Concurrency, Rely/Guarantee and Separation Logic

Cliff Jones

Newcastle University

Overture Workshop
2014-06-21
Expressive power must be a “good thing”
I beg to differ!

- (decidable) type systems
- $\{\text{pre}\} \ S \ \{\text{post}\}$
- data abstraction (which is a sort of leitmotiv)
  - benefit of making clear what can not be discussed!
- “power” can beget intractability
One message: start with concepts
cf. “to a man with a hammer, every problem is a nail”

- e.g. concept input/output relation of a program
- Hoare logic based on specifications
  - \( \{p\} S \{q\} \)
  - pre/post say less than implementation
  - but “extend the vocabulary” (using \(\land/\neg\))
  - easier to show “satisfaction of specification”
  - \ldots than equivalence of two programs

- post conditions are relations!
An important concept: separation

- separation = zero interference/visibility(!)
  - question: control reads (as well as writes)?
- in the case of normal (stack) variables . . .
  - just separating alphabets
  - cf. VDM \( rd/wr \) frames
  - new R/G presentation allows \( x: c \)
Separation Logic

- basic idea is simple
  - to prove things about $S_1 \parallel S_2$
  - would like to conjoin their pre/post conditions
- history
  - [Hoa75] tackles parallelism with “stack” variables
  - [Rey02] covers “Separation Logic” for “heap” variables
  - Concurrent Separation Logic — Peter O’Hearn [O’H07]
- “heap” variables harder than normal (stack) variables
  - SL designed for this case
  - could “bend” R/G with $s \leftarrow heap$ etc.
  - ... see below on using abstraction
- SL origin = bottom-up code analysis
  - heap variables
  - probably avoid SL for stack variables!
A key SL proof rule

“Separating conjunction” – $P \ast Q$ (only if $P$ and $Q$ are separate)

\[
\begin{array}{c}
\{ P_1 \} s_1 \{ Q_1 \} \\
\{ P_2 \} s_2 \{ Q_2 \} \\
\hline
\text{SL} \\
\{ P_1 \ast P_2 \} s_1 \parallel s_2 \{ Q_1 \ast Q_2 \}
\end{array}
\]

Example

\[
\begin{align*}
\{ x \mapsto \_ \ast y \mapsto \_ \} \\
[x] \leftarrow 3 \parallel [y] \leftarrow 4 \\
\{ x \mapsto 3 \ast y \mapsto 4 \}
\end{align*}
\]
SL’s “frame rule”

\[ \text{SL-frame} \quad \frac{\{P\} s \{Q\}}{\{P \ast R\} s \{Q \ast R\}} \]
Reynold’s example [Rey02] reconsidered

The following program (!) performs an in-place reversal of a list:

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

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

The post condition itself only has to require that some variable, say \(s\), is changed so that

\[ \exists \alpha, \beta \cdot \text{list}(\alpha, i) * \text{list}(\beta, j) \]
Re-do Reynold’s example with “Separation as an abstraction”!? 

\[ r, s: [r' = rev(s)] \]

\( s \) and \( r \) are \textit{assumed} to be distinct variables that they are separate is a (useful and) natural abstraction

It is straightforward to “posit & prove”:

\[
\begin{align*}
& r \leftarrow []; \\
& \textbf{while } s \neq [] \textbf{ do} \\
& \quad r, s: [r' = [\textsf{hd} s] \bowtie r \land s' = \textsf{tl} s] \\
& \quad \{ rev(s') \bowtie r' = rev(s) \bowtie r \} \\
& \textbf{od}
\end{align*}
\]
Step 2: reify $r, s$ onto John’s linked list

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

$$Rep :: h: Heap$$
$$i: \mathbb{N}$$
$$j: \mathbb{N}$$

$$coll: [\mathbb{N}] \times Heap \rightarrow X^*$$

$$retr : Rep \rightarrow (X^* \times X^*)$$

$$retr(mk-Rep(h, i, j)) \triangleq (coll(i, h), coll(j, h))$$
• separation is a (useful) abstraction
  • reification obligation is to preserve the abstraction
  • the invariant on \( Rep \) can use * (or a simple predicate)
• this differs from standard view of SL
  • what form would/will SL take in this view?
  • trying more (complicated) concurrent examples
• we are now working on concurrent DOM trees
SL extensions

- basic idea works well for “disjoint concurrency”
  - e.g. parallel merge sort
- many extensions — see [Par10]
  - “Next 700 Separation Logics”
- magic wand (fits algebraic view)
- fractional permissions Boyland
- (most papers) limit to “partial correctness”
- (concurrent) abstract predicates
An important concept: ownership

CSL

- Interesting examples involve “ownership”
- processes/threads can “exchange” ownership

\[ [10] \leftarrow x \parallel y \leftarrow [10] \]

- … given appropriate locking – reason about passing value
- could code ownership swapping in R/G!
- actually comes back to “what is ownership?”
- one attempt to demarcate scopes of SL and R/G
  a promising dichotomy – [O’H07]
    - use SL if proving (data) race freedom
    - use R/G for “racy” programs
Issue: interference

- how to express (constraints on) interference
- R/G background:
  - VDM
  - post conditions are relations (over $\Sigma$)
  - (total) correctness
  - “posit and prove” style of development
  - importance of data abstraction/reification
  - compositional development
  - didn’t handle concurrency

- Owicki/Gries
Rely/Guarantee “thinking”

- basic idea is simple
  - acknowledge “interference”
- rely conditions
  - record assumptions the designer can make
  - cf. pre conditions
- guarantee conditions
  - requirements on running code
  - cf. post conditions
- (see below: interplay with data abstraction)
NB: *rely*, *guar* (and *post*) conditions are relations
One form of R/G rule

\[
\begin{align*}
\{ P, R \lor G_2 \} & \underset{s_1}{\longrightarrow} \{ G_1, Q_1 \} \\
\{ P, R \lor G_1 \} & \underset{s_2}{\longrightarrow} \{ G_2, Q_2 \} \\
\{ P, R \} & \underset{\| - I}{\longrightarrow} \{ G_1 \lor G_2, Q_1 \land Q_2 \land (R \lor G_1 \lor G_2)^\ast \}
\end{align*}
\]
A more algebraic presentation of R/G

“pulling R/G apart”

- abandon 5-tuple: \( \{p, r\} S \{g, q\} \)
- present in “refinement calculus” style
  - specifications: \([p, q]\) (special case: \([q]\))
  - rely \(r \cdot c\)
  - guar \(r \cdot c\)
  - \(x: c\) rather than VDM rd/wr framing
(Some) Laws

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

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

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

Intro-multi-Par:
\[(\land i q_i) \sqsubseteq \|i (\text{guar } gr \cdot (\text{rely } gr \cdot [q_i]))\]
Example: Prime sieve

REM(2)

REM(3)

Concurrency, Rely/Guarantee and Separation Logic

Cliff Jones [20]
Refinement calculus style development

Set $s$ initially contains all natural numbers up to some $n$, $C$ is the set of all composite numbers

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

\[\sqsubseteq\] by Intro-G

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

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

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

\[=\] by set theory

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

\[\sqsubseteq\] by Intro-multi-Par

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

\[=\] Distribute-G

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

\[=\] Nested-G

\[\text{guar } s - s' \subseteq C \land s' \subseteq s \land (||_i \text{rely } s' \subseteq s \cdot [s' \cap c_i = \{\}])\]
Another look at Peter’s “dichotomy”
Using Simpson’s non-blocking “4-slot” algorithm

- Asynchronous Communication Mechanisms
  - one reader/writer
  - “lock free”
  - never read corrupt data (i.e. whist being written)
  - always read “most recently written”
- there are several algorithms, specifically . . .
  - there are several proofs of Simpson’s 4-slot algorithm
Hugo Simpson’s 4-slot idea

write(42) → x := read()

write(42) → x := read()

write(42) → x := read()

pair-r
pair-w
slot-w

✗
?
✓
Doubts about that neat dichotomy

- essence of 4-slot idea is race freedom on slots
- argue in terms of exchanging ownership (of slots)
- 4 (of many) papers on Simpson’s 4-slot algorithm
  - R/G Jones & Pierce
  - SL Bornat & Amjad
- Richard Bornat [BA10]
  - uses R/G as well . . . and serialisability!
  - SL *not* used for ownership
  - Wang & Wang do — but no freshness proof
- Jones & Pierce use R/G for race *freedom*
  - . . . at an abstract level
  - introduced “possible values” concept (below)
- have a new specification (using “possible values”)
Strategic messages

- start with the concepts/challenges
  - not with your pet notation
- abstraction, abstraction, abstraction
- identify issues/concepts in question
  - e.g. interference, separation
  - then select an apposite specific notation/approach
- reversing this order frequently . . .
  - bends an approach to do things that aren’t natural
  - encrypts real step
- “Ghost variables” a way to cheat on expressiveness
A (minor?) concept: possible values

- arose in Jones/Pierce work on 4-slot
- our first attempt (ABZ 2008) had an interesting flaw
  - \( \text{hold-}r = \text{fresh-w} \lor \text{hold-}r = \text{fresh-w} \)
  - but Write could actually change \text{fresh-w} many times
- actually need to say:
  - \text{READ} can set \text{hold-}r (only) to any value set in \text{fresh-w}
  - I prefer to avoid “ghost variables” (longer story)
  - \text{hold-}r \in \text{fresh-w}
- found a variety of other uses
- + link to Hayes’ work on
  “non-deterministic expression evaluation” (TCJ paper)
New project: “Taming Concurrency”
EPSRC (UK) funded

- “pull R/G and SL apart” — two papers submitted
  - CS-TR-1394 (short)
  - CS-TR-1395 (long)
- figure out what they express well
  - try for a *semantic* combination
  - . . . which might look like neither!
- (UK) project twinned with Australian (ARC) project
  - “Understanding concurrent programmes using rely-guarantee thinking”
  - led by Ian Hayes
References

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

C.A.R. Hoare.
Parallel programming: An axiomatic approach.

P. W. O’Hearn.
Resources, concurrency and local reasoning.

Matthew Parkinson.
The next 700 separation logics.

John Reynolds.
A logic for shared mutable data structures.