Jones CB, Velykis A, Yatapanage N.

General Lessons from a Rely/Guarantee Development.

***In: 3rd International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA)*. 2017, Changsha, China: Springer Verlag**

General Lessons from a Rely/Guarantee Development

Cliff B. Jones[1], Andrius Velykis[1], Nisansala Yatapanage[1,2]

[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; this prompts the need for a notion of *compositional* development methods; finding such methods for concurrent software is technically challenging because of the interference that characterises concurrency. This paper outlines the development of a difficult example in order to draw out lessons about such development methods. Although the "rely/guarantee" approach is employed in the example, the intuitions are more general.

1 Introduction

The aim of this paper is to contribute to the 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*.

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 obligation involving only the specifications of S and $S1/S2$. 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 1 *The notion of "compositionality" is best understood by thinking about a development process: specifications of separate components ought genuinely insulate them from one another (and from their context). The ideal is, faced with a specified task (module), propose a decomposition (combinator) and specify any sub-tasks; then prove the decomposition correct wrt (only) the specifications. The same process is then repeated on the sub-tasks.*

Because of the interference inherent in concurrency, this 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 subcomponents that are to be executed in parallel ($S1 \parallel S2$).

1.2 Rely/Guarantee thinking

The origin of the rely/guarantee (R/G) work is in [Jon81]. More than 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 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] manage 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]. (A fuller set of references is contained in [HJC14,JHC15].)

The original way of writing R/G specifications displayed the predicates of a specification delimited by keywords; some of the subsequent papers (notably those concerned with showing the soundness of the system) present specifications as 5-tuples. The reformulation in [HJC14,JHC15] 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 and diverse examples of R/G developments including:

- Susan Owicki's [Owi75] problem of finding the minimum index to an array at which an element can be found satisfying a predicate 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 [JH16a]

The first two represent 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 2 *By using relations to express interference, R/G conditions offer a plausible balance of expressiveness versus tractability — see Sections 3.2 and 4.*

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 [Owi75,OG76] not only postpones a final (*Einmischungsfrie*) check until the code of all concurrent processes is to hand — 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 that they overlap in time. Hoare [Hoa72] observes that, in this simple case, the conjunction of the post conditions of the individual threads is an acceptable post condition for their combination.
2. Over-simplifying, this is a basis for concurrent separation logic. 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 Owicki's example that is 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 R/G approach in [HJC14], however, the interference is adequately characterised by relation.
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,JH16b] 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 4. That their use is tempting in order to simplify reasoning about concurrent processes is attested by the number of proofs that employ them.

Lesson 3 *The use of "ghost" (aka "auxiliary") variables presents a subtle danger to compositional development (cf. Lesson 1). The case against is, however, clear: in the extreme, ghost variables can be used to record complete detail about the environment of a process. Few researchers would succumb to such extreme temptation but minimising the use of ghost variables ought be an objective.*

It might surprise readers who have heard the current authors inveigh against ghost variables that the development in this conference paper does in fact use such a variable. The acknowledgements report a relevant discussion on this point and a journal paper is planned to explore other options.

1.4 Plan of the paper

The bulk of this paper (Sections 2–5) offers a development of the concurrent garbage collector described in [BA84,vdS87]. At several points, "lessons" are identified. It is these lessons that are actually the main point of the paper and the example is chosen to give credence to these intuitions.

The development uses R/G ideas (often referred to as "R/G thinking") but the lessons have far wider applicability.

It is also worth mentioning that it is intended to write an extended journal version of this paper that will contain a full development from an abstract specification hopefully with proofs checked on Isabelle. That paper will also review various modelling decisions made in the development. Many of twhich were revised (in some cases more than once) e.g. the decision how to model *Heap* was revised several times![3]

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 3–5. 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 spec

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 4 *It is widely accepted that the use of abstract datatypes can clarify key concepts before discussion turns to 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 4 is common place for sequential programs but it actually has even 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.

[3] One plausible alternative is:

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

The abstract states[4] contain two sets of addresses: those that are in use (*busy*) and those that have been collected into a *free* set.

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

where

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

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 $(busy' - busy) \subseteq free \wedge free' \subseteq free$
guar $free \subseteq free'$
post $(Addr - busy) \subseteq \bigcup \widehat{free}$

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 *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:

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

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 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 *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*. The notation discussed in [JP11,HBDJ13,JH16b] for the set of possible values that can be observed by a component is $\widehat{free}$.

[4] 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].

Lesson 5 *The "possible values" notation is a useful addition to –at least– R/G.*

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.

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

where

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

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

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

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. When addresses are deleted from nodes, their position is set to the **nil** value.

The second conjunct of the invariant defines the upper bound of garbage collection; the final conjunct requires that free addresses map to empty nodes. The *roots* component of Σ_1 is taken to be constant.

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

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

$$\mathit{child\text{-}rel}(\mathit{hp}) \quad \triangleq \quad \{(a,b) \mid a \in \mathbf{dom}\,\mathit{hp} \wedge b \in (\mathbf{elems}\,\mathit{hp}(a)) \cap \mathit{Addr}\}$$

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

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

$$\mathit{reach}(s,\mathit{hp}) \quad \triangleq \quad \mathbf{rng}\,(s \lhd \mathit{child\text{-}rel}(\mathit{hp})^\star)$$

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).

A useful lemma is:

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

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 state underlying the chosen garbage collector has an additional component to record the addresses that have been marked (the third conjunct of the invariant ensures that all addresses in $(roots \cup free)$ are always marked).

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

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

3.1 Sequential algorithm

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 in Section 4. This section introduces (in Fig. 1) 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 (see Sections 3.2 and 4).

As observed above, the full garbage collector repeatedly iterates the code called here *Collector*. This can be split into three phases. Providing the invariant is respected, *Mark*/*Sweep* do not depend on how many addresses are marked initially (*Unmark* is there to ensure that garbage is collected in at most two passes) but, thinking of the *Collector* being run intermittently, it is reasonable to start by removing any surplus marks.

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

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; 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. (Specifications of *Mark-kids* are in Sections 3.4 and 4.3.)

```
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 is running with no interference, R/G reasoning is not required and the specification of *Mark* and proof that the code in Fig. 1 satisfies that specification are straightforward. (In fact, they are simplified cases of what follows in Section 3.2.) When the same code is considered in the interfering environments in Sections 3.2 and 4, (differing) R/Gs and, of course, proofs are needed. The elaboration of the R/Gs is particularly interesting.

Lesson 6 *Considering the sequential case is useful because it is then possible to note how the rely condition (nothing changes) and the guarantee condition (***true***) need to be changed to handle concurrency.*

3.2 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 4 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 conditions. The argument to establish the upper bound for marking (and thus the lower bound of garbage collection) is given in Section 5.

If the *Mutator* were able to update and mark atomically, specifications and proofs are straightforward; although this atomicity assumption is unrealistic, it is informative to compare this section with Section 4. As adumbrated in Section 1, the argument is split into a justification of the parallel decomposition (Section 3.3) and the decompositions of the *Collector*/*Mutator* sub-components, addressed in Sections 3.4 and 3.5 respectively.

3.3 Parallel decomposition

An R/G specification of the *Collector* is:

Collector

ext wr $free, marked$
  rd $roots, hp$

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 cover interference.

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.[5]

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. The lower bound for garbage collection requires setting an upper bound for marking addresses; this topic is postponed to Section 5.

Lesson 7 *Such splitting of what would be an equality in the specification of a sequential component is a common R/G tactic.*

The corresponding specification of the *Mutator* is:

Mutator

ext wr $hp, free, marked$
  rd $roots$

pre true

rely $free \subseteq free'$

guar $free' \subseteq free \wedge marked \subseteq marked' \wedge$
  $\forall (a,i) \in \mathbf{dom}\, hp \cdot$
    $hp'(a,i) \neq hp(a,i) \wedge hp'(a,i) \in Addr \;\Rightarrow\; hp'(a,i) \in marked'$

post true

The R/G proof obligation (PO) for concurrent processes requires that each one's guarantee condition implies the rely condition of the other(s); in this case they match identically so the result is immediate.

[5] 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 nodes in *free* as required by the final conjunct of *inv*-Σ_2. Changing the guarantee conditions is routine but uninformative.

3.4 Developing the *Collector* code

As outlined in Section 1, what remains to be done is to develop code that satisfies the specification of the *Collector* (in isolation from that of the *Mutator*) — i.e. show that the decomposition of the *Collector* into three phases given in Section 3.1 satisfies the *Collector* specification in Section 3.3 and then to develop code for *Mark*.

A post condition for a sequential version of *Unmark* could constrain $marked'$ to be exactly equal to $roots \cup free$ but, again, interference must be considered. The rely condition indicates that the environment can mark addresses so whatever *Unmark* removes from *marked* could be replaced. The possible values notation is again deployed so that *post-Unmark* 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* itself first to mark an address and then remove the marking; this erroneous behaviour is ruled out by *guar-Unmark*. The rely condition indicates that the *free* set can also change but, since it can only reduce, this poses no problem.

Unmark

ext wr *marked*
  rd *roots*, *free*

pre true

rely $free' \subseteq free$
guar $marked' \subseteq marked$
post $\forall a \in (Addr - (roots \cup free)) \cdot \exists m \in \overset{\frown}{marked} \cdot a \notin m$

The relaxing of the post condition again uses the idea in Lesson 7.

The post condition for *Mark* also has to cope with the interference absent from a sequential specification and this requires more thought. In the sequential case, *post-Mark* could 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 5).

Mark

ext wr *marked*
  rd *roots*, *hp*, *free*

pre true

rely *rely-Collector*
guar $marked \subseteq marked'$
post $reach(marked, hp') \subseteq marked'$

The relaxing of the post condition once again uses the idea in Lesson 7. Similar observations to those for *Unmark* relate to the specification of *Sweep* which, for the concurrent case, becomes:

Sweep

ext wr *free*
  rd *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* are distributed (with appropriate weakening/strengthening) over the three sub-components; all of the pre conditions are **true**; so the remaining PO is:

$$post\text{-}Unmark(\sigma,\sigma') \wedge post\text{-}Mark(\sigma',\sigma'') \wedge post\text{-}Sweep(\sigma'',\sigma''') \Rightarrow post\text{-}Collector(\sigma,\sigma''')$$

The proof is straightforward.

Turning to the decomposition of *Mark* (see Fig. 1), in order to prove *post-Mark*, a specification is needed for *Propagate* that copes with interference:

Propagate

ext wr *marked*
  rd *hp*

pre true

rely *rely-Collector*
guar $marked \subseteq marked'$
post $\bigcup\{\mathbf{elems}\, hp'(a) \cap Addr \mid a \in marked\} \subseteq marked' \wedge$
  $(marked \subset marked' \vee reach(roots, hp') \subseteq marked')$

The first conjunct of the post condition indicates the progress required of the wave of marking; the second triggers further iterations if any marking has occurred.

To prove the lower marking bound (i.e. must mark everything that is reachable from *roots*), we use an argument 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:

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

The PO is:

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

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 one (cf. Section 5).

Then trivially:

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

Pursuing the decomposition of *Propagate* (again, see Fig. 1) needs a specification of the inner operation:

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

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

$$\textit{so-far}(\sigma,\sigma') \;\triangleq\; \bigcup\{\textbf{elems}\, hp(a) \cap \textit{Addr} \mid a \in (\textit{marked} \cap \textit{consid}')\} \subseteq \textit{marked}'$$

The relevant PO is:

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

whose discharge is obvious.

The final obligation is to show:

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

The first conjunct of *post-Propagate* is straightforward; the fact that (unless the marking process is complete) some marking must occur in this iteration of *Propagate* follows from the lemma in Section 2.2.

3.5 Checking the interference from *Mutator*

The mutator is viewed as an infinite loop non-deterministically selecting one of *Redirect*, *Malloc*, *Zap* as specified below. At this stage, these are viewed as atomic

operations so no R/Gs are supplied here:[6] their respective post conditions must be shown to imply *rely-Mark*):

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

It follows trivially that:

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

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

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

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

Malloc preserves the invariant because *inv*-Σ_2 insists that free addresses are always marked. It follows trivially that:

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

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

It again follows trivially that:

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

4 Relaxing atomicity: reasoning using a ghost variable

The interesting challenge remaining is to consider the impact of acknowledging that the atomicity assumption in Section 3.2 about the mutator is unrealistic. Splitting the atomic assignment (on the two shared variables $hp, marked$) in Section 3.5 is delicate. The reader would be excused for thinking that performing the marking first would be safe but there is a counter example in the case where the *Collector* executes *Unmark* between the two steps of such an erroneous *Redirect*.

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 all we need to know about the mutator) with standard rely conditions.

[6] The all-important non-atomic case for *Redirect* is covered in Section 4.

Lesson 8 *Ghost variables can undermine compositionality/separation (cf. Lesson 3). Where they appear to be essential, it would be useful to have a test for this fact.*

The difficulty can be understood by considering the following scenario. *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* of Section 3.4 relies. There is in fact no danger since, even if b was not marked, there must be another path to b (see *pre-Redirect* in Section 3.5) and the *Collector* should perform the marking when that path (say $hp(c,j)$) is encountered. If, however, that $hp(c,j)$ could be destroyed before the *Collector* 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.[7]

It is precisely this three step argument that pinpoints the limitation of using two state relations in R/G reasoning.

Here, despite all of the reservations expressed in Section 1, a ghost variable is employed to complete the justification of the design;thus the state Σ_2 is extended with a ghost variable tbm: $\lceil Addr \rceil$ that can record an address as "to be marked".

Lesson 9 *Lesson 8 asks for a test that would justify the use of ghost variable; the need for a "3-state" justification is such a test.*[8]

4.1 Parallel decomposition

The rely condition used in Section 3.3 is replaced for the non-atomic interference from the mutator by:

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

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

Here again, the PO of the parallel introduction rule is trivial to discharge because the guarantee condition of the *Mutator* is identical with the rely condition of the *Collector*.

---

[7] This line of argument rules out multiple *Mutator* threads.

[8] A planned journal version of this paper will investigate other options. It is also hoped to compare the approaches with RGITL [STER11].

4.2 Developing *Mutator* code

As indicated in Section 1, it still remains to establish that the design of each component satisfies its specification. Looking first at the non-atomic *Mutator* argument, the only real challenge is:[9]

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-Collector*
post $hp' = hp \dagger \{(a,i) \mapsto b\} \wedge b \in marked'$

Redirect can satisfy this specification by executing the following two atomic steps (but the atomic brackets only surround one shared variable in each case):

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

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

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

4.3 Developing *Collector* code

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

Mark
ext wr *marked*
  rd *roots*, *hp*, *free*
pre true
rely *rely-Collector*
guar $marked \subseteq marked'$
post $reach(marked, hp') \subseteq marked'$

The code for *Mark* is still that in Fig. 1 — under interference, the post condition of *Propagate* has to be further weakened (from Section 3.4) 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:

[9] When removing a pointer, no *tbm* is set — see $Zap(a,i)$ in Section 3.5; also no *tbm* is needed in the *Malloc* case because *inv*-Σ_2 ensures that any *free* address is marked.

Propagate

ext wr *marked*
  rd *hp*

pre true

rely *rely-Collector*
guar $marked \subseteq marked'$
post $\bigcup\{\mathbf{elems}\, hp'(a) \cap Addr \mid a \in marked\} \subseteq (marked' \cup (\{tbm'\} \cap Addr)) \wedge$
  $(marked \subset marked' \vee reach(roots, hp') \subseteq marked')$

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 *rely-Collector*.

The correctness of this loop is interesting — it follows the structure of that in Section 3.4 using a *to-end* relation and, in fact, the relation is still:

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

The PO is now:

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

In comparison with the PO in Section 3.4, the difficult case is where $tbm' \neq \mathbf{nil}$ (in the converse case the earlier proof would suffice). What needs to be shown is that the stray address in tbm' will be marked. The lemma in Section 4.2 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 the lemma at the end of Section 2.2 which, combined with the second conjunct of *post-Propagate*, avoids premature termination.

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

Mark-kids (x:*Addr*)

ext wr *marked*
  rd *hp*

pre true

rely *rely-Collector*
guar $marked \subseteq marked'$
post $(\mathbf{elems}\, hp'(x) \cap Addr) \subseteq (marked' \cup (\{tbm'\} \cap Addr)))$

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

$$so\text{-}far(\sigma, \sigma') \quad \triangleq \quad \bigcup\{\mathbf{elems}\, hp(a) \cap Addr \mid a \in (marked \cap considered')\} \subseteq (marked' \cup (\{tbm'\} \cap Addr))$$

5 Lower limit of GC

Sections 3.2 and 4 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:

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

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

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".

6 Related work

The nine lessons are the real message of this paper; the (garbage collection) example illustrates and hopefully clarifies the issues for the reader. The current authors believe that examples are essential to drive research.

Many papers exist 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 a very large number 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 to our approach is [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 Collector. Similarly, in [VYB06], a series of transformations is used to derive various concurrent garbage collection algorithms from an initial algorithm.

Acknowledgements

We have benefitted from productive discussions with researchers including José Nuno Oliviera 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.

Our colleagues in Newcastle, Leo Freitas and Diego Machado Dias are currently formalising proofs of the lemmas and POs using Isabelle.

The authors gratefully acknowledge funding for their research from EPSRC grant *Taming Concurrency*.

References

[BA84] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages ans 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.

[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.

[JH16a] 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.

[JH16b] 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.

[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.

[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.