Proof methods for concurrent programs

3. Owicki-Gries, Rely-Guarantee

Francesco Zappa Nardelli

INRIA Paris-Rocquencourt, MOSCOVA project-team

francesco.zappa_nardelli@inria.fr

http://moscova.inria.fr/~zappa/teaching/mpri/2010/

Warm-up: concurrent separation logic

- Threads must be interference free:

$$\frac{\{ P_1 \}\ \mathtt{C_1}\ \{ Q_1 \} \qquad \{ P_2 \}\ \mathtt{C_2}\ \{ Q_2 \}}{\{ P_1 * P_2 \}\ \mathtt{C_1} \parallel \mathtt{C_2}\ \{ Q_1 * Q_2 \}}$$

- A resource invariant RI_r is associated to each resource. Acquiring the resource *imports* the resource invariant in the local scope.

$$\frac{\{ (P * RI_r) \wedge \mathtt{S} \}\ \mathtt{C}\ \{ Q * RI_r \}}{\{ P \}\ \mathtt{with\ r\ when\ S\ do\ C}\ \{ Q \}}$$

- A careful design of the resource invariants is the key to enable threads to share datas.

Warm-up: simple exercices

Prove the following triples, or explain why they cannot be proved.

```
{ empty }
  x := cons(3);
  z := cons(3);
  [x] := 4 || [z] := 5;
{ x ↦ 4 * z ↦ 5 }
```

```
{ empty }
  x := 4 || x := 5;
{ empty }
```

```
{ y = x+1 }
  x := 4 || y := y+1;
{ y = x+2 }
```

Warm-up: simple exercices

Prove the following triples, or explain why they cannot be proved.

{ empty }
```
  x := cons(3);
  z := cons(3);
  [x] := 4 || [z] := 5;
```
$\{ x \mapsto 4 * z \mapsto 5 \}$

{ empty }
```
  x := 4 || x := 5;
```
{ empty }

{ y = x+1 }
```
  x := 4 || y := y+1;
```
{ y = x+2 }

$$\frac{\{ P_1 \}\ c_1\ \{ Q_1 \} \qquad \{ P_2 \}\ c_2\ \{ Q_2 \}}{\{ P_1 * P_2 \}\ c_1 \parallel c_2\ \{ Q_1 * Q_2 \}}$$

if $\mathrm{modifies}(c_1) \cap \mathrm{fv}(P_2) = \mathrm{modifies}(c_2) \cap \mathrm{fv}(P_1) = \varnothing$, and
$\mathrm{modifies}(c_1) \cap \mathrm{fv}(c_2) = \mathrm{modifies}(c_2) \cap \mathrm{fv}(c_1) = \varnothing$.

Warm-up: a simple concurrent memory allocator

We implement a simple memory allocator using a shared list of memory cells.

```
init() { f := nil }

resource mm (f)
```

The resource invariant RI_{mm} is list `f`, where list `f` $\equiv$ `f = nil` $\vee \exists$ `j`. `f` $\longmapsto$ `j` $*$ list `j`.

```
alloc(x) {
  with mm when(true) do {
    if f=nil then x := cons(nil);
    else { x := f; f := [x]; }}
}
```

`alloc` attempts to return a cell from the list. If the list is empty, it invokes `cons`.

```
dealloc(y) {
  with mm when(true) do {
   [y] := f; f := y; }
}
```

`dealloc` puts the cell back into the list.

Warm-up: a simple concurrent memory allocator

We implement a simple memory allocator using a shared list of memory cells.

```
init() { f := nil }

resource mm (f)
```

The resource invariant RI_{mm} is list `f`, where list `f` ≡ $f = nil \lor (\exists j.\ f \longmapsto j * \text{list } j)$.

```
alloc(x) {
  with mm when(true) do {
    if f=nil then x := cons(nil);
    else { x := f; f := [x]; }}
}

dealloc(y) {
  with mm when(true) do {
   [y] := f; f := y; }
}
```

Prove that:

- { empty } `alloc(x)` { $x \mapsto \_$ }
- { $y \mapsto \_$ } `dealloc(y)` { empty }

Warm-up: clients of the memory allocator

We know that:

- { empty } `alloc(x)` { $x \mapsto \_$ }
- { $y \mapsto \_$ } `dealloc(y)` { empty }

Prove that

{ empty }

```
alloc(x); [x] := 13; dealloc(x) || alloc(y); [y] := 27; dealloc(y)
```

{ empty }

Observe that this proof does not mention the resource invariant (or resource).

Disclaimer

In 1965, Dijkstra introduces the `parbegin` statement.

In 1969, Hoare proposes a formal system of axioms and inference rules for the verification of imperative sequential programs.

In 1976, Susan Owicki and David Gries extend Hoare’s system for the verification of parallel programs with shared variables.

In 1981, Cliff Jones introduces the rely-guarantee method, a compositional version of the Owicki-Gries system.

Around year 2000, Peter O'Hearn introduces concurrent separation logic.

Today we travel back in time:

we ignore pointers and memory allocation (and separation logic).

Owicki-Gries reasoning

Can we reason about interfering processes?

Separation logic is about *absence of interference*:

- specifications are simple because they describe only the state that the program accesses;
- difficult to deal with interference.

How to reason about interferfering processes? For instance, consider:

$$P_1 :: x := x + 1 \;\|\; P_2 :: x := x + 2 \,.$$

Intuitively, if *assignement is atomic* and x initially holds 0, this program ends with x = 3.

A rule for parallel composition

Can we derive a specification of the parallel composition of two commands from the specifications of each command?

A first attempt:

$$\frac{\{ P_1 \}\ \mathtt{C_1}\ \{ Q_1 \} \qquad \{ P_2 \}\ \mathtt{C_2}\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ \mathtt{C_1} \parallel \mathtt{C_2}\ \{ Q_1 \wedge Q_2 \}}$$

Intuition: if we satisfy the preconditions of $\mathtt{C_1}$ and $\mathtt{C_2}$, their postconditions will be satisfied too.

Unsoundness of the first attempt

$$\frac{\{ P_1 \}\ \mathtt{c}_1\ \{ Q_1 \} \qquad \{ P_2 \}\ \mathtt{c}_2\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ \mathtt{c}_1 \parallel \mathtt{c}_2\ \{ Q_1 \wedge Q_2 \}}$$

Unsoundness of the first attempt

$$\frac{\{\, P_1 \,\}\ \mathtt{C_1}\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ \mathtt{C_2}\ \{\, Q_2 \,\}}{\{\, P_1 \wedge P_2 \,\}\ \mathtt{C_1} \parallel \mathtt{C_2}\ \{\, Q_1 \wedge Q_2 \,\}}$$

Consider:

$\{\, \mathtt{y = 1} \,\}\ \mathtt{x := 0}\ \{\, \mathtt{y = 1} \,\}$

$\{\, \mathrm{T} \,\}\ \mathtt{y := 2}\ \{\, \mathrm{T} \,\}$

It is not true that

$\{\, \mathtt{y = 1} \wedge \mathrm{T} \,\}\ \mathtt{x := 0} \parallel \mathtt{y := 2}\ \{\, \mathtt{y = 1} \wedge \mathrm{T} \,\}$

Second attempt

$$\frac{\{\, P_1 \,\}\ C_1\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ C_2\ \{\, Q_2 \,\}}{\{\, P_1 \wedge P_2 \,\}\ C_1 \parallel C_2\ \{\, Q_1 \wedge Q_2 \,\}}$$

if $\mathrm{mod}(C_1) \cap \mathrm{free}(P_2,Q_2) = \varnothing$ and $\mathrm{mod}(C_2) \cap \mathrm{free}(P_1,Q_1) = \varnothing$.

Second attempt

$$\frac{\{\, P_1 \,\}\ \mathtt{C}_1\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ \mathtt{C}_2\ \{\, Q_2 \,\}}{\{\, P_1 \wedge P_2 \,\}\ \mathtt{C}_1 \parallel \mathtt{C}_2\ \{\, Q_1 \wedge Q_2 \,\}}$$

if $\mathrm{mod}(C_1) \cap \mathrm{free}(P_2,Q_2) = \varnothing$ and $\mathrm{mod}(\mathtt{C}_2) \cap \mathrm{free}(P_1,Q_1) = \varnothing$.

Still unsound. Consider:

{ `z = 0` } `x := z; y := x` { `y = 0` }

{ T } `x := 2` { T }

It does not hold that

{ `z = 0` ∧ T } `x := z; y := x` ‖ `x := 2` { `y = 0` ∧ T }

$$\overset{\{\, \mathtt{x = 0} \,\}}{\swarrow}$$

Diagnose: `x := 2` interferes with the proof of { `z = 0` } `x := z; y := x` { `y = 0` }.

Interference

The intuition suggests that the command `x := 2` interferes with { `x = 0` }.

Question: does the command `bal := bal + 1` interfere with { `bal > 1000` } ?

What does *interfere* really means?

1) mod(`C`) $\cap$ free(P) $= \emptyset$

this will give a sound logic, but it is over-restrictive…

2) $\vdash$ { P } `C` { P }

Intuition: the assertion P is not invalidated by the execution of `C`.

Alternative intuition: if a thread went to a state where P holds, it is not a problem if another thread executes `C`.

Example: { `bal > 1000` } `bal := bal + 1` { `bal > 1000` }.

Interference freedom

Let a proof outline Δ of $\{P\}$ `c` $\{Q\}$ be given. A *critical formula* of Δ is either Q or a formula Q' appearing immediately before some statement in Δ.

Let proof outlines Δ_1 of $\{P_1\}$ c_1 $\{Q_1\}$ and Δ_2 of $\{P_2\}$ c_2 $\{Q_2\}$ be given.

Definition: Δ_2 does not interfere with Δ_1, if for every critical formula P of Δ_1 and triple $\{P_2\}$ c_2 $\{Q_2\}$ appearing in Δ_2, it holds $\{P \wedge P_2\}$ c_2 $\{P\}$.

Remark: need consider only those c_2 *that are assignments.*

We say that Δ_1 and Δ_2 are interference free, if Δ_1 and Δ_2 do not interfere with each other.

Owicki-Gries proof rule

$$\frac{\{\ P_1\ \}\ \mathtt{c}_1\ \{\ Q_1\ \} \qquad \{\ P_2\ \}\ \mathtt{c}_2\ \{\ Q_2\ \}}{\{\ P_1 \wedge P_2\ \}\ \ \mathtt{c}_1 \parallel \mathtt{c}_2\ \ \{\ Q_1 \wedge Q_2\ \}}$$

if the proofs of $\{\ P_1\ \}\ \mathtt{c}_1\ \{\ Q_1\ \}$ and $\{\ P_2\ \}\ \mathtt{c}_2\ \{\ Q_2\ \}$ are *interference free.*

Owicki & Gries method (on a simple example)

$$\{ x = 0 \}\ x := x + 1 \parallel x := x + 2\ \{ x = 3 \}$$

- Annotate an assertion to every control point, and show that the processes are locally correct (as if they were run in isolation) w.r.t. these assertions.

  Let $P_1 = (x = 0 \lor x = 2)$, $Q_1 = (x = 1 \lor x = 3)$,

  Let $P_2 = (x = 0 \lor x = 1)$, $Q_2 = (x = 2 \lor x = 3)$. We must verify that

$$\{ P_1 \}\ x := x + 1\ \{ Q_1 \}$$

$$\{ P_2 \}\ x := x + 2\ \{ Q_2 \}$$

- Prove interference freedom: every assertion used in the local verification is shown not invalidated by the execution of the other process.

$$\{ P_1 \land P_2 \}\ x := x + 2\ \{ P_1 \} \qquad \{ P_2 \land P_1 \}\ x := x + 1\ \{ P_2 \}$$

$$\{ Q_1 \land P_2 \}\ x := x + 2\ \{ Q_1 \} \qquad \{ Q_2 \land P_1 \}\ x := x + 1\ \{ Q_2 \}$$

Example

P_1:: `bal := bal + dep`

P_2:: `if bal > 1000 then credit := 1 else credit := 0`

Proof goal:

$$\{\, \mathtt{bal} = \mathtt{B} \land \mathtt{dep} > 0 \,\}$$
$$P_1 \parallel P_2$$
$$\{\, \mathtt{bal} = \mathtt{B} + \mathtt{dep} \land \mathtt{dep} > 0 \land (\mathtt{credit} = 1 \Rightarrow \mathtt{bal} > 1000) \}$$

Proof outline Δ_1

Proof outline Δ_1 of

$$\{\texttt{bal} = \texttt{B} \wedge \texttt{dep} > 0\}\ \texttt{bal := bal + dep}\ \{\texttt{bal} = \texttt{B} + \texttt{dep} \wedge \texttt{dep} > 0\}$$

$\{\texttt{bal} = \texttt{B} \wedge \texttt{dep} > 0\}$

$\{\texttt{bal} + \texttt{dep} = \texttt{B} + \texttt{dep} \wedge \texttt{dep} > 0\}$

```
bal := bal + dep
```

$\{\texttt{bal} = \texttt{B} + \texttt{dep} \wedge \texttt{dep} > 0\}$

Critical formulas:

$P_{1,1}$: $\texttt{bal} + \texttt{dep} = \texttt{B} + \texttt{dep} \wedge \texttt{dep} > 0$

$P_{1,2}$: $\texttt{bal} = \texttt{B} + \texttt{dep} \wedge \texttt{dep} > 0$

Proof outline Δ_2

{ T } `if bal > 1000 then credit := 1 else credit := 0` { `credit=1` ⇒ `bal>1000` }

```
{ T }
if bal > 1000 then
   { T ∧ bal > 1000 }
   { 1 = 1 ⇒ bal > 1000 }
   credit := 1
   { credit = 1 ⇒ bal > 1000 }
else
   { T ∧ bal <= 1000 }
   { 0 = 1 ⇒ bal > 1000 }
   credit := 0
   { credit = 1 ⇒ bal > 1000 }
{ credit = 1 ⇒ bal > 1000 }
```

Critical formulas:

$P_{2,1}$: `1 = 1 ⇒ bal > 1000`

$P_{2,2}$: `0 = 1 ⇒ bal > 1000`

$P_{2,3}$: { `credit = 1 ⇒ bal > 1000` }

Proving interference freedom

Need to prove, for each i {1,2} and j {1,2,3}:

1. $\{ P_{1,i} \wedge P_{2,1} \}$ `credit := 1` $\{ P_{1,i} \}$
2. $\{ P_{1,i} \wedge P_{2,2} \}$ `credit := 0` $\{ P_{1,i} \}$
3. $\{ P_{2,j} \wedge P_{1,1} \}$ `bal := bal + dep` $\{ P_{2,j} \}$

7 proof goals!

Triples of type 1 and 2 hold trivially since no $P_{1,i}$ mentions `credit`.

The type 3 goal $\{ P_{2,2} \wedge P_{1,1} \}$ `bal := bal + dep` $\{ P_{2,j} \}$ is trivially valid

Remain to prove:

- {(1=1 ⇒ bal > 1000) ∧ bal + dep = B + dep ∧ dep > 0} bal := bal + dep {1=1 => bal > 1000 }
- {(credit = 1 ⇒ bal > 1000) ∧ bal + dep = B + dep ∧ dep > 0} bal := bal + dep {credit = 1 ⇒ bal > 1000}

Remarks

- If P_1 had been withdrawal

```
bal := bal - wdr
```

  where `wdr > 0`, then the last step of the proof would not have gone through.

- A program which never grants credit would satisfy the specification.

- Same for a postcondition of the form

$$(\texttt{credit} = 1 \Rightarrow \texttt{bal} > 1000) \wedge (\texttt{credit} = 0 \Rightarrow \texttt{bal} \leq 1000)$$

  but this would lead to a violation of interference freedom. Why?

Exercise

Prove that:

$$\{x = 0\}\ \texttt{x := x + 1} \parallel \texttt{x := x + 1}\ \{x = 2\}$$

Exercise

Prove that:

$$\{x = 0\}\ \texttt{x := x + 1} \parallel \texttt{x := x + 1}\ \{x = 2\}$$

Did you find this unreasonably difficult? Do not worry (and read on)...

Auxiliary variables

Let `C` be a program and A a set of variables in `C`.

A is a set of *auxiliary variables* of `C` if

- variables in A occurs only in assignments
  not in assignment guards or tests in loops or conditionals

- If $x \in A$ occurs in an assignment $(x_1,\ldots,x_n)$:= $(E_1,\ldots,E_n)$ then x occurs in E_i only when $x_i \in A$
  variables in A cannot influence variables outside A

- *erase(*`C`*,A)* is defined as `C` where all assignments to auxillary variables in A, and all assignments `( ) := ( )`, have been erased.

Auxiliary variable rule

$$\frac{\{\,P\,\}\ \mathtt{C}\ \{\,Q\,\}}{\{\,P\,\}\ \mathtt{C'}\ \{\,Q\,\}}$$

if there is a set A of auxiliary variables of `C` such that

- `C'` = erase(`C`,A), and
- Q does not mention variables in A.

Auxiliary variables are often used to record the state of the computation at some point in time.

Exercise

$$\{ x = 0 \}\ x := x + 1 \parallel x := x + 1 \{ x = 2 \}$$

Idea: Add auxillary variables $done_1$, $done_2$ to catch when each of the assignments have been executed. Initially $(done_1, done_2) := (0,0)$.

Proof outline Δ_1:

$\{ done_1 = 0 \wedge (done_2 = 0 \Rightarrow x = 0) \wedge (done_2 = 1 \Rightarrow x = 1) \}$

$(x,done_1) := (x+1,1)$

$\{ done_1 = 1 \wedge (done_2 = 0 \Rightarrow x = 1) \wedge (done_2 = 1 \Rightarrow x = 2) \}$

Proof outline Δ_2:

$\{ done_2 = 0 \wedge (done_1 = 0 \Rightarrow x = 0) \wedge (done_1 = 1 \Rightarrow x = 1) \}$

$(x,done_2) := (x+1,1)$

$\{ done_2 = 1 \wedge (done_1 = 0 \Rightarrow x = 1) (done_1 = 1 \Rightarrow x = 2) \}$

Exercise (ctd.)

Check that Δ_1 and Δ_2 are interference free. By the Owicki-Gries rule and by the rule of consequence we obtain

$$\{\, \mathtt{x=0} \wedge \mathtt{done}_1 = 0 \wedge \mathtt{done}_2 = 0 \,\}\ \mathtt{C'}\ \{\, \mathtt{x} = 2 \,\}$$

where `C' = (x,done`$_1$`) := (x+1,1)` $\|$ `(x,done`$_2$`) := (x+1,1)`.

Observe that erase(`C'`) = `x := x + 1` $\|$ `x := x + 1`.

By Hoare logic reasoning:

$$\{\mathtt{x} = 0\}\ (\mathtt{done}_1,\mathtt{done}_2) := (0,0)\ ;\ \mathtt{C'}\ \{\mathtt{x} = 2\}\,.$$

By the auxillary variable rule:

$$\{\mathtt{x} = 0\}\ \mathtt{x := x + 1}\ \|\ \mathtt{x := x + 1}\ \{\mathtt{x} = 2\}\,.$$

Owicky-Gries

Great but…

requires reasoning on the whole program.

Uff, sequential reasoning is hard enough!

I would prefer to prove the correctness of a concurrent program one thread at a time, as we did for concurrent separation logic.

Rely-Guarantee reasoning

Owicki & Gries method from a different perspective

Consider again $\{\, x = 0 \,\}\; x := x + 1 \parallel x := x + 2 \;\{\, x = 3 \,\}$.

In the example, the transition of P_2 :: `x := x + 2` (which is the *environment* of P_1), is constrained by the predicate

$$(\underline{x} = 0 \land x = 2) \lor (\underline{x} = 1 \land x = 3)$$

where $\underline{x}$ and x refer to program states *before* and *after* a transition.

This fact suffices to prove that the assertion used in P_2's local proof are not invalidated by interference from P_1.

If we record interference information in a specification,
we can use it in the verification of constituent processes.

No additional interference freedom test will be needed.

Binary relations vs. predicates

We will specify how a program changes the state by means of *binary relations*.

Recall that a (unary) predicate P describes a set of system states.

A *binary relation* describe a set of actions (i.e. transitions) of the system. These are two-state predicates that relate the state just *after* the action (denoted σ) to the state just *before* the action (denoted $\underline{\sigma}$).

We will often call such binary relations *actions.*

Example: in the previous exercise, the transitions of P1:: `x := x + 1` can be described by the action:

$$\{ (x=0 , x=1);\ (x=2 , x=3) \} .$$

Notations:

- we will use predicates to denote actions, e.g. $(\underline{x}=0 \wedge x=1) \vee (\underline{x}=2 \wedge x=3)$;
- True will denote the action that changes arbitrarily the state, that is $\underline{\sigma} \times \sigma$.

Rely/guarantee specifications

A rely/guarantee specification is a quadruple

$$(P, R, G, Q) .$$

- The predicate P is the *precondition*, a single state predicate that describe what is assumed about the initial state;
- the predicate Q is the *postcondition*, a two-state predicate relating the initial state to the final state immediately after the program terminates;
- the *rely condition R* models all the atomic actions of the environment, describing the interference the program can tolerate from its environment.
- the *guarantee condition G* models the atomic actions of the program, and hence it describes the interference that it imposes on the other threads of the system.

A command that satisifies a RG specification: `C` sat (P,R,G,Q).

A R/G computation

The computation of a thread (in black), with the interleaved computations of the other threads of the system (in blue):

$$\underbrace{\overbrace{\sigma_0}^{\text{precondition}} \quad \dots \quad \underbrace{\sigma_i\ \sigma_{i+1}}_{\text{rely}} \quad \dots\ \sigma_k\ \sigma_{k+1}\ \dots \quad \underbrace{\sigma_j\ \sigma_{j+1}}_{\text{rely}} \quad \dots \quad \sigma_f}_{\text{postcondition}}$$

Key ideas:

- the action of the interleaved transitions of the other threads (e.g. the states σ_{i+1} and σ_{j+1}) is constrained by the rely condition;
- the postcondition relates the initial and final state, under the assumption that all other threads respect the rely constraints.

A R/G computation

The computation of a thread (in black), with the interleaved computations of the other threads of the system (in blue):

$$\underbrace{\overbrace{\sigma_0}^{\text{precondition}} \quad \ldots \quad \underbrace{\sigma_i\,\sigma_{i+1}}_{\text{rely}} \quad \ldots\ \sigma_k\,\sigma_{k+1}\ \ldots \quad \underbrace{\sigma_j\,\sigma_{j+1}}_{\text{rely}} \quad \ldots \quad \sigma_f}_{\text{postcondition}}$$

Key

- the σ_{i+1}
  an
- the t all
  oth

- Concurrent separation logic did not allow other threads to modify the state of a thread;

- R/G allows other threads to modify the state of a thread, provided that they respect the rely constraints.

Example: the FINDP algorithm

Problem:

given an array `v[1..n]` and a predicate P,
find the smallest `r` such that P(`v[r]`) holds.

A sequential specification in Hoare logic:

$\{\ \forall\ \mathtt{i}\ .\ P(\mathtt{v[i]})$ is defined $\}$

findp

$\{\ (\mathtt{r} = \mathtt{n} + 1 \wedge \forall\ \mathtt{i}\ .\ \neg P(\mathtt{v[i]})) \vee (1 \leq \mathtt{r} \leq \mathtt{n} \wedge P(\mathtt{v[r]}) \wedge \forall\ \mathtt{i} < \mathtt{r}.\ \neg P(\mathtt{v[i]})))$

Guarantee: this thread can modify the state arbitrarily.

An R/G specification of the *findp* algorithm:

findp $\vDash$ (*pre* , $\mathrm{v} = \underline{\mathrm{v}} \wedge \mathrm{r} = \underline{\mathrm{r}}$, True , *post*)

where *pre* and *post* are as above.

Rely: other threads cannot modify `v` or `r`.

Example: a concurrent FINDP algorithm

Idea:

- partition the array,
- multiple processes search concurrently, one process per partition.

Simple way: `even` and `odd` processes.

Naive concurrency: each process searches a partition, calculates the final result as the minimum of the result of the `even` and `odd` processes.

Problem: can perform worse than sequential (why?)

Example: a concurrent FINDP algorithm

Idea:

- partition the array,
- multiple processes search concurrently, one process per partition.

Simple way: `even` and `odd` processes.

Naive concurrency: each process searches a partition, calculates the final result as the minimum of the result of the `even` and `odd` processes.

Problem: can perform worse than sequential (why?)

Communicating processes:

- introduce a (shared) variable `top` that records the lowest index that satisifies P found so far;
- each thread checks at each iteration that it did not go past `top`.

Example: specification of concurrent FINDP

FindpWorker $\models$

pre: $\forall$ `i` $\in$ partition . P(`v[i]`)) is defined

rely: $\mathtt{v} = \underline{\mathtt{v}} \land \mathtt{top} \leq \underline{\mathtt{top}}$

guar: $\mathtt{top} = \underline{\mathtt{top}} \lor \mathtt{top} < \underline{\mathtt{top}} \land$ P(`v[top]`)

post: $\forall$ `i` $\in$ partition, $\mathtt{i} \leq \mathtt{top} \Rightarrow \neg$P(`v[i]`)

Rely: other threads cannot modify `v` and can only decrement `top.`

Guarantee: the other threads are guaranteed that, if this thread updates `top`, the new value is smaller than the older and is such that P(`v[top]`)holds.

It is then possible to prove that two FindpWorkers, running in parallel, satisfy the specificationof Findp described two slides ago.

(modulo setting up the partitions appropriately and copying the final value from top to r)

Rely/guarantee reasoning

Compare:

Hoare logic specification: $\{ P \}\ \mathtt{C}\ \{ Q \}$

Rely-guarantee specification: $\mathtt{C} \vDash (P, R, G, Q)$

- The rely condition R models all the atomic actions of the environment, describing the interference the program can tolerate from its environment.

The key idea is that if the environment performs actions declared in R, it does not invalidate the precondition P. This is captured by the notion of ***stable relations***.

- the *guarantee condition G* models the atomic actions of the program, and hence it describes the interference that it imposes on the other threads of the system.

The guarantee condition, although not useful to prove the correctness of C on its own, is vital to reason on the correctness of C || C'. *Intuition:* the guarantee of C morally is the rely of C', and viceversa.

Stability

In a specification (P, R, G, Q) we require that P is *stable* under the rely condition, that is, they are resistant to interference from the environment.

Definition (stability): A binary relation Q is *stable* under a *binary relation* R if and only if $(R; Q) \Rightarrow Q$ and $(Q; R) \Rightarrow Q$.

Intuition: doing an environment step before or after Q should not make Q invalid.

For single-state predicates the definition above can be simplified:

Lemma: A single state predicate P is *stable* under a *binary relation* R if and only $P(\sigma) \wedge R(\sigma, \sigma') \Rightarrow P(\sigma')$.

Examples of stable relations

- The predicate P = `bal > 1000` is stable under the action `bal = `$\underline{\texttt{bal}}$` + dep`, provided that `dep > 0` (that is, the action { (`bal`, `bal+dep`) | `dep > 0` }).

  We must show that P(σ) ∧ R(σ, σ') ⇒ P(σ'), which is easy.

- Let ID be the identity action, defined as: ∀ x. x = $\underline{x}$. Any predicate P or action R is stable under ID.

- Given a predicate P, the action

$$\text{preserve}(P) = \underline{P} \Rightarrow P$$

is the smallest action under which P is stable.

Proof rules: parallel composition

- Initially, the preconditions of both threads must hold;
- each thread must be immune to interference by all the other threads;
  this is checked by the $G_i \Rightarrow R_j$ conditions (remember that p_i is stable under R_i).
- the total action of the program is given by the composition of the actions of the two threads in either order, allowing for repeated environment interference $(R_1 \wedge R_2)^*$ in between.
- the concurrent threads can only guarantee the disjunction G1 ∨ G2.

$$\frac{\begin{array}{ll} C_1 \models (p_1, R_1, G_1, q_1) & G_1 \Rightarrow R_2 \\ C_2 \models (p_2, R_2, G_2, q_2) & G_2 \Rightarrow R_1 \end{array}}{C_1 \,\|\, C_2 \models (p_1 \wedge p_2,\ R_1 \wedge R_2,\ G_1 \vee G_2,\ q)}$$

$$q = (q_1; (R_1 \wedge R_2)^*; q_2) \vee (q_2; (R_1 \wedge R_2)^*; q_1)$$

Proof rules: atomic actions

$$\frac{\{p\}\ C\ \{q\}\text{ in Hoare-logic}}{C\ \models\ (p,\ \text{Preserve}(p),\ q \vee \text{ID},\ q)}$$

If C is an atomic action, then

- the precondition and the postcondition are determined by sequential reasoning;
- the rely condition states that the environment must at least preserve the precondition; (def preserve)
- the guarantee condition states that either the command terminates and its action is described by Q, or nothing happens on the global state (ID).

Proof rules: parallel composition, again

$$\frac{\begin{array}{ll} C_1 \models (p_1, R_1, G_1, q_1) & G_1 \Rightarrow R_2 \\ C_2 \models (p_2, R_2, G_2, q_2) & G_2 \Rightarrow R_1 \end{array}}{C_1 \parallel C_2 \models (p_1 \wedge p_2, R_1 \wedge R_2, G_1 \vee G_2, q)}$$

In plain English, proving the safety of a parallel program reduces to:

- a sequential proof of the post-condition and guarantee condition of each individual thread, assuming that its rely condition is true;
- a pairwise proof that every other thread's guarantee condition implies this thread's rely conditiion.

Proof rules: sequential composition and weaken

- The precondition of the second operand must follow from the postcondition of the first. The total action is given by the composition of the actions of its components accounting for environment interference in between.

$$\frac{\begin{array}{ll} C_1 \models (p_1, R, G, q_1) & \\ C_2 \models (p_2, R, G, q_2) & q_1 \Rightarrow p_2 \end{array}}{C_1 \,;\, C_2 \models (p_1, R, G, (q_1; R^*; q_2))}$$

- It is always safe to replace the specification by a stronger specification:

$$\frac{\begin{array}{ll} p' \Rightarrow p & R' \Rightarrow R \\ G \Rightarrow G' & q \Rightarrow q' \\ \multicolumn{2}{c}{C \models (p, R, G, q)} \end{array}}{C \models (p', R', G', q')}$$

Many R/G proof systems

Disclaimer:

- several (slightly) different proof systems have been proposed for Rely/Guarantee reasoning. The one presented here is taken from Vafeiadis et al.

For instance, Jones requires that, given a specification C ⊨ (P,R,G,Q), both the precondition and postcondition are stable under the rely guarantee R. This leads to a mildly different proof system and mildly different proofs.

All these proof systems enjoy soundness (proof non-trivial).

$\{ x = 0 \}\ x := x + 1 \parallel x := x + 2\ \{ x = 3 \}$

In RG, we would write this specification as:

$$x := x + 1 \parallel x := x + 2 \vDash (x{=}0,\ x = \underline{x},\ \text{True},\ x{=}3)$$

The specifications of the two threads are:

$x := x + 1 \vDash$

$$(x{=}0 \lor x{=}2,\ (\underline{x}{=}0 \land x{=}2) \lor (\underline{x}{=}1 \land x{=}3),\ (\underline{x}{=}0 \land x{=}1) \lor (\underline{x}{=}2 \land x{=}3),\ (\underline{x}{=}0 \land x{=}1) \lor (\underline{x}{=}2 \land x{=}3))$$

Observe that $(x{=}0 \lor x{=}2)$ is stable under the rely condition $(\underline{x}{=}0 \land x{=}2) \lor (\underline{x}{=}1 \land x{=}3)$.

$x := x + 2 \vDash$

$$(x{=}0 \lor x{=}1,\ (\underline{x}{=}0 \land x{=}1) \lor (\underline{x}{=}2 \land x{=}3),\ (\underline{x}{=}0 \land x{=}2) \lor (\underline{x}{=}1 \land x{=}3),\ (\underline{x}{=}0 \land x{=}2) \lor (\underline{x}{=}1 \land x{=}3)\})$$

Observe that $(x{=}0 \lor x{=}1)$ is stable under the rely condition $(\underline{x}{=}0 \land x{=}1) \lor (\underline{x}{=}2 \land x{=}3)$.

It is trivial to show that $G_i \Rightarrow R_j$. Can you complete the proof?

The budget example with rely-guarantee

P$_1$:: `bal := bal + dep`

P$_2$:: `if bal > 1000 then credit := 1 else credit := 0`

Proof goal:

$$P_1 \parallel P_2 \models$$

$$\{\, \mathtt{dep} > 0,$$
$$\mathtt{dep} = \underline{\mathtt{dep}} \wedge \mathtt{credit} = \underline{\mathtt{credit}},$$
$$\mathtt{True},$$
$$\mathtt{bal} = \underline{\mathtt{bal}} + \mathtt{dep} \wedge \mathtt{dep} > 0 \wedge (\mathtt{credit} = 1 \Rightarrow \mathtt{bal} > 1000) \,\}$$

Exercise: can you complete this proof?

You will need the rule for if/then/else:

$$\frac{\begin{array}{ll} C_1 \models (p \wedge b_1, R, G, q) & \overleftarrow{b} \wedge R^* \Rightarrow b_1 \\ C_2 \models (p \wedge b_2, R, G, q) & \neg\overleftarrow{b} \wedge R^* \Rightarrow b_2 \end{array}}{\textbf{if } \langle b \rangle \textbf{ then } C_1 \textbf{ else } C_2 \models (p, R, G, q)}$$

Case study:

linearisability of highly-concurrent data-structures

Concurrent data structures

- *Coarse grain*: a single owner thread (e.g. one big lock on the data structure);
- *Fine grain*: multiple threads inside the same object simultaneously.

Several patterns:

- *lock coupling*: locks are acquired and released in a "hand-over-hand" order, acquiring the next lock before releasing the previous;
- *optimistic*: a thread searches a data structure without acquiring locks, locks the sough-after component, and then validates;
- *lazy*: two phases to remove an object, *logical* (e.g. setting a flag) and *physical*.
- *lock-free*: some thread always complete in a finite number of steps, even in presence of failures or delays by other threads.

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.

Implementing the set data type

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

Concrete shared state:

- sorted linked list;
- two sentinel nodes: `Head` with value -∞ and `Tail` with value +∞;
- no duplicates;
- each node in the list is associated with a lock.

We will implement a lock-coupling algorithm.

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
```

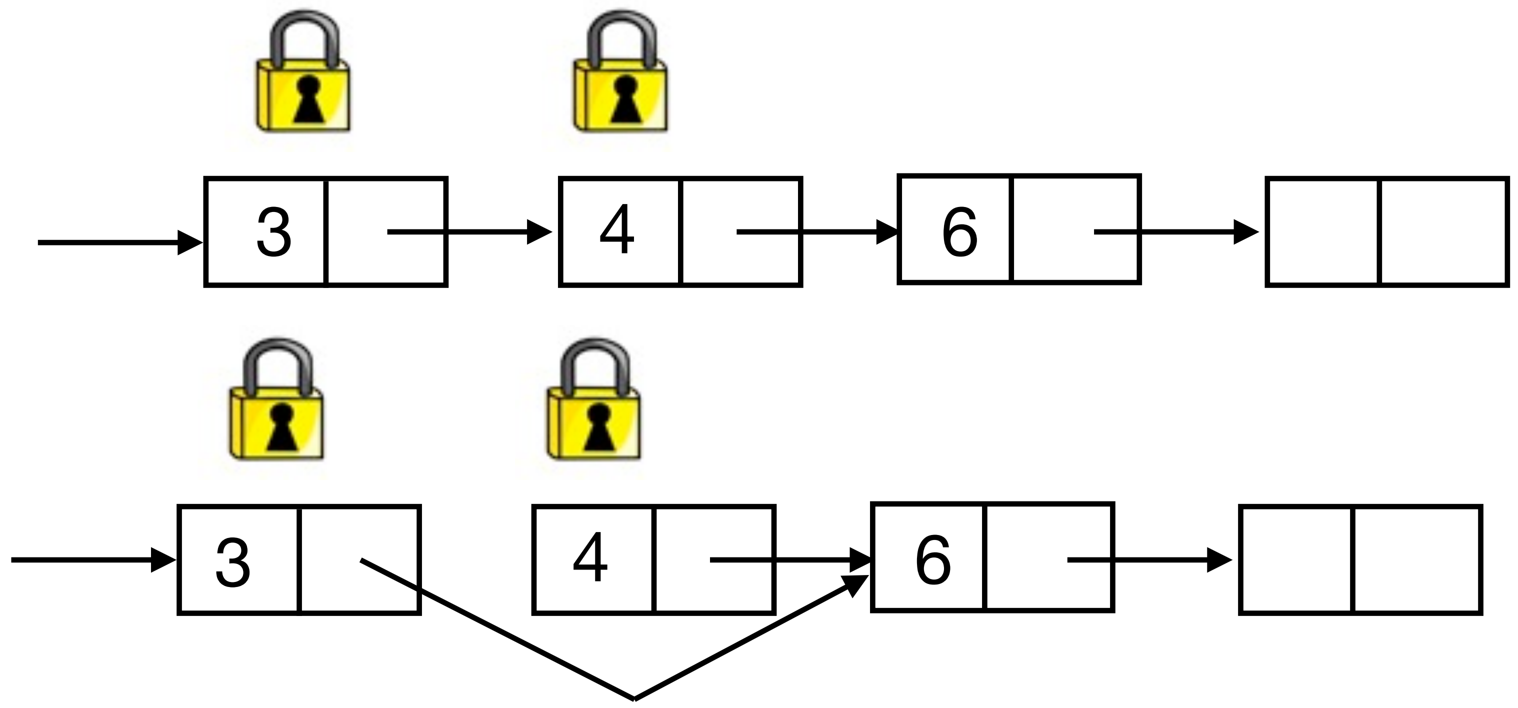
```
add(e) :
  n1, n3 := locate(e) ;
  if n3.val ≠ e then
    n2 := new Node(e) ;
    n2.next := n3 ;
    n1.next := n2    [*A] ;
    Result := true
  else
    Result := false    [*B]
  endif ;
  n1.unlock() ;
  n3.unlock() ;
  return Result
```

```
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
```

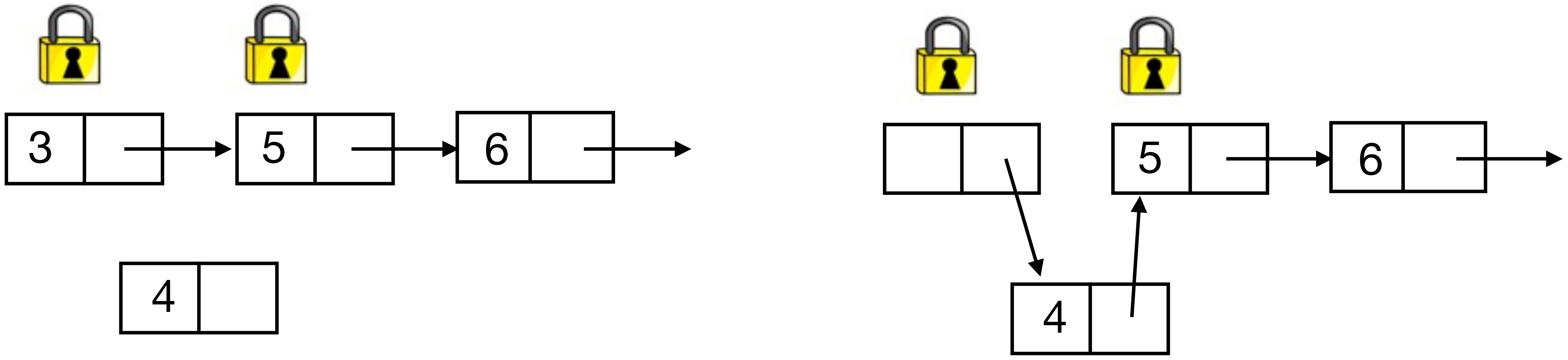
- *locate* uses *lock-coupling*: the lock on some node is not released until the next is locked. Returns the previous and current (that is the first node >= e) node, both locked.
- *add* inserts the new element while holding the locks of the previous and next node;
- *remove* updates the previous *next*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

Key property: locked nodes are always reachable

`remove(4)`:



add(4):



Lock-coupling and separation logic

Can we reason on lock-coupling algorithms using separation logic?

- must provide invariants for the locks.

The lock invariant for a node should state that when the lock is held, the node is reachable from the list.

Specifying this invariant in separation logic is painful… (the other threads must be allowed to read, lock, modify, the unlocked nodes the precede the current one).

An example of the VeriFast tool details this example: http://www.cs.kuleuven.be/~bartj/verifast/examples/lcset/lcset.c.html . However, there should be a simpler way...

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:

```
L.lock() = < L.owner = null → L.owner := self >

L.unlock() = < L.owner := null >
```

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

`L.lock()` ⊨ (`L.owner ≠ self` , *lockRely* , *lockGuar* , `L.owner = self`)

`L.unlock()` ⊨ (`L.owner = self` , *lockRely* , *lockGuar* , `L.owner ≠ self`)

where *lockRely* = ID(`L.owner = self`)

and *lockGuar* = (∀i ∉ {`self`, `null`}. ID(`L.owner = i`)).

Rely/Guarantee and thread identifiers

In Rely/Guarantee reasoning, threads identifiers are abstract and the only one of interest are `self` and non-`self`.

The special variable `self` must be instantiated with the proper `tid` value in the rule for parallel composition. In particular the

$$G_i \Rightarrow R_j$$

check becomes

$$G_i\,[i/\texttt{self}] \Rightarrow R_j\,[j/\texttt{self}]\ .$$

Pessimistic implementation of a set via a linked list

We write Node(`n`) for the assertion that `n` is a valid public node, and `n → m` for Node(`n.next`) ∧ `n.next = m`.

The data-structure invariant states that:

- `Head` and `Tail` contain the infinity values;
- if a (public) node other than `Tail` is unlocked, it points to a valid node;
- if two unlocked nodes follow each other, their values are sorted;
- the abstract set *Abs* and the values of non-sentinel reachable nodes are equal.

$$\mathsf{noOwn}(\mathrm{n}) \overset{\mathrm{def}}{=} \mathrm{n.owner} = \mathsf{null}$$

$$\begin{aligned} \mathit{ListInv} \overset{\mathrm{def}}{=}\ & \mathsf{Node}(\mathrm{Head}) \\ & \wedge\ \mathrm{Head.val} = -\infty \ \wedge\ \mathrm{Tail.val} = +\infty \\ & \wedge\ \forall_{\mathsf{Node}}\, \mathrm{n}.\ (\mathsf{noOwn}(\mathrm{n}) \wedge \mathrm{n.val} < +\infty) \Rightarrow \mathsf{Node}(\mathrm{n.next}) \\ & \wedge\ \forall_{\mathsf{Node}}\, \mathrm{n\,m}.\ (\mathsf{noOwn}(\mathrm{n,m}) \wedge \mathrm{n} \to \mathrm{m}) \Rightarrow \mathrm{n.val} < \mathrm{m.val} \\ & \wedge\ \mathit{Abs} = \{\mathrm{n.val} \mid \mathrm{Head} \to^{*} \mathrm{n} \wedge \mathrm{n.val} \neq \pm\infty\} \end{aligned}$$

Observe that locked nodes are not required to satisfy the invariant.

Rely/Guarantee proof of add and remove

It is then possible to prove that the operations `add(e)` and `remove(e)` operations preserve the list invariant:

$$\texttt{locate(e)} \models (\ \mathrm{ListInv} \wedge -\infty < \texttt{e} < +\infty\ ,\ R\ ,\ G\ ,\ \mathrm{ListInv} \wedge \mathrm{Head} \longrightarrow^{*} \mathrm{pred} \longrightarrow \mathrm{curr} \wedge \texttt{pred.val} < \texttt{e} \leq \texttt{curr.val} \wedge \textit{pred.owner} = \textit{curr.owner} = \textit{self}\ )$$

$$\texttt{add(e)} \models (\ \mathrm{ListInv} \wedge -\infty < \texttt{e} < +\infty\ ,\ R\ ,\ G\ ,\ \mathrm{ListInv} \wedge \texttt{result} = \textit{AbsResult}\ )$$

$$\texttt{remove(e)} \models (\ \mathrm{ListInv} \wedge -\infty < \texttt{e} < +\infty\ ,\ R\ ,\ G\ ,\ \mathrm{ListInv} \wedge \texttt{result} = \textit{AbsResult}\ )$$

where the *R* (resp. *G*) condition states that the environment (resp. thread) actions preserve the list invariant and use locks properly:

$$R \stackrel{\text{def}}{=} \forall_{\mathsf{Node}}\ \mathrm{n.}\ \mathrm{Preserve}(\mathit{ListInv}) \wedge \mathrm{n.}\mathit{LockRely} \wedge \overleftarrow{\mathrm{n.owner}} = \mathsf{self} \Rightarrow \mathrm{ID}(\mathrm{n.val}, \mathrm{n.next}, \mathrm{Head} \rightarrow^{*} \mathrm{n})$$

$$G \stackrel{\text{def}}{=} \forall_{\mathsf{Node}}\ \mathrm{n.}\ \mathrm{Preserve}(\mathit{ListInv}) \wedge \mathrm{n.}\mathit{LockGuar} \wedge \overleftarrow{\mathrm{n.owner}} \neq \mathsf{self} \Rightarrow \mathrm{ID}(\mathrm{n.val}, \mathrm{n.next}, \mathrm{Head} \rightarrow^{*} \mathrm{n})$$

Linearisability

The instructions marked with `[*A]` and `[*B]` are the *linearisation points* of `add(e)`.

To prove this, we embed the abstract implementation of the algorithm at that point, and check that the abstract implementation is implied by the post-condition of `[*A]` and `[*B]`.

```
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
```

$Abs = \underline{Abs} \cup \{e\}$

$Abs = \underline{Abs} \cup \{e\}$

```
I = ListInv ∧ -∞ < e < +∞

{ I }
n1,n3 = locate(e)
{ I ∧ Head ⟶* n1 ⟶ n3 ∧ n1.val<e≤n3.val ∧ n1.owner=n3.owner=self }
if n3.val /= e then
  { I ∧ Head ⟶* n1 ⟶ n3 ∧ n1.val<e<n3.val ∧ n1.owner=n3.owner=self }
  n2 = new Node(e)
  { … ∧ PNode(n2) ∧ n2.val = e }
  n2.next = n3;
  { … ∧ PNode(n2) ∧ n2.val = e ∧ n2.next = n3 } ⇒ { e ∉ Abs }
  n1.next = n2;
  { I ∧ n1.owner=n3.owner=self ∧ Node(n2) ∧ Abs = Abs ∪ {e} }
  Result := true;
  { I ∧ n1.owner=n3.owner=self ∧ Node(n2) ∧ Abs = Abs ∪ {e} ∧ AbsResult = e ∉ Abs }
else
  { I ∧ Head ⟶* n1 ⟶ n3 ∧ n3.val = e ∧ n1.owner=n3.owner=self }
  Result := false;
  { I ∧ n1.owner=n3.owner=self ∧ Abs=Abs∪{e} ∧ AbsResult=e∉Abs ∧ Result=AbsResult }
{ I ∧ n1.owner=n3.owner=self ∧ Result=AbsResult }
n1.unlock();
{ I ∧ n3.owner=self ∧ Result=AbsResult }
n3.unlock{};
{ I ∧ Result=AbsResult }
```

Must also check that R ⇒ all intermediate conditions, and that all the intermediate conditions imply G (easy).

See Vafeiadis, Herlihy, Hoare, Shapiro: A safety proof of a lazy concurrent list-based set implementation.

Perspectives

- Separation logic has difficulties dealing with interference, but its specifications are simpler because they describe only the relevant state that the program accesses.
- Rely-Guarantee copes naturally with interference, but its specifications describe the entire state.

Is a marriage between separation logic and rely-guarantee possible?

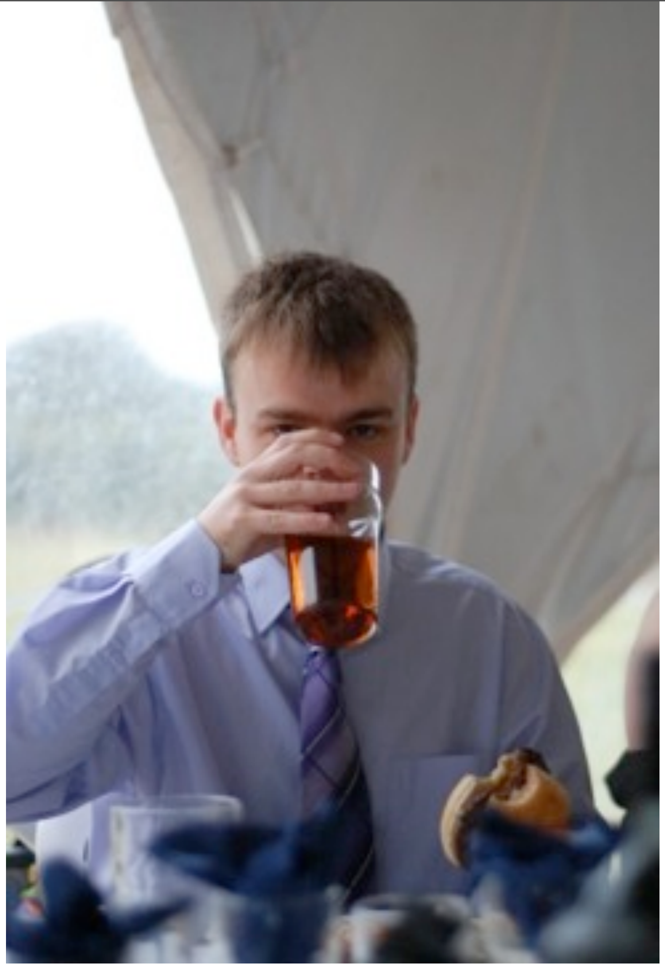
Recent works in this direction seem very promising:

Parkinson, Vafeiadis: *A marriage of rely/guarantee and separation logic.*

Doods, Feng, Parkinson, Vafeiadis: *Deny-Guarantee reasoning.*

Exercise: who is who?

I am extremely grateful to Cliff Jones, Matthew Parkinson and Viktor Vafeiadis for their help preparing this lecture.



References (two PhD thesis):

Leonor Prensa Nieto, *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL,* 2002.

Viktor Vafeiadis: *Modular fine-grained concurrency verification*, 2007.

available from http://moscova.inria.fr/~zappa/teaching/mpri/2010/ .

Next lecture: all lies!

(the semantics of a concurrent program is not the interleaving of its atomic actions)