Software Verification

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

November 24, 2025

Lecture 12

based on previous editions by João Seco, Luís Caires, and Bernardo Toninho

also based on lectures by Jean-Marie Madiot and Arthur Charguéraud

1. Presentation of the third handout.

Verification of Heap-Manipulating Programs (4/4)

---

2. Recap Construction of Concurrent ADTs
3. Linear Resource Usage and Sharing
4. Fractional Permissions
5. Fractional Permissions and Locks

Recap Construction of Concurrent ADTs

Concurrent ADT Construction Steps

Challenge: Can we systematically transform and verify correct a "sequential" ADT implementation into an efficient "concurrent" ADT implementation?

Concurrent ADT Recipe

1. Associate a monitor to the ADT (`ReentrantLock` - **`mon`**)
2. Define the sequential ADT Representation Invariant (**`RepInv`**) that talks about the shared state
3. Define the concurrent ADT Representation Invariant that talks about the monitor and associated conditions.
4. In the implementation of each operation of the CADT
   - 4.1 get access to the shared state representation invariant, use **`mon.lock()`**
   - 4.2 when done and only if the shared state representation invariant holds, use **`mon.unlock()`**
5. Replace the ADT operation pre-conditions by monitor conditions inside the monitor.
6. Design voluntary yielding points to implement the correct interleaving of operations.

Concurrent ADT Construction Steps

To replace ADT operation pre-conditions by monitor conditions inside the monitor, we must consider the following aspects:

- When a thread enters a CADT operation and gets ownership of the representation invariant, it should check the state to satisfy (or not) the pre-condition (*e.g.*, wants to decrement but counter value is zero).
- The thread must then await for the condition to hold (*e.g.*, for the value to be `> 0`).
- Conversely, whenever a thread running inside the CADT establishes any one of the monitor conditions (*e.g.*, `inc` establishes value `> 0`), it has the duty to signal the condition (so that the runtime system may awake a waiting thread).
- Note that signaling is there to help the system to progress, and simplify the implementation of monitors.

Linear Resource Usage and Sharing

Resource Sharing

Exclusive access to resources is sound in Concurrent ADTs, *i.e.*, it avoids any kind of interference.

However, not all interferences and sharing situations are harmful.

One crucial example where aliasing and sharing is safe, and many times necessary, is when one only reads information from memory.

Let us use a familiar example from previous lectures: `Account`.

Account ADT

```
public class Account {
  int balance;
  ...
  static void test2(Account a1, Account a2)
  //@ requires a1.AccountInv(?b1) &*& a2.AccountInv(_) ;
  //@ ensures a1.AccountInv(b1) &*& a2.AccountInv(_) ;
  {
    int v1;
    v1 = a1.getBalance();
    a2.deposit(v1);
  }
  ...
}
```

Separation Logic proves that a1 does not change.

The frame principle of SL and the separating conjunction supports local reasoning. Only focusing on the exact footprint of the program fragment.

Account ADT

```
public class Account {
  int balance;
  ...
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    b1.deposit(10);
    //@ assert b1.Account(?b) &*& b2.Account(_);
    test2(b1,b2);
    //@ assert b1.Account(b) &*& b2.Account(_);
  }
  ...
}
```

The precondition of `test2` holds: we are sure that b1 and b2 are not aliases (separated memory heap chunks).

Account ADT

```
public class Account {
  int balance;
  ...
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    b1.deposit(10);
    //@ assert b1.Account(?b) &*& b2.Account(_);
    test2(b1,b1); // ERROR!
  }
  ...
}
```

Precondition of `test2` is not valid now, we cannot show that `b1.Account(_)` and `b2.Account(_)` are separated.

Separation and Sharing

The use of $A \star B$ solves a limitation of Hoare Logic: it helps to keep track of aliases and memory footprints, as needed to check programs in C, Java, or even OCaml.

However, the use of $A \star B$ in Separation Logic forces all usages of memory references to be used linearly.

Each usage of a memory cell requires a permission $c.\mathtt{N} \rightsquigarrow v$, and there is only one permission around for each memory cell.

Thus memory cells cannot be shared or aliased, unless in trivial contexts (where the reference is passed around but not really used for reading or writing).

However, there are of course situations in which sharing and aliasing is safe and necessary.

Accound ADT

```
public class Account {
  static int sum(AccountS a1, AccountS a2)
  //@ requires ??;
  //@ ensures ??;
  {
    return a1.getBalance()+a2.getBalance();
  }
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1,b2);
 }
}
```

Which contract should we assign to method sum?

Accound ADT

```
public class Account {
  static int sum(AccountS a1, AccountS a2)
  //@ requires a1.Account(?b1) &*& a2.Account(?b2);
  //@ ensures a1.Account(b1) &*& a2.Account(b2);
  {
    return a1.getBalance()+a2.getBalance();
  }
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1, b2);
 }
}
```

The precondition of sum holds, so everything is OK.

Accound ADT

```
public class Account {
  static int sum(AccountS a1, AccountS a2)
  //@ requires a1.Account(?b1) &*& a2.Account(?b2);
  //@ ensures a1.Account(b1) &*& a2.Account(b2);
  {
    return a1.getBalance()+a2.getBalance();
  }
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1, b1);
 }
}
```

The precondition does not hold now, but “intuitively” the contract is OK. It works even if a1 and a2 are aliases.

Fractional Permissions

Sharing in Separation Logic

In "Classical" Logic, we have the law

$$A \Leftrightarrow A \wedge A$$

This is useful to reason about pure (aka "duplicable") values.

In Separation Logic, we do not have in general

$$A \not\Leftrightarrow A \star A$$

We have seen that the separation principle is key to control aliasing, and support frame (local) reasoning.

In Separation logic, we would like to "sometimes" allow

$$A \Leftrightarrow A \star A$$

(*E.g.*, if the various usages of A do not actually interfere)

Fractional Permissions [Boyland, 2003]

Answer: we may try to extract "partial views" of the whole permission, using "fractional permissions", as follows:

$$[f]A \Leftrightarrow [f_1]A \star [f_2]A$$

where $0 \leq f \leq 1 \wedge f = f_1 + f_2$.

Fractional permissions are rational numbers f, such that $0 \leq f \leq 1$.

A permission $[1]A$ means the "whole of A", so $[1]A \Leftrightarrow A$.

Writing requires full (*i.e.*, 1) permission, unlike reading.

Assignment and Lookup Rules (Fractions)

The assignment rule in separation logic is

$$\frac{}{\mathsf{SL}\{x \rightsquigarrow V\}\, x := t\, \{x \rightsquigarrow t\}}$$

The memory lookup rule is now:

$$\frac{}{\mathsf{SL}\{[f]L \rightsquigarrow V\}\, y := L\, \{[f]L \rightsquigarrow V \star y == V\}}$$

Here, we can allow $f < 1$. But always $f > 0$.

The frame rule still works.

Fractional Permissions

The key principle to keep in mind is

$$[f]A \Leftrightarrow [f_1]A \star [f_2]A$$

where $0 \leq f \leq 1 \wedge f = f_1 + f_2$.

Using this splitting rule, one can "duplicate" references and heap chunks, allowing sharing or aliasing safely.

We may recombine fractions later on, eventually regaining the full permission (uniqueness) again.

Mutable objects may also be safely shared using fractional permissions, if protected by invariants.

This is the case for concurrent abstract data types, where shared state use is coordinated by locks.

Account ADT

```
public class Account {
  static int sum(AccountS a1, AccountS a2)
  //@ requires [?f1]a1.Account(?b1) &*& [?f2]a2.Account(?b2);
  //@ ensures [f1]a1.Account(b1) &*& [f2]a2.Account(b2);
  {
    return a1.getBalance()+a2.getBalance();
  }
  static void main (String args[] )
  //@ requires true;
  //@ ensures true ;
  {
    Account b1 = new Account();
    Account b2 = new Account();
    int v = sum(b1, b1);
  }
}
```

The precondition of the sum call holds (a1 and a2 are aliases).
This is fine because getBalance only reads.

Fractional Permissions and Locks

Java Monitors Specification

```
package java.util.concurrent.locks;
/*@
predicate lck(ReentrantLock s; int p, predicate() inv);

predicate cond(Condition c; predicate() inv, predicate() p);

predicate enter_lck(int p, predicate() inv) = (p == 0 ? emp :
    inv()) ;

predicate set_cond(predicate() inv, predicate() p) = true;
@*/
```

Java Monitors Specification

```
public class ReentrantLock {

  public ReentrantLock();
  //@ requires enter_lck(1,?inv);
  //@ ensures lck(this, 1, inv);

  public void lock();
  //@ requires [?f]lck(?t, 1, ?inv);
  //@ ensures [f]lck(t, 0, inv) &*& inv();

  public void unlock();
  //@ requires [?f]lck(?t, 0, ?inv) &*& inv();
  //@ ensures [f]lck(t, 1, inv);

  public Condition newCondition();
  //@ requires lck(?t, 1, ?inv) &*& set_cond(inv, ?pred);
  //@ ensures lck(t, 1, inv) &*& result != null &*& cond(
      result,inv,pred);
}
```

Java Monitors Specification

```
package java.util.concurrent.locks;

public interface Condition {
  public void await();
  //@ requires cond(this,?inv,?acond) &*& inv();
  //@ ensures cond(this,inv, acond) &*& acond();

  public void signal();
  //@ requires cond(this,?inv,?acond) &*& acond();
  //@ ensures cond(this,inv,acond) &*& inv();
}
```

Counter ADT

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  void inc()
  //@ requires [?f]BCounterInv(this);
  //@ ensures [f]BCounterInv(this);
  {
    //@ open [f]BCounterInv(this);
    mon.lock();
    //@ open BCounter_shared_state(this)();
    if( N == MAX ) {
      //@ close BCounter_shared_state(this)();
      notmax.await();
      //@ open BCounter_nonmax(this)();
    }
    N++;
    //@ close BCounter_nonzero(this)();
    notzero.signal();
    mon.unlock();
    //@ close [f]BCounterInv(this);
  }
```

Counter ADT

```
class BCounter {
  int N;
  int MAX;
  ReentrantLock mon;
  Condition notzero;
  Condition notmax;

  void dec()
  //@ requires [?f]BCounterInv(this);
  //@ ensures [f]BCounterInv(this);
  {
    //@ open [f]BCounterInv(this);
    mon.lock();
    //@ open BCounter_shared_state(this)();
    if ( N == 0 ) {
      //@ close BCounter_shared_state(this)();
      notzero.await();
      //@ open BCounter_nonzero(this)();
    }
    N--;
    //@ close BCounter_nonmax(this)();
    notmax.signal();
    mon.unlock(); // release ownership of the shared state
    //@ close [f]BCounterInv(this);
  }
```

Counter ADT – Client

```
public class Main {

  public void doMain ()
  //@ requires true;
  //@ ensures true;
  {
    BCounter ccount = new BCounter();
    //@ assert BCounterInv(ccount);
    Inc_thread it = new Inc_thread(ccount);
    //@ assert [?f]BCounterInv(ccount) &*& f < 1.0;
    (new Thread(it)).start();
    Dec_thread dt = new Dec_thread(ccount);
    new Thread(dt).start();
  }
}
```

Verifast Interface for Threads

```
public interface Runnable {
  //@ predicate pre();                    -- to be redefined in sub-
      class
  //@ predicate post();                   -- to be redefined in sub-
      class
  public void run();
  //@ requires pre();
  //@ ensures post();
}

public class Thread {
  static final int MAX_PRIORITY = 10;
  //@ predicate Thread(Runnable r, boolean started);
  public Thread(Runnable r);
  //@ requires true;
  //@ ensures Thread(r, false);
  void start();
  //@ requires Thread(?r, false) &*& r.pre();
  //@ ensures Thread(r, true);
  void setPriority(int newPriority);
  //@ requires Thread(?r, false);
  //@ ensures Thread(r, false);
}
```

Counter ADT – Client

```
class Inc_thread implements Runnable {
  public BCounter loc_cc;
  //@ predicate pre() = Inc_threadInv(this);
  //@ predicate post() = true;
  public Inc_thread(BCounter cc)
  //@ requires cc != null &*& [1/2]CCounterInv(cc);
  //@ ensures Inc_threadInv(this);
  {
    loc_cc = cc;
  }
  public void run()
  //@ requires pre();
  //@ ensures post();
  {
    while(true)
    //@ invariant Inc_threadInv(this);
    { loc_cc.inc();}
  }
}
```

More Concurrency

More Concurrency

The basic monitor scheme does not allow more than 1 thread to ever touch the shared state.

However, it should be ok to let more than 1 thread access the shared state simultaneously, if they do not interfere in "unsafe" ways (*e.g.*, they only read).

This amounts to controlling sharing granularity, and there are many, more refined, ways to do this.

A simpler to use pattern is the N readers & 1 writer monitor idiom, which is already very useful.

The key idea is to protect the concurrent object with a more refined monitor wrapper, that coordinates read / write access to the shared state.

N Readers & 1 Writer

```
class NReaders1Writer {
  int readercount; // number of readers inside the CADT
  boolean busy; // busy means someone writing inside ADT
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;

  RW() { ... }
  void StartRead()
  {
    mon.lock();
    while (busy) { OKtoRead.wait();}
    readercount = readercount + 1;
    OKtoRead.signal();
    mon.unlock();
  }
  void EndRead()
  {
    mon.lock();
    if (readercount > 0) { readercount--; }
    if (readercount == 0) OKtoWrite.signal();
    mon.unlock();
  }
  ...
}
```

N Readers & 1 Writer

```
class NReaders1Writer {
  int readercount; // number of readers inside the CADT
  boolean busy; // busy means someone writing inside ADT
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;

  void StartWrite ()
  {
    mon.lock();
    if ((readercount == 0) || !busy) OKtoWrite.signal();
    busy = true;
    mon.unlock();
  }
  void EndWrite ()
  {
    mon.lock();
    busy = false;
    OKtoRead.signal();
    mon.unlock();
  }
}
```

N Readers & 1 Writer

```
/*@
  predicate_ctor NR1W_shared_state(NReaders1Writer rw)() = rw.
      busy |-> ?b &*&
    rw.readercount |-> ?n &*& n >= 0 &*& n > 0 ? b == false:
        true;

  predicate_ctor NR1W_canRead(NReaders1Writer rw)() = rw.busy
      |-> ?b &*&
    rw.readercount |-> ?n &*& n >= 0 &*& b == false;

  predicate_ctor NR1W_canWrite(NReaders1Writer rw)() = rw.busy
       |-> ?b &*&
    rw.readercount |-> ?n &*& n == 0 &*& b == false;
@*/

class NReaders1Writer {
  int readercount; // number of readers inside the CADT
  boolean busy; // busy means someone writing inside ADT
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;

  ...
}
```

N Readers & 1 Writer

```
/*@
predicate RepInv() =
  this.mon |-> ?l
  &*& this.canRead |-> ?r
  &*& this.canWrite |-> ?w
  &*& l != null
  &*& lck(l, 1, NR1W_shared_state(this))
  &*& r != null
  &*& cond(r, NR1W_shared_state(this), NR1W_canRead(this))
  &*& w != null
  &*& cond(w, NR1W_shared_state(this), NR1W_canWrite(this));
@*/

class NReaders1Writer {
  int readercount; // number of readers inside the CADT
  boolean busy; // busy means someone writing inside ADT
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;

  ...
}
```

N Readers & 1 Writer

```
class NReaders1Writer {
  int readercount; // number of readers inside the CADT
  boolean busy; // busy means someone writing inside ADT
  ReentrantLock mon;
  Condition OKtoRead, OKtoWrite;
  // { INV = busy ==> readercount == 0 }
  // { OktoRead = INV && not busy }
  // { OktoWrite = INV && not busy && (readercount == 0) }

  void StartRead ()
  //@requires RepInv();
  //@ensures RepInv();
  { mon.lock();
    //@close NR1W_shared_state(this)();
    if (busy) { OKtoRead.wait(); }

    readercount = readercount + 1;
    //@close NR1W_canRead(this)();
    OKtoRead.signal();
    //@close NR1W_shared_state(this)();
    mon.unlock();
  }
}
```

Summary

Exclusive access to monitor is essential to ensure consistency of ADT.

More flexible mechanisms can be used in the case of safe interference and aliasing.

Safe sharing (for reading) can be checked with fractional permissions and disciplined with monitors anyway.

Monitors implementing N-readers and 1-Writer can be used to extend sequential ADTs to concurrent ones.

Other, flexible, concurrency control mechanisms are possible, but more challenging. Verification of such mechanisms is an open research area.