VeriFast

Software Fiável

Mestrado em Engenharia Informática
Mestrado em Informática
Faculdade de Ciências da Universidade de Lisboa

2019/2020

Vasco T. Vasconcelos

VeriFast

- VeriFast is a program verifier for Java
- Uses the design-by-contract approach to modular verification
- VeriFast is based on separation logic, an extension of the Hoare logic

Verification

- Java methods are annotated with pre and postconditions and other specifications describing assumptions made by the developer
- VeriFast checks whether the assumptions hold in each execution of the program for arbitrary input
- If VeriFast deems a Java program to be correct, then that program
  - does not contain assertion violations
  - data races
  - divisions by zero
  - null dereferences
  - array indexing errors
  - and the program makes correct use of the Java API

Contracts

- Conventional Java code is not analysed by VeriFast
- In order to call VeriFast's attention add a contract to a method:

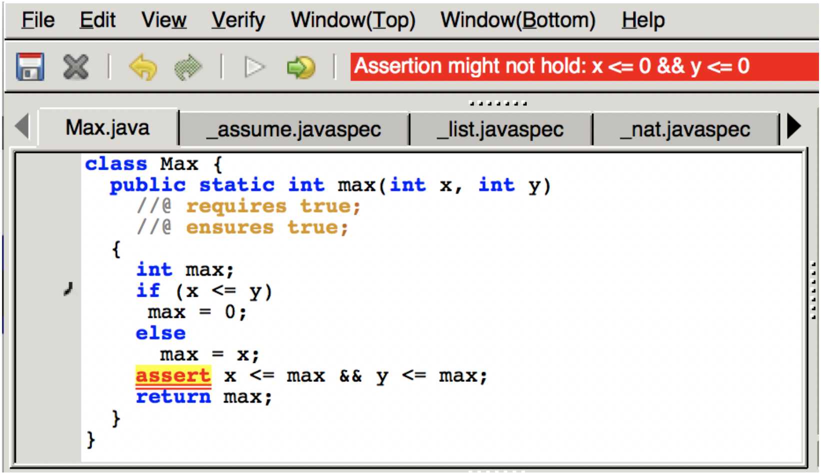
```
static int toInt (Integer i)
  //@ requires true;
  //@ ensures true;
{
  return i.intValue();
}
```

- A contract for a method is a pair `requires`/`ensures` placed in line comments, `//@`, or block comments, `/*@ ... @*/`

The assert statement

- A Java assert statement consists of the keyword `assert` followed by a boolean expression
- By inserting an `assert` statement in the code, a developer indicates that she expects the corresponding boolean expression to evaluate to true whenever the statement is reached during the program's execution
- If the expression evaluates to false, an `AssertionError` is thrown (provided assertion checking is enabled)
- VeriFast, on the other hand, checks `assert` statements *without* evaluating any code

Max


File Edit View Verify Window(Top) Window(Bottom) Help
Assertion might not hold: x <= 0 && y <= 0
Max.java
_assume.javaspec
_list.javaspec
_nat.javaspec
class Max {
public static int max(int x, int y)
//@ requires true;
//@ ensures true;
{
int max;
if (x <= y)
max = 0;
else
max = x;
assert x <= max && y <= max;
return max;
}
}

Method contracts

- VeriFast performs modular verification: each method call is verified with respect to the callee's signature
- The current contract of method `max`, namely

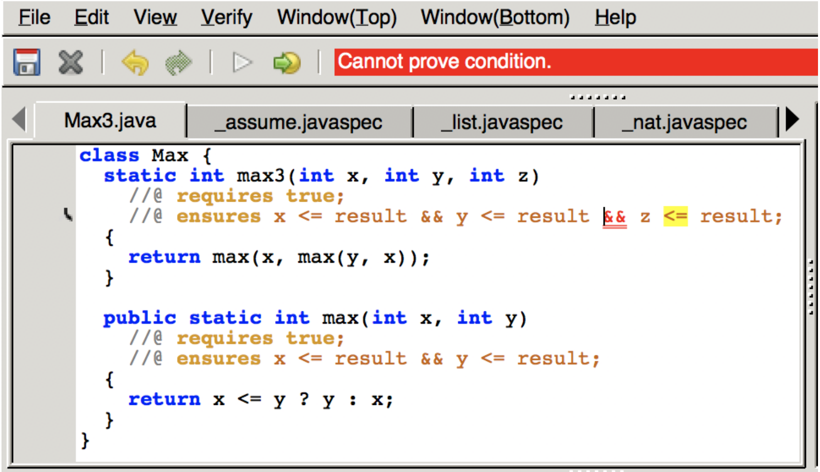
  ```
  //@ requires true;
  //@ ensures true;
  ```

  tells very little about the behaviour of the method
- Instead we "move" the `assert` expression to the post condition,

  ```
  //@ ensures x <= result && y <= result
  ```

  so that we may use the contract in another method
- Notice the *ghost variable* `result` used to denote the value of the method

Max3


File Edit View Verify Window(Top) Window(Bottom) Help
Cannot prove condition.
Max3.java
_assume.javaspec
_list.javaspec
_nat.javaspec
class Max {
static int max3(int x, int y, int z)
//@ requires true;
//@ ensures x <= result && y <= result && z <= result;
{
return max(x, max(y, x));
}
public static int max(int x, int y)
//@ requires true;
//@ ensures x <= result && y <= result;
{
return x <= y ? y : x;
}
}

Partial correctness

- A method body *satisfies a contract* if for each program state *s* that satisfies the precondition, execution of the method body starting in *s* does not trigger illegal operations (such as assertion violations and divisions by zero) and the postcondition holds when the method terminates
- VeriFast *only checks partial correctness* so methods are not required to terminate

Ciências ULisboa | Informática

Symbolic execution

- VeriFast uses *symbolic* rather than concrete execution
- It constructs a symbolic state that represents an arbitrary concrete pre-state which satisfies the precondition
- Checks that the body satisfies the contract for this symbolic state
- Symbolically executes the body starting in the initial symbolic state
- At each statement encountered during symbolic execution, checks that the statement cannot go wrong and updates the symbolic state to reflect execution of that statement
- Finally, when the method returns, VeriFast checks that the postcondition holds for *all* resulting symbolic states

Symbolic state

- A symbolic state is a triple composed of
  - A *symbolic store* (right frame on the IDE)
  - A path condition, or *assumptions* (bottom-centre frame on the IDE)
  - A *symbolic heap* (right-centre frame on the IDE)
- Each *symbolic value* is a first-order term, i.e., a symbol, or a literal number, or an operator ($+$, -, $<$, $=$, $\ldots$) or a function applied to first-order terms.
- The path condition is a set of first-order formulas describing the conditions that hold on the path being verified
- The symbolic heap is a multi-set of heap chunks (more later)

Assertions

- An assertion is a side-effect free, heap-independent Java boolean expression (extensions to be introduced later)
- *Consuming an assertion*—`ensures`, `assert`—means symbolically evaluating the expression yielding a first-order formula and checking that the formula is derivable from the path condition
- VeriFast relies on an SMT solver, a kind of automatic theorem prover, to discharge such proof obligations
- *Producing an assertion*—`requires`, `assume`—corresponds to evaluating that expression yielding a first-order formula and adding it to the path condition

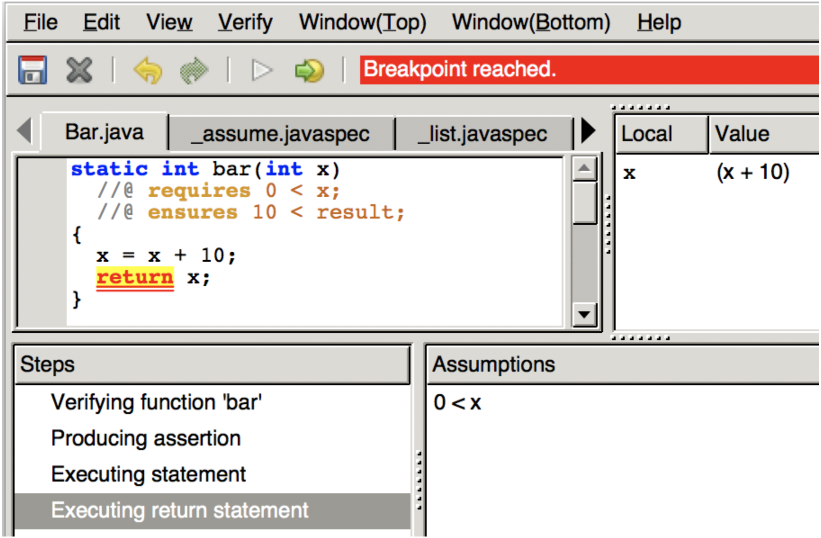
The pre-state

- Symbolic execution of each method starts by initializing the symbolic store by assigning a fresh first-order symbol to each parameter
- VeriFast selects the symbol x as the fresh term representing the symbolic value of a parameter `x`
- The resulting symbolic state thus represents an arbitrary concrete pre-state
- Notice that the symbol and its symbolic value are rendered in different fonts: x and `x`

Assignment

- Disable overflow warnings by unchecking Check arithmetic overflow in the Verify menu
- Place the cursor in `return` statement
- Press the "Run to cursor" button
- Check the Symbolic Store frame (top left) and the Assumptions frame (bottom centre)
- The postcondition holds as the corresponding first-order formula, $10 < x + 10$, is derivable from the path condition, $0 < x$, by the SMT solver

Assignment example



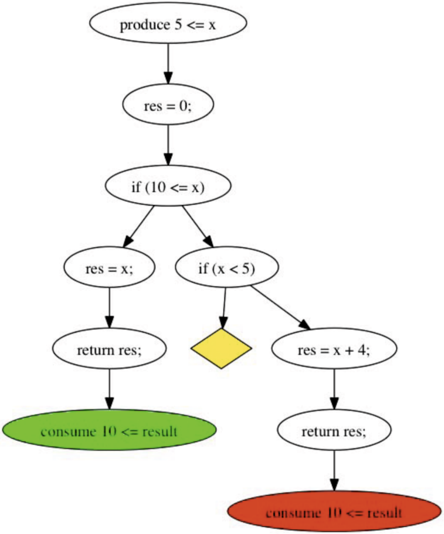
Conditional

- Symbolic evaluation of the condition of an if statement results in a first-order formula
- Based on this formula, it is generally not possible to decide which branch must be taken
- Given a statement `if (x<10)S1; else S2;`, statement `S1` is verified under the assumption that $10 \leq x$, while `S2` is verified assuming the negation of the condition, $10 > x$

Conditional example

```
static int foo(int x)
  //@ requires 5 <= x;
  //@ ensures 10 <= result;
{
  int res = 0;
  if (10 <= x)
    res = x;
  else if (x < 5)
    assert false;
  else
    res = x + 4;
  return res;
}
```

Symbolic execution tree



Inconsistent assumptions

- The diamond node represents a symbolic state with an inconsistent path condition
- Such states are not reachable during concrete executions of the program
- VeriFast does not examine infeasible paths any further
- The formula representing the postcondition, $10 \leq x + 4$, is not derivable from the path condition $5 \leq x$, $10 > x$ and $x \geq 5$
- VeriFast does not report all problems on all paths; it stops when it finds the first error or when all paths successfully verify

Method calls

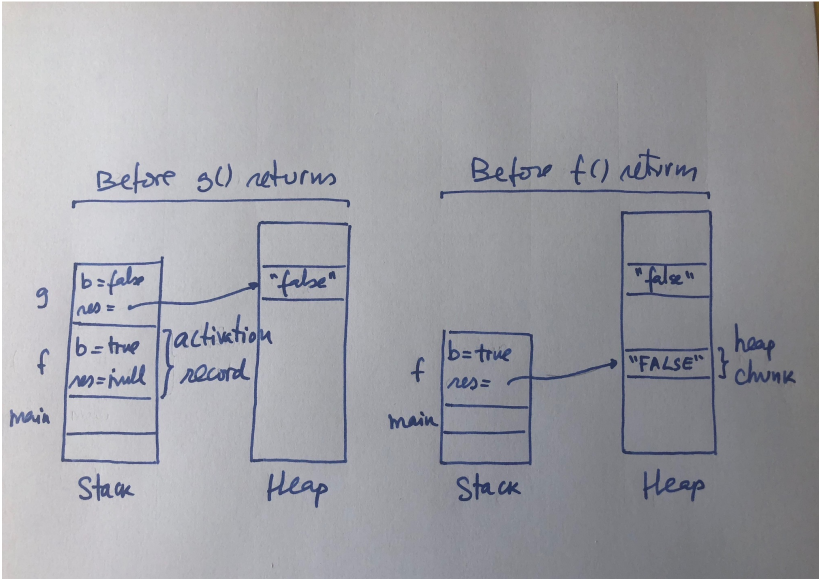
- The symbolic execution of a call consists of two steps:
  1. Consumption of the callee's precondition and
  2. Production of its postcondition
- Both steps are executed under the callee's symbolic store
- During production of the postcondition, the callee's return value is represented by the ghost variable `result`

The memory layout of a Java program

- Before we address classes and objects we need to understand the memory layout of a Java program

```
class MemoryLayout {
  static String g (boolean b) {
      return Boolean.toString(!b);
  }
  static String f (boolean b) {
    return g(b).toUpperCase();
  }
  public static void main (String[] args) {
    System.out.println(f(true));
  }
}
```

The memory layout of a Java program



Aliasing and heap chunks

- Modular verification in the presence of aliasing is challenging
- VeriFast applies an *ownership* regime
- VeriFast tracks during symbolic execution what part of the program state is owned by the method
- The symbolic heap is a *multiset of heap chunks*
- Each heap chunk represents a memory region that is owned by a method
- The chunk can contain information on the *state* of that memory region. For example, the heap chunk `C_f(o,v)` represents
  - exclusive ownership of the field `C.f` of object `o` and
  - the property that the field's current value is v

Heap chunks in a symbolic heap

- All heap chunks in the symbolic heap represent mutually disjoint memory regions
- If the heap contains two field chunks $C\_f(o_1, v)$ and $C\_f(o_2, w)$, then o_1 and o_2 are distinct
- As chunks on the symbolic heap do not share hidden dependencies, the verifier can safely assume that an operation that only affects a particular chunk does not invalidate the information in the remaining chunks
- **Invariant:** any time each heap location is exclusively owned by at most one activation record
- A method is only allowed to access a heap location if it owns that location

Acquiring and releasing ownership

- Acquiring
  1. Constructor
  2. Precondition
  3. Call
- Releasing:
  1. Call
  2. Return

Aquiring ownership via construction

```
class Account {
  private int balance;
  public Account()
    //@ requires true;
    //@ ensures balance ↦ 0;
  {
    super();
    balance = 0;
  }...}
```

- A constructor gains ownership of the fields of the new object right after calling the superclass constructor
- The constructor of the class `Account` is allowed to initialize balance to zero

Ciências ULisboa | Informática

Aquiring ownership via precondition

```
public void deposit(int amount)
  //@ requires balance ↦ ?b;
  //@ ensures balance ↦ b + amount;
{
  this.balance += amount;
}
```

- Ownership of the field `f` of an object `e1` with value `e2` is denoted as `e1.f ↦ e2`. Read "`e1.f` points to `e2`"
- The method body of method `deposit` is allowed to read and write `this.balance`
- Note the `?b` to introduce a new ghost variable `b`. This is typical of the precondition; the variable can then be used in the postcondition

Aquiring ownership via call

```
1 public Account copy()
2   //@ requires balance ↦ ?b;
3   //@ ensures balance ↦ b &*& result != null
      &*& result.balance ↦ b;
4 {
5   Account copy = new Account();
6   copy.balance = balance;
7   return copy;
8 }
```

- A new `Account` is created in line 5. Assignment to `balance` (line 6) is allowed because the constructor's postcondition includes `this.balance ↦...`
- The postcondition of the constructor specifies that ownership of field `balance` of the new object is transferred to its caller (`copy()` in this case) when the constructor terminates

Releasing ownership via call

```
1  public void release()
2    //@ requires this.balance ↦ _;
3    //@ ensures true; // balance not released
4  {}
5  public void m()
6    //@ requires this.balance ↦ _;
7    //@ ensures true;
8  {
9    release();
10   this.balance += 0; // Error _ No matching heap
         chunks: Account_balance(this, _)
11 }
```

- At each method call, ownership of the memory locations described by the callee's precondition is transferred from the caller to the callee
- Ownership of `balance` is lost in line 9

Releasing ownership via return

```
public void deposit(int amount)
  //@ requires balance ↦ ?b;
  //@ ensures balance ↦ b + amount;
{ this.balance += amount; }
public void m ()
  //@ requires this.balance ↦ _;
  //@ ensures true;
{
  deposit(100);
  this.balance += 0; // OK: there is a matching
     h. chunk released by deposit() in its post
}
```

- When a method returns, the method loses ownership of all memory locations enumerated in its postcondition
- Ownership of those locations is transferred from the method to its caller when it returns

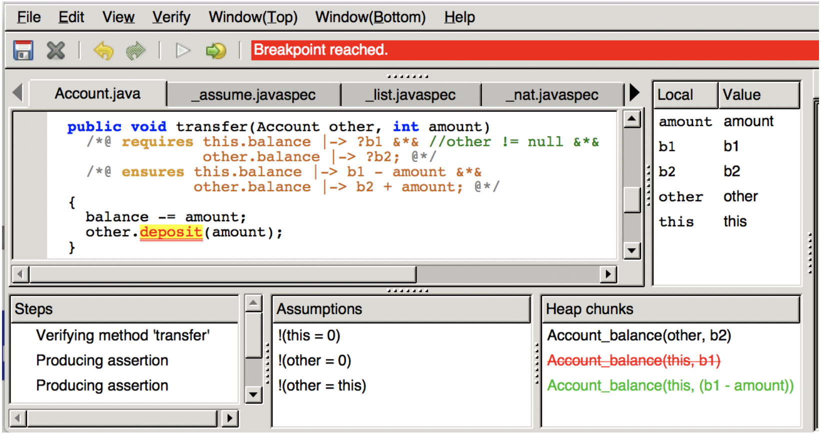
Spatial assertions

- *Pure assertions* such as `0 <= x` specify constraints on local variables
- *Spatial assertions* such as `o.f` $\mapsto$ `v` denote ownership of a heap subregion and information about that region
- *Production of a spatial assertion* corresponds to the acquisition of ownership by the current activation record of the memory regions described by the assertion
- *Consumption of a spatial assertion* corresponds to the current activation record relinquishing ownership of the memory regions described by the assertion

Separating conjunction

- Multiple atomic assertions can be conjoined via the separating conjunction, denoted `&*&`
- Semantically, `A &*& B` holds if both `A` and `B` hold and `A`'s footprint is disjoint from `B`'s footprint
- The footprint of an assertion is the set of memory locations for which that assertion claims ownership
- Consuming (respectively producing) `A &*& B` is implemented by first consuming (respectively producing) `A` and afterwards `B`
- Note that if `A` is a pure assertion, then `A &*& A` is equivalent to `A`. However, this property does not necessarily hold for spatial assertions

Ownership transfer



Ownership transfer

- Place the cursor in the method's closing brace; press "Run to cursor"
- Click on a step (lower-left box) and observe the "Heap chunks" frame; use the up/down key to see the verification progress

One heap chunk per field

- Different references may hold different parts of an object
- Consider the following excerpt of class `FieldSeparation`

```
int a;
boolean b;
int getA ()
  //@ requires a ↦ _;
  //@ ensures true; // do not release a
{ return a; }
boolean getB ()
  //@ requires b ↦ _;
  //@ ensures true; // do not release b
{ return b; }
```

Different references may hold different parts of an object

```
FieldSeparation f = new FieldSeparation();
f.getA(); f.getB(); // OK

FieldSeparation f1 = new FieldSeparation();
FieldSeparation f2 = f1;
f1.getA(); f2.getB(); // OK

FieldSeparation f3 = new FieldSeparation();
f3.getA(); f3.getA(); // No mathing heap chunks:
    FieldSeparation_a(f3, _)

FieldSeparation f4 = new FieldSeparation();
FieldSeparation f5 = f4;
f4.getA(); f5.getA(); // No mathing heap chunks:
    FieldSeparation_a(f4, _)
```

Data abstraction

- The contract for class `Account` is built around the value of field `balance`
- But `balance` is a `private` field, hence should not appear in contracts
- If we choose a different representation for the class (e.g. a list of transactions), we would have to
  - Update the method contracts and consequently
  - Have to reconsider the correctness of all clients

Predicates

- How can we specify the observable behaviour of a class or interface without exposing its internal representation?
- VeriFast's answer to this question is **predicates**
- Assertions describing the state associated with instances of a class can be hidden inside predicates
- A predicate is a named, parameterised, assertion

```
//@ predicate account(int b) = this.balance ↦ b
    &*& b >= 0;
```

- We now use predicate `account` in all method contracts
- This predicate is defined **within** class `Account`

Class Account with predicate account()

```
class Account {
  private int balance;
  //@ predicate account(int b) = this.balance ↦
      b &*& b >= 0;
  public Account()
    //@ requires true;
    //@ ensures account(0);
  {}
  public void deposit(int amount)
    //@ requires account(?b);
    //@ ensures account(b + amount);
  { this.balance += amount; }
}
```

- Contracts are now written in terms of predicate `account()` and not field `balance`

Static predicates

- Predicates that talk about possibly `null` references cannot be declared inside a class (`this != null`)
- In this case we declare the predicate **outside** the class and pass the object as parameter

```
//@ predicate account(Account a, int b) = a.
    balance ↦ b &*& b >= 0;
```

- Calls to this predicate require a new parameter, typically `this`:

```
public void deposit(int amount)
  //@ requires account(this, ?b);
  //@ ensures account(this, b + amount);
```

Predicates are class members

- A call to a member `account()` abbreviates `this.account()`
- To call the other predicate on an object `o` use `o.account()`

```
public void transfer(Account target, int amount)
  /*@ requires amount >= 0 &*&
        this.account(?b1) &*& amount <= b1 &*&
        target != null &*& target.account(?b2);
            @*/
  /*@ ensures this.account(b1 - amount) &*&
        target.account(b2 + amount); @*/
```

Folding and unfolding predicates

- VeriFast by default does not automatically fold and unfold predicates
- Developers must explicitly use ghost statements to switch between the external, abstract view offered by the predicate and the internal definition of the predicate

Folding (closing) a predicate

- The `close` ghost statement *folds* a predicate: it consumes the body of the predicate, and afterwards adds a chunk representing the predicate to the symbolic heap (see method `Account`)
- Without the ghost statement, the constructor does not verify as the heap does not contain a chunk that matches the postcondition

```
public Account(int initialBalance)
  //@ requires initialBalance >= 0;
  //@ ensures account(this, initialBalance);
{
  balance = initialBalance;
  //@ close account(this, initialBalance);
}
```

Unfolding (opening) a predicate

- The `open` ghost statement unfolds a predicate: it removes a heap chunk that represents the predicate from the symbolic heap and produces its body
- As the necessary chunk is nested inside `account(this, ?b)`, the predicate must opened first
- If we omit the ghost statement, VeriFast would no longer find a chunk that matches the field assertion `Account_balance(this, _)` on the heap and report an error

```
public void deposit(int amount)
  //@ requires account(this, ?b);
  //@ ensures account(this, b + amount);
{
  //@ open account(this, b);
  this.balance += amount;
  //@ close account(this, b + amount);
}
```

Precise predicates

- Inserting `open` and `close` ghost statements is tedious
- To alleviate this burden, programmers can mark certain predicates as **precise**
- VeriFast automatically opens and closes precise predicates (in many cases) whenever necessary during symbolic execution
- A predicate can be marked as precise by using a semicolon instead of comma somewhere in the parameter list

```
//@ predicate account(Account a; int b) = a.
    balance ↦ b;
```

- The semicolon separates the input from the output parameters: `a` is input; `b` is output

Account with precise predicates

- No `open` or `close` ghost statements required
- Nevertheless, if possible declare predicates inside the class

```
class Account {
  private int balance;
  public Account()
    //@ requires true;
    //@ ensures account(this, 0);
  {}
  public void deposit(int amount)
    //@ requires account(this, ?b);
    //@ ensures account(this, b + amount);
  { this.balance += amount; }
}
```

An alternative implementation for class Account

- Now that we have the contract for class `Account` defined on top of a predicate, we can easily change the implementation without touching the contract
- The implementation:
  - Stores the deposit/withdraw transactions on a linked list,
  - Extracts the balance from the list of transactions, rather than storing it explicitly on a field, and
  - Introduces a new definition for `predicate account`

Class Transaction

```
class Transaction {
  final Transaction next;
  final int amount;
  public Transaction(int a, Transaction t)
    //@ requires true;
    //@ ensures amount ↦ a &*& next ↦ t;
  { amount = a; next = t; }
}
```

- Class `Transaction` implements a linked list of integer values

Account with a list of transactions

```
class Account {
  private Transaction transactions;
  public Account()
    //@ requires true;
    //@ ensures account(0);
  {}
  ...
}
```

- The contract for the constructor (and other methods) remains unchanged

The new predicate for the class

```
/*@
predicate account(int b) =
  this.transactions ↦ ?ts &*&
  transactions(ts, b);
@*/
```

- Field `transactions` points to `ts` and predicate `transactions` (below) is true of `ts` and balance `b`
- Again, note `?ts` to introduce variable `ts`

The predicate for class Transaction

```
/*@
predicate transactions(Transaction t; int total)
    =
  t == null ?
    b == 0
  :
    t.amount ↦ ?a &*&
    t.next ↦ ?n &*&
    transactions(n, ?ntotal) &*&
    total == a + ntotal;
@*/
```

- `transactions()` is a recursive predicate that traverses the list collecting the amounts in the transactions
- Declared outside the class so that it may be used with `null` references

The balance of an Account with transactions

```
public int getBalance ()
  //@ requires account(?b);
  //@ ensures account(b);
{
  return getTotal(transactions);
}
private int getTotal(Transaction t)
  //@ requires transactions(t, ?total);
  //@ ensures transactions(t, total) &*& result
      == total;
{
  //@ open transactions(t, total);
  return t == null ? 0 : t.amount + getTotal(t.
      next);
}
```

- In this case an `open transactions()` is mandatory for the success of Verifast

Inheritance

- A Java interface defines a set of methods; each non-abstract class that implements the interface provides code for each method
- In order to modularly verify client code, each interface method is annotated with a contract
- We have used predicates to hide the internal representation of classes
- Verifast interfaces allows `predicate` declarations in interfaces
- A class that implements an interface is a subtype of the interface; contracts for subtypes can be refined
- In any case they must be restated in the subtype (even if they remain unchanged)

A stack interface that only speaks about the size

```
interface StackSize {
  //@ predicate stack(int size);
  void push(Object o);
    //@ requires stack(?s);
    //@ ensures stack(s+1);
  int size();
    //@ requires stack(?s);
    //@ ensures stack(s) &*& result == s;
  Object peek ();
    //@ requires stack(?s) &*& s > 0;
    //@ ensures stack(s);
  void pop ();
    //@ requires stack(?s) &*& s > 0;
    //@ ensures stack(s - 1);
}
```

Array ownership

- Recall: ownership of field `f` of an object `e1` with value `e2` is denoted as `e1.f` $\mapsto$ `e2`. Read "`e1.f` points to `e2`"
- For arrays one writes: `a[from..to]` $\mapsto$`v` to mean "the portion of array `a` between indices `from` (inclusive) to `to` (exclusive) points to `v`"
- Example for a stack implemented with an array `elements`, variable `elems` denotes the list of the elements in the stack:

```
this.size ↦ ?s &*& this.elements ↦ ?e &*&
e[0..s] ↦ elems &*& ...
```

ArrayStack

```
public class ArrayStack implements StackSize {
  private Object[] elements;
  private int size;
  /*@
  predicate stack(int s) =
    this.size ↦ s &*&       // Acquire size
    this.elements ↦ ?e &*& // Acquire reference
    e[0..e.length] ↦ _ &*& // Acquire all array
        elems
    0 <= s && s <= e.length; // The invariant
  @*/
```

Some ArrayStack methods

```
ArrayStack(int initialCapacity)
  //@ requires initialCapacity >= 0;
  //@ ensures stack(0);
{ elements = new Object[initialCapacity]; }
void pop()
  //@ requires stack(?s) &*& s > 0;
  //@ ensures stack(s - 1);
{ elements[--size] = null; }
Object peek()
  //@ requires stack(?s) &*& s > 0;
  //@ ensures stack(s);
{ return elements[size - 1]; }
```

- Contracts written with `predicate stack()` only

Copying an array

```
Object[] a = new Object [size * 2 + 1];
//@ close array_slice_dynamic(array_slice_Object
    , elements, 0, size, _); // get hold of the
    elems array
//@ close array_slice_dynamic(array_slice_Object
    , a, 0, size, _); // same for array a
//@ close arraycopy_pre(array_slice_Object,
    false, 1, elements,
0,size, _, a, 0); // fold the pre
System.arraycopy(elements, 0, a, 0, size);
//@ open arraycopy_post(_, _, _, _, _, _, _, _,
    _); // unfold the post
//@ open array_slice_dynamic(array_slice_Object,
     a, _, _, _); // release array a
elements = a;
```

Method push: growing the stack

```
void push(Object x)
  //@ requires stack(?s);
  //@ ensures stack(s + 1);
{
  if (size == elements.length) {
    Object[] a = new Object [size * 2 + 1];
    ...
    System.arraycopy(elements, 0, a, 0, size);
    ...
  }
  elements[size++] = x;
}
```

A better invariant

- The invariant for the stack, and consequently the contracts, only talk about the size
- We never know if the `push()` indeed places the value in the stack, let alone if the value is placed at the top
- For more precise invariants, we use **models**

Model-based specifications

- Modelling is an abstraction technique for system design and specification
- A **model** is a representation of the desired system
- A **formal model** is one that has a precise description in a formal language
- A model differs from an implementation in that it might:
  - capture only some aspects of the system
  - be partial, leaving some parts unspecified
  - not be executable
- An implementation of the system can be compared to the model

The list datatype: one of the Verifast models

- See `<verifast>/bin/rt/_list.javaspec`
- Operations on datatypes (introduced with the keyword `fixpoint`) must be total

```
inductive list<t> = nil | cons(t, list<t>);

fixpoint t head<t>(list<t> xs) {
  switch (xs) {
    case nil: return default_value<t>;
    case cons(x, xs0): return x;
  }
}
```

Ciências ULisboa | Informática

Some list predefined operations

```
fixpoint t head<t>(list<t> xs)
fixpoint list<t> tail<t>(list<t> xs)
fixpoint int length<t>(list<t> xs)
fixpoint list<t> append<t>(list<t> xs, list<t> ys)
fixpoint list<t> reverse<t>(list<t> xs)
fixpoint boolean mem<t>(t x, list<t> xs)
fixpoint t nth<t>(int n, list<t> xs)
fixpoint list<t> store<t>(list<t> xs, int index, t v)
fixpoint boolean distinct<t>(list<t> xs)
fixpoint list<t> take<t>(int n, list<t> xs)
fixpoint list<t> drop<t>(int n, list<t> xs)
fixpoint list<t> remove<t>(t x, list<t> xs)
fixpoint list<t> remove_nth<t>(int n, list<t> xs)
fixpoint list<t> remove_every<t>(t x, list<t> xs)
fixpoint list<t> remove_all<t>(list<t> xs, list<t> ys)
fixpoint int index_of<t>(t x, list<t> xs)
fixpoint boolean all_eq<t>(list<t> xs, t x0)
fixpoint list<t> update<t>(int i, t y, list<t> xs)
fixpoint boolean forall<t>(list<t> xs, fixpoint(t,
    boolean) p)
```

An interface that talks about the elements in the stack

```
interface Stack {
  //@ predicate stack(list<Object> elems);
  void push(Object x);
    //@ requires stack(?e);
    //@ ensures stack(append(e, cons(x, nil)));
  int size();
    //@ requires stack(?e);
    //@ ensures stack(e) &*& result == length(e)
        ;
  Object peek ();
    //@ requires stack(?e) &*& length(e) > 0;
    //@ ensures stack(e) &*& result == nth(
        length(e) - 1, e);
  void pop ();
    //@ requires stack(?e) &*& length(e) > 0;
    //@ ensures stack(take(length(e) - 1, e));
}
```

The abstraction function for an ArrayStack

```
public class ArrayStack implements Stack {
  /*@
  predicate stack(stack<Object> elems) =
    this.size ↦ ?s &*&
    this.elements ↦ ?e &*&
    e[0..s] ↦ elems &*& // get hold of the
       elems in the stack
    e[s..e.length] ↦ _ &*& // get hold of the
       remaining elems in the array
    s == length(elems);  // the invariant
  @*/
```

Some ArrayStack methods

```
ArrayStack(int initialCapacity)
 //@ requires initialCapacity >= 0;
 //@ ensures stack(nil);

boolean isEmpty ()
 //@ requires stack(?elems);
 //@ ensures stack(elems) &*& result == (length(
     elems) == 0);
```

The push method

```
void push(Object x)
 //@ requires stack(?elems);
 //@ ensures stack(append(elems, cons(x, nil)));
     // Cannot prove condition.
```

- Recall the invariant:

```
this.size ↦ ?s &*&
this.elements ↦ ?e &*&
e[0..s] ↦ elems &*& // get hold of the stack
   elems
s == length(elems);  // the invariant
```

The verification condition for push()

- In this case, `elems` in the invariant is a complicated expression

```
append(elemsOld, cons(x, nil))
```

  where `elemsOld` is the list at method entry
- So Verifast needs to prove that

```
s == length(append(elemsOld, cons(x, nil)));
```

- that is, that the length of the append of two lists is the sum of the length of the lists:

```
length(append(xs, ys)) == length(xs) +
    length(ys)
```

- And this is a bit too much for VeriFast

Helping Verifast

- Lemma functions allow developers to prove properties about predicates
- A lemma is a method without side effects. A lemma is *pure* if its contract does not contain spatial assertions
- A particular useful variant is the `lemma_auto` that does not require to write a body:

```
/*@
lemma_auto void length_append<t>(list<t> xs,
    list<t> ys)
  requires true;
  ensures length(append(xs, ys)) == length(xs) +
      length(ys);
{
  length_append(xs, ys);
}
@*/
```

Inheritance

- Contracts are not inherited: there exists a relation between contracts in the subtype and that of the supertype
- In particular, the two contracts may coincide

```
interface Parent {
  int triple(int n);
    //@ requires n >= 0;
    //@ ensures result >= 0;
}
class Child1 implements Parent {
  int triple(int n)
    //@ requires n >= 0;      // As in supertype
    //@ ensures result >= 0; // As in supertype
  { return 3 * n; }
}
```

The precondition may be weakened

```
class Child2 implements Parent {
  int triple(int n)
    //@ requires n > -5;
    //@ ensures result >= 0;
  { return n > 0 ? 3 * n : -3 * n; }
}
```

- Precondition `n > -5` **implies** `n > 0`, and so we are good

The postcondition may be strengthened

```
class Child3 implements Parent {
  int triple(int n)
    //@ requires n >= 0;
    //@ ensures result >= n;
  { return 3 * n; }
}
```

- Postcondition `result >= n` **is implied by** `result >= 0`, and so we are good

Weak the pre; strengthen the post

```
class Child4 implements Parent {
  int triple(int n)
    //@ requires true;
    //@ ensures result >= n;
  { return n > 0 ? 3 * n : -3 * n; }
}
```

- The precondition in the subtype implies that in supertype
- The postcondition in the subtype is implied by that in supertype

Bibliography

Jan Smans, Bart Jacobs, and Frank Piessens.
Verifast for Java: A tutorial.
In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 407–442. Springer, 2013.