Lectures Notes on Abstract Data Types

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

5 April, 2022

These notes focus on the implementation and verification of abstract data types (ADT). ADTs were introduced in the 70s of last century, by Liskov and Zilles [2], as building blocks for program construction that extend data types that are usually built-in in any programming language. We will add to this the construction and verification of ADTs using the object-oriented features of Dafny. You can (and should) use external references such as [1] to complement your knowledge of the language besides these notes.

Before we get into ADTs, lets first briefly address the issue of verification with machine integers (discussed in last week's lecture).

1 Verification with Machine Integers

All automated or semi-automated reasoning we have done so far with numbers has involved the Dafny type `int`, which denotes mathematical, arbitrary precision integers. This means that all our correctness proofs up until now have *assumed* that there is no largest or smallest integer, and that adding two positive (respectively negative) integers will always result in a positive (resp. negative) integer. This does not fundamentally break our proofs, but makes them only valid in settings where these assumptions hold true: whenever the manipulated integer values are "small enough".

We can strengthen our results, or address this potential limitation, in a few ways:

- We can bypass the limitation entirely and simply work with arbitrary precision integer libraries (e.g. Java's BigInteger library). This solution is plausible, but suffers from performance considerations and may not be generally applicable. For instance, one cannot create an array whose size is determined by such a number.
- We can complementary or separately verify that all our operations on integers preserve the appropriate machine bounds.

Focusing on the latter, our goal is to then prove that a program is safe with respect to integer overflow. Focusing on 32-bit signed integers in twos-complement representation, we must ensure that all integers are between -2^{31} and $2^{31} - 1$. Specifically, if the *mathematical* (i.e. arbitrary precision) result of an operation fits in the range, then it will be the *computed* result. Otherwise, an *overflow* or *underflow* occurs. The concrete program behavior that arises in this situation is machine and language dependent. We say that a program is *safe* with respect to overflows if no overflow (or underflow) can occur.

1.1 Bounded Integers in Dafny

Dafny allows us to introduce new types to the language, which are refined versions of existing base types:

```
newtype int32 = x: int | -2_147_483_648 ≤ x ≤ 2_147_483_647
```

The code above defines a type `int32` which consists of integers within the specified range, corresponding to the 32-bit encoding of signed machine integers. We can then use type `int32` over **`int`** when we want to verify that an implementation is safe with respect to overflows.

For instance, we can verify that a naive implementation of binary search is **not** safe (we make use of the *sorted* predicate from before):

```
method BSearch(a: array<int>, n: int32, value: int) returns (pos: int32)
requires 0 ≤ n as int ≤ a.Length ∧ sorted(a, n as int)
  ∧ a.Length ≤ 2_147_483_647
ensures 0 ≤ pos ⟹ pos < n ∧ a[pos] = value
ensures pos < 0 ⟹ ∀ i • (0 ≤ i < n) ⟹ a[i] ≠ value
{
 var low: int32 := 0;
 var high: int32 := n;
 while low < high
 decreases high - low
 invariant 0 ≤ low ≤ high ≤ n
 invariant ∀ i • 0 ≤ i < n ∧ i < low ⟹ a[i] ≠ value
 invariant ∀ i • 0 ≤ i < n ∧ high ≤ i ⟹ a[i] ≠ value
 {
   var mid: int32 := (high+low)/2;
   if a[mid] < value     { low := mid + 1; }
   else if value < a[mid]   { high := mid; }
   else  { return mid; }
 }
 return -1;
}
```

The code above will signal an error in the addition `high+low` since the operation might overflow. Changing the operation to `low + (high-low)/2` makes the verification go through.

We highlight the following: when verifying code with machine integers, we generally need to assume data structures manipulate quantities that can be represented with the appropriate precision. In the case of binary search and 32-bit signed integers, the algorithm can only be correct for arrays with up to $2^{31} - 1$ positions. Also note that in operations that expect mathematical integers, we use the Dafny explicit conversion operator **`as`**, to convert between type `int32` and **`int`**.

Dafny also supports *bit vectors* to reason directly about bit-level representations. We will not discuss bit-vectors in these notes.

2 Abstract Data Types

An abstract data type consists of a denomination for the data elements of said data type, and a set of operations over the elements of said data type. A fullfledge software system can be defined as a composition of abstract data types at different levels of abstraction. ADTs promote crucial properties of software development such as modularity, information hiding, and hence promote the reusability of code, modifiability, and the functional correctness of their implementations. Quoting the original work of Liskov, we may observe that the user of an ADT is not usually concerned with the internal representation of a data type, but only with the available operations and how they can be used.

Also, one can generally not decompose ADT values into its constituents but only use constructor and destructor operations to define and inspect them.

Taking an external view of an ADT, it comprises a public and opaque type, usually bound to an identifier in the programming language and a set of operations that create, manipulate, and inspect the newly defined data type. Notice that by being opaque, they may be used in any program as if it was a primitive type.

The theory of ADTs says that the type is opaque, and its internal details are hidden from the developer using it in a program. In practice, the visibility mechanisms incorporated in the programming languages are used in the definition of the ADTs so that their internal representation cannot be inspected or modified, except through the available operations.

For example, in the programming language C, a module with a header file (`.h`) can have the declarations

```
typedef *Date;

Date* createDate(int day, int month, int year);
Date* today();
int daysBetween(Date* d1, Date*d2);
int daysOf(Date* d);
int monthOf(Date* d);
int yearOf(Date* d);
```

in this case, the internal representation is unknown to the client module, and the recommended way to manipulate values of type `Date` is through the operations declared by the module

The pre-conditions and post-conditions of each operation can only be expressed using the ADT and its (pure) operations. They form a contract between the client module and the implementation of the ADT that allows one to switch between implementations without breaking the functionality of the client module. The external and opaque view, is complemented by an internal view (and implementation). Each ADT is implemented using a representation data type in the underlying level of abstraction. For instance the ADT `Date` above can be internally represented by three values of type **`int`**, or by a single value of type `long`. One particularly important observation is that, not all possible valuations for the internal representation form a valid ADT value. For instance, the tuple of three integer values $(0, 0, 0)$ does not form a valid date.

Each one of the available operations is implemented with relation to the internal representation data type. The implementation must maintain the consistency of the abstract state, that is, they must always produce valid ADT values. Thus, the specification of an ADT (pre-conditions and post-conditions) that are visible in the client module must be mapped onto assertions expressed in the representation state. The representation states of the ADT, that are observable by the client module, must be valid ADT values.

Besides the opaque type and the all operations, the ADT implementation also comprises a *representation invariant*, which is a condition that denotes the allowed values of the representation type. The condition defines a set of possible values, the domain and target sets of all operations on the ADT. This set defines the set of safe concrete states for the ADT internal representation. It also includes an abstraction function, which maps abstract states to concrete states. The pre- and post-conditions must be expressed for both the representation type and the abstract type.

A functional way of defining an ADT, of name *Set* is to define a list of functions that accompany it. Each operation has a name, a set of parameters and a return value.

$$\begin{array}{l}
\textsf{constructor} : \textsf{void} \longrightarrow Set \\
\textsf{add} : Set \times \textsf{int} \longrightarrow Set \\
\textsf{contains} : Set \times \textsf{int} \longrightarrow \textsf{bool} \\
\textsf{size} : Set \longrightarrow \textsf{int} \\
\textsf{maxSize} : Set \longrightarrow \textsf{int}
\end{array}$$

3 Implementing ADTs with objects

Object-oriented languages provide native isolation and visibility mechanisms that are suitable to implementing ADTs. For instance, consider the following class prototype written in Dafny that describes the ADT *Set*.

```
class Set {
  // creation operation given a max capacity
  constructor(n: int)

  // adds a new element to the set if the set capacity permits
  method add(v:int)

  // checks if a given element is in the set
  method contains(v:int) returns (b:bool)

  // returns the number of elements in the set
  function size() : int

  // returns the maximum number of elements in the set
  function maxSize() : int
}
```

The concrete state of a possible implementation of the ADT *Set* uses as representation type an array `store` of integer values, initialized with a number of positions greater than one, and an integer variable `nelems` that counts the significant positions in the array, corresponding to the elements stored in the set.

The domain of the variables in the representation type are restricted by the following conditions, in order to fullfil the properties of a set:

```
0 < store.Length
∧ 0 ≤ nelems ≤ a.Length
∧ ∀ i,j • 0 ≤ i < j ≤ nelems ⟹ store[i] ≠ store[j]
```

All valid arrays are the ones with more than one position of capacity, the ones where the integer value in variable `nelems` is between `0` and the actual size of the array. Moreover, since a set does not store multiple copies of the same value, we restrict the contents of the array to only allow distinct values to be stored.

This condition can be isolated properly in a predicate that defines the representation invariant of this ADT (recall that predicates in Dafny are no more than functions whose return type is `bool`):

```
predicate RepInv()
  reads this,a;
{
  0 < store.Length
  ∧ 0 ≤ nelems ≤ a.Length
  ∧ ∀ i,j • 0 ≤ i < j ≤ nelems ⟹ store[i] ≠ store[j]
}
```

The predicate `RepInv` is to be used in every operation to explicitly specify that all operations should assume and subsequently enforce a sound state of the ADT (concrete and abstract). This boilerplate can actually be circumvented by means of special declarations, which we will detail in subsequent lectures when we study the problem of framing in Hoare logic and Dafny.

In the case of the constructor, the representation invariant is used as post-condition:

```
constructor(n: int)
    requires 0 < n
    ensures RepInv()
{
    store := new int[n];
    nelems := 0;
}
```

All other (public) operation should include the invariant in their pre-condition and post-condition. Consider the implementation below for class `Set`:

```
class Set {

    var store:array<int>;
    var nelems: int;

    predicate RepInv()
    reads store, `store, `nelems
    {
        0 < store.Length
        ∧ 0 ≤ nelems ≤ store.Length
        ∧ ∀ i,j • 0 ≤ i < j < nelems ⟹ store[i] ≠ store[j]
    }

    // the construction operation
    constructor(n: int)
    requires 0 < n
    ensures RepInv()
    {
        store := new int[n];
        nelems := 0;
    }

    // returns the number of elements in the set
    function method size():int
        requires RepInv()
        ensures RepInv()
        reads store, `store, `nelems
    { nelems }

    // returns the maximum number of elements in the set
    function method maxSize():int
        requires RepInv()
        ensures RepInv()
        reads store, `store, `nelems
    { store.Length }

    // checks if the element given is in the set
    method contains(v:int) returns (b:bool)
        requires RepInv()
        ensures RepInv()
        ensures b ⟺ ∃ j • (0 ≤ j < nelems) ∧ v = store[j];
    {
        var i := find(v);
        return i ≥ 0;
    }

    // adds a new element to the set if space available
    method add(v:int)
        requires RepInv()
        requires size() < maxSize()
        ensures RepInv()
        modifies store, `nelems
    {
        var f:int := find(v);
        if (f < 0) {
            store[nelems] := v;
            nelems := nelems + 1;
        }
    }

    // private method that should not be in the
    method find(x:int) returns (r:int)
        requires RepInv()
        ensures RepInv()
        ensures r < 0 ⟹ ∀ j • (0≤j<nelems) ⟹ x ≠ store[j];
        ensures r ≥0 ⟹ r < nelems ∧ store[r] = x;
    {
        var i:int := 0;
        while (i<nelems)
            decreases nelems-i
            invariant 0≤i≤nelems;
            invariant ∀ j • (0≤j<i) ⟹ x ≠ store[j];
        {
            if (store[i]==x) { return i; }
            i := i + 1;
        }
        return -1;
    }
}
```

Notice functions `size` and `maxSize` in lines 24 and 31, which simply return some information. These functions are used to abstract the real implementation of the Set, and can be used in the specification of all other operations without revealing the existence of fields `size` and `store`. Each of these functions has a **`reads`** clause, which specifies the values from dynamically allocated memory that are touched (but not modified) by the function. As before, when some mutable memory location is read by a program fragment is read, it must be declared to enable the reasoning about assignments in general. In this case, we must refer separately to the fields of the class, and to the contents of the array, which are in a different area of the heap. The reads clause

```
reads `store
```

specifies that the function will read the value of the field `store`, which amounts to the pointer to the array and its size. It is equivalent to

```
reads this`store
```

And is subsumed by the more general declaration:

```
reads this
```

Note that the read clauses in lines 27 and 34 include also the field `nelems`. They also include the clause

```
reads `store
```

which amount to reading the contents of the array. This can also be written

```
reads this.store
```

4 Mapping to an Abstract State

Notice that in the example above, not much information about the elements of the set is given to the client code of class *Set*.

To define an abstract state, one that can be used to verify client code and does not reveal the representation type (the internal structure of the ADT), we use *ghost* code. Ghost variables and methods are definitions that are used at verification time only. Ghost code is not included in the compiled output code.

Thus, we can define an abstract representation using Dafny's native set data structures, and use their declarative operations to specify our class methods. We need to define the abstract invariant based on an abstract state.

Consider the implementation below:

```
class Set {

    var store: array<int>;
    var nelems: int;

    ghost var s: set<int>;

    predicate RepInv()
    reads store, `store, `nelems
    {
        0 < store.Length
        ∧ 0 ≤ nelems ≤ store.Length
        ∧ ∀ i,j • 0 ≤ i < j < nelems ⟹ store[i] ≠ store[j]
    }

    predicate Sound()
        reads `store, store, `nelems, `s
        requires RepInv()
    { ∀ x • (x in s) ⟺ ∃ p • (0≤p<nelems) ∧ (store[p] = x) }

    predicate Valid()
        reads `store, store, `nelems, `s
    { RepInv() ∧ Sound() }

    // the construction operation
    constructor(n: int)
    requires 0 < n
    ensures AbsInv() ∧ s = {}
    {
        store := new int[n];
        nelems := 0;
        s := {};
    }

    // returns the number of elements in the set
    function method size():int
        requires AbsInv()
        ensures AbsInv()
        reads store, `store, `nelems, `s
    { nelems }

    // returns the maximum number of elements in the set
    function method maxSize():int
        requires AbsInv()
        reads store, `store, `nelems, `s
    { store.Length }

    method find(x:int) returns (r:int)
        requires AbsInv()
        ensures AbsInv()
        ensures r < 0 ⟹ ∀ j • (0≤j<nelems) ⟹ x ≠ store[j];
```

```
52          ensures r ≥0  ⟹  r < nelems ∧ store[r] = x;
53      {
54          var i:int := 0;
55          while (i<nelems)
56              decreases nelems-i
57              invariant 0≤i≤nelems;
58              invariant ∀ j • (0≤j<i)  ⟹  x ≠ store[j];
59          {
60              if (store[i]=x) { return i; }
61              i := i + 1;
62          }
63          return -1;
64      }
65
66      // checks if the element given is in the set
67      method contains(v:int) returns (b:bool)
68          requires AbsInv()
69          ensures AbsInv() ∧ b  ⟺  v in s
70      {
71          var i := find(v);
72          return i ≥ 0 ;
73      }
74
75      // adds a new element to the set if space available
76      method add(v:int)
77          requires AbsInv()
78          requires nelems < store.Length
79          ensures AbsInv() ∧ s = old(s) + {v}
80          modifies store, `nelems, `s
81      {
82          var f:int := find(v);
83          if (f < 0) {
84              store[nelems] := v;
85              nelems := nelems + 1;
86              s := s + {v};
87              assert ∀ i  •  (0 ≤ i < nelems-1)  ⟹  (store[i] = old(store[i]));
88          }
89      }
90  }
```

The abstract state is defined in line 6, by a set of integer values. The correspondence between abstract and concrete state is expressed by predicate `Sound`, lines 16-19, which states that all elements of the array are included in the set and vice-versa. Then, predicate `Valid` expresses that the abstract invariant comprises the representation invariant and its correspondence to an abstract state.

Notice that all functions manipulate the abstract state in sync with the concrete state. You can find that in lines 32 and 86. Notice also that the abstract state is used in the public specification of the ADT, in lines 28, 69, and 79.

In a last note, notice the **`assert`** in line 87, which is a lemma (intermediate proof) that Dafny needs to prove its pre-condition. As a rule of thumb, it is always good to ensure that Dafny is capable of *framing* the unchanged positions in arrays.

In subsequent lectures we will discuss the frame problem and framing in more detail, as well a general design pattern that we can use to specify an object's memory footprint while preserving modularity in verification.

References

[1] K. Rustan M. Leino. Developing verified programs with dafny. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 1488–1490. IEEE Press, 2013.

[2] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA, 1974. Association for Computing Machinery.