IP Lecture 12: Implementing Abstract Datatypes

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Implementing abstract datatypes

So far, we have used implementations of abstract datatypes from the Scala API, e.g. `HashSet` in `Dictionary`, and `Map` in `MapBook`. Now we will see various implementations of the phone-book trait `Book`.

- It's not that we can provide a *better implementation* than the Scala collections. . . we can't.
- This is to demonstrate data refinement.
- Sometimes we will have to provide our own implementation.

What does it mean for one of these implementations to be correct? Informally, the implementation should have externally visible behaviour that is consistent with the specification.

Reminder: the Book trait

```
/** The state of a phone book, mapping names (Strings) to
  * numbers (also Strings).
  * state: $book : \mathtt{String} \nrightarrow \mathtt{String}$
  * init: $book = \{\}$ */
trait Book{
  /** Store number against name in the phone book.
    * post: $book = book_0 \oplus \{name \to number\}$ */
  def store(name: String, number: String)

  /** The number stored against name.
    * pre: $name \in \mathrm{dom}\, book$
    * post: $book = book_0 \land$ returns $book(name)$ */
  def recall(name: String): String

  /** Does name appear in the book?
    * post: $book = book_0 \land$ returns $name \in \mathrm{dom}\, book$ */
  def isInBook(name: String): Boolean }
```

A different implementation of `Book`

We'll consider an implementation of `Book` based on storing the names and numbers in corresponding positions of two arrays. For example the abstract mapping

$$book = \{\text{“Gavin”} \to \text{“1234”}, \text{“Mike”} \to \text{“4567”}, \text{“Pete”} \to \text{“5443”}\}.$$

will be represented by the two arrays

$$\texttt{names} : \begin{array}{|c|c|c|c|}\hline Gavin & Mike & Pete & \quad\ldots\quad \\ \hline \end{array}$$

$$\texttt{numbers} : \begin{array}{|c|c|c|c|}\hline 1234 & 4567 & 5443 & \quad\ldots\quad \\ \hline \end{array}$$

or consistent permutations of these.

The arrays will be of some finite size `MAX`, so this places a restriction on the number of entries we can store.

The datatype invariant

We also need a variable, `count` say, that records the number of entries (3 in the above example), so we know to ignore the data in the arrays beyond this point. This will satisfy the property: $0 \leq$ `count` $\leq$ `MAX`.

Further, we do not want to record two different numbers against the same name. Hence we ensure that the entries in `names[0..count)` are distinct.

The above two properties represent a datatype invariant (DTI): a property that is true initially, and is maintained by each operation, and hence is true after each operation. Datatype invariants can be useful to help us come up with an efficient implementation. Datatype invariants should be stated explicitly and precisely.

The abstraction

The concrete state represents the mapping

$$\mathit{book} = \{\texttt{names}(0) \to \texttt{numbers}(0), \texttt{names}(1) \to \texttt{numbers}(1), \ldots, \\ \texttt{names}(\texttt{count} - 1) \to \texttt{numbers}(\texttt{count} - 1)\}$$

or more concisely

$$\mathit{book} = \{\texttt{names}(i) \to \texttt{numbers}(i) \mid i \in [0..\texttt{count})\}$$

The implementation will act on
$\{\texttt{names}(i) \to \texttt{numbers}(i) \mid i \in [0..\texttt{count})\}$ in the same way that the specification acts on the abstract state *book*.

The abstraction

We call this relation between the concrete and abstract states the abstraction function. More precisely, the abstraction function is the function abs that takes the concrete state c and gives back the corresponding abstract state a; we write $a = abs(c)$ in this case. (We'll only consider the case where this relationship is functional from the concrete state to the abstract state.)

The implementation of each function should change c so that $a = abs(c)$ changes in a way allowed by the abstract specification.

The state of the ArraysBook object

```
object ArraysBook extends Book{
  private val MAX = 1000 // max number of names we can store
  private val names = new Array[String](MAX)
  private val numbers = new Array[String](MAX)
  private var count = 0
  // Abs:  book = {names(i) → numbers(i) | i ∈ [0..count)}
  // DTI:  0 ≤ count ≤ MAX ∧
  //        ∀i, j ∈ [0..count)(i ≠ j ⇒ names(i) ≠ names(j))
  ...
}
```

Note that all the object's variables are private.

Stating the second clause of the DTI informally would have been fine.

Note that $\mathrm{dom}\, \mathit{book} =$ `names[0..count)`.

Some checks

We need to check that the abstraction function really does produce a result matching the type of the abstract state, i.e. a function of type $String \nrightarrow String$. This follows from the second clause of the DTI, that the names in `names[0..count)` are distinct.

We also need to check that the initial concrete state c_0 agrees with the abstract initialisation condition, i.e. $abs(c_0) = \{\}$. This is immediate since we set `count = 0`.

Finding a name

All the operations will involve searching for a particular name. We want to avoid repeated code, so let's encapsulate that searching in a private function `find`:

```
// Return the index i<count s.t. names(i) = name; or
//              return count if no such index exists
private def find(name: String) : Int = {
  // Invariant: name not in names[0..i) && i <= count
  var i = 0
  while(i < count && names(i) != name) i += 1
  i
}
```

The `isInBook` operation

The `isInBook` operation is then easy.

```
/** Is name in the book? */
def isInBook(name: String): Boolean = find(name) < count
```

Does this match the specification?

$$\textbf{post:}\ \ book = book_0 \land \textbf{returns}\ \ name \in \operatorname{dom} book$$

The `recall` operation

```
/** Return the number stored against name */
def recall(name: String) : String = {
  val i = find(name)
  assert(i<count)
  numbers(i)
}
```

Does this match the abstract specification?

pre: $name \in \operatorname{dom} book$

post: $book = book_0 \wedge$ **returns** $book(name)$

The `store` operation

```
/** Add the maplet name -> number to the mapping */
def store(name: String, number: String) = {
  val i = find(name)
  if(i == count){
    assert(count < MAX); names(i) = name; count += 1
  }
  numbers(i) = number
}
```

Does this match the specification?

post: $book = book_0 \oplus \{name \to number\}$

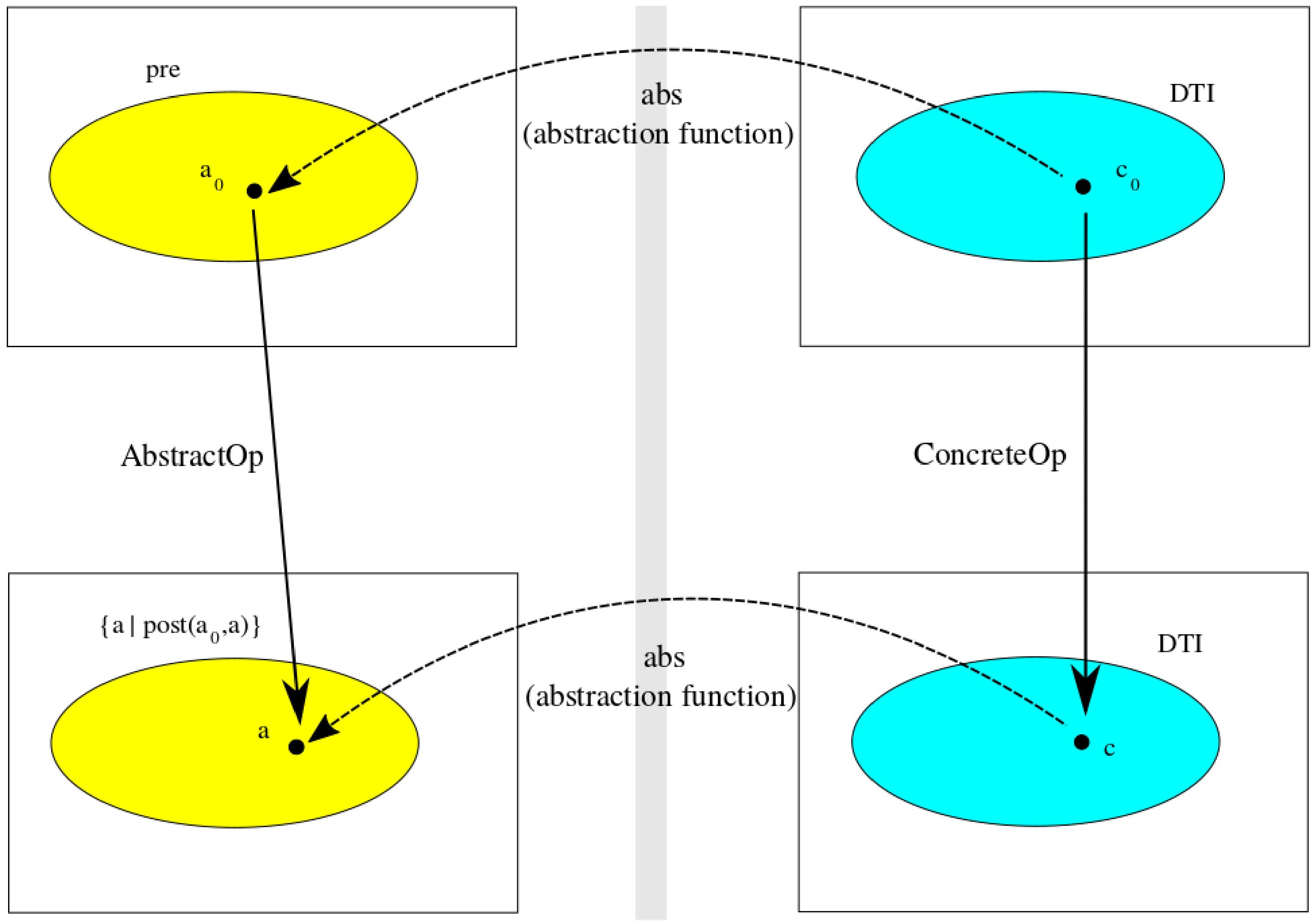
Correctness conditions

In general, to prove that a concrete implementation meets a specification over the abstract state space we need to proceed as follows.

Suppose that c_0 is an initial concrete state that satisfies the DTI, and suppose $a_0 = abs(c_0)$ satisfies the abstract precondition. Then the concrete operation must terminate (without error) in a concrete state c and return a result res such that:

- c satisfies the DTI
- the abstract postcondition is satisfied by a_0, $a = abs(c)$ and $abs(res)$ (where $abs(res)$ applies abs to each part of res that is of the type we are specifying).

A graphical representation



(Ignoring the returned value.)

Pre- and post-conditions

If the abstract operation has pre- and post-conditions

pre: $pre(a)$

post: **returns** res **s.t.** $post(a_0, a, res)$

then the concrete operation should have pre- and post-conditions

pre: $pre(abs(c)) \wedge DTI$

post: **returns** res **s.t.** $post(abs(c_0), abs(c), abs(res)) \wedge DTI$

Pre-conditions

Recall that the user of a function is responsible for ensuring that the pre-condition is satisfied.

If the pre-condition is not satisfied, then the code may do anything, including:

- giving a sensible answer;
- giving a non-sensible answer;
- throwing an exception;
- failing to terminate.

Which of these is best?

Criticism of `ArrayBook`

Our implementation stored the names and numbers in the order in which they are added. If we were to have a `delete` operation, then we would need to worry about closing gaps. (Exercise).

However, if we store the entries in order of the names, we can implement `find` using a binary search. (Exercise).

The datatype invariant between the pair of arrays requires that when we alter `names` then we have to operate on `numbers` in the same way. How about an array of pairs? Can we do better?

Abstract and concrete

Abstract:

- Abstract datatype (ADT) gives the generic description;
- Corresponds well with Scala `trait`;
- Preconditions and postconditions.

Concrete:

- Concrete datatype gives an implementation;
- Corresponds well with Scala `class`;
- Obeys preconditions and postconditions;
- Datatype invariant (DTI): how variables maintain consistency.

Connection:

- Abstraction function shows the correspondence. A function from the concrete implementation to the abstract state: $a = abs(c)$.

Summary

- Implementing abstract datatypes;
- Abstraction functions and datatype invariants;
- Correctness conditions.
- Next time: Other implementations of datatype.