Rely/Guarantee-thinking and Separation Logic

Viktor Vafeiadis and Cliff Jones

MPI Kaiserslautern and Newcastle University

FM-2011 Tutorial
Limerick, Ireland
2011-06-21

Part II. Rely/Guarantee thinking

Part 1 (moving to Part 2)

- you've heard from Viktor about "separation"
  - he's shown it works on code that handles pointers (heap storage)
  - typically, this is low-level code
  - IMHO pointer handling is nearly always a reification of more abstract data structures
- switch now to "rigorous design" (by layers of abstraction)
- a dichotomy: avoiding races / reasoning about races
  - SL for avoiding races
  - R/G for reasoning about races
  - we'll see later, it's not that crisp a distinction

Complex systems

- (IMHO) the only tool to master complexity is abstraction
- complex systems are likely to exhibit concurrency
  - in detailed code
  - . . . and inherent in the application
- the essence of concurrency is *interference*

Successful abstractions

- key abstractions
  1. pre/post-conditions (sequential programs)
  2. abstract objects (crucial, pervasive)
  3. “framing” (cf. Separation Logic)
  4. recording interference (rely/guarantee thinking)
  5. “fiction of atomicity” + splitting atoms safely
- revisit known abstractions to look for lessons
- BUT when we abstract, we ignore some things
  - be aware what we ignore — and consider its impact
  - e.g. model of message system built on CSP/CCS
  - atomicity: atomic operations
  - . . . (even) assignment — cf. “relaxed memory” models
  - we’ll be careful about atomicity!

Abstraction: pre/post-conditions (as in VDM/Z/B/...)

design by: *sequential* "operation decomposition rules"

- Floyd/Hoare-like rules
  - even here, differences possible
    - e.g. weakening built in/separate
    - emphasise composition or decomposition
    - "total correctness": termination
  - coping with relational post-conditions (≠ [Hoare69])

    $post\text{-}OP_i : \Sigma \times \Sigma \to \mathbb{B}$ cf. SLAyer
  - "satisfiability"

    $$\forall \sigma \in \Sigma \cdot pre\text{-}OP_i(\sigma) \;\Rightarrow\; \exists \sigma' \in \Sigma \cdot post\text{-}OP_i(\sigma, \sigma')$$
  - this (slight) "expressive weakness" can be useful!
  - allowed to widen pre
  - allowed to narrow $post$ (respecting satisfiability)
- role of non-determinism: postpone design decisions
- *compositional* development
- ...

pre/post-conditions (continued)

- a rule for relational post-conditions:

$$\text{While-I}\ \frac{\begin{array}{l}\{P \wedge b\}\ S\ \{P \wedge W\} \\ P \wedge \neg\, b \wedge W^* \;\Rightarrow\; Q \\ P \;\Rightarrow\; \delta_l(b)\end{array}}{\{P\}\ \mathit{mk\text{-}While}(b, S)\ \{Q\}}$$

  termination “for free” with well-founded W
  (cf. “variant function”)

- don’t record unintended split then force equivalence proof
- ensure meaningful split (come back to this!)

decomposition vs composition

... this becomes more important with R/G

Contrast:

$$\boxed{\textit{While-I}}\ \frac{\{P \wedge b\}\ S\ \{P \wedge W\} \quad P \wedge \neg\, b \wedge W^* \ \Rightarrow\ Q}{\{P\}\ \textit{mk-While}(b, S)\ \{Q\}}$$

$$\boxed{\textit{While-I}}\ \frac{\{P \wedge b\}\ S\ \{P \wedge W\}}{\{P\}\ \textit{mk-While}(b, S)\ \{P \wedge \neg\, b \wedge W^*\}}$$

Interference

- *interference is the essence of concurrency*
- even with communication-based concurrency
  - obvious: as soon as shared variables can be simulated
  - trace assertions convenient for deadlock reasoning?
- “compositional” rules much harder to devise
  - than for sequential constructs
- rely/guarantee thinking faces up to interference
  - history below
  - remember lessons from sequential decomposition

R/G "thinking"

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

- assumptions $pre/rely$
- commitments $guar/post$
- typical R/G conditions:
  - x unchanged (but prefer to use "framing")
  - $\overleftarrow{s} \subseteq s$
  - more commonly

    $$flag \Rightarrow \cdots$$

  - use a $flag$ to signal between two processes (cf. locking)
- proof rules below

A simple example: *FINDP*

- a warming up example
- simple searching problem
- classic example from Sue Owicki's thesis
- concurrent version is non-trivial
- illustrates the importance of data representation

Overview: *FINDP* Algorithm

- a sequence of values: v
- a predicate: *pred*
  - e.g. first non-zero element
    concurrency would make more sense with complex *pred*
- task
  - find the lowest index in v satisfying *pred*
  - if none is found, result is one greater than the length of v
  - vs. sentinel/assumption
- use (simple) VDM notation
  - stop me if unclear

Specification

$FINDP$

rd $v\colon Value^*$

wr $r\colon \mathbb{N}_1$

pre $\forall i \in \textbf{inds}\, v \cdot \delta(pred(v(i)))$

rely $v = \overleftarrow{v} \wedge r = \overleftarrow{r}$

guar $\textbf{true}$

post $(r = \textbf{len}\, v + 1 \vee r \in \textbf{inds}\, v \wedge pred(v(r))) \wedge$
$\forall i \in \{1\colon r - 1\} \cdot \neg\, pred(v(i))$

Concurrent implementation

- partition indexes
  $p_1 \cup \cdots \cup p_n = \{1, \ldots, \mathbf{len}\, v\}$
  $FINDP = SEARCH(p_1) \mid\mid \cdots \mid\mid SEARCH(p_n)$
- concurrent processes search, one process per partition
- any partition would do
- but simplest with two processes: even/odd indexes

Naive Concurrency

- disjoint concurrency!
- each process checks indexes in its partition
- final result = minimum of even and odd result
- problem: this can perform worse than sequential
    - because one process may continue unnecessarily

Interfering Processes

- allow processes to share variables
- introduce *top*
- *top* records the lowest index found so far that satisfies *pred*
- each process tests/updates *top*

Concurrent Specification

$SEARCH(part: \mathbb{N}_1\text{-}\mathbf{set})$
rd $v: Value^*$
wr $r: \mathbb{N}_1$
pre $\forall i \in part \cdot \delta(pred(v(i)))$
rely $v = \overleftarrow{v} \wedge top \leq \overleftarrow{top}$
guar $top = \overleftarrow{top} \vee top < \overleftarrow{top} \wedge pred(v(top))$
post $\forall i \in part \cdot i \leq top \Rightarrow \neg pred(v(i))$

One possible R/G (decomposition) rule

remember, real message is “R/G thinking”

In the spirit of $\{P\}\ S\ \{Q\}$ we write $\{P, R\}\ S\ \{G, Q\}$

$$
\boxed{Par\text{-}I}\ \frac{\begin{array}{l}\{P, R_l\}\ s_l\ \{G_l, Q_l\}\\ \{P, R_r\}\ s_r\ \{G_r, Q_r\}\\ R \vee G_r \ \Rightarrow\ R_l\\ R \vee G_l \ \Rightarrow\ R_r\\ G_l \vee G_r \ \Rightarrow\ G\\ \overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \ \Rightarrow\ Q\end{array}}{\{P, R\}\ s_l \mid\mid s_r\ \{G, Q\}}
$$

scope for variation in rules *much* larger (than in Hoare logics)
here: for decomposition (∃ more compact presentations)

Using the proof rule (i)

So:

$$FINDP = SEARCH(odds) \mid\mid SEARCH(evens)$$

$$pre\text{-}FINDP \;\Rightarrow\; pre\text{-}SEARCH$$

is:

$$\forall i \in \mathbf{inds}\, v \cdot \delta(pred(v(i))) \;\Rightarrow\; \forall i \in part \cdot \delta(pred(v(i)))$$

Using the proof rule (ii)

$$rely\text{-}FINDP \vee guar\text{-}SEARCH \;\Rightarrow\; rely\text{-}SEARCH$$

is

$$top = \overleftharpoon{top} \vee top < \overleftharpoon{top} \;\Rightarrow\; top \leq \overleftharpoon{top}$$

Using the proof rule (iii)

$$post\text{-}SEARCH(odds) \wedge post\text{-}SEARCH(evens) \wedge$$
$$guar\text{-}SEARCH^* \;\Rightarrow$$
$$post\text{-}FINDP$$

is

$$(\forall i \in odds \cdot i \leq top \;\Rightarrow\; \neg\, pred(v(i)) \wedge$$
$$\forall i \in evens \cdot i \leq top \;\Rightarrow\; \neg\, pred(v(i)) \wedge$$
$$top = \overleftarrow{top} \vee top < \overleftarrow{top} \wedge pred(v(top))) \;\Rightarrow$$
$$(top = \mathbf{len}\, v + 1 \vee top \in \mathbf{inds}\, v \wedge pred(v(top))) \wedge$$
$$\forall i \in \{1 \colon top - 1\} \cdot \neg\, pred(v(i)))$$

Interesting link between R/G and data reification [Jon07]

- in *FINDP*
  - $top \leftarrow min(top, local)$ in two (or n) parallel processes
  - assuming don't want to "lock" top
  - find a representation that helps us to realise R/G conditions
  - (simple) represent as t as $min(et, ot)$
- (pattern repeated below — with less obvious reifications)

Sieve of Eratosthanes

This example:

- gives insight into the trade-offs between $pre/rely$ and $guar/post$
- more dramatic in concurrent $QREL$ —- cf. [CJ00]
- shows importance of choosing the representation ("reifying") to achieve (more complex) G

Interfaces need *thought* (even sequential)

"achieve real split"

$$post\text{-}PRIMES(\overleftarrow{s}, s) \triangleq \{1 \leq i \leq n \mid is\text{-}prime(i)\}$$

$(INIT; SIEVE)$ **satisfies** $PRIMES$

$$post\text{-}INIT(\overleftarrow{s}, s) \triangleq s = \{1, \ldots, n\}$$

$$pre\text{-}SIEVE(s) \triangleq post\text{-}INIT(\overleftarrow{s}, s)$$

$$post\text{-}SIEVE(\overleftarrow{s}, s) \triangleq post\text{-}Primes(\overleftarrow{s}, s)$$

versus . . .

$pre\text{-}SIEVE \triangleq$ **true**

$$post\text{-}SIEVE(\overleftarrow{s}, s) \triangleq s = \overleftarrow{s} - \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}$$

Sequential implementation of *SIEVE*

$PRIMES$:

$\quad \textbf{for } i \leftarrow \cdots$

$\quad\quad \textit{post-BODY}: s = \overleftarrow{s} - mults(i)$

$\quad\quad \textbf{for } j \leftarrow \cdots$

$\quad\quad\quad s \leftarrow s - \{i * j\}$

Parallel implementation of *SIEVE*

repeat message: "achieve real split"

$$post\text{-}PRIMES(\overleftarrow{s}, s) \triangleq \{1 \le i \le n \mid is\text{-}prime(i)\}$$

$$\Big\|_{i=2}^{\lfloor\sqrt{n}\rfloor} REM(i) \textbf{ satisfies } SIEVE$$

$REM(i)$
pre true
rely $s \subseteq \overleftarrow{s}$ — can't achieve post unless
guar $(\overleftarrow{s} - s) \subseteq mults(i)$ — upper bound $\wedge\ s \subseteq \overleftarrow{s}$ to match rely
post $s = \overleftarrow{s} - mults(i)$ — sequential exact set

Another proof rule ($n$ary version)

remember, real message is “R/G thinking”

$$\textit{Par-I}\quad \frac{\begin{array}{l}\{P, R \vee \bigvee_j G_j\}\ s_i\ \{G_i, Q_i\} \\ \bigvee_i G_i \ \Rightarrow\ G \\ \overleftarrow{P} \wedge \bigwedge_j Q_j \wedge (R \vee \bigvee_j G_j)^* \ \Rightarrow\ Q\end{array}}{\{P, R\}\ \|_i\ s_i\ \{G, Q\}}$$

Using the proof rule (i)

$$\big\|_{i=2}^{\lfloor\sqrt{n}\rfloor} REM(i) \textbf{ satisfies } SIEVE$$

$$rely\text{-}SIEVE \vee \bigvee_i guar\text{-}REM(i) \;\Rightarrow\; rely\text{-}SIEVE$$

is:

$$s \subseteq \overleftarrow{s} \;\Rightarrow\; s \subseteq \overleftarrow{s}$$

Using the proof rule (ii)

$$\overset{\lfloor\sqrt{n}\rfloor}{\underset{i=2}{\Vert}} REM(i) \textbf{ satisfies } SIEVE$$

$$\bigvee_i \textit{guar-REM}(i) \;\Rightarrow\; \textit{guar-SIEVE}$$

is:

$$\cdots \;\Rightarrow\; \textbf{true}$$

Using the proof rule (iii)

$\displaystyle\mathop{\|}_{i=2}^{\lfloor\sqrt{n}\rfloor} REM(i)$ **satisfies** $SIEVE$

$$\bigwedge_j post\text{-}REM(j) \wedge \bigvee_j guar\text{-}REM(j)^* \;\Rightarrow\; post\text{-}SIEVE$$

is:

$$\begin{array}{l}\forall i \in \{2 \colon \lfloor\sqrt{n}\rfloor\} \cdot s \cap mults(i) = \{\,\} \wedge \\ \quad (i \in \{s - \overleftarrow{s}\} \;\Rightarrow\; \exists j \in \{2 \colon \lfloor\sqrt{n}\rfloor\} \cdot i \in mults(j)) \;\Rightarrow \\ \qquad s = \overleftarrow{s} - \bigcup\{mults(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}\end{array}$$

(again) Interesting link between R/G and data reification

- achieving monotonic reduction in s
  - requires a suitable representation
  - a representation that helps realise R/G conditions $s \subseteq \overleftarrow{s}$

$Rem(i)$:
  for $j \leftarrow \cdots$
    $s \leftarrow s - \{i * j\}$

- don't want to "lock" s (it's big!)
- represent s by a vector of bits

$Rem(i)$:
  for $j \leftarrow \cdots$
    $s(i * j) \leftarrow$ **false**

- residual atomicity assumptions:
  - care if 8 bits packed into one byte (memory access/change)

Concurrent set

(source Francesco Zappa Nardelli)

- concurrent access to a (linked list) representation of a set
- see slides from [Nar10]
- although he uses R/G, my approach differs from Francesco's
- there are places where R/G (thinking) is too heavy!
- ... and it brings out another piece of work

Concurrent set: Specification

[Nar10, Slide 54]

Abstract and concrete state

Abstract specification of a *set* data type:

$$
\begin{array}{lll}
AbsContains(e): & < AbsResult := e \in Abs & > \\
AbsAdd(e): & < AbsResult := e \notin Abs\ ; & \\
 & \quad Abs := Abs \cup \{e\} & > \\
AbsRemove(e): & < AbsResult := e \in Abs\ ; & \\
 & \quad Abs := Abs \setminus \{e\} & >
\end{array}
$$

A module implements the abstract specification using local state and methods.

Sequential code: prove that the concrete methods are equivalent to their abstract counterpart.

Concurrent code: must also establish that the externally visible effect of each method takes place at some instant, atomically with respect to other threads.

This property is called *linearisability*:

each operation appears to take effect instantaneously.

Concurrent set: Implementation

[Nar10, Slide 56]

Pessimistic implementation of a set via a linked list

```
locate(e) :
  pred := Head ;
  pred.lock() ;
  curr := pred.next ;
  curr.lock() ;
  while (curr.val < e) {
    pred.unlock() ;
    pred := curr ;
    curr := curr.next ;
    curr.lock()
  } ;
  return pred, curr
```

```
add(e) :
  n1, n3 := locate(e) ;
  if n3.val ≠ e then
    n2 := new Node(e) ;
    n2.next := n3 ;
    n1.next := n2   [*A] ;
    Result := true
  else
    Result := false   [*B]
  endif ;
  n1.unlock() ;
  n3.unlock() ;
  return Result
```

```
remove(e) :
  n1, n2 := locate(e) ;
  if n2.val = e then
    n3 := n2.next   [*C] ;
    n1.next := n3 ;
    Result := true
  else
    Result := false   [*D]
  endif ;
  n1.unlock() ;
  n2.unlock() ;
  return Result
```

- *locate* uses *lock-coupling*: the lock on some node is not released until the next is locked. Returns the previous and current (that is the first node >= e) node, both locked.
- *add* inserts the new element while holding the locks of the previous and next node;
- *remove* updates the previous *next* pointer while holding the locks on previous and current

Concurrent set: R/G for locks

[Nar10, Slide 59]

Rely/Guarantee specification of locks

A mutex L is just a variable that holds the thread id (`tid`) of its owner, or `null`.

The semantics of lock and unlock can be formalised as:

$$\texttt{L.lock()} = < \texttt{L.owner = null} \rightarrow \texttt{L.owner := self} >$$

$$\texttt{L.unlock()} = < \texttt{L.owner := null} >$$

where $< \texttt{C} >$ denotes that `C` is executed atomically (and $< \texttt{B} \rightarrow \texttt{C} >$ is a CCR), and the distinguished variable `self` stands for the `tid` of the current thread.

$$\texttt{L.lock} \models (\texttt{L.owner} \neq \texttt{self}, \mathit{lockRely}, \mathit{lockGuar}, \texttt{L.owner} = \texttt{self})$$

$$\texttt{L.unlock()} \models (\texttt{L.owner} = \texttt{self}, \mathit{lockRely}, \mathit{lockGuar}, \texttt{L.owner} \neq \texttt{self})$$

where $\mathit{lockRely} = \mathrm{ID}(\texttt{L.owner} = \texttt{self})$

and $\mathit{lockGuar} = (\forall i \notin \{\texttt{self}, \texttt{null}\}.\ \mathrm{ID}(\texttt{L.owner} = i))$.

Concurrent set

an alternative approach

- use “fiction of atomicity”
- “splitting atoms safely”
- the approach to “refining atomicity” is (also) covered in [Jon96]
- . . . it fits with development by “layers of abstraction”

$\pi o \beta \lambda$

- $\pi o \beta \lambda$ is a concurrent object-based language
- synchronisation: only one method active per object (instance)
- effectively: atomic behaviour
- equivalence rules to introduce concurrency
  - “islands”
- no observable difference
- ... but relies on power of observers
- ... (thus) of observation language
- cf. “synchronisation points” / linearisability

R/G comments

- meaningful notion of compositionality
- scope for variation in rules *much* larger (than in Hoare logics)
  - e.g. “stability” (Coleman, Dodds *et al.*)
- odd variants

$$rely\text{-}OP_i \colon \Sigma \times \Sigma \to \mathbb{B}$$
$$guar\text{-}OP_i \colon \Sigma \times \Sigma \to \mathbb{B}$$
$$post\text{-}OP_i \colon \Sigma \to \mathbb{B}$$

- even (deprecated)
  but Stirling was looking for meta results

$$rely\text{-}OP_i \colon \Sigma \to \mathbb{B}$$
$$guar\text{-}OP_i \colon \Sigma \to \mathbb{B}$$
$$post\text{-}OP_i \colon \Sigma \to \mathbb{B}$$

R/G comments (continued)

- expressive weakness more marked!
  - there are things (transitive) relations can't express
  - R/G "thinking"
- "phasing" (as a way to increase expressiveness)
  - roughly: using PL constructs in specifications
  - (drastically) simplifies R/G
  - consider interference in two phases:
    x increases; x decreases
  - "4-slot" (in Part 4)
- proving soundness of R/G rules
  - joint paper with Joey Coleman: [CJ07]
  - language with nested parallel construct
  - . . . and fine granularity (+ STM in Coleman's thesis)
    - cf. Prensa Nieto's mechanically checked soundness proofs
  - my specific form of R also useful in our proof

Framing

There are several ways of achieving $x = \overleftarrow{x}$:

- locking
- local scope
- we can conjoin pre/post with independent frames
- what SL buys us is a concise notation for doing this
- (perhaps less for “stack” variables, but) for heap variables

Disjoint concurrency

Hoare

- all around us (e.g. paging)
- Hoare in 1971
  - check alphabet disjointness
  - use sequential proof rules
  - straight conjunction of pre/post conditions
- see “framing”
- cf. separation logic
  - usual origin: Reynolds
  - O’Hearn pointed to Hoare (at April 2009 event)

Interference

Ashcroft/Manna

- interference (i.e. shared alphabets)
- 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"

Interference

Owicki/Gries

- interference (i.e. shared alphabets)
- separate sequential reasoning
- *post facto*: final “Einmischungsfrei” PO
- non compositional
- arbitrary/fixed granularity assumption
- of course, disjoint frames remove risk of interference

Rely/Guarantee conditions

- compositional
- takes “interference” head on
- no fixed view of granularity (atomicity)
- saw later, R/G “thinking”
- easiest reference [Jon96]
- thesis now on-line [Jon81]
- see also [Jon07]

(more) R/G comments

- meaningful notion of compositionality
  - R/G for reasoning about "racey" programs
  - but also (see later) handling "abstract races"
- significant literature on extensions/variants (cf. www....)
  - $rely/guar$ both transitive and reflexive (zero/multiple steps)
  - other versions of R/G rules use "dynamic invariants" [CJ00]
  - "progress" conditions — Stølen
  - RGSep — see Viktor's Part 3
  - "Deny/Guarantee" Parkinson *et al.*
- look for synergy — not competition

References

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.

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.

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.
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.

Francesco Zappa Nardelli.
Proof methods for concurrent programs (slides part 3), 2010.
slides: `http://moscova.inria.fr/~zappa/teaching/mpri/2010/fzn-mpri-2010-3.pdf`.