IP Lecture 13: Refining Datatypes

Joe Pitt-Francis
—with thanks to Mike Spivey & Gavin Lowe—

Reminders from last time

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: $a = abs(c)$.

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:  $book = book_0 \wedge \mathtt{returns}\ \ book(name)$ */
  def recall(name: String): String

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

Two implementations of `Book`

1. `MapBook`. This object was just an adaptor to convert `Book` operations into operations on Scala library's own `Map` collection.

2. `ArraysBook`. This object was based on storing the names and numbers in corresponding positions of a pair of arrays of finite size `MAX`.

`names` :	*Gavin*	*Mike*	*Pete*	...
`numbers` :	1234	4567	5443	...

The datatype invariant included the variable `count` which denoted how many of the array slots were to be included in the abstract mapping.

Keeping the names and numbers is consistent order

Our implementation stored the names and numbers in the order in which they are added. However, if we store the entries in order of the names, we can implement `find` using a binary search.

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?

Pairs

Scala supports pairs, in a similar way to Haskell. If `T1` and `T2` are types then `(T1,T2)` (also written as `Tuple2[T1,T2]`) represents the type of all pairs `(x,y)` for `x` in `T1` and `y` in `T2`.

The elements of a pair can be extracted using the operators `_1` and `_2`, e.g.:

```
val x = p._1; val y = p._2
```

Alternatively, you can use pattern matching (but only with `val`s):

```
val (x,y) = p
```

Scala also supports tuples with more elements (up to 22).

We can produce a new implementation of `Book` that uses an array of pairs, instead of a “pair” of arrays.

An implementation using an array of pairs

```
object PairArrayBook extends Book{
  private val MAX = 1000 // max number of names we can store
  private val entries = new Array[(String,String)](MAX)
  private var count = 0
  // Abs:  $book = \{\texttt{entries}(i).\_1 \to \texttt{entries}(i).\_2 \mid i \in [0..\texttt{count})\}$
  // or Abs:  $book = \{x \to y \mid (x,y) \in \texttt{entries}[0..\texttt{count})\}$
  // DTI:  $0 \le \texttt{count} \le \texttt{MAX} \wedge$
  //        $\forall i,j \in [0..\texttt{count})(i \ne j \Rightarrow \texttt{entries}(i).\_1 \ne \texttt{entries}(j).\_1)$

  /** Return the index i<count s.t. entries(i)._1 = name;
    * or return count if no such index exists */
  private def find(name: String) : Int = {
    // Invariant: name not in { entries(j)._1 | j <- [0..i) }
    //            and   0 <= i <= count
    var i = 0
    while(i < count && entries(i)._1 != name) i += 1
    i
  }
```

Implementing the operations

```
/** Return the number stored against name */
def recall(name: String) : String = {
  val i = find(name)
  assert(i < count); entries(i)._2
}

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

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

Using special-purpose classes

The previous code was not as clear as it might have been: we have to remember that the first entry of each pair is the name, and the second entry is the number. If would be nicer if we could write, e.g., `entries(i).name` and `entries(i).number`. We could define a class

```
class Entry {
  var name : String
  var number : String
}
```

and store an array `entries` of `Entrys`. To store a new name and number we would do something like

```
  entries(i) = new Entry
  entries(i).name = name; entries(i).number = number
```

That's all a bit long-winded.

Class parameters

Scala allows classes to have class parameters[a]. For example

```
class Entry(name: String, number: String)
```

In this case, the class has an empty body, so we miss off the curly brackets.

We can provide values for these parameters when we create an object:

```
  entries(i) = new Entry(name, number)
```

(Note that we're using the identifier `name` for two different purposes: (1) the name of a field within `Entry` objects; (2) the name of the variable that we're storing in this `Entry`. This is a common style.)

[a] Java provides a more long-winded way of achieving the same thing, for those programmers who enjoy typing tedious boiler-plate code.

Class parameters

In fact, the fields of each `Entry` can be set only when the object is created, and can't be changed subsequently; the fields are effectively `val`s. We say that the object is immutable.

It is often more convenient to deal with immutable objects. They can safely be shared between different datatypes. If a mutable object is shared, then an update in one place can affect another place. Thus immutable objects are often easier to reason about.

If we do want to be able to update the fields we can mark them as `var`s:

```
class Entry(var name: String, var number: String)
```

Case classes

We can define `Entry` as a case class as follows:

```
case class Entry(name: String, number: String)
```

This offers various advantages:

- We can omit the word `new` when constructing new objects, e.g., `entries(i) = Entry(name, number)`.
- The compiler provides "natural" definitions of operators `==`, `toString` and `hashCode`.
- We can perform pattern matching; e.g.

  ```
  val Entry(theName, theNumber) = entries(i)
  ```

  which binds `theName` and `theNumber` to the appropriate fields.

The cost is that the classes and objects are slightly larger. (See Chapter 15 of Programming in Scala for more about case classes.)

A phone book using the Entry class

```
object EntryArrayBook extends Book{
  private val MAX = 1000 // max number of names we can store
  private case class Entry(name: String, number: String)
  private val entries = new Array[Entry](MAX)
  private var count = 0
  // Abs: book =
  //  { entries(i).name -> entries(i).number | i <- [0..count) }
  // ={ name -> number | Entry(name,number) in entries[0..count)}
  // invariant: 0 <= count <= MAX &&
  // values of entries(i).name distinct, for i in [0..count)

  ...
}
```

Implement the operations (`find`), `store`, `recall` and `isInBook`.

Linked lists

The array-based implementations of the phone book that we've seen so far have only been able to deal with a bounded number of names and numbers. (In principle, if the arrays become full, we could create new bigger arrays and copy the data across.) Also, they waste space when the arrays are only partially full.

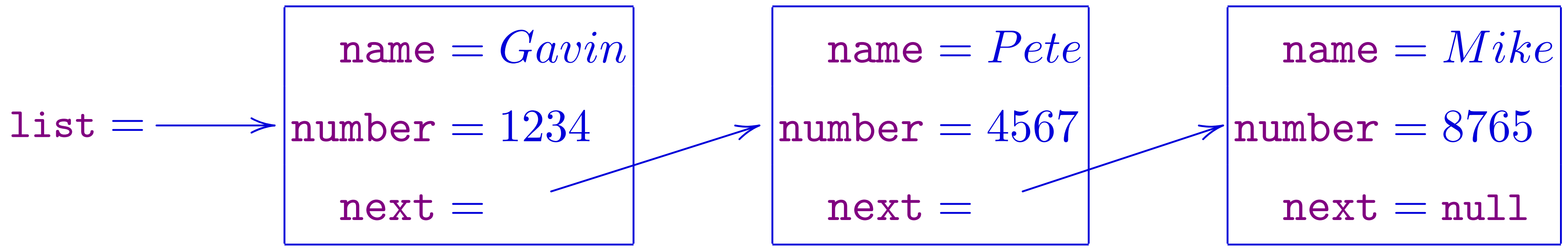
Next we'll see a different implementation that overcomes these shortcomings: the data structure will grow as required to store the data.

The implementation will use several "nodes", each containing a name and number, and linked to the next node.

We will implement a class, each object of which will represent a phone book, encapsulating such a linked list.

Linked lists

A linked list can be pictured as follows.



Each `next` field is a reference to the next node in the list. A reference simply records where the node is stored in memory, plus some typing information. (In some languages, references are called pointers.)
There is a special reference `null` which indicates that there is no next node.

Also there is a program variable `list` which holds a reference to the first node in the list.

Objects and references

All program variables that correspond to objects actually store references to those objects. For example, after the declaration `val xs = List(1,2,3)`, the variable `xs` stores a reference to the place in memory where the `List` is stored. If `xs` is passed to a function, in fact it is the reference that is passed, rather than the whole list; this requires far less copying of data.

Scala automatically follows the reference where appropriate, so the referencing is transparent.

However, if a reference variable holds the value `null`, so doesn't point to a real object, then calling an operation on that reference is an error (it will give a `NullPointerException`).

Defining linked lists

We can define nodes of a linked list as follows

```
class Node(var name: String, var number: String, var next: Node)
```

Each node contains a reference to the next, so collectively they represent a list.

Representing the phone book using a linked list

We can create an implementation of the phone book by encapsulating a linked list.

```
class LinkedListBook extends Book{
  private var list : Node = null
  // list represents the mapping composed of (n.name -> n.number)
  // maplets, when n is a node reached by following 0 or more
  // next references.
  // Abs: book = { n.name -> n.number | n <- L(list,null) }
  ...
}
```

The abstraction function

Informally, each `LinkedListBook` object represents the mapping composed of $n.\texttt{name} \rightarrow n.\texttt{number}$ maplets, where n is a node reached by following 0 or more `next` references from `list`.

More formally, let's define the list of nodes reachable by following `next` references, starting from a, and continuing up to but not including b (using Haskell-style list notation).

$$\begin{aligned} L(a,b) = {} & [\,], && \text{if } a = b, \\ & a : L(a.\texttt{next}, b), && \text{if } a \neq b. \end{aligned}$$

We will sometimes abbreviate $L(a,\texttt{null})$ to $L(a)$, which could be defined:

$$\begin{aligned} L(\texttt{null}) &= [\,], \\ L(a) &= a : L(a.\texttt{next}), \quad && \text{if } a \neq \texttt{null}. \end{aligned}$$

The abstraction function and datatype invariant

Then the abstraction function is

Abs: $book = \{n.\texttt{name} \to n.\texttt{number} \mid n \in L(\texttt{list})\}$

Our intention is that the lists are finite, which implies that they are acyclic; and that names are not repeated:

DTI: $L(\texttt{list})$ is finite, and the names in $L(\texttt{list})$ are distinct

Finding a name

For each operation, we need to search to see if the given name is already in the list. So let's encapsulate this in a function. The function traverses the list, comparing the name in the node with the name we're looking for.

```
/** Find the node containing name, if it exists
  * Post: book = book_0 && returns n s.t.
  *       (n in L(list) && n.name=name)
  *       or n=null if no such Node exists */
private def find(name: String) : Node = {
  var n = list
  // Invariant: name does not appear in the nodes before n:
  // for all n1 in L(list,n), n1.name != name
  while(n != null && n.name != name) n = n.next
  n
}
```

isInBook and recall

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

/** Return the number stored against name */
def recall(name: String) : String = {
  val n = find(name); assert(n!=null); n.number
}
```

`store`

```
/** Add the maplet name -> number to the mapping */
def store(name: String, number: String) = {
  val n = find(name);
  if(n==null){
    val n1 = new Node(name, number, list)
    list = n1
  }
  else n.number = number
}
```

In `store`, if the name is new, we add the new name to the front of the list. The code for this could be shortened to `list = new Node(name, number, list)`.

Summary

- Array of pairs;
- Array of objects;
- Class parameters;
- Case classes;
- Linked lists;
- References; `null`;
- Next time: Programming with linked lists.