Verifying concurrent programs

UNIVERSITY OF CAMBRIDGE
Computer Laboratory
Theory and Semantics Group

Concurrency is coming!

Concurrency means *interaction between processes*.

Classic problems:

- Reading and writing from shared memory
- Communicating asynchronously

Concurrency has been regarded with suspicion:

> *"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."*
>
> - Java Sun Tutorial

Concurrency makes errors hard to replicate.

But in a multi-core world, concurrency is inevitable:

- Multi-core processors are taking over
- Concurrency built in from the hardware level

Verification

Programs often do not do what we want. Two solutions are:

- Testing possible inputs. Too many possible inputs!
- Verifying for *all* possible inputs

Verification: *prove* that program conforms to its specification.

However, current verification approaches do not work well with fine-grained concurrency.

Rely/guarantee & separation logic

Concurrency verification is about two things:

(1) *partitioning* state into local and shared areas

(2) controlling *interference* between processes

Separation logic is a logic for shared mutable data-structures:

- Reason locally about data-structures.
- Partitioning of state makes proofs tractable.

Rely/guarantee models interference as actions:

- *Rely* = 'interference from the environment'
- *Guarantee* = 'interference caused by the process'

We use separation logic for partitioning and rely/guarantee for interference.

Each process has its own local state, and they all share a single global state.

Actions only operate over the shared state, making proofs simpler.

People

- **Viktor Vafeiadis**, Microsoft Research Cambridge
- **Matthew Parkinson**, University of Cambridge
- **Cristiano Calcagno**, Imperial College
- **Mike Dodds**, University of Cambridge

Lock coupling list

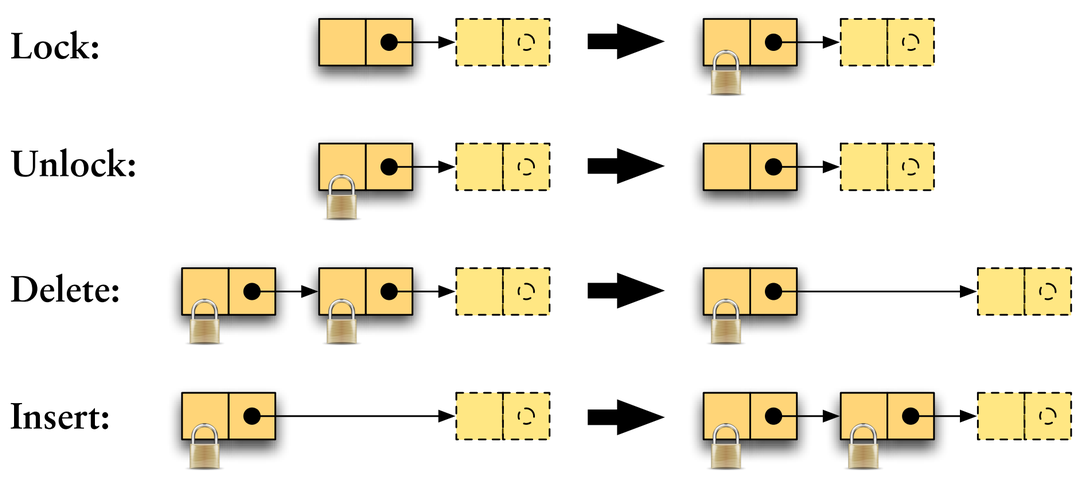
Two ways to concurrently access a list:

- Coarse-grained: lock the *whole list*
- Fine-grained: lock *individual nodes*

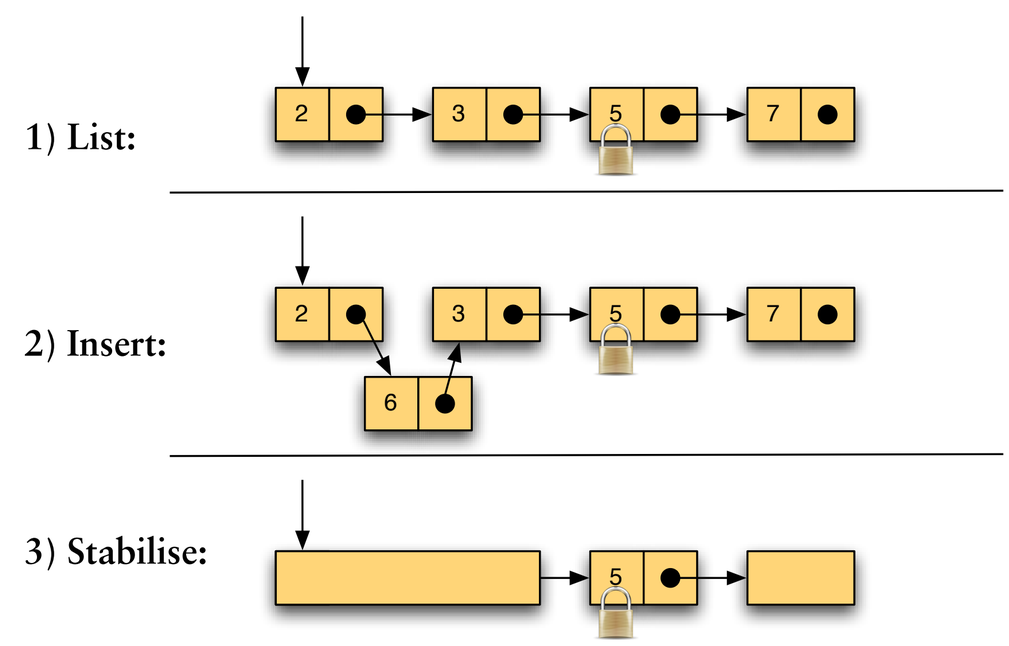
Lock-coupling list algorithm locks nodes *hand-over-hand*, i.e. lock a node only after locking the preceding node:

1) Lock the list head
2) Until target node discovered, do:
   A. lock the next node
   B. release the current node
3) Insert / delete node

Algorithm relies and guarantees defined by actions:



Check *stability* under interference from rely actions. After applying an action we must get back a list.



The result of an **Insert** operation is a list, meaning the action preserves the invariant.

Tool support

Logic is tractable for automatic checking.

SmallfootRG operates by *symbolic execution*, that is, execution over an abstract domain.

- Input: specification and actions
- Stability inference is automatic
- Output: proof of correctness