Lectures Notes on
Separation Logic

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

18 April, 2023

1 Aliasing and Hoare Logic

As we have discussed in the previous lecture, Hoare logic's basic framework is not generally suited for reasoning about programs with dynamically allocated memory that may be shared (i.e., aliased) across different parts of the program. For instance, if we revisit the Hoare logic rule for assignment:

$$\frac{}{\{A[{}^{E}/_{x}]\}\ x := E\ \{A\}}$$

We can observe that this reasoning principle is only sound if no other variable can alias x. The following Hoare triple is valid

$$\frac{}{\{y \geq 0 \wedge x \geq 0\}\ x := -1\ \{y \geq 0 \wedge x < 0\}}$$

but only in a setting where x and y are not treated as dynamic memory references which may refer to the same underlying memory region. In such setting, assigning to x would mutate y and we would somehow require the Hoare triple to be:

$$\frac{}{\{y \geq 0 \wedge x \geq 0\}\ x := -1\ \{y < 0 \wedge x < 0\}}$$

In the presence of language features that provide abstraction or information hiding (e.g. functions/methods, objects, interfaces, modules), the problem is further aggravated. Object references may be aliased, object fields may be shared across multiple objects, methods may alias memory references, etc.

A language such as Dafny, which does include these features, extends the basic principles of Hoare logic with framing conditions, where each function and method must precisely specify which memory regions it may modify. To ensure soundness, Dafny (and related approaches) are conservative with respect to the treatment of aliasing. In method `M` below:

```
  method M(a:C, b:C)
    modifies a
{
    a.f := 2+b.f;
    assert b.f = old(b.f);
}
```

The assertion fails because Dafny's underlying logic makes no assumption on the memory locations denoted by `a` and `b` beyond the fact that they are (non-null) references to objects of type `C`. In particular, after the assignment `a.f := 2+b.f` we cannot assure that the field `f` of `b` has not been changed because `a` and `b` might refer to the *same* object. For the assertion to hold, we must be explicit in our specification, enforcing that `a` and `b` are not aliases to the same memory region:

```
  method M(a:C, b:C)
   modifies a
   requires a ≠ b
{
    a.f := 2+b.f;
    assert b.f = old(b.f);
}
```

When defining object interfaces in this style, since objects may refer to shared mutable state (e.g. in fields), this means that we must either expose the object's fields in the **`modifies`** clauses of its methods or we have to use a technique such as the dynamic frames approach, where the memory frame of each object is explicit but abstracted in the object's ghost state.

An alternate approach, adopted by the formalism of Separation Logic [Rey02], is the so-called *small footprint* reasoning, which allows us to reason in terms of (implicitly) disjoint or *separate* portions of the heap (i.e. dynamically allocated memory). As we will see, this provides significant streamlining of reasoning about shared mutable state.

2 Separation Logic

Separation logic [Rey02] is an extension of Hoare logic specifically designed to facilitate reasoning about programs that manipulate shared mutable state through pointer manipulation (including information abstraction mechanisms) via reasoning about ownership transfer. The latter point plays a key role, since it avoids the need for explicit framing that arises in related Hoare-logic based approaches.

Assertions in separation logic allow reasoning about shared mutable state by providing a precise description of dynamically allocated memory and providing a general theory for modular reasoning about both concurrent and sequential programs, where different components may potentially *interfere* with one another through shared memory. The two key principles of separation logic are:

Implicit Framing A program need not specify in an explicit way the properties of state that is changed or unchanged by the program (i.e., no reads/modifies clauses).

Small Footprint The precondition to any given program or code fragment describes the components of dynamic memory (i.e. the heap) that the program or code fragment uses.

To achieve this, Separation Logic extends Hoare Logic assertions with two new key connectives, the separating conjunction $A * B$ and the memory access assertion $L \mapsto V$. More generally, the language of separation logic we will use throughout this course is given by the following grammar:

$$\begin{array}{rcll}
A, B & ::= & & \text{Separation Logic Assertions}\\
 & & L \mapsto V & \text{Memory Access}\\
 & | & A * B & \text{Separating Conjunction}\\
 & | & \texttt{emp} & \text{Empty heap}\\
 & | & C & \text{Boolean condition (pure)}\\
 & | & C \,?\, A : A & \text{Conditional}\\
 \\
C & ::= & \multicolumn{2}{l}{B \wedge C \mid C \vee C \mid V = V \mid V \neq V \mid \ldots}\\
V & ::= & \ldots & \text{Pure Expressions}\\
L & ::= & x.\ell \mid [x] & \text{Memory reference}
\end{array}$$

A pure expression, ranged over by the meta-variable V, is any expression that does not depend on memory locations. Separation logic assertions are meant to be assertions on the contents of dynamic memory, as follows: `emp` holds true only of the *empty* heap or memory; the separating conjunction $A * B$ holds iff we can find two *disjoint* memory regions such that one satisfies A and the other satisfies B; the memory access assertion $L \mapsto V$ holds in a state where memory location denotes by L contains the value denoted by V.

The underlying model of separation logic assumes that dynamic memory (the *heap*) is a set of memory locations (L) storing contents V. Memory contents themselves are values of basic type or references (i.e. pointers) to memory locations.

Precise Assertions An assertion A is called *precise* if it uniquely specifies a part of the heap. For instance, the assertion `emp` is precise since it holds exactly of the empty heap. The separating conjunction $A * B$ is precise when both A and B are precise. Dually, the assertion `true` is *not* precise since it holds of any heap; neither are the assertions $\texttt{emp} \vee x \mapsto 10$ or $x \mapsto 10 \vee y \mapsto 10$. Perhaps less obviously, the assertion $\exists x.x \mapsto 10$ is also imprecise. This is the case since $\{\texttt{emp}\}\ x := \textbf{new}\ \mathit{Cell}(10); x := \textbf{null}\ \{\exists x.x \mapsto 10\}$ is valid.

2.1 Proof Rules

The proof rules of separation logic are exactly those of Hoare logic, with the following exceptions.

Assignment Assignment is now expressed easily in terms of the memory access assertion, with the precondition referring exactly to the memory region changed by the assignment operator. The rule is:

$$\frac{}{\{x \mapsto V\}\ x := E\ \{x \mapsto E\}}$$

Memory Access Since the programming language models dynamic memory, we need a reasoning principle for memory lookups:

$$\frac{}{\{L \mapsto V\}\ y := L\ \{L \mapsto V \wedge y = V\}}$$

The rule states that in a memory configuration where the location L contains V, the assignment $y := L$ (note that y is a *stack* variable, not a heap location) results in a state where the contents of L unchanged and where y is V.

Frame Rule Due to the meaning of separating conjunction, the frame rule allows us to preserve all information about "the rest of the world", effectively enabling *local reasoning* about the effects of a program:

$$\frac{\{A\}\ P\ \{B\}}{\{A * C\}\ P\ \{B * C\}}$$

Note that, unlike the frame rule for Hoare logic, we need no side-conditions to this rule. The memory footprint of P is given precisely by assertion A and so we need not specify explicitly what has been modified by P (it must be mentioned in A) nor what is unchanged (it is framed away as C).

3 Separation Logic in Verifast

Verifast is a verification tool for C and Java code based on Separation Logic and a notion of symbolic execution and abstract predicates [VJP15]. We will cover some specific aspects of the Verifast approach in upcoming lectures. For now, lets see how to specify a simple counter-like class in Java:

```
  public class Account {

    int balance;

    /*@
    predicate AccountInv(int b) = this.balance |-> b &*& b >= 0;
    @*/

    public Account()
    //@ requires true;
    //@ ensures AccountInv(0);
    {
      balance = 0;
    }
    ...
}
```

Just as in Dafny, methods specifications are given in terms of pre- and post-conditions, given by requires and ensures clauses in comments which are then processed by the verifier. Unlikely Dafny, however, we do not specify the memory regions modified or read by methods. Instead, we characterize memory by a combination of separation logic assertions and *abstract* predicates.

In the code above, we are defining an *AccountInv* predicate that takes as argument an integer value. The predicate uses separating conjunction (written as `&*&`) as a traditional conjunction to specify that the predicate holds of a memory configuration where the memory location `this.balance` contains the value b and b is non-negative. Note that since $b \geq 0$ is pure, the use of separating conjunction is equivalent to using standard conjunction. In fact, Verifast does not have any other form of conjunction for this precise reason.

Moving to the modifier methods, we have:

```
  public class Account {

    int balance;

    /*@
    predicate AccountInv(int b) = this.balance |-> b &*& b >= 0;
    @*/

    void deposit(int v)
    //@ requires AccountInv(?b) &*& v >= 0;
    //@ ensures AccountInv(b+v);
    {
      balance += v;
    }

    void withdraw(int v)
    //@ requires AccountInv(?b) &*& b >= v;
    //@ ensures AccountInv(b-v);
    {
      balance -= v;
    }
}
```

Note the use of `AccountInv(?b)` in the pre-conditions. This mode of use states that the parameter `b` is an *output* parameter of the predicate, which warrants its use in the post-condition as a way of referring to the old value of the account balance.

3.1 Input and Output Parameters in Predicates

Consider the following code, specifying a Node structure such as that of a linked list.

```
/*@
   predicate Node(Node n; Node nxt, int v) = n.next |-> nxt &*& n.val |-> v;
   predicate List(Node n;) = n == null ? emp : Node(n,?h,_) &*& List(h);
@*/

public class Node {
    Node next;
    int val;

    public Node()
    //@ requires true;
    //@ ensures Node(this,null,0);
    {
        next = null;
        val = 0;
    }
...
}
```

In the definitions of the `List` and `Node` predicates above, the first parameter of the predicate is identified as an *input* parameter, which means that any use of the predicate must provide a concrete Node value. The parameters after the **;** may be used as input or *output*, meaning that we may use them in a similar style to the account balance example of the previous section. One way of thinking about output parameters is as existentially quantified variables.

This warrants the use of the `Node` predicate in the definition of `List`, where we inductively state (in the non-null case) that a part of the heap is a `Node`, whose next field (used as an output parameter) must itself be a `List`. It is important to note that this definition of `List` excludes cycles in the list, since the memory of each cell must be disjoint due to the use of separating conjunction.

3.2 Typestates in Verifast

We can easily model the typestate approach in Verifast via the abstract predicate model:

```
/*@
    predicate StackInv(Stack s;) = s.head |-> ?h &*& List(h);
    predicate NonEmptyStack(Stack s;) = s.head |-> ?h &*& h != null &*& List(h);
@*/
```

```
public class Stack {

    Node head;

    public int pop()
    //@ requires NonEmptyStack(this);
    //@ ensures StackInv(this);
    {
    int v = head.getval();
    head = head.getnext();
    return v;
    }

    public boolean isEmpty()
    //@ requires StackInv(this);
    //@ ensures (result ? StackInv(this):NonEmptyStack(this));
    {
      return head == null;
    }

    public void push(int v)
    //@ requires StackInv(this);
    //@ ensures NonEmptyStack(this);
    {
     Node n = new node();
     n.setval(v);
     n.setnext(head);
     head = n;
    }
}
```

In the Stack example above, we can characterize the abstract (sub)state that represents a non-empty stack. The idea is that abstract substates should logically imply the general representation invariant. In the example above, we have that `NonEmptyStack(s)` implies `StackInv(s)`.

As will become clearer in the next lecture, while the tool can often derive this implication automatically, its automation often requires programmer assistance, in the form of *open* and *close* annotations. The statement `open P(x)` instructs the verifier to expand the definition of $P(x)$, which must be known to hold (in many situations, Verifast maintains predicates abstract and does not replace them with their definitions outright). Dually, the annotation `close P(x)` tries to use the ambient assumptions to establish the validity of $P(x)$.

4 Exercises

1. Install the Verifast tool.
2. Implement a savings account class in Java, maintaining a savings and a checking balance (check last week's exercises for the constraints on the two balances). Verify your implementation in Verifast.
3. Add a static method to your implementation that transfers money between two savings accounts.
4. Implement and specify the Counter class from previous lecture that uses two integer Cell fields.

References

[Rey02] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[VJP15] Frédéric Vogels, Bart Jacobs, and Frank Piessens. Featherweight verifast. *Log. Methods Comput. Sci.*, 11(3), 2015.