Providing the invariant is respected, the initial marking is unimportant but, thinking of the *Collector* being run intermittently, it is reasonable to start by removing any surplus marks.

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

(These operation names are decorated with subscripts below to distinguish the sequential, atomic and truly concurrent versions.)

The main interest is in the marking phase. As shown in Fig. 1, the outer loop propagates a wave of marking over the *hp* relation;[5] it iterates until no new addresses are marked. The inner *Propagate* iterates over all addresses: for each address that is itself marked, all of its children are marked.

```
Mark ≜
    repeat
      mc ← card marked;
      Propagate
    until card marked = mc
```

```
Propagate ≜
    consid ← { };
    do while consid ≠ Addr
      let x ∈ (Addr − consid) in
      if x ∈ marked then Mark-kids(x) else skip;
      consid ← consid ∪ {x}
    od
```

Fig. 1. Code for *Mark*

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. 1 satisfies that specification are straightforward. (In fact, they are simplified cases of what follows in Section 4.) When the same code is placed in environments that admit interference, R/Gs and different POs are needed (see Sections 4 and 5). The evolution of the R/G conditions is particularly interesting.

[5] It would be more elegant to write:

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

but the set *consid* is useful to express some assertions below.

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

Here, the operation names are subscripted with s to mark them as the sequential versions.

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

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

$Sweep_s$
ext wr $hp, free, marked$
pre true
rely $marked' = marked \wedge free' = free \wedge hp' = hp$
guar true
post $free' = free \cup (Addr - marked) \wedge hp' = hp \dagger \{a \mapsto [\,] \mid a \in marked\}$

Theorem 4. *The sequential composition PO is (all pre conditions are true):*

$$\boxed{;\text{-}I}\ \frac{\begin{array}{l} post\text{-}Unmark_s(\sigma, \sigma') \\ post\text{-}Mark_s(\sigma', \sigma'') \\ post\text{-}Sweep_s(\sigma'', \sigma''') \end{array}}{post\text{-}Collector(\sigma, \sigma''')}$$

Proof. Without interference, the proof is straightforward ($\widehat{free}$ in *post-Collector* being the special case of $free'''$). But the main result (upper bound of what is collected) is expressed in *inv*-Σ_2 and follows from the lower bound of marking.

The outer loop (cf. Figure 1) propagates a wave of marking over the hp relation.

$Propagate_s$
ext wr *marked*
  rd *hp*
pre true
rely $marked' = marked \wedge free' = free \wedge hp' = hp$
guar true
post $marked' = marked \cup \bigcup\{Addr \cap \mathbf{elems}\, hp(a) \mid a \in marked\}$

To prove the lower marking bound (i.e. must mark everything that is reachable from *roots*), a *to-end* induction is employed;[6] essentially the $to\text{-}end_s$ relation says that the remaining iterations of the loop will mark everything reachable from what is already marked:

$$to\text{-}end_s(\sigma,\sigma') \triangleq marked' = marked \cup reach(marked, hp)$$

Theorem 5. *Thus:*

$$\boxed{\textit{loop-right-compose}}\ \frac{\begin{array}{l}post\text{-}Propagate_s(\sigma,\sigma')\\ \mathbf{card}\, marked < \mathbf{card}\, marked'\\ to\text{-}end_s(\sigma',\sigma'')\end{array}}{to\text{-}end_s(\sigma,\sigma'')}$$

Proof. The proof is straightforward.

Theorem 6. *And finally:*

$$\boxed{;\text{-}I}\ \frac{\begin{array}{l}post\text{-}Unmark_s(\sigma,\sigma')\\ to\text{-}end_s(\sigma',\sigma'')\end{array}}{post\text{-}Mark_s(\sigma,\sigma'')}$$

Proof. The proof is trivial.

The body of the inner loop (cf. Figure 1) has to satisfy:

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

[6] There is an interesting point here. In the standard 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 *to-end* version overwrites the initial value.

Lemma 2. *With:*

$$\textit{so-far}_s(\sigma,\sigma') \triangleq \\ \quad \textit{marked}' = \textit{marked} \cup \bigcup\{\textit{Addr} \cap \mathbf{elems}\, \textit{hp}(a) \mid a \in (\textit{marked} \cap \textit{consid}')\}$$

$$\boxed{\textit{loop-left-compose}}\ \frac{\begin{array}{l}\textit{so-far}_s(\sigma,\sigma') \\ \textit{consid}' \neq \textit{Addr} \\ \textit{post-Mark-kids}_s(\sigma',x,\sigma'')\end{array}}{\textit{so-far}_s(\sigma,\sigma'')}$$

Proof. The proof is straightforward.

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.

Theorem 7. *Finally:*

$$\boxed{\textit{Propagate}_s}\ \frac{\begin{array}{l}\textit{so-far}_s(\sigma,\sigma') \\ \textit{consid}' = \textit{Addr}\end{array}}{\textit{post-Propagate}_s(\sigma,\sigma')}$$

Proof. The proof is immediate.

4 Concurrent GC with atomic interference

The complication in the concurrent case is that the *Mutator* can interfere with the marking strategy of the *Collector* by redirecting pointers. This can be accommodated providing the *Mutator* marks appropriately whenever it makes a change.

The development is tackled in two stages: firstly, this section assumes a *Mutator* that atomically both redirects a pointer in a *Node* and marks the new address; Section 5 shows that even separating the two steps still allows the *Collector* code of Fig. 1 to achieve the lower bound of marking but the argument is more delicate and indicates an expressive limitation 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 Section 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 this section with Section 5. As proposed in Section 1, the argument is split into a justification of the parallel decomposition (Section 4.1) and the decompositions of the *Collector*/*Mutator* sub-components, addressed in Sections 4.2 and 4.3 respectively.

4.1 Parallel decomposition

In this section, the operation names have the subscript a to record the atomicity assumption.

Given the atomicity assumption, an R/G specification of the collector is:

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

Here again, the notation for possible values is used to cope with interference. In a sequential formulation, *post-Collector*$_a$ would set the lower bound for garbage collection by requiring that any addresses not reachable (in the initial *hp*) from *roots* would be in the final *free* set. To cope with the fact that a concurrent *Mutator* can acquire addresses from *free*, the correct statement is that all unreachable addresses should appear in some value of *free*.

It should be noted that an implied guarantee comes from the component having only read access — e.g. the *Collector*$_a$ cannot change the *hp* component.[7] 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.

The lower bound of addresses to be collected is one part of the requirement; the upper bound is constrained by the second conjunct of *inv*-Σ_2.

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 lower bound for garbage collection requires setting an upper bound for marking addresses; this topic is postponed to Section 6.

The corresponding specification of the *Mutator*$_a$ is:

$Mutator_a$
ext wr $hp, free, marked$
  rd $roots$
pre true

[7] Strictly, the fact that the *Collector*$_a$ (in particular, its *Sweep*$_a$ component) does not have write access to *hp* means that it cannot clean up the nodes in *free* as required by the final conjunct of *inv*-Σ_2. Changing the guarantee conditions is uninformative. An alternative would be to perform the cleanup in *Malloc*.

rely $free \subseteq free'$
guar $rely\text{-}Collector_a$
post true

Theorem 8. *The R/G PO for concurrent processes requires that each one's guarantee condition implies the rely condition of the other(s); in this case they are identical so the result is immediate.*

4.2 Developing the $Collector_a$ code

As outlined in Section 1, 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 $Collector_a$ into three phases ($Unmark_a$;$Mark_a$;$Sweep_a$) given in Section 3.1 satisfies the $Collector_a$ specification in Section 4.1.

The post condition for the sequential version of $Unmark_s$ constrains $marked'$ to be exactly equal to $roots \cup free$ (cf. the third conjunct of $inv\text{-}\Sigma_2$) but, again, interference must be considered. The rely condition indicates that the environment can mark addresses so whatever $Unmark_a$ removes from *marked* could be replaced. The possible values notation is again deployed so that $post\text{-}Unmark_a$ requires that, for every address which should not be marked, a possible value of *marked* exists which does not contain the address. However, 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 again uses the idea in Lesson VI.

$Unmark_a$
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 second conjunct of $inv\text{-}\Sigma_2$). This lower bound is recorded in the post condition. (The use of hp' is, of course, challenging but the post condition is stable [CJ07,WDP10] under the rely condition.) 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-mog* in Section 6).

$Mark_a$
ext wr *marked*
  rd $hp, roots, free$
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, for the concurrent case, becomes:

$Sweep_a$
ext wr *free*
  rd $hp, marked$
pre true
rely $free' \subseteq free \wedge marked \subseteq marked'$
guar $free \subseteq free'$
post $(free' - free) \cap marked = \{\} \wedge$
  $\forall a \in (Addr - marked) \cdot \exists f \in \widehat{free} \cdot a \in f$

The rely and guarantee conditions of $Collector_a$ are distributed (with appropriate weakening/strengthening) over the three sub-components;

Theorem 9. *Since all of the pre conditions are* **true***; so the remaining PO for the composition is:*

$post\text{-}Unmark_a(\sigma, \sigma') \wedge post\text{-}Mark_a(\sigma', \sigma'') \wedge post\text{-}Sweep_a(\sigma'', \sigma''')$
$\Rightarrow\ post\text{-}Collector_a(\sigma, \sigma''')$

Proof. The proof is straightforward.

It is useful to define a predicate for "completely marked" nodes:

$cm\text{-}n : Node \times Addr\text{-}\mathbf{set} \to \mathbb{B}$

$cm\text{-}n(n, s) \quad \triangleq \quad (Addr \cap \mathbf{elems}\, n) \subseteq s$

Turning to the decomposition of $Mark_a$ to an iteration (see Fig. 1), in order to prove $post\text{-}Mark_a$, a specification is needed for $Propagate_a$ that copes with interference:

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

The first conjunct of the post condition indicates the progress required of the wave of marking. 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 as in [Jon90]: essentially the *to-end* relation states that the remaining iterations of the loop will mark everything reachable from what is already marked:

$$\textit{to-end}_a(\sigma,\sigma') \triangleq \textit{reach}(\textit{marked},\textit{hp}') \subseteq \textit{marked}'$$

Theorem 10. *The PO is:*

$$\textit{post-Propagate}_a(\sigma,\sigma') \wedge \sigma.\textit{marked} \neq \sigma'.\textit{marked} \wedge \textit{to-end}_a(\sigma',\sigma'') \Rightarrow \textit{to-end}_a(\sigma,\sigma'')$$

Proof. whose 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. Section 6).

Theorem 11. *Then:*

$$\sigma.\textit{marked} = \sigma.\textit{roots} \wedge \textit{to-end}(\sigma,\sigma') \Rightarrow \textit{post-Mark}_a(\sigma,\sigma')$$

Proof. This follows trivially.

Pursuing the decomposition of $\textit{Propagate}_a$ to a nested iteration (again, see Fig. 1) needs a specification of the inner operation:

$\textit{Mark-kids}_a$ (x:*Addr*)
ext wr *marked*
  rd *hp*
pre true
rely $\textit{rely-Collector}_a$
guar $\textit{marked} \subseteq \textit{marked}'$
post $\textit{cm-n}(\textit{hp}'(x), \textit{marked}')$

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

$$\textit{so-far}_a(\sigma,\sigma') \triangleq \forall a \in (\textit{marked} \cap \textit{consid}') \cdot \textit{cm-n}(\textit{hp}(a), \textit{marked}')$$

Theorem 12. *The relevant PO is:*

$$\textit{so-far}_a(\sigma,\sigma') \wedge \textit{consid}' \neq \textit{Addr} \wedge \textit{post-Mark-kids}_a(\sigma',x,\sigma'') \wedge \textit{consid}'' = \textit{consid}' \cup \{x\} \\ \Rightarrow\ \textit{so-far}_a(\sigma,\sigma'')$$

whose discharge is obvious.

Theorem 13. *The final obligation is to show:*

$$\textit{so-far}_a(\sigma,\sigma') \wedge \textit{consid}' = \textit{Addr} \ \Rightarrow\ \textit{post-Propagate}_a(\sigma,\sigma')$$

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

4.3 Checking the interference from $\textit{Mutator}_a$

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 $\textit{rely-Mark}_a$:

Redirect $(a{:}\textit{Addr}, i{:}\mathbb{N}_1, b{:}\textit{Addr})$
ext wr $hp, \textit{marked}$
pre $\{a,b\} \subseteq \textit{reach}(\textit{roots}, hp) \wedge i \in \textbf{inds}\, hp(a)$
post $hp' = hp \dagger \{(a,i) \mapsto b\} \wedge \textit{marked}' = \textit{marked} \cup \{b\}$

Lemma 3. *It follows trivially that:*

$$\textit{post-Redirect}(\sigma,\sigma') \ \Rightarrow\ \textit{guar-Mutator}_a(\sigma,\sigma')$$

For this atomic case, the code (using multiple assignment) would be:

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

Malloc $(a{:}\textit{Addr}, i{:}\mathbb{N}_1, b{:}\textit{Addr})$
ext wr $hp, \textit{free}$
pre $a \in \textit{reach}(\textit{roots}, hp) \wedge i \in \textbf{inds}\, hp(a) \wedge b \in \textit{free}$
post $hp' = hp \dagger \{(a,i) \mapsto b\} \wedge \textit{free}' = \textit{free} - \{b\}$

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

Lemma 4. *It follows trivially that:*

$$\textit{post-Malloc}(\sigma,\sigma') \ \Rightarrow\ \textit{guar-Mutator}_a(\sigma,\sigma')$$

Zap $(a{:}\textit{Addr}, i{:}\mathbb{N}_1)$
ext wr hp
pre $a \in \textit{reach}(\textit{roots}, hp) \wedge i \in \textbf{inds}\, hp(a)$
post $hp' = hp \dagger \{(a,i) \mapsto \textbf{nil}\}$

Lemma 5. *It again follows trivially that:*

$$\textit{post-Zap}(\sigma,\sigma') \;\Rightarrow\; \textit{guar-Mutator}_a(\sigma,\sigma')$$

5 Relaxing atomicity

The remaining challenge is to consider the impact of removing the unrealistic atomicity assumption about $Mutator_a$ in Section 4. Splitting the atomic assignment (on the two shared variables $hp, marked$) in

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

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 for compositionality (see Lesson I) extremely challenging; some form of global argument is difficult to avoid. However, facing that challenge and looking at alternative extensions of R/G thinking is informative and minimising that global argument is interesting.

It is worth first disposing of a non-solution. The reader would be excused for thinking 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 $Mutator_c$ 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 the mutator 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:

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

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 Section 4.2 relies. There is in fact no danger because, even if b was not marked, there must be another path to b (see *pre-Redirect* in Section 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* cannot make another change without waking up and marking b.

The case considered in Scenario B 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:

- Section 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 other approaches;
- Section 5.2 discusses an alternative that suggests an extension to R/G;
- Section 5.3 outlines a way of avoiding a shared ghost variable but still, in some sense, uses a non-compositional argument.

5.1 Abstracting details with auxiliary variables

It might surprise readers who have heard the current authors inveigh against ghost variables that the development in Section 5.1 does in fact use such a variable (see Lesson VII). The state Σ_2 is extended with a variable $tbm: \lfloor Addr \rceil$ that can be used to record an address as "to be marked".

(In this section, the subscript c on operations and their predicates marks the fact that they cover true concurrency.)

The rely condition used in Section 4.1 is replaced for the truly concurrent (non-atomic interference from the *Mutator*) case by:

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

$$\begin{aligned} &rely\text{-}Collector_c(\sigma, \sigma') \triangleq \\ &\quad free' \subseteq free \wedge marked \subseteq marked' \wedge \\ &\quad (\forall (a,i) \in hp \cdot \\ &\qquad hp'(a,i) \neq hp(a,i) \wedge hp'(a,i) \in Addr \\ &\qquad \Rightarrow\ hp'(a,i) \in marked' \vee tbm' = hp'(a,i)) \wedge \\ &\quad (tbm \neq \mathbf{nil} \wedge tbm' \neq tbm\ \Rightarrow\ tbm \in marked' \wedge tbm' = \mathbf{nil}) \end{aligned}$$

The third conjunct of $rely\text{-}Collector_c$ records 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 **nil** at exactly the point in time when the delayed marking occurs.

Parallel decomposition

Theorem 14. *Here again, the parallel introduction PO is trivial to discharge because the guarantee condition of the $Mutator_c$ is identical to the rely condition of the $Collector_c$.*

Developing *Mutator* code As indicated in Section 1, it still remains to be established that the design of each component satisfies its specification.

Redirect will include the steps:

$< hp(a), tbm := hp(a) \dagger \{i \mapsto b\}, b >;$
$< marked, tbm := marked \cup \{b\}, \mathbf{nil} >$

Essentially *tbm* records the delayed marking that is considered in Scenario B. Notice that the atomic brackets now only surround one shared variable in each case.

This not only guarantees *rely-Collector*, but also preserves the following invariant:

Lemma 6.
$tbm \neq \mathbf{nil} \Rightarrow$
$\quad \exists\{(a,i),(b,j)\} \subseteq \mathbf{dom}\, hp \cdot (a,i) \neq (b,j) \wedge hp(a,i) = hp(b,j) = tbm$

Looking first at the non-atomic *Mutator* argument, the only real challenge is:[8]

Redirect $(a{:}Addr, i{:}\mathbb{N}_1, b{:}Addr)$
ext wr $hp, marked$
pre $\{a,b\} \subseteq reach(roots, hp) \wedge i \in \mathbf{inds}\, hp(a)$
rely $hp' = hp$
guar $rely\text{-}Collector_c$
post $hp' = hp \dagger \{(a,i) \mapsto b\} \wedge b \in marked'$

Developing *Collector* code Turning to the development of *Collector*, code must be developed relying only on the revised *rely-Collector*. The only challenge is the mark phase whose specification is:

$Mark_c$
ext wr $marked$
  rd $hp, roots, free$
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. 1 — under interference, the post condition of *Propagate* has to be further weakened (from Section 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:

[8] When removing a pointer, no *tbm* is set — see $Zap(a,i)$ in Section 4.3; also no *tbm* is needed in the *Malloc* case because $inv\text{-}\Sigma_2$ ensures that any *free* address is marked.

Propagate$_c$

ext wr *marked*
  rd *hp*

pre true

rely *rely-Collector*$_c$
guar $marked \subseteq marked'$
post $\forall a \in marked \cdot cm\text{-}n(hp'(a), (marked' \cup (\{tbm'\} \cap Addr))) \wedge$
  $(marked = marked' \;\Rightarrow\; reach(marked, hp') \subseteq marked')$

Notice that *post-Propagate* 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-Collector*.

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

$$to\text{-}end_c(\sigma, \sigma') \;\triangleq\; reach(marked, hp') \subseteq marked'$$

Theorem 15. *The PO is now:*

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

Proof. In comparison with the PO in Section 4.2, the difficult case is where $tbm' \neq$ **nil** (in the converse case the earlier proof would suffice). What needs to be shown is that the stray address in tbm' will be marked. Lemma 6 ensures there is another path to the address in tbm'; 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-Propagate*, avoids premature termination.

The code in Fig. 1 shows how *Propagate* uses *Mark-kids*$_c$ in the inner loop.

Mark-kids$_c$ (x:*Addr*)

ext wr *marked*
  rd *hp*

pre true

rely *rely-Collector*$_c$
guar $marked \subseteq marked'$
post $cm\text{-}n(hp'(x), (marked' \cup (\{tbm'\} \cap Addr)))$

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

$$so\text{-}far_c(\sigma, \sigma') \;\triangleq\; \forall a \in (marked \cap consid') \cdot cm\text{-}n(hp'(a), (marked' \cup (\{tbm'\} \cap Addr))$$

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 record 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 Section 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*-1/*mr*-2

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

There are actually two roles for *tbm* in the definition of *rely-Collector*$_c$ (Section 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 **nil** and *non*-**nil** values of *tbm* pinpoint the crucial point in the execution between *mr*-1 and *mr*-2.

Lemma 6 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:

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

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 Sections 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[9] 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 Section 5.2.

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

Thus, in the approach here, 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 Section 5.1, this avoids sharing *tbm* although a similar but local variable is used in $Mutator_c$.

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 *Collect*). A predicate "no marked old garbage" can be used for the upper bound of marking:

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

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

[9] 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.

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

7 Related work and conclusions

There 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. 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.

Perhaps the closest approach to the development of the current paper is contained in [PPS10], which presents a refinement-based approach for deriving various garbage collection 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.

To do: Add something on (forthcoming) Isabelle proofs

Acknowledgements

We have benefitted from productive discussions with researchers including José Nuno Oliviera, Ian Hayes and attendees at the January 2017 *Northern Concurrency*

Working Group held at Teesside University. 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; this is true but looking at an example at the fringe of R/G expressivity has informed the notion of compositional development.

To do: Leo (and Diego?) attempts to prove using Isabelle anonymous referees??

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

References

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, 1984.

[BA10] Richard Bornat and Hasan Amjad. Inter-process buffers in separation logic with rely-guarantee. *Formal Aspects of Computing*, 22(6):735–772, 2010.

[BA13] Richard Bornat and Hasan Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, 25(6):893–931, 2013.

[BvW98] R.-J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, New York, 1998.

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 277–307. MIT Press, 2000.

[CJ07] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[Col08] Joseph William Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics*. PhD thesis, Newcastle University, January 2008.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer Berlin / Heidelberg, 2009.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP: Programming Languages and Systems*, pages 173–188. Springer, 2007.

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

[HBDJ13] Ian J. Hayes, Alan Burns, Brijesh Dongol, and Cliff B. Jones. Comparing degrees of non-determinism in expression evaluation. *The Computer Journal*, 56(6):741–755, 2013.

[HJ18] I. J. Hayes and C. B. Jones. A guide to rely/guarantee thinking. In Jonathan Bowen, Zhiming Liu, and Zili Zhan, editors, *Engineering Trustworthy Software Systems – Second International School, SETSS 2017*, LNCS. Springer-Verlag, 2018.

[HJC14] I. J. Hayes, C. B. Jones, and R. J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, July 2014.

[HL10] Wim H. Hesselink and Muhammad Ikram Lali. Simple concurrent garbage collection almost without synchronization. *Formal Methods in System Design*, 36(2):148–166, 2010.

[Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

[JH16] Cliff B. Jones and Ian J. Hayes. Possible values: Exploring a concept for concurrency. *Journal of Logical and Algebraic Methods in Programming*, 85(5, Part 2):972–984, August 2016. Articles dedicated to Prof. J. N. Oliveira on the occasion of his 60th birthday.

[JHC15] C. B. Jones, I. J. Hayes, and R. J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27(3):475–497, May 2015.

[Jon81] C. B. Jones. *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.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[JP11] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.

[JY15] Cliff B. Jones and Nisansala Yatapanage. Reasoning about separation using abstraction and reification. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, volume 9276 of *LNCS*, pages 3–19. Springer, 2015.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[NE00] Leonor Prensa Nieto and Javier Esparza. Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In *MFCS 2000*, volume 1893 of *LNCS*, pages 619–628. Springer, 2000.

[NPW09] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2009.

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

[O'H07] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975.

[Par10] Matthew Parkinson. The next 700 separation logics. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *LNCS*, pages 169–182. Springer, 2010.

[Pie09] Ken Pierce. *Enhancing the Useability of Rely-Guaranteee Conditions for Atomicity Refinement*. PhD thesis, Newcastle University, 2009.

[PPS10] Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Formal derivation of concurrent garbage collectors. In *MPC 2010*, volume 6120 of *LNCS*, pages 353–376. Springer, 2010.

[Pre01] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Institut für Informatic der Technischen Universitaet München, 2001.

[STER11] G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In *TIME*, pages 99–106, 2011.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

[TSBR08] Noah Torp-Smith, Lars Birkedal, and John C. Reynolds. Local reasoning about a copying garbage collector. *ToPLaS*, 30:24:1–24:58, July 2008.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[vdS87] Jan L.A. van de Snepscheut. "Algorithms for on-the-fly garbage collection" revisited. *Information Processing Letters*, 24(4):211–216, 1987.

[VYB06] Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI*, pages 341–353, 2006.

[WDP10] J. Wickerson, M. Dodds, and M. J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In A. D. Gordon, editor, *ESOP*, volume 6012 of *LNCS*, pages 610–629. Springer, 2010.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.