

Rely-Guarantee

● ● ● assertions for safe concurrent programming

Marc Shapiro, INRIA Rocq & LIP6

Based on joint work with:

Viktor Vafeiadis, U. of Cambridge

Maurice Herlihy, Brown U.

Tony Hoare, Microsoft Research Cambridge



Who is this Marc Shapiro? Google search: lots of colourful characters...



Marc Shapiro, INRIA Rocq.

Distributed systems

- But I won't talk about that today

Concurrent programming

- Hard
- No / misguided compiler support
- Language community diverted
- No single right way

Needed: tools for reasoning about general concurrent programs

CC = concurrency control

I am not a language or theory person. This is a practitioner's take on a bit of theory.

● ● ● | Concurrency is hard...

Shared-memory parallelism:

- Other threads unpredictable
- Many, many possible interleavings
- Combinatorial explosion

Parallel computations don't compose

- Inherent!
- How about modularity?
- Opaque libraries, uncontrolled events

Undocumented: concurrency-control discipline

- “ X and Y are protected by lock L ”

ETH – 2007-05-14

R-G for safe concurrent programming

7

Concurrent programming is **inherently** hard

This is made even worse by bad support

- Concurrency control disciplines not documented
- ... and of course not verified

● ● ● | ...but concurrency is necessary

Responsiveness to:

- Hardware IT, page fault, GC
- New data, application events

Concurrency is everywhere:

- Clusters
- Multi-core CPUs
- Web server applications

Developers ill prepared

Concurrency is not just for OS and GC programmers any more.

Lots of (uneducated) application developers stumble into / are seduced by it.

Multi-core: Intel wants programmers to use them, but Intel is scared they will screw up



Roadmap

Introduction

Sequential Hoare logic

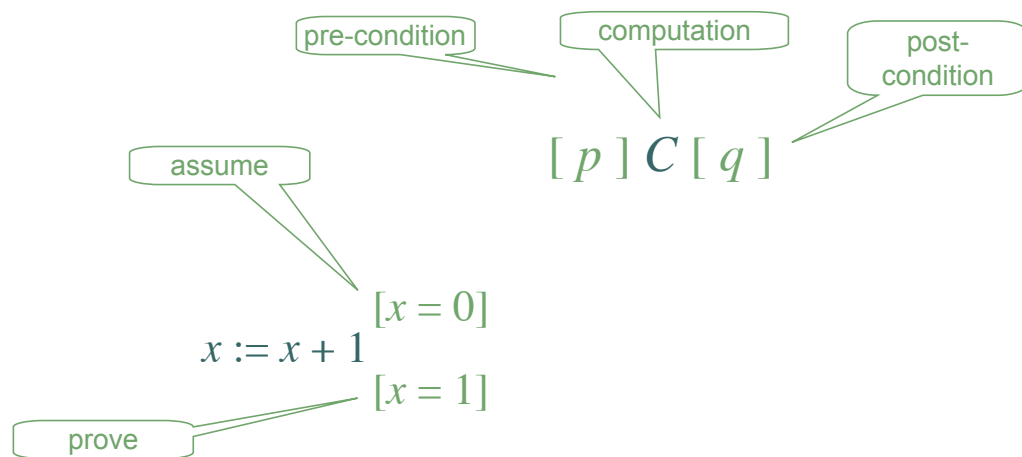
Rely-Guarantee logic

A specific example

Conclusion & call to arms

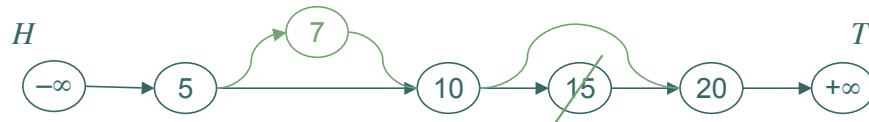
Sequential Hoare logic = boring background material

• • • | Sequential Hoare logic



Pre-condition ; computation $\Rightarrow$ post-condition

Sorted linked list



$contains_s(e), add_s(e), remove_s(e)$

$p = (e \neq \pm\infty) \wedge inv$

$q = effect \wedge inv$

$inv =$

Singly-linked linear sorted list

at entry & exit

R-G for safe concurrent programming

12

Animation: here is how the sorted linked list works

- The pseudo-code is left as an exercise for the reader
- Also left as an exercise is the proof that $p \implies q$ for every method

Sequential implementation of the Set abstraction. Inv, p, q

- At beginning and end of methods
- May be violated during execution

Invariant: (no need to look at the details)

- Structural: it has the properties expected of a sorted, non-repeating singly-linked list

● ● ● | Spec# contracts

```
interface Set {  
    invariant n == size () && ... ;  
    boolean add (elem! e)  
        assumes e.val <= MAX;  
        ensures inSet(e) && !retval = old (inSet (e));  
}
```

Compiler contains a theorem prover:

- Often proved at compile time!
- Otherwise, run-time + test case generation

How about the same for concurrent programs?

Eiffel: run-time only

Java/ESC



Sequential composition rule (no concurrency)

$$\frac{[p_1] C_1 [q_1] \qquad [p_2] C_2 [q_2]}{[p_1] C_1 ; C_2 [q_2]} \quad q_1 \Rightarrow p_2$$

To prove $C_1 ; C_2$, prove:

- C_1 and C_2 separately
- and the implication

Just a simple rule, we will need it later. Also define the inference notation.

Inference rule:

- If C_1 and C_2 are correct (pre ; comp implies post)
- And C_1 's post implies C_2 's pre
- Then it follows that $C_1;C_2$ is correct, under C_1 's pre and C_2 's post

• • • | Concurrency $\Rightarrow$ interference

$[x = 0]$
 $x := x + 1$
 ~~$[x = 1]$~~

x shared between threads

Pre-condition ; effects of computation

+ *non-interference* $\Rightarrow$ post-condition

Ex.: “No concurrent thread changes x ”

ETH – 2007-05-14

R-G for safe concurrent programming

15

Correctness: fundamental issue with concurrency is *interference*

Non-interference:

- Owicki-Gries: no concurrent thread violates the precondition (in this case, that $x == 0$)
- Simplified: no concurrent thread writes x

● ● ● | Rely-Guarantee reasoning



Arbitrary interleaving between atomic steps of C with atomic steps of the environment (=other threads)

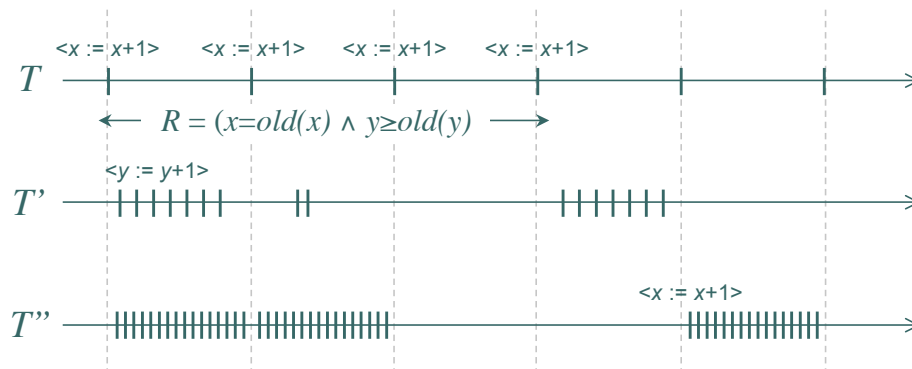
- R = what environment may do between steps of C
- G = what C may do between steps of environment

Prove properties $\implies$ assertions, invariants $\implies$ adapt Hoare to concurrent

The environment may interfere AT ANY POINT during the execution of C



Concurrency model: Interleaved atomic steps



ETH – 2007-05-14

R-G for safe concurrent programming

17

T = current thread

T' , T'' = environment

Non-deterministic scheduler interleaves steps arbitrarily

Between two atomic statements of T , the environment may interfere 0, 1, ..., any number of times

Because $0 \implies R$, G include identity

Note: Usually, I don't necessarily know how T decomposes into atomic statements (the compiler and the HW does that), only for reasoning purposes.

Occasionally however I do need to reason about low-level properties.



Examples

Envt. does not
modify x

$[x = 0, ID(x)]$
 $t := x+1; x := t$
 $[x \in \{0,1\}, x = 1]$

Envt. preserves
truth

Thread preserves
truth

$[true, Preserve(x \geq old(x))]$
 $t := x+1; x := t$
 $[Preserve(x \geq old(x)), x \geq old(x)+1]$

Thread modifies
no more than n

$[H \rightarrow^* n, Preserve(H \rightarrow^* n)]$
 $n.next := n'$
 $[Mod(n), H \rightarrow^* n']$



Basic RG axiom

$$[p] \ C(R, W) \ [q]$$

$$[p, ID(R, W)] \ C(R, W) \ [Mod(W), q]$$

If $\forall i \neq j, (R_i \cup W_i) \cap W_j = \emptyset$, then

- R trivially satisfied
- Sequential logic sufficient

Separation logic to the rescue?

- Logic for reasoning about the heap
- Global proofs: disjoint read/write-sets
- cf. Vafeiadis, Parkinson, Calcagno

ETH – 2007-05-14

R-G for safe concurrent programming

20

R = set of variables and locations that C depends on (C reads them)

W = set of vars/locs that C affects (C writes them)

- $p = p(R)$
- $C = C(R, W)$
- $q = q(R, W)$

If the write set of every thread is disjoint from the read-write sets of every other set, then (obviously) independent, and the sequential rule applies.

In many cases, disjoint memory obvious: scope rules, Unix processes.

Separation logic is a possible tool (still immature) to reason about the border cases.

R-G axioms justify separation

• • • | Atomic bracket axiom

$$\frac{[p] C(R, W) [q]}{[p, \emptyset] \langle C(R, W) \rangle [Mod(W), q]}$$

Environment
can't interfere
with C

Concurrent $\langle C \rangle$ equivalent to sequential C

$\langle C \rangle$ stands for "Atomic execution of C "

(Atomic: read "Isolated" in the ACID nomenclature)

• • • | Parallel composition

$$\begin{array}{c}
 [p_1, R_1] C_1 [G_1, q_1] \qquad [p_2, R_2] C_2 [G_2, q_2] \\
 G_1 \Rightarrow R_2 \\
 G_2 \Rightarrow R_1 \\
 \hline
 [p_1 \wedge p_2, R_1 \wedge R_2] C_1 \parallel C_2 [G_1 \vee G_2, q_1 \wedge q_2]
 \end{array}$$

Safety:

- Each thread safe in context of its *Rely*.
- *Rely* of any one thread implied by *Guarantee* of all others.

ETH – 2007-05-14

R-G for safe concurrent programming

23

I don't plan to explain the details of this slide.

- P,R,G,q of $C_1 \parallel C_2$ some combination of each thread's
- Each thread promises not to interfere with the others

Then, a safety proof of a parallel program reduces to

- Standard, sequential proof of each thread in isolation
- Pairise $G \Rightarrow R$ implications
- No combinatorial explosion
- Lots of book-keeping.

Proving parallel programs is EASY -- well, easier than you'd expect.

If you want to understand the technicalities, here it is.

Let's first assume there are only two threads in the system. We compose them in parallel.

The first one must not interfere with the second: $G_1 \Rightarrow R_2$. And vice-versa.

Now consider there are other threads around. They must not interfere with the composition. Hence the composed *Rely*: $R_1 \wedge R_2$.

In order to run, both pre-conditions better be true. Conversely, at the end, both threads have finished successfully, so both post-conditions are true.

What is the *Guar* of the composition? At some point in time, you don't know whether thread 1 or thread 2 has executed.

Therefore at most we can guarantee $G_1 \vee G_2$.

(But, since *G* is transitive, G_1 includes nil, so it's probably the case that $G_1 \vee G_2 \Rightarrow G_1 \wedge G_2$. So I could have just as well promised $G_1 \wedge G_2$.)

In summary:

- the guarantee condition of each thread
- Must implies the rely condition of the others.

No combinatorial explosion!



Proving safety of a concurrent program

- Sequential proof of each thread

Assuming: $[p, R]$

Prove: $[q, G]$

- Pairwise proof between threads

Prove: your G implies my R

$\forall t, t' \neq t: G_{t'} \Rightarrow R_t$

“If you don't shoot, I won't shoot”: circular reasoning?

Sound for safety [Abadi & Lamport 1995]

In plain English, and generalising to any number of threads...

Despite the apparent circular reasoning, Abadi and Lamport proved that this procedure is sound for safety proofs. Under certain conditions, it is also sound for liveness proofs.

Abadi & Lamport, “Conjoining Specifications”, TOPLAS 17(3), May 1995
<http://doi.acm.org/10.1145/203095.201069>

● ● ● | Lock axioms

Lock data type

$[L.owner = null, \emptyset] L.acquire$
 $[Mod(L), L.owner = self]$
 $[L.owner = self, \emptyset] L.release$
 $[Mod(L), L.owner = null]$

Convention: lock L protects object x

Ensure, for all accesses to x :

$L.R = ((L.owner = self) \Rightarrow ID(x))$

$L.G = ((L.owner \neq self) \Rightarrow ID(x))$

Just to make it a bit more concrete here are axioms of some common CC constructs.

If all threads correctly wrap accesses to x within regions where they own L , then $G \Rightarrow R$ and all is OK.

You knew that, but now we can formalise it (and hopefully some day have the machine find the bugs for you.)

If however the programmer forgot a lock, then the $G \Rightarrow R$ proof will fail.

At last, we have a language for documenting and checking the CC discipline!

Not very useful for simple locking, but there are a lot of more complex disciplines, e.g. if x is locked I don't lock y .

You will be able to document that, and to prove correctness.

Empty rely: acquire/release always execute atomically

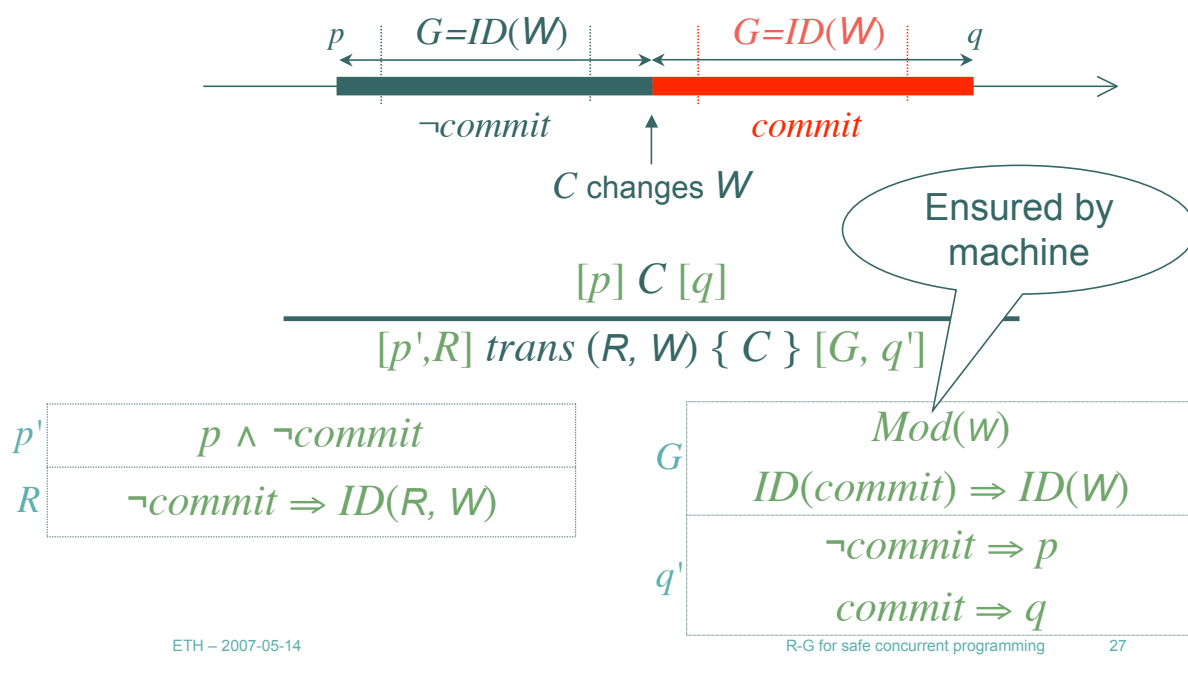
● ● ● | Concurrency control axioms

R-G is agnostic to CC mechanism

- Atomic Bracket
- Locks
- Transactions
- Messages
- Etc.

$R \Rightarrow G$ proof verifies discipline

R-G Transaction axiom



Here is the spec for transactions. Basically it says that the effect of a trans occur atomically at its commit point; if it doesn't commit it has no effect. Ensured at the physical level (memory reads and writes).

No need to understand the detail. The important message:

- If all access to RW is transactional, then $G \Rightarrow R$ for all pairs, and it's safe.
- Non-transactional access may co-exist with transactional access, as long as the non-trans accesses also guarantee the transactions' R.

Can't reuse the spec of $\langle C \rangle$ because in most SW transaction environments non-transactional code co-exists with transactions.

commit_i = sticky atomic boolean, lifetime = $\text{trans} \Rightarrow$ only one trans can commit at a time

- q: at end either abort with no effects (p true), or commit with normal execution of C (q true)
- I don't restrict which thread may set commit_i to true, because this is non-deterministic.
- Commit is a ghost variable, not a program variable. Doesn't exist outside of transactions.

Durability = Well-formed transactions:

- R, G: Once committed, stays committed

Effect of transaction:

- G: trans does not write outside of W
- G: trans writes only at the instant of commitment: W changes only over time intervals that include the commitment point

Isolation = transactions appear mutually instantaneous

- R: other threads do not overwrite R,W after p established and until this trans commits or terminates

The transaction scheduler ensures the G property is true. If all accesses are transactional, then the R property is true, which establishes isolation. But if transactions co-exist with non-transactional threads, then you must prove G! If CC discipline violated, no proof...

Fallacy: "With transactions the problem goes away"

- $\text{Trans}\{C1\}; \text{trans}\{C2\}$ same issue as $\langle C1 \rangle; \langle C2 \rangle$
- In addition: non-transactional access allowed, easy to forget CC discipline if not enforced
- Same interference problems as locks, etc. (but a lot nicer: composable, automated)

I could also show you the axioms for other constructs, e.g. messages, join patterns, etc.

● ● ● | Example: IntSet

Abstract sequential specification of integer set:

p	C	q
$true$	$contains(e)$	$ret = (e \in abs)$
$true$	$add(e)$	$abs = old(abs) \cup \{e\}$ $\wedge ret = (e \in old(abs))$
$true$	$remove(e)$	$abs = old(abs) \setminus \{e\}$ $\wedge ret = (e \notin old(abs))$

ETH – 2007-05-14

R-G for safe concurrent programming

28

We will now consider a simple example.

We will specify an abstract IntSet.

We will prove that a seq linked list implementation

- is safe
- does correctly implement the abstraction.

Then we will make the linked list more and more concurrent, and re-use the previous proof and the CC axioms to prove again safety and linearisability.

Of course this is just a simplified example. The logic can do much better.



Sequential implementation: IntSet as sorted linked list



$contains_s(e), add_s(e), remove_s(e)$

$p = (e \neq \pm\infty) \wedge inv$

$q = abs.q \wedge inv$

$inv =$

Linear, sorted, singly-linked list

$\wedge$

Abstraction Clause

ETH – 2007-05-14

R-G for safe concurrent programming

29

Animation: here is how the sorted linked list works

- The pseudo-code is left as an exercise for the reader
- Also left as an exercise is the proof that $p \implies q$ for every method

Sequential implementation of the Set abstraction. Inv, p, q

- At beginning and end of methods
- May be violated during execution

Invariant: (no need to look at the details)

- Structural: it has the properties expected of a sorted, non-repeating singly-linked list
- Linking invariant: LL implements IntSet abstraction
- Proving $p \implies q$ contains the proof that LL indeed implements the abstraction



Coarse-grain concurrent linked list

Coarse grain:

- atomic *contains, add, remove*

Next slide:

$contains_L = acquire; contains_s; release$

Exercise for the reader:

$contains_T = trans \{contains_s\}$

• • • | List + coarse-grain lock

$$[p] \text{ add}_s(e) \quad [q]$$

$$[p, ID(list)] \text{ add}_s(e) \quad [\dots, q]$$

$$\text{add}_L(e) = \text{acquire}(L); \text{add}_s(e); \text{release}(L)$$

$$[p', L.\text{owner} = \text{self} \Rightarrow ID(list)] \quad \text{add}_L(e) \quad [L.\text{owner} \neq \text{self} \Rightarrow ID(list), q']$$

lock axioms

ETH – 2007-05-14

R-G for safe concurrent programming

31

We can re-use the sequential implem in a concurrent setting. Putting together

- seq spec
- The basic R-G rule
- The lock axioms

If clients never call addS directly but only addL, then $G \Rightarrow R$, and this slide is a (hand-waving) proof.

Could do the same reasoning with transactions (there is nothing specific to locks in the logic.)

R/G documents the *concurrency control discipline* (usually left implicit). Concurrency correct: global property

Technicalities:

Suppose we have a proof of the sequential add in the absence of interference (1st line)

We define locking add to be the same as sequential add preceded by acquire and followed by release of lock L

Then we can prove the locking version (using the lock axioms).

- The pre/post condition proof follows from the sequential result and the rely
- Symmetry of rely/guar clauses, pairwise proof obviously correct.

Additional obligation: all threads must follow the lock discipline.



Fine-grain concurrent linked list implementations

Fine grain:

- *contains, add, remove* concurrently access to different parts of list
- Linearisable: Externally, behave *as if* atomic $\Rightarrow$ modular

Issues:

- `< dereference; test; assign >`
- Updates elsewhere must not interfere with my work: invariant true *all the time*

ETH – 2007-05-14

R-G for safe concurrent programming

32

"Externally, behave as if atomic" = i.e., w.r.t. the abstract specification

Linearisable: composes $\rightarrow$ can be hidden within a library

Issue 2

- Not identified in the literature
- Long conversations with Maurice to realise it



Fine-grain: lock coupling



$traverse_{lc}(e)$ = hand-over-hand search for e
return locked pair $(n1, n2)$

$remove_{lc}(e) = traverse_{lc}(e)$
 $n1.next := n2.next$
unlock $n1, n2$

$add_{lc}(e) = traverse_{lc}(e)$
 $n1.next := newnode(e, n2.next)$
unlock $n1, n2$

ETH – 2007-05-14

R-G for safe concurrent programming

34

Let's simulate the desired transactional effect with locks.

Why need

- lock *two adjacent nodes*
- Never allow a state where no node is locked



Lock-coupling proof

Invariant true *at all times*

$R \approx G$

$R = \text{Preserve}(\text{inv})$
 $\wedge \text{old}(L.\text{owner}) = \text{self} \Rightarrow \text{ID}(n.\text{next}, H \rightarrow^* n)$

$G = \text{Preserve}(\text{inv})$
 $\wedge \text{old}(L.\text{owner}) \neq \text{self} \Rightarrow \text{ID}(n.\text{next}, H \rightarrow^* n)$

p, q unchanged

Modified
while locked

$\text{inv} = H.\text{val} = -\infty \wedge T.\text{val} = +\infty \wedge \text{valid}(H)$
 $\wedge (H \rightarrow^* n \wedge n \neq T \wedge \neg \text{locked}(n)) \Rightarrow \text{valid}(n.\text{next})$
 $\wedge (n \rightarrow m \wedge \neg \text{locked}(n, m)) \Rightarrow n.\text{val} < m.\text{val}$
 $\wedge$ **Abstraction Clause**

ETH – 2007-05-14

R-G for safe concurrent programming

35

The assertions are basically the same as before, except:

- R, G: invariant true all the time!!! Because don't know when other threads may access list
- Structural invariant: a node that is locked need not maintain
- Abstraction invariant: another thread can kick in at any time $\Rightarrow$ must be always true

Proof:

- Per thread, $[p, R]$ C implies $[G, q]$
- Pairwise, $G \Rightarrow R$

● ● ● | Linearisability “proof”

$R = \text{Preserve}(inv) \wedge \dots$

$G = \text{Preserve}(inv) \wedge \dots$ // true at every step

$inv = \text{Structural Clause}$
 $\quad \wedge \text{Abstraction Clause}$

-- in addition must prove that e stays in the list once there

Note: I proved that add is linearisable

Similarly for the other methods

However I have not shown a linearisation point!

● ● ● | Also proved...

Optimistic

- Locate without locks; lock node pair; locate again
- If still reachable OK
- Rep statements true continuously

Lazy

- Optimistic
- 2-step remove: set “invisible” bit; fix pointer
- Linearisable

Lock-free

- Lazy; no locks
- Compare-and-swap for atomicity

(Manual proofs)

ETH – 2007-05-14

R-G for safe concurrent programming

37

In the PPOP article, we proved some complex algorithms.

This demonstrates the power of R-G.

But the *real* point is R-G is simple (enough) extension to Hoare, (relatively) easy to use, can prove real programs.



Fallacy

“If you just use X , concurrency is not an issue”

where $X \in$

{ transactions,
synchronised classes,
non-pre-emptive scheduling,
active objects,
events,
join patterns,
... }

Underlying truth: you need some concurrency control construct

None of these “well-structured” constructs can express ad-hoc concurrency control (such as hand-over-hand)

- None is perfect
- Depending on the job (and personal taste) some are more appropriate than others

Fundamental issue shared by all:

- Disallow bad interactions
- Allow good ones
- But which are which? Programmer must tell the system, and programmers make mistakes.

Locks, transactions, non-blocking, messages, events, patterns, joins: all useful

- No single “right” approach to CC

● ● ● | Related work

Rely-Guarantee theory:

- [Cliff Jones, 1981++](#)
- [Abadi, Lamport: Toplas 1995](#)

Highly concurrent algorithms:

- Recent PODC, PPOPP, etc.

Software transactions

- e.g. [Harris et al.: PPOPP05](#)

Separation logic

- e.g. [Parkinson et al.: POPL 07](#)
- R-G + separation: [Vafeiadis, Parkinson](#)

R-G pretty old technology

Theorists never put it into practice: no real algos, no axioms!

● ● ● | R-G reasoning

Proofs:

- Per-thread reasoning: sequential techniques
- Pairwise $G \Rightarrow R$: by symmetry + book-keeping

CC discipline made explicit, checkable

No combinatorial explosion

Errors $\Rightarrow$ lack of proof

(Difficulty: coming up with invariants)



Conclusion

Proof of linearisability

- Family of highly-concurrent algos
- Found bugs.

R-G practical tool for reasoning about concurrent code

- Modular
- (Relatively) simple and intuitive
- Explains *why* algorithm works
- Specifying remains difficult, but more modular

● ● ● | Call to Arms!

Multi-core $\Rightarrow$ concurrency *inescapable*

Surely language research can help

- Developers need tools, *you* can provide them
- R-G: small piece of theory begging to be used

Invariants

- Engineers: use them!
- Eiffel: prove them! + R-G!
- Microsoft: add R-G to Spec#!!!
- Open Source: add invariants, R-G, proofs into Java!!!

Plenty of publishable work

The end



Citations

- Flanagan, Freund, Qadeer, Seshia: “*Modular verification of multithreaded programs.*” Theoretical Computer Science 338 (1-3) pp. 153--183, June 2005. <http://dx.doi.org/10.1016/j.tcs.2004.12.006>
- Harris, Fraser: “*Language support for lightweight transactions.*” OOPSLA 2003, Anaheim, USA. <http://doi.acm.org/10.1145/949305.949340>
- Harris, Marlow, Peyton-Jones, Herlihy “*Composable memory transactions.*” PPOPP 2005, Chicago, USA, pp. 48--60 . <http://doi.acm.org/10.1145/1065944.1065952>
- Henzinger, Jhala, Majumdar, Qadeer: “*Thread-modular abstraction refinement.*” CAV 2003, LNCS 2725, pp. 262-274. http://mtc.epfl.ch/~tah/Publications/thread-modular_abstraction_refinement.html
- Herlihy, Wing: “*Linearizability: a Correctness Condition for Concurrent Objects.*” TOPLAS 12(3) July 1990. <http://doi.acm.org/10.1145/78969.78972>
- Parkinson, Bornat, O'Hearn: “*Modular Verification of a Non-Blocking Stack.*” POPL 2007. <http://www.cl.cam.ac.uk/~mjp41/>
- Vafeiadis, Herlihy, Hoare, Shapiro: “*Proving Correctness of Highly-Concurrent Linearisable Objects*”, PPOPP 2006. <http://doi.acm.org/10.1145/1122971.1122992>



Other common CC structures

Message passing:

$[true, \emptyset] \text{ send}(m) [\dots, m \in M]$

$[m \in M, \emptyset] \text{ receive}(m) [\dots, m \notin M]$

Linda: $\langle \text{put} \rangle, \langle \text{take} \rangle, \langle \text{read} \rangle$

Join



Assertions

Imperative concurrency code

- Explicit concurrency important for efficiency

Declarative specification

- Mutual check
- Useful to think two ways
- Like test cases (assert: summarises a class of test cases)

Note: ≠ Eiffel: they derive scheduling from spec

Optimistic list

Only lock holder may set next ptr

$$\begin{aligned} ListInv &\stackrel{\text{def}}{=} \text{Node}(\text{Head}) \wedge \text{Head.val} = -\infty \\ &\wedge \forall_{\text{Node } n}. n.\text{val} < +\infty \Rightarrow \text{Node}(n.\text{next}) \\ &\wedge \forall_{\text{Node } n\ m}. n \rightarrow m \Rightarrow n.\text{val} < m.\text{val} \\ &\wedge Abs = \{n.\text{val} \mid \text{Head} \rightarrow^* n \wedge n.\text{val} \neq \pm\infty\} \end{aligned}$$

Rep invariant preserved by every step

Reachability preserved by every step

$$\begin{aligned} R &\stackrel{\text{def}}{=} \forall_{\text{Node } n\ m}. \text{Preserve}(ListInv) \\ &\wedge n.LockRely \wedge ID(n.\text{val}) \\ &\wedge \overline{n.owner} = \text{self} \Rightarrow ID(n.\text{next}) \wedge \text{Preserve}(m \rightarrow^* n) \\ G &\stackrel{\text{def}}{=} \forall_{\text{Node } n\ m}. \text{Preserve}(ListInv) \\ &\wedge n.LockGuar \wedge ID(n.\text{val}) \\ &\wedge \overline{n.owner} \neq \text{self} \Rightarrow ID(n.\text{next}) \wedge \text{Preserve}(m \rightarrow^* n) \end{aligned}$$

ETH – 2007-05-14

R-G for safe concurrent programming

47

ListInv = Rep in previous slides

Rely:

- Rep invariant preserved at every atomic step!
- Only lock holder may modify a next pointer
- If there is a path from m to n, preserve it -- whether locked or not

Note locks do not appear in Rep Invariant -- may not be violated even temporarily

Lazy list

```

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

Check if removed after
traverse

ETH – 2007-05-14

```

contains(e) :
  curr := Head ;
  while (curr.val < e)
    curr := curr.next ;
  if curr.marked then
    return false
  else
    return curr.val = e

validate(pred, curr) :
  if ¬pred.marked
    and ¬curr.marked
    and pred.next = curr then
    return true
  else
    return false
  
```

Traverse: no locking

If so, give up and start
over

Remove: 2 separate
atomic statements

```

add(e) :
  same as lock-coupl

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

R-G for safe concurrent programming

48

Traverse without locking

- Locate
- Contains

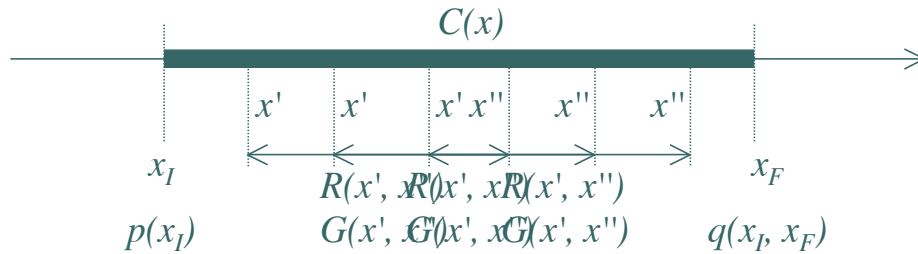
● ● ● | Atomic sticky boolean

Data type

- Initially false
- Only operation: *set*
- Atomic: no two concurrent *set* ops
- Sticky: No reset
- Limited lifetime



R and G are transitive



R holds over arbitrary successive atomic statements (including zero): $R = R^*$

Similarly for G

• • • | R-G Sequential composition

$$\begin{array}{c}
 [p_1] C_1 [q_1] \qquad \qquad \qquad [p_2] C_2 [q_2] \\
 \hline
 q_1 ; R \Rightarrow p_2 \\
 \hline
 [p_1, R] \langle C_1 \rangle ; \langle C_2 \rangle [\dots, q_2]
 \end{array}$$

$$\begin{array}{c}
 [p_1, R] C_1 [G, q_1] \qquad \qquad \qquad [p_2, R] C_2 [G, q_2] \\
 p_1 ; R \Rightarrow p_1 \qquad \qquad q_1 ; R \Rightarrow p_2 \qquad \qquad q_2 ; R \Rightarrow q_2 \\
 \hline
 [p_1, R] C_1 ; C_2 [G, q_2]
 \end{array}$$

ETH – 2007-05-14

R-G for safe concurrent programming

51

Remember the good old Hoare seq composition rule... Now let's consider what happens when the environment interferes.

If each computation is atomic, there can still be interference at the semi-colon.

However if R respects $\$p_2\$$ then the composition remains correct.

Now, let's suppose C1 and C2 are internally composed of a sequence of atomic statements.

The same reasoning applies recursively. To simplify assume all interference is limited by the same R and promises the same G. Then the second formula applies.

Note that there may be a delay, i.e. an implicit "skip" statement at the beginning and the end of the composed statement.

● ● ● | ACID transaction

Transaction:

- Arbitrary instruction sequence.
- Eventually terminates.
- Attomic: all (commit) or nothing (abort).
- Consistent: $p; C \Rightarrow q$
- Isolated: no intermediate effects.
(Called “atomic” herein).
- Durable: once committed, stays committed

Note: "atomic" means something slightly different for DB people.

What we have been calling atomic, they call isolation.

We'll stick to atomic.



Fallacy

Fallacy: “With transactions the concurrency problem goes away”

- $\text{trans}\{C1\}; \text{trans}\{C2\}$ same issue as $\langle C1 \rangle; \langle C2 \rangle$
- In addition: non-transactional access allowed (CC discipline not enforced).
- Just like locks, etc.

Improvements: automated, composable



Linearisable data type

“All its effects appear to take place atomically at some instant between the method's invocation and return.” [HW 1990]

Sequential effect = (p, q) of *linearisation point*
(Execution order preserved for non-overlapping method calls.)

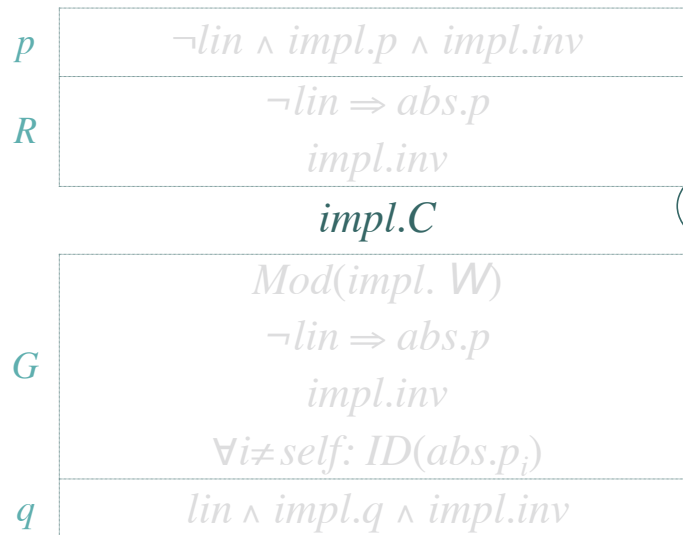
$$\frac{[impl.p] \text{ impl.C } [impl.q] \quad impl \text{ linearisable}}{[abs.p] <abs.C> [abs.q]}$$

If the implementation of a data type is linearisable

- i.e. all methods appear to have instantaneous effects (w.r.t the abstract spec)

Then you can replace the specification of the implementation with the spec of the abstraction

Linearisable



ETH – 2007-05-14

R-G for safe concurrent programming

55

Similar to transactions, except not RW but pqRG

0) Trivially the frame condition is guaranteed.

1) *p*: initially concrete pre-condition and invariant true (otherwise won't work)

Similarly for post-condition at end.

Post-condition occurs at *lin* point, lifted to the end: since effect is atomic, anything after *lin* point is irrelevant (???)

2) *G*: No effects before *lin* point: *ID(p)*, and therefore *p*.

R: The environment mustn't mess up *p*.

3) *G*: another thread may access this data structure at any time during execution. If we don't guarantee the (concrete) invariant, the other threads are in trouble.

R: Conversely we can rely on the invariant. What a relief!

4) And by the way, we better promise not to mess up with other thread's pre or post conditions

Transactions vs. Linearisability :

- Non-interference: RW sets protected by trans mechanism / abstract properties ensured by programmer
- *G*: ensured by transaction scheduler / to be ensured by class programmer