Compositionality and Expressiveness (with Rely/Guarantee notation)

Cliff Jones

(joint work with: Andrius Velykis and Nisansala Yatapanage)
Newcastle University

NCWG @ Teesside
2017-01-13

Compositionality

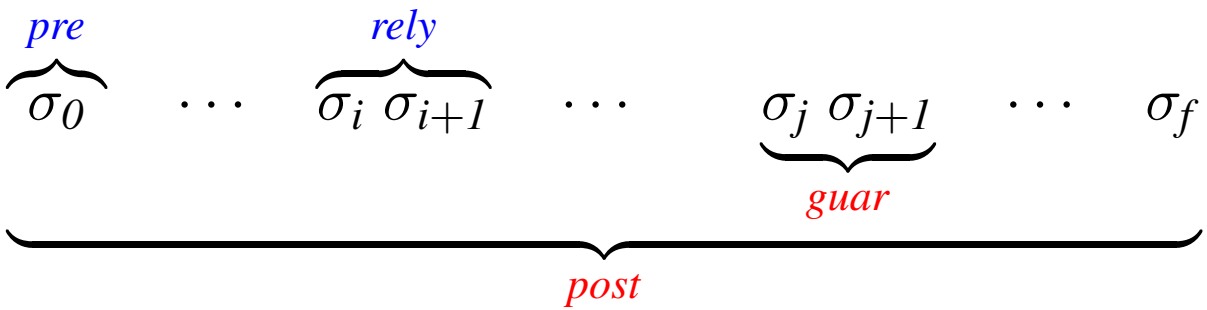
- when we write:
  $\{\textbf{true}\}SORT\{s\text{-}ordered(s') \wedge is\text{-}permutation(s', s)\}$
  we accept any (proven) implementation
- we have chosen what to express! (relation between states)
- post conditions take a position on what is observable
- notice (engineering):
  $\{\textbf{true}\}SIEVE\{s' = s - Composites\}$ is more useful than
  $\{s = \{1..n\}\}SIEVE\{s' = s - Composites\}$
  because the former is less reliant on the environment
  $INIT: s \leftarrow \{1..n\}$ being only one possibility
- compositionality is **crucial** for concurrency
- as with $SIEVE$ above, it's about separation
  - moot with symmetric processes (e.g. $SIEVE =||_i REM(i)$)
  - clearer with asymmetric processes: different R/Gs
  - e.g. $QREL$ (aka "union/find")

Document interference with Rely/Guarantee (R/G)

- using R/G also takes a position on what can be expressed
- *rely* relations abstract the interference to be tolerated
- R/G is expressively weak
  - only relations
  - (same as for post conditions)
  - reflexive (for no interference); transitive (for multiple steps)
- maybe the weakness is their strength?
  - simple(r) POs
  - clear separation
- the other extreme
  - the code of the environment
  - makes it possible to reason about all interleavings
  - unattractive: non-compositional and brittle

Rely/Guarantee (R/G) idea is simple

. . . but requires adopting an R/G mindset!



- assumptions $pre/rely$
- commitments $guar/post$
- reduce to non-concurrent by rely: $x' = x$; guar: **true**

Prejudices

- post conditions should be relations [Jon80, Jon86]
  - well-founded loop relation vs. “variant function” arguments
- data abstraction/reification
  - is at least as important as control structures
  - this is reinforced in concurrency [Jon07, JY15]

Undermining compositionality

- minimal information about the (concurrent) environment
  - R/G facilitates this
  - clearest in examples of asymmetric concurrent processes
  - Owicki/Gries reasons about the final code!
- "ghost" variables dent/destroy compositionality
  - in the extreme, they can dictate the entire environment code
    e.g. $(PC = 1 \Rightarrow x = 5) \wedge (PC = 2 \Rightarrow x = 6) \wedge \cdots$
  - in many cases, such "auxiliary" variables can be avoided
  - if I have to use them, I want a test
    (cf. homomorphic rule vs. "biased" specifications)

Possible values

- in the presence of interference
  $\{P\}x \leftarrow y\{x' = y \vee x' = y'\}$
  is not enough — write:
  $\{P\}x \leftarrow y\{x' \in \widehat{y}\}$
- need identified in [JP11]
  - development of Simpson's "4-slot" algorithm for ACMs
  - (asymmetric $WRITE \mid\mid READ$)
- [JH16] has a beautifully clear specification of ACMs
- posvals extends expressive power
- . . . avoids some apparent "needs" for ghost variables
- have to decide if this limits implementations too much!

Example: GC

it is the underlying points, not the example, that matter

- concurrent Garbage Collection: Ben Ari [BA84]
- an “Owicki/Gries” proof by Jan van de Snepscheut [vdS87]
- illustrates:
  - asymmetric processes (and therefore R/G): $Collector \,||\, Mutator$
  - three $Mutator$ options: $Redirect/Malloc/Zap$
- highlights a limitation(!) of R/G
  - let’s us explore options

(Concurrent) garbage collection: abstract specification

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

where

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

GC: $Collector^*$

$Collector$
wr $free$
rd $busy$
pre true
...
post $(Addr - busy) \subseteq \bigcup \overset{\frown}{free}$ lower bound for GC

data reification gets us to:

$$\Sigma_1 :: roots: Addr\text{-}\mathbf{set}$$
$$hp: Heap$$
$$free: Addr\text{-}\mathbf{set}$$

where

$$inv\text{-}\Sigma_1(mk\text{-}\Sigma_1(roots, hp, free)) \triangleq$$
$$\mathbf{dom}\, hp = Addr \wedge$$
$$free \cap reach(roots, hp) = \{\}$$

upper bound for GC

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

$$Node = Addr^*$$

Simplification: here **nil** is ignored

Mark any *Addr* reachable from *roots*

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

$\Sigma_2 :: roots: Addr\text{-}\mathbf{set}$
$hp: Heap$
$free: Addr\text{-}\mathbf{set}$
$marked: Addr\text{-}\mathbf{set}$

where

$inv\text{-}\Sigma_2(mk\text{-}\Sigma_2(roots, hp, free, marked)) \triangleq$
$\mathbf{dom}\, hp = Addr \wedge$
$free \cap reach(roots, hp) = \{\} \wedge$
$(roots \cup free) \subseteq marked$

Stage 1: non-interfering case

(transition from sequential to interfering shows an R/G "pattern")

$Mark$

wr $marked$

rd $hp, roots, free$

pre true

rely $marked' = marked \wedge free' = free \wedge hp' = hp$

guar true

post $marked' = (free \cup reach(roots, hp))$

$Mark \triangleq$

repeat

$mc \leftarrow$ **card** $marked$;

$Propagate$

until card $marked = mc$

$Propagate$:

$consid \leftarrow \{\}$;

do while $consid \neq Addr$

let $x \in (Addr - consid)$ **in**

if $x \in marked$ **then** $Mark\text{-}kids(x)$ **else skip**;

$consid \leftarrow consid \cup \{x\}$

od

proofs of (sequential) loops are straightforward

Stage 2: accept concurrent changes to *heap*

have to enlist *Mutator* to help with marking process!

Unmark
wr *marked*
rd *free*
pre true
rely $marked \subseteq marked' \wedge free' \subseteq free$
guar $marked' \subseteq marked$
post $\forall a \in (Addr - (roots \cup free)) \cdot \exists m \in \overbrace{marked} \cdot a \notin m$

simplification: upper bound of marking (lower of GC) ignored for now

Mark: (simplified)

Pretend $Mutator(Redirect)$ can make change and mark atomically

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

Mark
wr $marked$
rd $hp, roots, free$
pre true
rely $marked \subseteq marked' \wedge$
$\forall (a,i) \in \mathbf{dom}\, hp \cdot hp'(a,i) \neq hp(a,i) \Rightarrow hp'(a,i) \in marked'$
guar $marked \subseteq marked'$
post $reach(roots, hp') \subseteq marked'$

slightly more complicated but previous proof *strategy* works
(with exactly the same code — just different R/G)

Stage 3: concurrent case

. . . the *Mutator* cannot redirect/mark atomicaly

- I have worked on "fiction of atomicity/atomicity refinement" but I don't think that's the right approach here
- the essence of the correctness is a "three state" argument:
  Mutator can change $hp(a, i))$ to point to b
  but then go to sleep
  if the $Collector/Proagate$ has already passed a, its (obvious) post condition fails
  but we will get to another pointer to b
  unless the *Mutator* destroys that link

  but in that case, the *Mutator* must have marked b to get to that point
- such a three state argument is not expressible as relations

Concurrent case: alternatives

- reject the restrictions of R/G and use Temporal Logic
  - I worry that RGITL is too expressive!
  - slippery slope to arguing about the combined code
- concede a ghost variable!

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

but we would have a test!

- either of these alternatives damage compositionality!
- I think that one could specify that the *Mutator* preserves *can-be-completed*
- destroys compositionality! the designer of *Mutator* . . .

(My) current preferred alternative

Consider:

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

With:
$Collector \mid\mid (< mr\text{-}1 >; < mr\text{-}2 >)^*$
the $Collector$ has to tolerate interference in the following cases:
$(< mr\text{-}1 >; < mr\text{-}2 >)^*$ (which is covered above)
$(< mr\text{-}1 >; < mr\text{-}2 >)^*; < mr\text{-}1 > (guar\text{-}mr\text{-}2 \Leftrightarrow hp' = hp)$
$(< mr\text{-}2 >)$ (OK here this is like the counter-example if the $Collector$ marked first)

Clearly this also limits compositionality — but . . .
Comment (most) welcome

Conclusions

- the layered R/Gs are rather nice!
- compositionality = good engineering
- compositionality wrt a notation
- R/G can't specify GC fully compositionally
  - the 3-state argument looks like a clear test
- the "jury is still out" on the least damaging solution
- actually, I think all candidate workarounds tell us something
- (for me) examples are essential!

Background R/G literature

- [Jon81, Jon83a, Jon83b] used keywords (VDM style)
- elsewhere presented as $\{P, R\}\ S\ \{G, Q\}$
- . . . especially for the proof rules
- useful intro [Jon96]
- many uses — including Bornat's 4-slot [BA10, BA11]
- more recently, with Ian Hayes [HJC14, JHC15]:
  - R/G completely recast in refinement calculus style
  - Ian's newer semantics gets more out of algebraic properties
- links to Separation Logic [JY15]


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


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


Richard Bornat and Hasan Amjad.
Explanation of two non-blocking shared-variable communication algorithms.
Formal Aspects of Computing, pages 1–39, 2011.


Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin.
Laws and semantics for rely-guarantee refinement.
Technical Report CS-TR-1425, Newcastle University, July 2014.


Cliff B. Jones and Ian J. Hayes.
Possible values: Exploring a concept for concurrency.
Journal of Logical and Algebraic Methods in Programming, 2016.


Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin.
Balancing expressiveness in formal approaches to concurrency.
Formal Aspects of Computing, 27(3):465–497, 2015.


C. B. Jones.
Software Development: A Rigorous Approach.
Prentice Hall International, Englewood Cliffs, N.J., USA, 1980.


C. B. Jones.
Development Methods for Computer Programs including a Notion of Interference.
PhD thesis, Oxford University, June 1981.
Printed as: Programming Research Group, Technical Monograph 25.


C. B. Jones.
Specification and design of (parallel) programs.
In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.


C. B. Jones.
Tentative steps toward a development method for interfering programs.
Transactions on Programming Languages and System, 5(4):596–619, 1983.


C. B. Jones.
Systematic Software Development Using VDM.
Prentice Hall International, 1986.


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.


C. B. Jones.
Splitting atoms safely.
Theoretical Computer Science, 375(1–3):109–119, 2007.


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.


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.


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