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 $\sum$) are essential for post conditions (…) 
- ‘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 $\pi$-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)

\[\begin{align*}
\text{pre} & : \sigma_0 \quad \cdots \quad \sigma_i \quad \sigma_{i+1} \quad \cdots \quad \sigma_j \quad \sigma_{j+1} \quad \cdots \quad \sigma_f \\
\text{rely} & : \\
\text{guar} & : \\
\text{post} & : 
\end{align*}\]

- assumptions \textit{pre/rely}
- commitments \textit{guar/post}

\textit{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

\[
\begin{array}{c}
\{P, R \lor G_2\} \quad S_1 \quad \{G_1, Q_1\} \\
\{P, R \lor G_1\} \quad S_2 \quad \{G_2, Q_2\}
\end{array}
\]

\[
\frac{\{P, R\} \quad S_1 \ || \ S_2 \quad \{G_1 \lor G_2, Q_1 \land Q_2 \land (R \lor G_1 \lor G_2)\}}{Par-I}
\]

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’:
\begin{align*}
x: & \ [P, Q] \\
x: & \ [Q] \\
= & \ / \subseteq
\end{align*}

and wrap \textit{rely/guar} around any statement

\begin{align*}
\text{rely} & \ R \bullet c \\
\text{guar} & \ G \bullet c
\end{align*}
(Some) Laws of the new algebraic R/G

Nested-G:  
\[(\text{guar } g_1 \circ (\text{guar } g_2 \circ c)) = (\text{guar } g_1 \land g_2 \circ c)\]

Intro-G:  
\[c \sqsubseteq (\text{guar } g \circ c)\]

Trading-G-Q:  
\[(\text{guar } g \circ [g^* \land q]) = (\text{guar } g \circ [q])\]

Intro-multi-Par:  
\[\land_i[q_i] \sqsubseteq \|_i (\text{guar } gr \circ (\text{rely } gr \circ [q_i]))\]

(Asymmetric version below)
Reasoning about Interference (or lack thereof)
Refinement calculus style development

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

\[
[s' = s - C] = [s' \subseteq s \land s - s' \subseteq C \land s' \cap C = \{\}]
\]

\[\sqsubseteq \text{ by Intro-G}
\]

\[
\text{guar} \ s' \subseteq s \land s - s' \subseteq C \bullet [s' \subseteq s \land s - s' \subseteq C \land s' \cap C = \{\}]
\]

\[= \text{ by Trading-G-Q (} s - s' \subseteq C \text{ is transitive)}
\]

\[
\text{guar} \ s' \subseteq s \land s - s' \subseteq C \bullet [s' \land C = \{\}]
\]

\[\sqsubseteq \text{ by Intro-multi-Par}
\]

\[
\text{guar} \ s' \subseteq s \land s - s' \subseteq C \bullet
\[
(||_i \text{ guar} s' \subseteq s \bullet \text{ rely} s' \subseteq s \bullet [s' \cap c_i = \{\}])
\]

\[= \text{ Nested-G}
\]

\[
\text{guar} \ s - s' \subseteq C \land s' \subseteq s \bullet (||_i \text{ rely} s' \subseteq s \bullet [s' \cap c_i = \{\}])
\]
R/G observations

• asymmetric rely/guarantee conditions are important:
\[ [q_1 \land q_2] \subseteq (\text{guar } g_1 \bullet (\text{rely } g_2 \bullet [q_1])) \parallel (\text{guar } g_2 \bullet (\text{rely } g_1 \bullet [q_2])) \]

• nice bonus of new style: \textbf{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 \textbf{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)

\[
\begin{array}{c}
\{ P_1 \} s_1 \{ Q_1 \} \\
\{ P_2 \} s_2 \{ Q_2 \} \\
\{ P_1 * P_2 \} s_1 || s_2 \{ Q_1 * Q_2 \}
\end{array}
\]

Frame rule

\[
\begin{array}{c}
\{ P \} s \{ Q \} \\
\{ P * R \} s \{ Q * R \}
\end{array}
\]
‘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:

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

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

He then derives a post condition using \(\exists \alpha, \beta \cdot list(\alpha, i) \ast list(\beta, j)\)
Re-do the example with ‘Separation as an abstraction’

\[ r, s: \left[ r' = \text{rev}(s) \right] \]

\[ \text{rev}: X^* \rightarrow X^* \]

\[ \text{rev}(s) \triangleq \ldots \]

\( s \) and \( r \) are \textit{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’:

\[
\begin{align*}
    r & \gets []; \\
    \textbf{while } s \neq [] \textbf{ do} \\
    & \quad \text{STEP}_0 \\
    & \quad \{ \text{rev}(s') \sim r' = \text{rev}(s) \sim r \land \text{len } s' < \text{len } s \} \\
    \textbf{od}
\end{align*}
\]

\[
\begin{align*}
    \text{STEP}_0 \quad r, s: [s \neq \{ \}, \ r' = [\text{hd } s] \sim r \land s' = \text{tl } s]
\end{align*}
\]

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

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

Inductive definition of \(Srep \subseteq Heap\)

\[
\{\} \in Srep \\
\text{sr} \in Srep \land p \in \text{Ptr} \land p \notin \text{dom sr} \Rightarrow \\
\left(\{p \mapsto (v, \text{start(sr)})\} \cup \text{sr}\right) \in Srep
\]

\[
\text{start(\{\})} = \text{nil} \\
\text{start(\{p \mapsto (v, \text{start(sr)})\} \cup \text{sr})} = p
\]

Could develop a theory of \(Srep\) (e.g. Isabelle)
Reify \((X^* \times X^*)\) as \(\Sigma_I\)

\[
\Sigma_I = (Srep \times Srep)
\]

where

\[
\text{inv-}\Sigma_I((sr, rr)) \triangleq \text{sep}(sr, rr)
\]

\[
\text{sep} : Srep \times Srep \to \mathbb{B}
\]

\[
\text{sep}(sr, rr) \triangleq \text{dom} sr \cap \text{dom} rr = \{\}
\]
**STEP on $\Sigma_1$**

\[
\text{STEP}_1 \quad rr, sr: \left[ \begin{array}{l}
\text{let } p = \text{start}(sr) \text{ in } \\
\quad sr' = \{p\} \triangleleft sr \land \\
\quad rr' = rr \cup \{p \mapsto (sr(p), \text{start}(rr))\}
\end{array} \right]
\]

**Lemma 1** \(\text{STEP}_1\) preserves \(\text{inv-}\Sigma_1\)
Data reification (homomorphic rule)

Reasoning about Interference (or lack thereof)
Data reification proof — standard (VDM) rule

\[ retr_0 : \Sigma_1 \rightarrow \Sigma_0 \]
\[ retr_0((sr, rr)) \triangleq (\text{gather}(sr), \text{gather}(rr)) \]

\text{gather} : Srep \rightarrow X^*
\text{gather}{} = []
\text{gather}(\{p \mapsto (v, \text{start}(sr))\} \cup sr) = [v] \sim \text{gather}(sr)

**Lemma 2** (‘Adequacy’) There is a \( \Sigma_1 \) representation of any \( \Sigma_0 \)

**Lemma 3** (‘Commutativity’) \( \text{STEP}_1 \) models (under \( retr_0 \)) the abstract \( \text{STEP}_0 \)
Reification to a single *Heap*

\[ \Sigma_2 = (\text{Heap} \times \text{Ptr} \times \text{Ptr}) \]

where

\[ \text{inv-} \Sigma_2((hp, i, j)) \triangleq \exists sr, rr \in Srep \cdot sr \cup rr \subseteq hp \land i = \text{start}(sr) \land j = \text{start}(rr) \]

- 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 $\Sigma_2/\Sigma_1$

\[ retr_1 : \Sigma_2 \rightarrow \Sigma_1 \]

\[ retr_1((hp, i, j)) \triangleq (\text{trace}(hp, i) \triangleleft hp, \text{trace}(hp, j) \triangleleft hp) \]

\[ \text{trace} : \text{Heap} \times \text{Ptr} \rightarrow \text{Ptr-set} \]

\[ \text{trace}(hp, p) \triangleq \begin{cases} \text{if } p = \text{nil} & \text{then } \{\} \\ \text{else } \{p\} \cup \text{trace}(hp, hp(p)_2) \end{cases} \]

**Lemma 4** \(\text{trace}\) from \(\text{start}(sr)\) characterises \(sr\)

**Lemma 5** (‘Adequacy’) of \(\Sigma_2\) wrt \(\Sigma_1/retr_1\)

**Theorem 1** (‘Commutativity’) \(STEP_2\) models (under \(retr_1\)) \(STEP_1\)

Reasoning about Interference (or lack thereof)
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

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

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

$\text{ordered} : X^* \rightarrow \mathbb{B}$

$\text{ordered}(s) \triangleq \ldots$

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

$\text{permutes}(s, s') \triangleq \ldots$

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 RG-Sep [Vaf07]

A key (abstract) R/G law

\[ [q_1 \land q_2] \sqsubseteq (\text{guar} \; g_1 \bullet (\text{rely} \; g_2 \bullet [q_1])) \parallel (\text{guar} \; g_2 \bullet (\text{rely} \; g_1 \bullet [q_2])) \]

\ldots \text{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.

Richard Bornat and Hasan Amjad.
Inter-process buffers in separation logic with rely-guarantee.

Dines Bjørner and Cliff B. Jones, editors.
*Formal Specification and Software Development*.

Cristiano Calcagno, Peter O’Hearn, and Hongseok Yang.
Local action and abstract separation logic.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang.
Views: compositional reasoning for concurrent programs.
Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis.
Concurrent abstract predicates.

Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin.
Laws and semantics for rely-guarantee refinement.

C. A. R. Hoare.
An axiomatic basis for computer programming.

C.A.R. Hoare.
Towards a theory of parallel programming.

Balancing expressiveness in formal approaches to concurrency.
Reasoning about Interference (or lack thereof)

C. B. Jones.

*Software Development: A Rigorous Approach.*

Cliff B. Jones.
The early search for tractable ways of reasoning about programs.

C. B. Jones.
Splitting atoms safely.

Cliff B. Jones and Nisansala Yatapanage.
Reasoning about separation using abstraction and reification.

S. S. Owicki and D. Gries.
An axiomatic proof technique for parallel programs I.
P. W. O’Hearn.

Resources, concurrency and local reasoning.


S. Owicki.

*Axiomatic Proof Techniques for Parallel Programs.*


Matthew Parkinson.

The next 700 separation logics.


John Reynolds.

A logic for shared mutable data structures.


Viktor Vafeiadis.

*Modular fine-grained concurrency verification.*