Software Verification

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

October 14, 2025

Lecture 6

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

also based on lectures by Andrei Paskevich and Claude Marché

Verification of Imperative Programs (3/n)

1. Loop invariants – recap
2. Case study: fast exponentiation
3. Case study: the McCarthy 91 function
4. Sorting algorithms
5. Verification with bounded integers

Loop invariants – recap

Loop invariant approximate state assertions before the loop, between iterations, and at the end of the loop.

```
let max_array (a: array int) : int
  requires { 0 < a.length }
  ensures  { forall k. 0 <= k < a.length -> a[k] <= result }
= let ref m = a[0] in
  let ref i = 1 in
  while i < a.length do
    variant   { ??? }
    invariant { ??? }
    invariant { ??? }
    if m < a[i] then
      m <- a[i];
    i <- i + 1
  done;
  m
```

Loop invariant approximate state assertions before the loop, between iterations, and at the end of the loop.

```
let max_array (a: array int) : int
  requires { 0 < a.length }
  ensures  { forall k. 0 <= k < a.length -> a[k] <= result }
= let ref m = a[0] in
  let ref i = 1 in
  while i < a.length do
    variant   { a.length - i }
    invariant { ??? }
    invariant { ??? }
    if m < a[i] then
      m <- a[i];
    i <- i + 1
  done;
  m
```

Loop invariant approximate state assertions before the loop, between iterations, and at the end of the loop.

```
let max_array (a: array int) : int
  requires { 0 < a.length }
  ensures  { forall k. 0 <= k < a.length -> a[k] <= result }
= let ref m = a[0] in
  let ref i = 1 in
  while i < a.length do
    variant   { a.length - i }
    invariant { 1 <= i <= a.length }
    invariant { ??? }
    if m < a[i] then
      m <- a[i];
    i <- i + 1
  done;
  m
```

Loop invariant approximate state assertions before the loop, between iterations, and at the end of the loop.

```
let max_array (a: array int) : int
  requires { 0 < a.length }
  ensures  { forall k. 0 <= k < a.length -> a[k] <= result }
= let ref m = a[0] in
  let ref i = 1 in
  while i < a.length do
    variant   { a.length - i }
    invariant { 1 <= i <= a.length }
    invariant { forall k. 0 <= k < i -> a[k] < m }
    if m < a[i] then
      m <- a[i];
    i <- i + 1
  done;
  m
```

```
predicate sorted (a: array int)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int) (value: int) : (pos: int)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { ??? }
    invariant { ??? }
    invariant { ??? }
    let ref mid = div (high + low) 2 in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

```
predicate sorted (a: array int)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int) (value: int) : (pos: int)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { 0 <= low <= high <= a.length }
    invariant { ??? }
    invariant { ??? }
    let ref mid = div (high + low) 2 in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

```
predicate sorted (a: array int)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int) (value: int) : (pos: int)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { 0 <= low <= high <= a.length }
    invariant { forall k. 0 <= k < low -> a[k] <> value }
    invariant { forall k. high <= k < a.length -> a[k] <> value }
    let ref mid = div (high + low) 2 in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

```
predicate sorted (a: array int)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int) (value: int) : (pos: int)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { 0 <= low <= high <= a.length }
    invariant { forall k. 0 <= k < a.length -> a[k] = value -> low <= k < high }
                (* low >= high -> a[k] <> value *)
    let ref mid = div (high + low) 2 in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

Case study: fast exponentiation

Compute an exponent of an integer using fewer multiplications:

- $\lfloor \log(n) \rfloor$ squares
- $\lfloor \log(n) \rfloor$ multiplications

Repeated squaring: to compute b^{2^n}, for some positive n,
we compute a square n times
(e.g. $3^{16} = 3^{2^4} = (3^2)^8 = 9^8 = (9^2)^4 = (81^2)^2 = 6561^2 = 43046721$).

Magical property:

$$x^n = \begin{cases} x \times (x^2)^{\frac{n-1}{2}} & \text{if } n \text{ is odd} \\ \\ (x^2)^{\frac{n}{2}} & \text{if } n \text{ is even} \end{cases}$$

Exponentiation as a pure function in Why3:

```
let rec function exp (x: int) (n: int) : int
  requires { n >= 0 }
  variant  { n }
= if n = 0 then 1
  else x * exp x ( n - 1)
```

Now, the actual implementation:

```
let fast_exp (x: int) (n: int) : int


= let ref r = 1 in
  let ref c = x in
  let ref i = n in
  while i > 0 do
    variant   { i }
    invariant { 0 <= i <= n }
    invariant { r * exp c i = exp x n }
    if mod i 2 <> 0 then begin
        r <- r * c;
        i <- i - 1
    end;
    c <- c * c;
    i <- div i 2;
  done;
  r
```

Now, the actual implementation + specification:

```
let fast_exp (x: int) (n: int) : int
  requires { n >= 0 }
  ensures  { result = exp x n }
= let ref r = 1 in
  let ref c = x in
  let ref i = n in
  while i > 0 do
    variant   { i }
    invariant { 0 <= i <= n }
    invariant { r * exp c i = exp x n }
    if mod i 2 <> 0 then begin
        r <- r * c;
        i <- i - 1
    end;
    c <- c * c;
    i <- div i 2;
  done;
  r
```

Fast exponentiation – main loop

```
while i > 0 do
  variant   { ??? }
  invariant { ??? }
  invariant { ??? }
  if mod i 2 <> 0 then begin
      r <- r * c;
      i <- i - 1
  end;
  c <- c * c;
  i <- div i 2;
done;
```

```
while i > 0 do
  variant   { i }
  invariant { ??? }
  invariant { ??? }
  if mod i 2 <> 0 then begin
      r <- r * c;
      i <- i - 1
  end;
  c <- c * c;
  i <- div i 2;
done;
```

```
while i > 0 do
  variant   { i }
  invariant { 0 <= i <= n }
  invariant { ??? }
  if mod i 2 <> 0 then begin
      r <- r * c;
      i <- i - 1
  end;
  c <- c * c;
  i <- div i 2;
done;
```

```
while i > 0 do
  variant   { i }
  invariant { 0 <= i <= n }
  invariant { r * exp c i = exp x n }
  if mod i 2 <> 0 then begin
      r <- r * c;
      i <- i - 1
  end;
  c <- c * c;
  i <- div i 2;
done;
```

Magical property:

$$x^n = \begin{cases} x \times (x^2)^{\frac{n-1}{2}} & \text{if } n \text{ is odd} \\ \\ (x^2)^{\frac{n}{2}} & \text{if } n \text{ is even} \end{cases}$$

Auxiliary lemma:

```
let rec lemma exp_lemma (x n: int)
  requires { n >= 0 }
  variant  { n }
  ensures  { mod n 2 =  0 -> exp x n = exp (x * x) (div n 2) }
  ensures  { mod n 2 <> 0 -> exp x n = x * exp (x * x) (div (n-1) 2) }
=
```

Magical property:

$$x^n = \begin{cases} x \times (x^2)^{\frac{n-1}{2}} & \text{if } n \text{ is odd} \\ \\ (x^2)^{\frac{n}{2}} & \text{if } n \text{ is even} \end{cases}$$

Auxiliary lemma:

```
let rec lemma exp_lemma (x n: int)
  requires { n >= 0 }
  variant  { n }
  ensures  { mod n 2 =  0 -> exp x n = exp (x * x) (div n 2) }
  ensures  { mod n 2 <> 0 -> exp x n = x * exp (x * x) (div (n-1) 2) }
= if mod n 2 =  0 && n > 1 then exp_lemma x (n - 2);
  if mod n 2 <> 0 && n > 1 then exp_lemma x (n - 2)
```

Case study: the McCarthy 91 function

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ \\ M(M(n+11)) & \text{if } n \leq 100 \end{cases}$$

Interesting facts:

- $M(n) = 91$,
  forall $n \leq 100$
- $M(n) = n - 10$,
  forall $n > 100$.

(Homework: prove this by induction on n.)

A recursive implementation in Why3:

```
let rec mcCarthy (n: int) : int
  ensures { if n <= 100 then result = 91
                        else result = n - 10 }
  variant { 101 - n }
= if n > 100 then n - 10
  else mcCarthy(mcCarthy(n + 11))
```

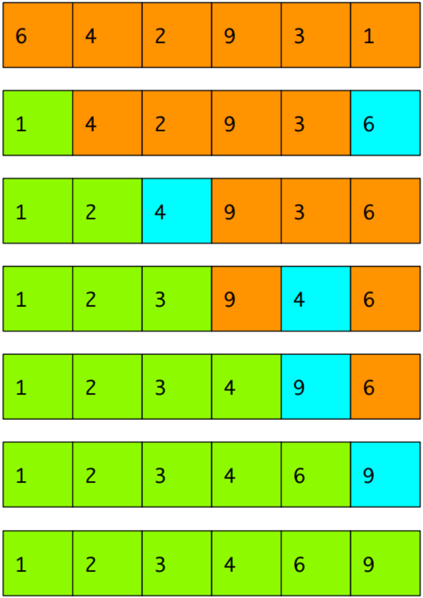
Sorting algorithms

Sorting – rationale

Sorting algorithms often iteratively traverse the data structure gradually sorting its elements.

Loop invariants capture the transient status of the sorting algorithm.

We illustrate that using Selection sort.


6 4 2 9 3 1
1 4 2 9 3 6
1 2 4 9 3 6
1 2 3 9 4 6
1 2 3 4 9 6
1 2 3 4 6 9
1 2 3 4 6 9

```
let selection_sort (a: array int)

  let min = ref 0 in
  for i = 0 to length a - 1 do



    min := i;
    for j = i + 1 to length a - 1 do


      if a[j] < a[!min] then min := j
    done;
    label L in
    if !min <> i then swap a !min i;
  done
```

Selection sort – specification

```
let selection_sort (a: array int)
  ensures { sorted a
  let min = ref 0 in
  for i = 0 to length a - 1 do
    (* a[0..i[ is sorted; now find minimum of a[i..n[ *)
    invariant { sorted_sub a 0 i /\
        forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }
    min := i;
    for j = i + 1 to length a - 1 do
      invariant {
        i <= !min < j && forall k: int. i <= k < j -> a[!min] <= a[k] }
      if a[j] < a[!min] then min := j
    done;
    label L in
    if !min <> i then swap a !min i;
  done
```

Some operations can fail if their safety preconditions are not met:

- arithmetic operations: division par zero, overflows, etc.
- memory access: `NULL` pointers, buffer overruns, etc.
- assertions

A correct program must not fail:

$\{P\}\,e\,\{Q\}$ if we execute e in a state that satisfies P,
then there will be no run-time errors
and the computation either diverges
or terminates normally in a state that satisfies Q

A new kind of expression:

$$
\begin{array}{lcll}
e & ::= & \dots & \\
 & | & \texttt{assert}\ R & \text{fail if } R \text{ does not hold}
\end{array}
$$

The corresponding weakest precondition rule:

$$\mathrm{WP}(\texttt{assert}\ R, Q) \quad \equiv \quad R \wedge Q \quad \equiv \quad R \wedge (R \to Q)$$

The second version is useful in practical deductive verification.

Selection sort – not quite done yet!

What if `selectionSort` just fills the array with zeros?

The specification we wrote is still provable!

The need for a better specification:

- the elements of the sorted array
- are a permutation of the
- elements in the input array

```
let selection_sort (a: array int)
  ensures { sorted a /\ permut_all (old a) a }
  let min = ref 0 in
  for i = 0 to length a - 1 do
    (* a[0..i[ is sorted; now find minimum of a[i..n[ *)
    invariant { sorted_sub a 0 i /\ permut_all (old a) a /\
        forall k1 k2: int. 0 <= k1 < i <= k2 < length a -> a[k1] <= a[k2] }
    min := i;
    for j = i + 1 to length a - 1 do
      invariant {
        i <= !min < j && forall k: int. i <= k < j -> a[!min] <= a[k] }
      if a[j] < a[!min] then min := j
    done;
    label L in
    if !min <> i then swap a !min i;
  done
```

Selection sort – permutation of elements

Notation **old a**:

- the elements of the array at function call point
- the pre-state

Verification with bounded integers

Back to binary search

```
predicate sorted (a: array int)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int) (value: int) : (pos: int)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { 0 <= low <= high <= a.length }
    invariant { forall k. 0 <= k < a.length -> a[k] = value ->
                  low <= k < high }
    let ref mid = div (high + low) 2 in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

In **(high + low) / 2**, what happens if

- **high == MAX_INT**
- **low == MAX_INT** - **1**?

INTEGER OVERFLOW!

> *Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken*
>
> Joshua Bloch, Software Engineer (02/06/2006)

So, have we been proving erroneous programs?

Until now all our proofs have assumed mathematical integers

- unbounded arithmetic

Our reasoning is sound, provided the code executes using arbitrary precision integers.

- `why3 extract -D ocaml64 bin_search.mlw`
- Use an external library of arbitrary precision integers

What if we want to use 32-bit or 64-bit integers?

What do we know about our implementations?

- Overflow?
- What happens? (Modulo arith., saturated arith., abrupt termination, etc.)

Verification with Machine Integer

Goal: prove that a program is safe with respect to overflows.

32-bit signed integers in two-complement representation: integers between -2^{31} and $2^{31}-1$.

If the mathematical result of an operation fits in the range, that is the computed result.

Otherwise, an overflow occurs:

- Behavior depends on language and environment.

A program is safe wrt overflow if no overflow can occur.

Idea: Replace all arith. operations by methods with preconditions.

`x+y` becomes `int32_add(x, y)`, and so on:

```
let int32_add (x y: int) : int
  requires { -2_147_483_648 <= x+y <= 2_147_483_647 }
  ensures  { r = x+y }
= x+y
```

Not great: range constraints of integer must be added explicitly everywhere...

Better Idea: Replace int with a range type `int32`:

```
type int32 = < range -0x8000_0000 0x7fff_ffff >

function to_int (x : int32) : int = int32'int x
```

The `to_int` function is a projection back to **int**.

Why3 features other models of precision: `Int63`, `Int64`, `UInt64`, etc.

Can even replace operations by custom methods:

```
predicate in_bounds (n:int) = min <= n <= max

val (+) (a:int32) (b:int32) : int32
  requires { [@expl:integer overflow] in_bounds (a + b) }
  ensures  { result = a + b }
```

Beware! There is an implicit conversion from `int32` to `int`.

Binary Search – Overflow

```
predicate sorted (a: array int32)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int32) (value: int32) : (pos: int32)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0:int32 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { 0 <= low <= high <= a.length }
    invariant { forall k. 0 <= k < a.length -> a[k] = value ->
                  low <= k < high }
    let ref mid = (high + low) / (2:int32) in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

```
predicate sorted (a: array int32)
= forall i j. 0 <= i <= j < a.length -> a[i] <= a[j]

let bin_search (a: array int32) (value: int32) : (pos: int32)
  requires { 0 <= a.length /\ sorted a }
  ensures  { 0 <= pos -> pos < a.length /\ a[pos] = value }
  ensures  { pos < 0 -> forall i. 0 <= i < a.length -> a[i] <> value }
= let ref low = 0:int32 in
  let ref high = a.length in
  while low < high do
    variant   { high - low }
    invariant { 0 <= low <= high <= a.length }
    invariant { forall k. 0 <= k < a.length -> a[k] = value ->
                    low <= k < high }
    let ref mid = low + (high - low) / (2:int32) in
    if a[mid] < value then low <- mid + 1
    else if value < a[mid] then high <- mid
    else (* value = a[mid] *) return mid
  done;
  -1
```

Caveat: Reasoning with machine integers in this way is challenging.

What if we want to reason with overflow?

Do we really need to always prove absence of overflow?

```
let incr (x: int32) : int32
= x + 1
```

If some variable `c`

- is initially set to 0
- is only incremented using method `incr`
- it would take 584 years to reach overflow.

Caveat: Reasoning with machine integers in this way is challenging.

What if we want to reason with overflow?

Do we really need to always prove absence of overflow?

```
let incr (x: int32) : int32
= x + 1
```

A solution: "*How to avoid proving the absence of integer overflows*".
M. Clochard, J-C. Filliâtre, and A. Paskevich (VSTTE 2015).

General idea:

- only require the absence of overflow during the first n steps
- n being large enough for all practical purposes.