Software Verification

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

November 4, 2025

Lecture 9

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

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

Verification of Heap-Manipulating Programs (1/n)

1. Interference and Aliasing
2. Separation Logic
3. Separation Logic in Verifast
4. Case Study – Stack ADT

Interference and Aliasing

Account ADT in Why3

```
type t = { mutable balance: int; }

let create () : t
= { balance = 0 }

let get_balance (t: t) : int
= t.balance

let deposit (amount: int) (t: t) : unit
= t.balance <- t.balance + amount

let withdraw (amount: int) (t: t) : unit
= t.balance <- t.balance - amount

let transfer (target: t) (amount: int) (t: t) : unit
= withdraw amount t;
  deposit amount target
```

Account ADT in Why3

```
type t = { mutable balance: int; }
invariant { balance >= 0 }

let create () : t
= { balance = 0 }

let function get_balance (t: t) : int
= t.balance

let deposit (amount: int) (t: t) : unit
  requires { amount >= 0 }
  ensures  { t.balance = (old t).balance + amount }
= t.balance <- t.balance + amount

let withdraw (amount: int) (t: t) : unit
  requires { 0 <= amount <= t.balance }
  ensures  { t.balance = (old t).balance - amount }
= t.balance <- t.balance - amount

let transfer (target: t) (amount: int) (t: t) : unit
  requires { 0 <= amount <= t.balance }
  ensures  { t.balance = (old t).balance - amount }
  ensures  { target.balance = (old target).balance + amount }
= withdraw amount t;
  deposit amount target

let main ()
= let a = create () in
  deposit 1000 a;
  let ba = get_balance a in
  assert { ba = 1000 } (* Is this provable? *)
```

Account ADT in Why3

```
type t = { mutable balance: int; }
invariant { balance >= 0 }

let create () : t
= { balance = 0 }

let function get_balance (t: t) : int
= t.balance

let deposit (amount: int) (t: t) : unit
  requires { amount >= 0 }
  ensures  { t.balance = (old t).balance + amount }
= t.balance <- t.balance + amount

let withdraw (amount: int) (t: t) : unit
  requires { 0 <= amount <= t.balance }
  ensures  { t.balance = (old t).balance - amount }
= t.balance <- t.balance - amount

let transfer (target: t) (amount: int) (t: t) : unit
  requires { 0 <= amount <= t.balance }
  ensures  { t.balance = (old t).balance - amount }
  ensures  { target.balance = (old target).balance + amount }
= withdraw amount t;
  deposit amount target

let main ()
= let a = create () in
  let b = create () in
  deposit 1000 a;
  let ba = get_balance a in
  assert { ba = 1000 } (* Is this provable? *)
```

Account ADT in Why3

```
type t = { mutable balance: int; }
invariant { balance >= 0 }

let create () : t
= { balance = 0 }

let function get_balance (t: t) : int
= t.balance

let deposit (amount: int) (t: t) : unit
  requires { amount >= 0 }
  ensures  { t.balance = (old t).balance + amount }
= t.balance <- t.balance + amount

let withdraw (amount: int) (t: t) : unit
  requires { 0 <= amount <= t.balance }
  ensures  { t.balance = (old t).balance - amount }
= t.balance <- t.balance - amount

let transfer (target: t) (amount: int) (t: t) : unit
  requires { 0 <= amount <= t.balance }
  ensures  { t.balance = (old t).balance - amount }
  ensures  { target.balance = (old target).balance + amount }
= withdraw amount t;
  deposit amount target

let main ()
= let a = create () in
  let b = create () in
  deposit 1000 a;
  transfer a 500 a;
  let ba = get_balance a in
  assert { ba = 1000 } (* Is this provable? *)
```

Account ADT in Why3

Consider the following code fragment and Hoare triple, intended to transfer `v` from `a1` to `a2`.

```
requires v > 0 && get_balance a1 >= v
{
  let v1 = get_balance a1 in
  if v1 >= v then begin
    withdraw v a1;
    deposit v a2
  end
}
ensures get_balance a1 < old(get_balance a1)
```

Is this Hoare triple valid?

Account ADT in Why3

Consider the following code fragment and Hoare triple, intended to transfer `v` from `a1` to `a2`.

```
requires v > 0 && get_balance a1 >= v
{
  let v1 = get_balance a1 in
  if v1 >= v then begin
    withdraw v a1;
    deposit v a2
  end
}
ensures get_balance a1 < old(get_balance a1)
```

Only if **a1** and **a2** refer to different `Account` objects!

If they are aliases, the Hoare triple is invalid.

Account ADT in Why3

Tracking aliasing is challenging, e.g.,

```
let safe_transfer (v: int) (a1 a2: t) : unit
= let v1 = get_balance a1 in
  if v1 >= v then begin
    withdraw v a1;
    deposit v a2
  end

let main ()
= let b1 = create () in
  let b2 = create () in
  deposit 10 b1;
  safe_transfer 2 b1 b2;
  safe_transfer 2 b1 b1
```

Evaluation of `if` body depends on whether `a1` and `a2` are aliases.

Aliasing and Interference

Two programming language expressions are aliases if they refer to the same memory location.

Aliasing occurs in any programming language with pointers (e.g., C, C++) or references (Java, C#, OCaml)

Two program fragments interfere if the execution of one may change the effect of the other.

Interference is particularly important in the context of concurrent programs.

Interference and aliasing is hard to detect syntactically, and hard to reason about semantically.

Recall the ASSIGNMENT rule from Hoare Logic:

$$\frac{}{\{P[x \mapsto t]\}\, x := t\, \{P\}}$$

The soundness of this reasoning principle is rooted on the fact that no other variable aliases x. We have:

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

But, on the presence of aliasing:

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

In our Account ADT example:

```
requires x.balance = K && y.balance > 0
                withdraw K x
ensures x.balance = 0 && y.balance > 0
```

If `x` and `y` are aliases, we would have, e.g.:

```
requires x.balance = K && y.balance > 0
                withdraw K x
ensures x.balance = 0 && y.balance = 0
```

To reason about programs with interference (aliasing or concurrency) a different approach is needed.

Aliasing in Why3 & Others

Why3 imposes that all mutable values are disjoint in memory.

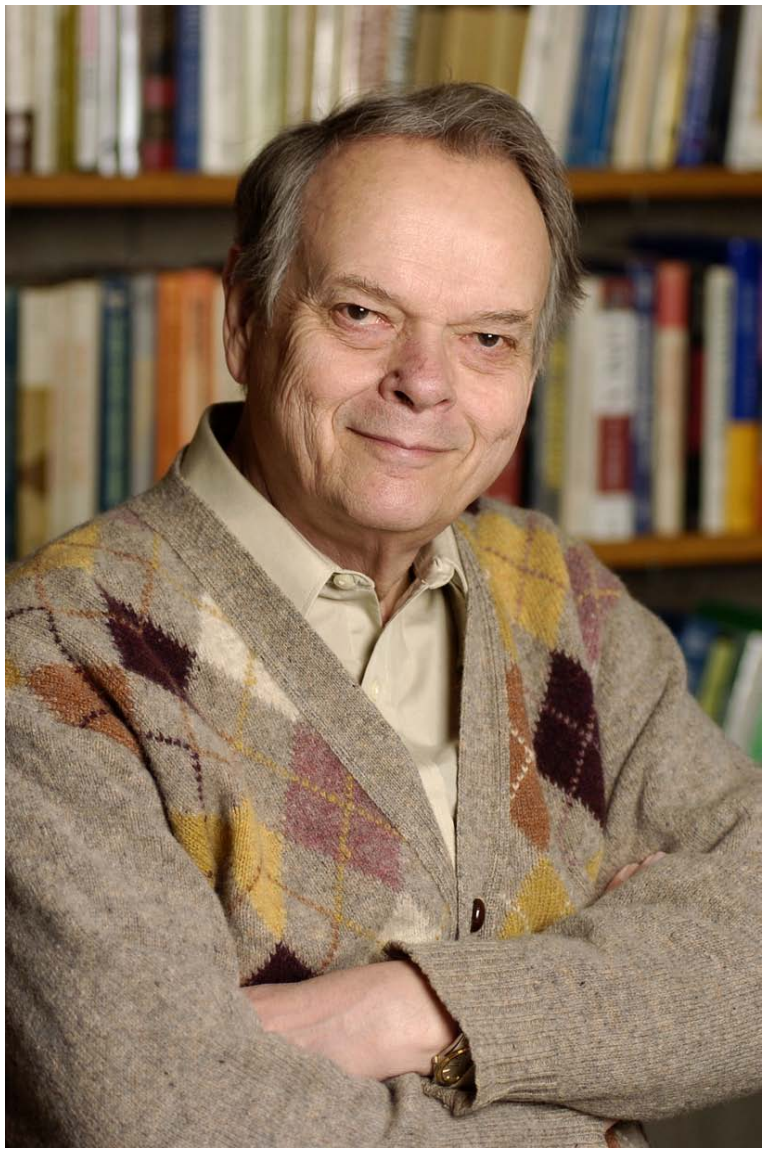
```
let foo (a b: C)
  (* a and b are implicitly not aliased *)
= a.f <- 2 + b.f;
  assert { b.f = (old b).f }

let bar (x: C)
  (* This application creates an illegal alias *)
= foo x x
```

Framing conditions (i.e., the **reads** and **writes** effects) help to manage knowledge about objects.

Other approaches include dynamic frames (Dafny), ownership hierarchy (Spec#), Ownership and data groups (JML), and **separation logic** (up next) which uses small footprint reasoning.

Separation Logic

Separation Logic – How it all began



John Reynolds

Peter O'Hearn

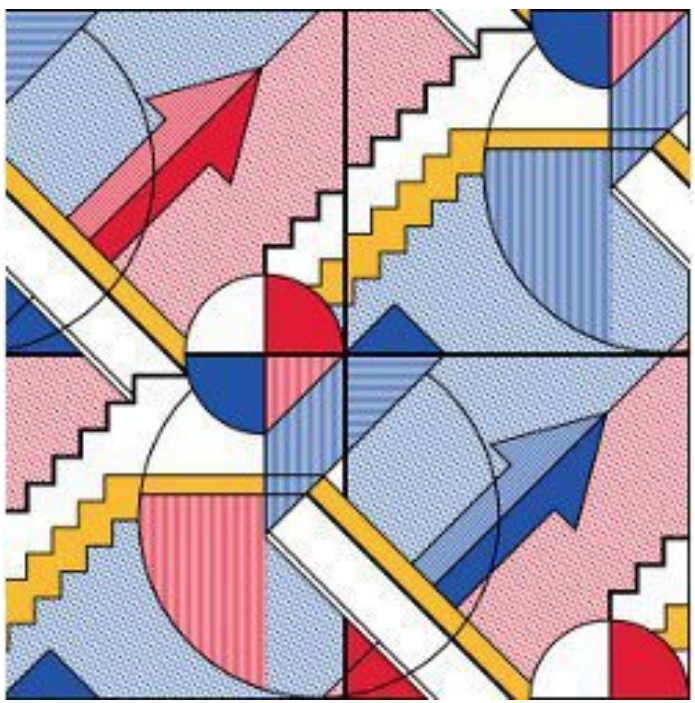
Separation Logic – Mandatory Reading!

DOI:10.1145/3211968

Separation logic is a key development in formal reasoning about programs, opening up new lines of attack on longstanding problems.

BY PETER O'HEARN

Separation Logic



Peter O'Hearn. 2019. **Separation logic**. Commun. ACM 62, 2 (February 2019), 86–95. https://doi.org/10.1145/3211968

Separation Logic – Mandatory Reading!

Separation Logic for Sequential Programs (Functional Pearl)

ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France

This paper presents a simple mechanized formalization of Separation Logic for sequential programs. This formalization is aimed for teaching the ideas of Separation Logic, including its soundness proof and its recent enhancements. The formalization serves as support for a course that follows the style of the successful *Software Foundations* series, with all the statement and proofs formalized in Coq. This course only assumes basic knowledge of λ-calculus, semantics and logics, and therefore should be accessible to a broad audience.

Arthur Charguéraud. 2020. **Separation logic for sequential programs (functional pearl)**. Proc. ACM Program. Lang. 4, ICFP, Article 116 (August 2020). `https://doi.org/10.1145/3408998`

Separation Logic

Extension of classic Hoare Logic.
An axiomatisation of language operations on pointers.

Allowing the verification of access of unrelated parts of the code to the shared memory locations.

Deals with aliasing by providing a precise shape of the dynamically allocated memory.

Provides a basis for a general theory for modular reasoning about concurrent and sequential programs.

Separation logic is based in two key principles:

(1) **Small footprint**
The precondition of a code fragment describes the part of the memory (heap) that the fragment needs to use.

(2) **Implicit framing**
No need to explicitly specify the properties of state that is changed / not changed by the program (modifies)

It adds to Hoare Logic two key novel primitives

- the separating conjunction operator $H_1 \star H_2$
- the points-to memory access assertion $L \rightsquigarrow V$

A heap predicate, written H, is a predicate over the mutable state

- `Heap` denotes the type of states
- `Prop` denotes the type of logical propositions
- a heap predicate H has type `Heap` $\rightarrow$ `Prop`

In the following Hoare Triple

$$\{H\}\, t\, \{H'\}$$

H and H' describe the whole input and output states.

In contrast, Separation Logic allows specifying only the fragment of the state that is relevant to the execution of the program.

We will use the following grammar for Separation Logic assertions:

A	$::=$		Separation Logic Assertions
		$L \rightsquigarrow V$	Points-to
	$\mid$	$H_1 \star H_2$	Separating cojunction
	$\mid$	$\texttt{emp}$	Empty heap
	$\mid$	B	Boolean condition (pure predicate)
	$\mid$	$B \ ? \ A \ : \ A$	Conditional

$$B ::= B \wedge B \mid B \vee B \mid V = V \mid V \neq V \mid \ldots$$

V	$::=$	$\ldots$	Pure expressions
L	$::=$	$x.\ell$	Object field

Points-to assertion

$$L \rightsquigarrow V$$

Assertion $L \rightsquigarrow V$ holds for the singleton memory where location L holds value V.

Empty assertion

`emp`

Assertion `emp` holds for the empty heap.

Separating conjunction

$$H_1 \star H_2$$

Assertion $H_1 \star H_2$ holds for the heap that can be disjointly decomposed into a sub-heap that satisfies H_1 and another sub-heap that satisfies H_2.

Digression: the Stack and the Heap

Stack

- Stores local variables and method parameters.
- The so-called call-stack, also stores return addresses.
- Memory recover discipline: LIFO, release on block exit.

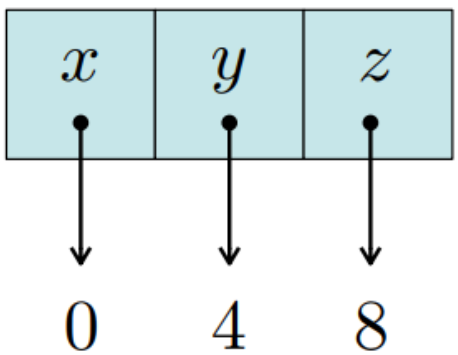
Heap

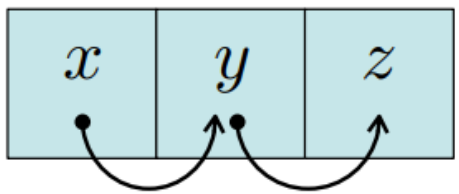
- Stores dynamically allocated objects (e.g. `new` / `malloc`)
- Explicit release (`free` in C/C++) or garbage collection (Java or OCaml).

Abstract view of the heap

- A sequence of memory locations (L) with their contents (V).
- Memory contents are values:
  - of basic types (*e.g.*, integers)
  - or pointers to other memory locations

Separation Logic – in a Picture





$\{x \mapsto 0 * y \mapsto 4 * z \mapsto 8\}$ $\{x \mapsto y * y \mapsto z * z \mapsto \ldots\}$

Rules of Separation Logic – Assignment

Recall the Hoare Logic assignment rule:

$$\frac{}{\mathsf{Hoare}\,\{P[x \mapsto t]\}\, x := t\, \{P\}}$$

The assignment rule in Separation Logic is

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

Note the small footprint principle, the precondition refers exactly to the part of the memory used by the fragment.

Rules of Separation Logic – Frame Rule

A key principle in SL is the Frame Rule

$$\frac{\mathsf{SL}\{H\}\,t\,\{H'\}}{\mathsf{SL}\{H \star H''\}\,t\,\{H' \star H''\}}\ \text{FRAME}$$

This frame rule allows us to preserve info about the "rest of the world", and locally reason about the effects of a program that only manipulates a given piece of the state.

The given piece footprint is specified by precondition H.

There is no need to specify what is `modified`

- it is clear from the pre-condition H

neither what is `not modified`

- it is framed away, as H'' above

Rules of Separation Logic – Lookup Rule

The lookup rule in Separation Logic is

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

Here, $y = V$ is a pure predicate.

Also, y is a stack variable, not a heap location

Separation Logic in Verifast

Verifast

VeriFast is a verifier for singlethreaded and multithreaded C and Java programs annotated with preconditions and postconditions written in separation logic.

Jacobs, Smans, Piessens, 2010-14

```
public void broadcast_message(String message) throws IOException
    //@ requires room(this) &*& message != null;
    //@ ensures room(this);
{
    //@ open room(this);
    //@ assert foreach(?members0, _);
    List membersList = this.members;
    Iterator iter = membersList.iterator();
    boolean hasNext = iter.hasNext();
    //@ length_nonnegative(members0);
    while (hasNext)
        /*@
        invariant
            foreach<Member>(?members, @member) &*& iter(iter, membersList, members, ?i)
            &*& hasNext == (i < length(members)) &*& 0 <= i &*& i <= length(members);
        @*/
    {
        Object o = iter.next();
        Member member = (Member)o;
        //@ mem_nth(i, members);
        //@ foreach_remove<Member>(member, members);
        //@ open member(member);
        Writer writer = member.writer;
        writer.write(message);
        writer.write("\r\n");
        writer.flush();
        //@ close member(member);
        //@ foreach_unremove<Member>(member, members);
        hasNext = iter.hasNext();
    }
    //@ iter_dispose(iter);
    //@ close room(this);
}
```

Where to get it: `https://github.com/verifast/verifast`

What to read: `https://people.cs.kuleuven.be/~bart.jacobs/verifast/`

Account ADT (Java + Verifast)

```
public class Account {
  int balance;
/*@
predicate AccountInv(int b) = this.balance |-> b &*& b >= 0;
@*/

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

Account ADT (Java + Verifast)

```
public class Account {
  int balance;

  ...

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

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

Account ADT (Java + Verifast)

```
public class Account {
  int balance;

  ...

  int getBalance()
  //@ requires AccountInv(?b);
  //@ ensures AccountInv(b) &*& result==b ;
  {
    return balance;
  }
}
```

Case Study

Stack ADT

Stack ADT, Using a Linked List

Separation logic allows us to precisely define the shape and memory layout (footprint) of arbitrary heap allocated data structures.

The framing principles allow us to easily express the functionality of heap manipulating operations using separation logic pre- and post-conditions.

Let's look to a simple example, a stack ADT implemented using a linked list.

Stack ADT, Using 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;
    }
```

Stack ADT, Using a Linked List

The predicate `Node(node n; node nxt, int v)` represents a single node object in the heap.

Note: In Verifast, the `";"` separates the input parameters, from I/O parameters, which may be queried using `"?"`.

The predicate `List(node n;)` represents the shape and layout of properly formed linked list in the heap.

Note: `List(node n;)` is recursively defined. A list is either empty (no memory) or contains an allocated head node linked to a possibly empty list.

Our definition forbids cycles in the list.

Stack ADT, Using 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 void setnext(Node n)
    //@ requires Node(this, ?nn, ?vv);
    //@ ensures Node(this, n, vv);
    {
        next = n;
    }
    public void setval(int v)
    //@ requires Node(this, ?nn, ?vv);
    //@ ensures Node(this, nn, v);
    {
        val = v;
    }
```

Stack ADT, Using 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 getnext()
    //@ requires Node(this, ?nn, ?vv);
    //@ ensures Node(this, nn, vv) &*& result == nn;
    {
        return next;
    }
    public int getval()
    //@ requires Node(this, ?nn, ?vv);
    //@ ensures Node(this, nn, vv) &*& result == vv;
    {
        return val;
    }
```

Stack ADT, Using a Linked List

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

public class Stack {
    Node head;

    public Stack()
    //@ requires true;
    //@ ensures StackInv(this);
    {
        head = null;
    }
```

Stack ADT, Using a Linked List

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

public class Stack {
    Node head;

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

Stack ADT, Using a Linked List

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

public class Stack {
    Node head;

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

Verifast does not check this code. **Why**?

Stack ADT, Using a Linked List

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

public class Stack {
    Node head;

    public int pop()
    //@ requires StackInv(this);
    //@ ensures StackInv(this);
    {
        if(head!=null) {
            int v = head.getval();
            head = head.getnext();
            return v;
        }
        return -1; // ??? how to deal with partiality ?
    }
```

Stack ADT, Using a Linked List

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

class StackEmptyE extends Exception {}

public class Stack {
    Node head;

    public int pop_maybe()
        throws StackEmptyE //@ ensures StackInv(this);
    //@ requires StackInv(this);
    //@ ensures StackInv(this);
    {
        if(head!=null) {
            int v = head.getval();
            head = head.getnext();
            return v;
        } else throw new StackEmptyE();
    }
```

Exceptions in Method Contracts

Java method contracts need to deal with exceptional behaviour, so Verifast allows postconditions to be specified at every exception thrown.

In the previous example, the `pop_maybe` method is partial and throws StackEmptyE if the stack is empty.

Instead of making a method partial, we may try to define an appropriate pre-condition.

To make `pop` total we may add a special pre-condition expressing the typestate "non-empty-stack".

This will require client code to be able to dynamically check the ADT state, *e.g.*, via an `isEmpty` method.

Stack ADT, Using a Linked List

```
/*@
  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;
    }
```

Stack ADT, Using a Linked List

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

Stack ADT, Using a Linked List

```
/*@
  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 void push2(int v)
    //@ requires StackInv(this);
    //@ ensures NonEmptyStack(this);
    {
        Node n = new Node();
        n.setval(v);
        n.setnext(head);
        head = n;
    }
```

Abstract States and Substates

In general, it is very useful to introduce abstract substates representing particular cases of the general representation invariant of an ADT.

All abstract substates logically imply the most general representation invariant, for instance, in our examples, for every `Stack s`, the following implication is valid:

$$\texttt{NonEmptyStack(this)} \implies \texttt{StackInv(this)}$$

Often these implications are not automatically derived by `Verifast`, but the programmer may give an `hint`, using `open` and `close` annotations.

Stack ADT, Using a Linked List

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

public class Stack {
    static void main()
    //@ requires true;
    //@ ensures true;
    {
        Stack s = new Stack();
        s.push(1);
        if (! s.isEmpty()) {
            //@ open(NonEmptyStack(s));
            s.pop();
        }
        s.push(2);
        s.push(3);
        s.pop();
    }
```