Software Verification

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

October 21, 2025

Lecture 7

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 (4/n)

1. Verification of Abstract Data Types
2. Ghost Code for Modeling Abstract Data Types

Abstract Data Types

Abstract Data Types (Liskov, 1978)

ADTs are the building blocks for software construction.

- A description of the data elements of the type
- A set of operations over the data elements of the ADT

A software system is a composition of ADTs.

ADTs behave like regular types in a programming language.

Promotes

- modularity,
- encapsulation,
- information hiding,
- reuse,
- modifiability,
- correctness.

Abstract Data Types – how it began

Abstract types are intended to be very much like the built-in types provided by a programming language. The user of a built-in type, such as integer or integer array, is only concerned with creating objects of that type and then performing operations on them. He is not (usually) concerned with how the data objects are represented, and he views the operations on the objects as indivisible and atomic when in fact several machine instructions may be required to perform them. In addition, he is not (in general) permitted to decompose the objects. Consider, for example, the built-in type integer. A programmer wants to declare objects of type integer and to perform the usual arithmetic operations on them. He is usually not interested in an integer object as a bit string, and cannot make use of the format of the bits within a computer word. Also, he would like the language to protect him from foolish misuses of types (e.g., adding an integer to a character) either by treating such a thing as an error (strong typing), or by some sort of automatic type conversion.

Abstract Data Type – external view

External view

- A public opaque data type (that clients will use)
  Note: opaque = behaves as a primitive type
- A set of operations on this data type
- Operations must neither reveal, nor allow a client to invalidate the internal representation of the ADT
- pre- and post-conditions on these operations must be expressed in terms of the abstract type (the only type known to the client)
- This is why ADTs promote reuse, modifiability, and correctness:
  - the developer can change the implementation anytime,
  - without breaking contracts

Abstract Data Type – internal view

Internal view

- a representation data type, hidden from clients
- a set of operations on the representation data type

Key remarks

- A programmer must define the operations in such a way that the
  - representation state is kept consistent
  - with the intended abstract state
- Pre-conditions on the public operations
  - are expressed on the abstract state
  - map into pre-conditions of the representation state
- The same for post-conditions
- The concrete state must always represent a sound abstract state. (otherwise something is wrong!)

Abstract Data Type – Example (Positive Set)

```
module PSet
  type t (* an abstract set of positive numbers *)

  val create (sz: int) : t
  (** initializes the set *)

  val add (v: int) (t: t) : unit
  (** adds v to the set if space available *)

  val function size (t: t) : int
  (** returns number of elements in the set *)

  val function contains (v: int) (t: t) : bool
  (** checks whether v is in the set *)

  val function max_size (t: t) : int
  (** returns max number of elements allowed in the set *)
end
```

Technical Ingredients in ADT Design (1/2)

1. The abstract state
   - defines how client sees the object

2. The representation type
   - the internals of the ADT
   - concrete data structures (*e.g.*, arrays, trees, lists)
   - algorithms (*e.g.*, sorting, searching)

3. The concrete state
   - in general, not all representation states are legal concrete states
   - *example*: lists without duplicates, Binary Search Trees
   - a concrete state must represent a well-defined abstract state

Technical Ingredients in ADT Design (2/2)

4. The type invariant
   - the type invariant is a condition that restricts the representation type to the set of safe concrete states
   - if the ADT rep type falls off the rep invariant, something is wrong

5. The abstraction function
   - maps every concrete state into some abstract state

6. The operation pre- and post-conditions
   - expressed for the representation type
   - also expressed for the abstract type (client view)

Abstract Data Types, Verified

Bank Account ADT

Abstract state

- the account balance, **`bal`**
- **`bal`** is of type **`int`**, constrained by **`bal >= 0`**

Representation type

- an integer **`bal`**
- the representation type is the same as the abstract state (pure coincidence)
- however, the semantics of rep and abstract types are different
- not all operations on integers are valid on account balances (*e.g.*, to multiply bank accounts)

Type invariant

- **`bal >= 0`**

Bank Account in Why3

```
module Account

  use int.Int

  type t = {
    mutable bal: int;
  } invariant { bal >= 0 } (* the type invariant *)
    by { bal = 0 }          (* the witness *)

  let create () : t
  (* ensures { result.bal >= 0 } is automatically added *)
  = { bal = 0 }
```

Bank Account in Dafny

```
(* All operations (auto.) require the type invariant *)
(* All operations (auto.) ensure the type invariant *)
let deposit (v: int) (t: t) : unit
  requires { v >= 0 }
= t.bal <- t.bal + v

let withdraw (v: int) (t: t) : unit
  requires { v >= 0 }
= if t.bal >= v then t.bal <- t.bal - v
```

Bank Account in Why3 – better specification for `withdraw`

```
let function get_bal (t: t) : int
  ensures { result = t.bal }
= t.bal

let withdraw (v: int) (t: t) : unit
  requires { 0 <= v <= get_bal t }
= if t.bal >= v then t.bal <- t.bal - v
```

Set ADT – recap

```
module PSet
  type t (* an abstract set of positive numbers *)

  val create (sz: int) : t
  (** initializes the set *)

  val add (v: int) (t: t) : unit
  (** adds v to the set if space available *)

  val function size (t: t) : int
  (** returns number of elements in the set *)

  val function contains (v: int) (t: t) : bool
  (** checks whether v is in the set *)

  val function max_size (t: t) : int
  (** returns max number of elements allowed in the set *)
end
```

Set ADT

Abstract state

- a bounded set of positive integers **`aset`**

Representation type

- an array of integers **`store`** with sufficient large size
- an integer **`nelems`** counting the elements in **`store`**

Type invariant

```
(0 <= nelems <= store.length) &&

forall k j. 0 <= k < nelems -> k < j < nelems -> store[k] <> store[j]
```

Abstraction mapping

<nelems=n, store=[$v_0, v_1, \ldots, v_{\texttt{store.Length}-1}$]> $\mapsto \{v_0, \ldots, v_n\}$

- more on this in a few minutes

Set ADT in Why3

```
module ASet : PSet (* refinement proof --> more later *)
  use int.Int
  use array.Array

  type t = {
    store: array int;
    max: int;
    mutable nelems: int;
  } invariant { store.length > 0 }
    invariant { max = store.length }
    invariant { 0 <= nelems <= store.length }
    invariant { forall k j.
                  0 <= k < nelems -> k < j < nelems ->
                  store[k] <> store[j] }
    by { store = make 1 0; max = 1; nelems = 0 }

  let create (sz: int) : t
    requires { sz > 0 }
  = { store = make sz 0; max = sz; nelems = 0; }
```

Set ADT in Why3

```
let function size (t: t) : int
= t.nelems

let function max_size (t: t) : int
= t.store.length

let add (x: int) (t: t) : unit
  requires { size t < max_size t }
= let f = find x t in
  if f > 0 then begin
    t.store[t.nelems] <- x;
    t.nelems <- t.nelems + 1
  end
```

Set ADT in Why3

```
let find (x: int) (t: t) : (r: int)
  ensures { -1 <= r <= t.nelems }
  ensures { r < 0 -> forall j. 0 < j < t.nelems ->
                x <> t.store[j] }
  ensures { r >= 0 -> t.store[r] = x }
= for i = 0 to t.nelems - 1 do
    invariant { forall j. 0 <= j < i -> x <> t.store[j] }
    if t.store[i] = x then return i
  done;
  -1
```

Set ADT in Why3

```
let function contains (v: int) (t: t) : bool
  ensures { result <->
            exists j. 0 <= j < t.nelems /\
                      v = t.store[j] }
= let p = find v t in
  p >= 0
```

Soundness and Abstraction Map

Soundness and Abstraction Map

We have learned how to express the type invariant and make sure that no unsound states are ever reached.

We have informally argued that the representation state in every case represents the right abstract state, but how to make sure?

We next see how the correspondence between the representation state and the abstract state is expressed in Why3 using

- ghost state,
- specification operations,
- abstraction map soundness check.

Ghost Code – Example

Compute a Fibonacci number using a recursive function in $O(n)$:

```
let rec aux (a b n: int) : (r: int)
  requires { 0 <= n }
  requires { exists k. 0 <= k /\ a = fib(k) /\
                       b = fib (k+1) }
  ensures  { exists k. 0 <= k /\ a = fib(k) /\
                       b = fib (k+1) /\ r = fib (k + n) }
  variant  { n }
= if n = 0 then a
  else aux b (a+b) (n-1)

let fib_rec (n: int) : (r: int)
  requires { 0 <= n }
  ensures { r = fib n }
= aux 0 1 n
(* fib_rec 5 = aux 0 1 5 = aux 1 1 4 = aux 1 2 3 =
                aux 2 3 2 = aux 3 5 1 = aux 5 8 0 = 5 *)
```

Ghost Code – Example

Instead of an existential we can use a ghost parameter:

```
let rec aux (a b n: int) (ghost k: int) : (r: int)
  requires { 0 <= n }
  requires { 0 <= k /\ a = fib(k) /\ b = fib (k+1) }
  ensures  { 0 <= k /\ a = fib(k) /\ b = fib (k+1) /\
                   r = fib (k + n) }
  variant  { n }
= if n = 0 then a
  else aux b (a+b) (n-1) (k+1)

let fib_rec (n: int) : (r: int)
  requires { 0 <= n }
  ensures { r = fib n }
= aux 0 1 n 0
```

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome.

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome.

Consequently:

- regular code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome.

Consequently:

- regular code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too
- ghost code cannot modify regular data
  - if r is a regular reference, then $r := \texttt{ghost}\ k$ is forbidden

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome.

Consequently:

- regular code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too
- ghost code cannot modify regular data
  - if r is a regular reference, then $r := \texttt{ghost}\ k$ is forbidden
- ghost code cannot alter the control flow of regular code
  - if c is ghost, then $\texttt{if}\ c\ \texttt{then}\ \ldots$ and $\texttt{while}\ c\ \texttt{do}\ \ldots$ are ghost

The spirit of ghost code

Ghost code is used to facilitate specification and proof

$\Rightarrow$ the principle of non-interference:

> We must be able to eliminate the ghost code
> from a program without changing its outcome.

Consequently:

- regular code cannot read ghost data
  - if k is ghost, then $(k+1)$ is ghost, too
- ghost code cannot modify regular data
  - if r is a regular reference, then $r := \texttt{ghost}\ k$ is forbidden
- ghost code cannot alter the control flow of regular code
  - if c is ghost, then `if` c `then ...` and `while` c `do ...` are ghost
- ghost code cannot diverge
  - we can prove `while true do skip ; assert false`

Ghost Code – ADT Abstract State

```
module ASet : PSet
  use int.Int
  use array.Array
  use set.Fset

  type t = {
    store: array int;
    max: int;
    mutable nelems: int;
    ghost mutable s: fset int;
  } invariant { store.length > 0 }
    invariant { max = store.length }
    invariant { 0 <= nelems <= store.length }
    invariant { forall k j. 0 <= k < nelems -> k < j < nelems ->
                  store[k] <> store[j] }
    invariant { forall x. mem x s <->
                          exists k. 0 <= k < nelems /\ store[k] = x }
    invariant { cardinal s = nelems }
    by { store = make 1 0; max = 1; nelems = 0; s = Fset.empty }
```

The type invariant now specifies the precise relationship between

- abstract
- and the concrete state

ADT with Abstract State

We must now express, in all operations

- how the abstract state changes, and
- how it is kept well related with a proper representation state.

Benefit: pre- and post-conditions expressed in terms of abstract state.

ADT with Abstract State

```
let create (sz: int) : t
  requires { sz > 0 }
  ensures  { result.s = Fset.empty }
= { store = make sz 0;
    max = sz;
    nelems = 0;
    s = Fset.empty; }
```

ADT with Abstract State

```
let add (x: int) (t: t) : unit
  requires { size t < max_size t }
  ensures  (* what about abstract state? *)
= let f = find x t in
  if f < 0 then begin
    t.store[t.nelems] <- x;
    t.nelems <- t.nelems + 1;
    (* update abstract state? *)
  end
```

ADT with Abstract State

```
let add (x: int) (t: t) : unit
  requires { size t < max_size t }
  ensures  { t.s = Fset.add x (old t.s) }
= let f = find x t in
  if f < 0 then begin
    t.store[t.nelems] <- x;
    t.nelems <- t.nelems + 1;
    t.s <- Fset.add x t.s
  end
```

Note the use of (`old t.s`)

Meaning: the elements in `t.s` before the function call.

We already saw this last week.

ADT with Abstract State

```
let function size (t: t) : int
  ensures { result = cardinal t.s }
= t.nelems

let function contains (v: int) (t: t) : bool
  ensures { result <-> mem v t.s }
= let p = find v t in
  p >= 0
```

Back to the PSet Interface

We must also provide specification to the ADT interface.

In fact, this is what an external client is supposed to see.

```
module PSet
  type t

  val create (sz: int) : t

  val function size (t: t) : int

  val function contains (v: int) (t: t) : bool

  val function max_size (t: t) : int

  val add (v: int) (t: t) : unit
end
```

ADT – Specified Interface

```
module PSet
  use int.Int, set.Fset

  type t = abstract { mutable s: fset int; max: int; }
    invariant { cardinal s <= max }
    by { s = empty; max = 0 }

  val create (sz: int) : t
    requires { sz > 0 }
    ensures  { result.s = empty }

  val function size (t: t) : int
    ensures { result = cardinal t.s }

  val function max_size (t: t) : int
    ensures { result = t.max }

  val add (v: int) (t: t) : unit
    requires { size t < max_size t }
    writes   { t.s } (* function with side-effects *)
    ensures  { t.s = Fset.add v (old t.s) }

  val function contains (v: int) (t: t) : bool
    ensures { result <-> mem v t.s }
end
```

The Implementation Refines the Interface

One can actually prove that `ASet` is a good implementation of `PSet`.

```
module ASet : PSet
```

This means:

- every function in interface has been implemented, with the correct type of arguments and return value.
- the specification in the implementation refines the specification in the interface.
- all the types are correctly implemented.

Specification Refinement

In the interface:

```
val f (x1: t1) ... (xn: tn) : (r: t)
  requires { P x1 ... xn }
  ensures  { Q x1 ... xn r }
```

In the implementation:

```
let f (x1': t1) ... (xn': tn) : (r': t)
  requires { P' x1' ... xn' }
  ensures  { Q' x1' ... xn' r' }
```

Refinement proof:

$$\begin{array}{lcl} \texttt{P x1 ... xn} & \Longrightarrow & \texttt{P}' \texttt{ x1}' \texttt{ ... xn}' \\ \texttt{Q}' \texttt{ x1}' \texttt{... xn}' \texttt{ r}' & \Longrightarrow & \texttt{Q x1... xn r} \end{array}$$

Does it remind you of something?

Specification Refinement

In the interface:

```
val f (x1: t1) ... (xn: tn) : (r: t)
  requires { P x1 ... xn }
  ensures  { Q x1 ... xn r }
```

In the implementation:

```
let f (x1': t1) ... (xn': tn) : (r': t)
  requires { P' x1' ... xn' }
  ensures  { Q' x1' ... xn' r' }
```

This is exactly the application of the Consequence Rule, from Hoare Logic.

What About Types?

Specification refinement is also applied to type invariants.

In the interface:

```
type t = abstract {
  f1: t1;
  ...
  fn: tn;
} invariant { I f1 ... fn }
```

In the implementation:

```
type t = {
  f1: t1;
  ...
  fn: tn;
  f1': t1';
  ...
  fn': tn';
} invariant { I' f1 ... fn f1' ... fn' }
```

Refinement Proof for Types

Refinement proof:

$$\texttt{I}' \ \texttt{f1} \ \ldots \ \texttt{fn} \ \texttt{f1}' \ \ldots \ \texttt{fn}' \quad \Longrightarrow \quad \texttt{I} \ \texttt{f1} \ \ldots \ \texttt{fn}$$

Additional checks:

- all fields from the interface type are implemented, with the same type.
- ghost hierarchy is respected:
  - an interface ghost field can remain ghost
  - an interface ghost field can become non-ghost
  - additional fields in the implementation can be either ghost or non-ghost.