Reasoning about Interference (or lack thereof)

Cliff Jones

Newcastle University

TUV
2016-03-08

My prejudices

- verification tackled by von Neumann and Turing
  - the search has been for ‘tractability’ [Jon03]
- the real payoff from formal methods is in design
  - code analysis can detect errors
  - unfortunately it does!
  - is it more productive to avoid their insertion?
  - formalism should support/check intuition
  - ‘posit and prove’
  - furthermore: a genie offers you a choice ...
- we understand complex systems from the top-down
  - i.e. via abstractions
  - ‘compositionality’ is a practical concern
- what if one is stuck with 10^n lines of legacy code?
  - top-down abstractions are useful in bottom-up analysis

Some lessons from VDM

VDM program development [Jon80]; VDM language description [BJ82]

- predicate restricted types (thanks to Lockwood Morris)
- relations (over Σ) are essential for post conditions ($\ldots$)
- 'real world' specifications don't fit on a line
    - keywords for structure
    - ... also for defining (**rd**/**wr**) 'frame'
- data abstraction/reification crucial in design
    - retrieve functions
    - 'adequacy'
- (avoid) 'implementation bias'
    - but there are understood cases where one can't
- cf. reservations about ghost (auxiliary) variables

Concurrency

- my main interest is in ‘shared variable’ concurrency
  - others have developed similar ideas for processes (communication-based) concurrency
  - interference: because construct ‘variables’ in π-calculus
- the real payoff from formal methods is in design
- concurrency
  - inherent in some top-level applications
  - for performance = distribute computation or data
- ‘compositionality’
  - is a practical concern
  - is difficult to achieve for concurrency

Issues in concurrency

look at issues (don't start with notation(s))

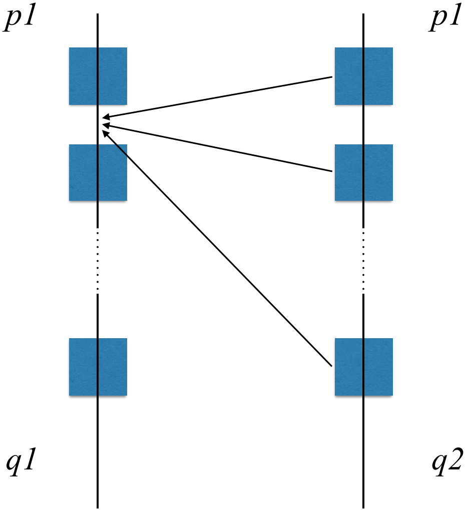
- interference
- separation
- ownership
- 'linearisability' (vs. splitting atoms)
- progress
- ...
- distributing data

Concurrency: pre Owicki

- Hoare (post *Axiomatic Basis*)
  - [Hoa69] — pick up [Hoa72] under "separation"
- interference (i.e. shared alphabets)
- Ashcroft [Ash75] (TR in 1973)
  - proof of "cross product" of control points
  - labour intensive!
- completely *post facto*
- non compositional
- arbitrary/fixed granularity assumption
  - assignments taken to be atomic
  - cf. so-called "Reynold's rule"

Susan Owicki [Owi75]

supervised by David Gries



Owicki/Gries [Owi75, OG76]

- interference (i.e. shared alphabets)
- separate sequential reasoning
- *post facto*: final ‘Einmischungsfrei’ PO
- non compositional
- arbitrary/fixed granularity assumption

Rely/Guarantee (R/G) idea is simple

face interference (in specifications and design process)

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

- assumptions $pre/rely$
- commitments $guar/post$

rely relations are an abstraction of interference to be tolerated
their expressive weakness might be a good thing!
'power' can beget intractability

One 5-tuple proof rule

many other possible rules

$$\boxed{Par\text{-}I}\ \frac{\begin{array}{l}\{P, R \vee G_2\}\ S_1\ \{G_1, Q_1\}\\ \{P, R \vee G_1\}\ S_2\ \{G_2, Q_2\}\end{array}}{\{P, R\}\ S_1 \mid\mid S_2\ \{G_1 \vee G_2, Q_1 \wedge Q_2 \wedge (R \vee G_1 \vee G_2)^*\}}$$

scope for variation in rules *much* larger (than in Hoare logic)
here: for composition (more compact than decomposition)
but, actually, less useful

Developments from R/G

- early
  - over 20 theses
  - zB Ketil Stoelen deals with progress
  - ...
  - Leonor Prensa Nieto formalise soundness proof in Isabelle
- ...
- recent
  - RGSep [Vaf07]
  - Bornat (twice) on Simpson's '4-slot' algorithm
  - (Concurrent) Kleene Algebras
  - Armstrong (2016) looks at soundness via Kleene Algebras

R/G rethought [JHC15, HJC14]

"pulling apart" R/G

Original rely/guarantee (R/G) can be presented as 5-tuples

But instead of
$\{P, R\}\ S\ \{G, Q\}$

We now follow the 'refinement calculus':
$x{:}\ \left[P, Q\right]$
$x{:}\ \left[Q\right]$
$=\ /\ \sqsubseteq$

and wrap **rely**/**guar** around *any* statement
rely $R \bullet c$
guar $G \bullet c$

(Some) Laws of the new algebraic R/G

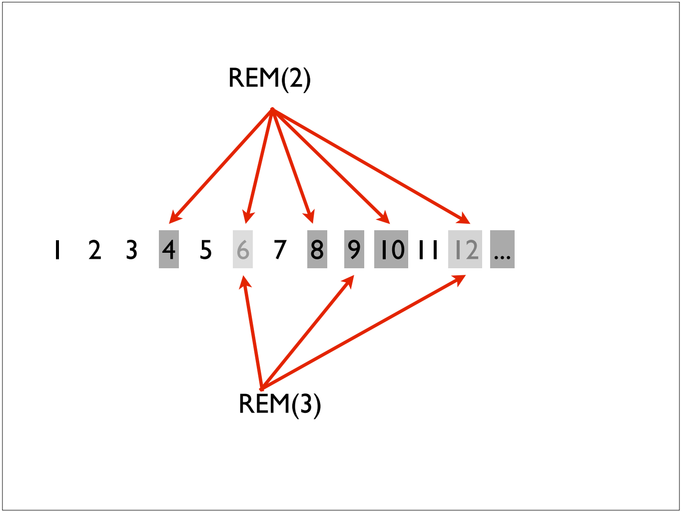
Nested-G: $(\textbf{guar}\ g_1 \bullet (\textbf{guar}\ g_2 \bullet c)) = (\textbf{guar}\ g_1 \wedge g_2 \bullet c)$

Intro-G: $c \sqsubseteq (\textbf{guar}\ g \bullet c)$

Trading-G-Q: $(\textbf{guar}\ g \bullet [g^* \wedge q]) = (\textbf{guar}\ g \bullet [q])$

Intro-multi-Par: $\wedge_i [q_i] \sqsubseteq \|_i\ (\textbf{guar}\ gr \bullet (\textbf{rely}\ gr \bullet [q_i]))$
(asymmetric version below)

(Parallel) Sieve of Eratosthenes



Refinement calculus style development

Set s might initially contain all natural numbers up to some n
C is the set of all composite numbers

$$[s' = s - C] = [s' \subseteq s \wedge s - s' \subseteq C \wedge s' \cap C = \{\,\}]$$

$\sqsubseteq$ by Intro-G

$$\textbf{guar}\; s' \subseteq s \wedge s - s' \subseteq C \bullet [s' \subseteq s \wedge s - s' \subseteq C \wedge s' \cap C = \{\,\}]$$

$=$ by Trading-G-Q ($s - s' \subseteq C$ is transitive)

$$\textbf{guar}\; s' \subseteq s \wedge s - s' \subseteq C \bullet [s' \cap C = \{\,\}]$$

$\sqsubseteq$ by Intro-multi-Par

$$\textbf{guar}\; s' \subseteq s \wedge s - s' \subseteq C \bullet \\ (\|_i \;\textbf{guar}\; s' \subseteq s \bullet \textbf{rely}\; s' \subseteq s \bullet [s' \cap c_i = \{\,\}])$$

$=$ Nested-G

$$\textbf{guar}\; s - s' \subseteq C \wedge s' \subseteq s \bullet (\|_i \;\textbf{rely}\; s' \subseteq s \bullet [s' \cap c_i = \{\,\}])$$

R/G observations

- asymmetric rely/guarantee conditions are important:
  $[q_1 \wedge q_2] \sqsubseteq (\textbf{guar}\ g_1 \bullet (\textbf{rely}\ g_2 \bullet [q_1])) \parallel (\textbf{guar}\ g_2 \bullet (\textbf{rely}\ g_1 \bullet [q_2]))$
- nice bonus of new style: **guar-inv** $g \bullet c$
- apposite representations often key to avoiding locking
  - FINDP
  - QREL — (spotted in design of CLEANUP)
  - SIEVE
  - 4-SLOT
  - only fully realised in [Jon07]
- (yet more) 'abstract R/G' (Ian Hayes looking at **guar** $c \bullet c'$)
  - cf. 'phasing', but
  - remember 'expressive weakness' point!

Another issue: ‘separation’

- ‘separation’ = ‘non-interference’
  - return to this later
- see what (data) abstraction can do for interference
  - retaining my top-down prejudice
- ordinary (‘scoped’) variables assumed to be separate
  - leave aside ‘by location’ parameter passing!
- towards a different view
  - ‘heap’ variables as representations of scoped variables?
- this exercise in the same spirit as ‘taking apart’ R/G

Separation Logic (SL)

refresher!

- basic idea is, again, simple
  - to prove things about $S_1 \parallel S_2$
  - would like to conjoin their pre/post conditions
- history
  - parallelism with ‘scoped’ variables — [Hoa72]
  - mentioned by Peter O’Hearn at Tony’s 2009 event
  - ‘Separation Logic’ for ‘heap’ variables — [Rey02]
  - Concurrent Separation Logic — Peter O’Hearn [O’H07]
- ‘heap’ variables can’t be handled by ‘alphabets’
  - SL designed for this case
- origin: bottom-up code analysis
  - heap variables
  - “probably avoid SL for ‘scoped’ variables!”

Two key SL proof rules

'Separating conjunction' – $P * Q$ (only if P and Q are separate)

$$\boxed{\text{SL-concurrency}}\ \frac{\begin{array}{l}\{P_1\}\ s_1\ \{Q_1\}\\ \{P_2\}\ s_2\ \{Q_2\}\end{array}}{\{P_1 * P_2\}\ s_1 \mid\mid s_2\ \{Q_1 * Q_2\}}$$

Frame rule

$$\boxed{\text{SL-frame}}\ \frac{\{P\}\ s\ \{Q\}}{\{P * R\}\ s\ \{Q * R\}}$$

‘Separation as an abstraction’

[JY15]

two examples:

- Reynold’s simple sequential (in-place) list reversal
- a concurrent merge sort

Example 1: list reversal example (Reynolds)

John started . . .

The following *program* performs an in-place reversal of a list:

$j := \textbf{nil};\ \textbf{while}\ i \neq \textbf{nil}\ \textbf{do}$
$\quad (k := [i + 1];\ [i + 1] := j;\ j := i;\ i := k).$

(Here the notation $[e]$ denotes the contents of the storage at address e.)

He then derives a post condition using $\exists \alpha, \beta \cdot list(\alpha, i) * list(\beta, j)$

Re-do the example with
‘Separation as an abstraction’

$$r, s \colon \left[r' = rev(s) \right]$$

$$rev : X^* \to X^*$$
$$rev(s) \quad \triangleq \quad \cdots$$

s and r are *assumed* to be distinct (‘scoped’) variables
that they are separate is a (useful and) natural abstraction

first step of design

$$\Sigma_0 = X^* \times X^*$$

it is straightforward to 'posit & prove':

$r \leftarrow [\,]$;
while $s \neq [\,]$ **do**
  $STEP_0$
  $\{rev(s') \frown r' = rev(s) \frown r \wedge \textbf{len}\, s' < \textbf{len}\, s\}$
od

$$STEP_0 \;\; r, s\colon \left[s \neq \{\,\},\; r' = [\textbf{hd}\, s] \frown r \wedge s' = \textbf{tl}\, s\right]$$

We have finished thinking about reversing sequences!
We now think about data (representation)
NB: s and r are still assumed to be distinct variables

($Heap$ but) $Srep$ is a useful abstraction

$$Heap = Ptr \xrightarrow{m} (X \times [Ptr])$$

Inductive definition of $Srep \subseteq Heap$

$$\{\,\} \in Srep$$

$$sr \in Srep \land p \in Ptr \land p \notin \mathbf{dom}\, sr \;\Rightarrow\; (\{p \mapsto (v, start(sr))\} \cup sr) \in Srep$$

$$start(\{\,\}) = \mathbf{nil}$$

$$start(\{p \mapsto (v, start(sr))\} \cup sr) = p$$

Could develop a theory of $Srep$ (e.g. Isabelle)

Reify $(X^* \times X^*)$ as Σ_1

$$\Sigma_1 = (Srep \times Srep)$$

where

$$inv\text{-}\Sigma_1((sr, rr)) \quad \triangleq \quad sep(sr, rr)$$

$$sep : Srep \times Srep \to \mathbb{B}$$

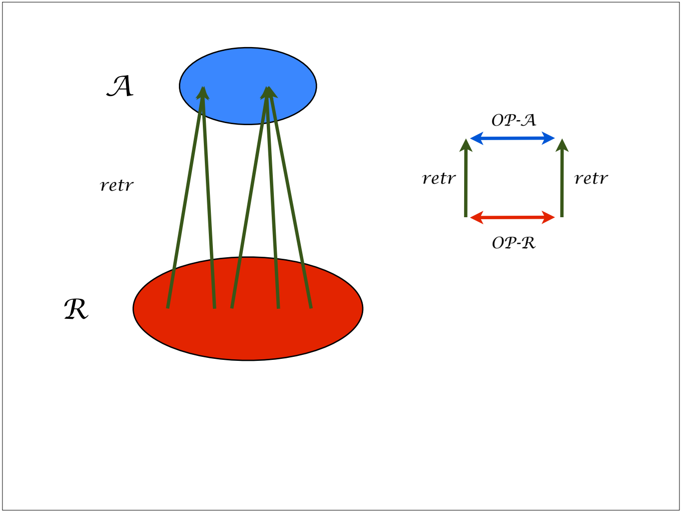
$$sep(sr, rr) \quad \triangleq \quad \mathbf{dom}\, sr \cap \mathbf{dom}\, rr = \{\,\}$$

$STEP$ on Σ_1

$$
STEP_1 \quad rr, sr: \left[\begin{array}{l}
sr \neq \{\,\}, \\
\textbf{let } p = start(sr) \textbf{ in} \\
\quad sr' = \{p\} \triangleleft sr \wedge \\
\quad rr' = rr \cup \{p \mapsto (sr(p)_1, start(rr))\}
\end{array}\right]
$$

Lemma 1 $STEP_1$ preserves $inv\text{-}\Sigma_1$

Data reification (homomorphic rule)



Data reification proof — standard (VDM) rule

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

$$retr_0((sr, rr)) \quad \triangleq \quad (gather(sr), gather(rr))$$

$$gather: Srep \to X^*$$
$$gather(\{\ \}) = [\,]$$
$$gather(\{p \mapsto (v, start(sr))\} \cup sr) = [v] \curvearrowright gather(sr)$$

Lemma 2 ('Adequacy') There is a Σ_1 representation of any Σ_0

Lemma 3 ('Commutativity') $STEP_1$ models (under $retr_0$) the abstract $STEP_0$

Reification to a single $Heap$

$$\Sigma_2 = (Heap \times Ptr \times Ptr)$$

where

$$\begin{array}{l} inv\text{-}\Sigma_2((hp, i, j)) \quad \triangleq \\ \qquad \exists sr, rr \in Srep \cdot sr \cup rr \subseteq hp \ \wedge \ i = start(sr) \wedge j = start(rr) \end{array}$$

- another exercise in data reification
- it is mandatory that sep holds between the two sub-heaps because their union is used in $(sr \cup rr) \subseteq hp$
- NB $\subseteq$ admits the possibility of other information in the heap

Relating Σ_2/Σ_1

$$retr_1 : \Sigma_2 \to \Sigma_1$$

$$retr_1((hp, i, j)) \quad \triangleq \quad (trace(hp, i) \lhd hp, trace(hp, j) \lhd hp)$$

$$trace : Heap \times Ptr \to Ptr\text{-}\mathbf{set}$$

$$trace(hp, p) \quad \triangleq \quad \begin{array}{l} \mathbf{if}\ p = \mathbf{nil} \\ \mathbf{then}\ \{\,\} \\ \mathbf{else}\ \{p\} \cup trace(hp, hp(p)_2) \end{array}$$

Lemma 4 $trace$ from $start(sr)$ characterises sr

Lemma 5 ('Adequacy') of Σ_2 wrt $\Sigma_1/retr_1$

Theorem 1 ('Commutativity') $STEP_2$ models (under $retr_1$) $STEP_1$

Comments on Example 1

- *separation is an abstraction*
  - wot, no Separation Logic?
  - representation shown to preserve the abstraction
  - standard reification process
- wot, no R/G?
  - no concurrency
  - therefore, no interference
- layered design
  - (only) first step is concerned with list reversal
  - second is (only) about data representation
- (C++) code in the paper

Example 2: *concurrent* merge sort

SKIP!

$$is\text{-}sort : X^* \times X^* \to \mathbb{B}$$

$$is\text{-}sort(s, s') \quad \triangleq \quad ordered(s') \wedge permutes(s', s)$$

$$ordered : X^* \to \mathbb{B}$$

$$ordered(s) \quad \triangleq \quad \cdots$$

$$permutes : X^* \times X^* \to \mathbb{B}$$

$$permutes(s, s') \quad \triangleq \quad \cdots$$

Because concurrent processes are used, employ *Intro-par*

What has been achieved?

- reason about *separation as an abstraction*
  - only standard (long-established) notions
- (like all reification steps) argue properties preserved
- key sorting ideas proved on the abstraction
  - only need to show the implementation mirrors steps
  - echoes Wirth: *Algorithms + Data = Programming*
- minimal use of R/G, mainly abstraction!
- this is *not* an argument against SL
  - … a nice definition of $Srep$ uses separating conjunction (*)
  - as with ‘pulling apart’ R/G, get to issue (of separation)

A (non-specialist's) view of SL

- *basic idea* works well for 'disjoint concurrency'
  - e.g. parallel merge sort
- (most papers) limit to 'partial correctness'
- (too?) many extensions
  - magic wand (fits algebraic view)
  - fractional permissions Boyland
  - Concurrent Abstract Predicates [DYDG$^+$10]
  - *Next 700 Separation Logics* [Par10]
- conceptual framework
  - monoids
  - Abstract Separation Logic [COY07]
  - Views [DYBG$^+$13]
- is it better to have everything under one (conceptual) roof?
  - vs. (?)
  - 'natural abstractions'

Relating interference/separation

compare with RGSep [Vaf07]

A key (abstract) R/G law

$[q_1 \wedge q_2] \sqsubseteq (\textbf{guar}\ g_1 \bullet (\textbf{rely}\ g_2 \bullet [q_1])) \parallel (\textbf{guar}\ g_2 \bullet (\textbf{rely}\ g_1 \bullet [q_2]))$

... covers complete or partial separation

Conclusions

- don't take position:
  "my notation (aka hammer) solves every problem"
- beware the siren the call of 'universality'
- but 'abstraction' is a/the key to understanding
- start with the issues
  - interference
  - separation
  - ownership
  - progress
  - 'linearisability' (vs. splitting atoms)
  - ...
  - distributing data

Edward A Ashcroft.
Proving assertions about parallel programs.
Journal of Computer and System Sciences, 10(1):110–135, 1975.

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

Dines Bjørner and Cliff B. Jones, editors.
Formal Specification and Software Development.
Prentice Hall International, 1982.

Cristiano Calcagno, Peter O'Hearn, and Hongseok Yang.
Local action and abstract separation logic.
In *LICS 2007*, pages 366–378. IEEE, 2007.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang.
Views: compositional reasoning for concurrent programs.
In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–300. ACM, 2013.

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.

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.

C. A. R. Hoare.
An axiomatic basis for computer programming.
Communications of the ACM, 12(10):576–580, 583, October 1969.

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

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.

Cliff B. Jones.
The early search for tractable ways of reasonning about programs.
IEEE, Annals of the History of Computing, 25(2):26–49, 2003.

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

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.

S. S. Owicki and D. Gries.
An axiomatic proof technique for parallel programs I.
Acta Informatica, 6:319–340, 1976.

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

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

Matthew Parkinson.
The next 700 separation logics.
volume 6217 of *LNCS*, pages 169–182. Springer, 2010.

John Reynolds.
A logic for shared mutable data structures.
In Gordon Plotkin, editor, *LICS 2002*. IEEE Computer Society Press, July 2002.

Viktor Vafeiadis.
Modular fine-grained concurrency verification.
PhD thesis, University of Cambridge, 2007.