Elements of Programming Languages
Lecture 9: Programs, modules and interfaces

James Cheney
University of Edinburgh
October 19, 2017

Overview

- So far we have covered programming “in the small”
  - simple functional programming
  - abstractions: parametric polymorphism and subtyping
- Next few lectures: programming “in the large”
- Today
  - “Programs” as collections of definitions
  - Namespace management — packages
  - Abstract data types — modules and interfaces
- We will mostly work “by example” using Scala — formalizing modules, interfaces involves a lot of bureaucracy.

Programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Namespaces and Packages</th>
<th>Modules and Interfaces</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Declarations and Programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Namespaces and Packages</th>
<th>Modules and Interfaces</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- What is a program?
  - In \( \text{L}_\text{Poly} \), a program is an expression; any functions defined in \( \text{L}_\text{Poly} \) are local to the expression
    
    \[
    \text{let fun } f(x : \tau) = e_1 \text{ in } \\
    \text{let fun } g(y : \tau') = e_2 \text{ in } \\
    : \\
    e
    \]

- Scope management is easier with these simplistic forms, but isn’t very modular
- In particular, we can’t easily split a program up into parts that do unrelated work.

- Most languages support \textit{declarations}
  \[
  \text{Decl} \supset d \ ::= \text{let } x = e; | \text{let fun } f(y : \tau) = e; \\
  | \text{let rec } f(y : \tau) : \tau' = e; \\
  | \text{type } T = \tau; | \text{deftype } T = \tau;
  \]

- A \textit{program} is a sequence of declarations. The names \( x, f, T \) are in scope in the subsequent declarations.
  - Variation: In some languages (Haskell, Scala), the order of declarations within a program is unimportant, and names can be referenced before they are used.
  - Variation: In some languages, only certain “top-level” declarations are allowed (e.g. classes/interfaces in Java)
**Entry points**

- The *entry point* is the place where execution starts when the program is run

```
public static void main(String[] args) {...}
```

- Can be specified in different ways:
  - Executable: specify a particular function that is called first (e.g. `main` in C/C++, Java, Scala)
  - Scripting: entry point is start of program, expressions or statements run in order
  - Web applications: entry points are functions such as `doGet`, `doPost` in Java's `Servlet` interface
  - Reactive: provide *callbacks* to handle one or more *events* (e.g. JavaScript handlers for mouse actions)

**Programming in the large**

- What is the largest program you've written (or maintained)?
  - 1000 lines — 1 file?
  - 10,000 lines? 10 files?
  - 100,000 lines? 100 files?

- Sooner or later, someone is going to want to use the same name for different things.

- If there are \( n \) programmers, then there are \( O(n^2) \) possible sources of name conflicts.

**Namespaces** provide a way to compartmentalize names to avoid ambiguity.

**Example: Packages in Java**

```java
// com/widget/round/Widget.java
package com.widget.round
class Widget {...}

// com/widget/square/Widget.java
package com.widget.square
class Widget {...}
```

- We can reuse Widget and disambiguate:
  
  ```java
  com.widget.square.Widget vs. com.widget.round.Widget
  ```

  *(Package names track the directory hierarchy in Java.)*

**Importing**

- Given a namespace, we can *import* it
  
  ```java
  import com.widget.round.Widget
  ```

  - This brings a *single* name defined in a namespace into the current scope

  ```java
  import com.widget.round.*
  ```

  - This brings *all* names defined in a namespace into the current scope

- In Java, importing can only happen at the top level of a file, and imported names are always classes or interfaces.

  - (Scala is more flexible, as we'll see)
Another important concern for programming in the large is **code reuse**.

We’d like to implement (or reuse) certain key data structures once and for all, in a **modular way**

- Examples: Lists, stacks, queues, sets, maps, etc.

An abstract data type (ADT) is a type together with some operations on it

- Abstract means the type definition (and operation implementations) are not visible to the rest of the program
- Only the types of the operations are visible (the interface)
- An ADT also has a specification describing its behavior

### Running example: priority queues in Scala

Using Scala objects, here is an initial priority queue ADT:

```scala
object PQueue {
  type T = ...
  val empty: T
  def insert(n: Int, pq: T): T
  def remove(pq:T): (Int,T)
}
```

(Similar to Java class with only static members)

Specification:
- A priority queue represents a set of integers.
- `empty` corresponds to the empty set
- `insert` adds to the set
- `remove` removes the least element of the set

### Implementing priority queues

One implementation: sorted lists (others possible)

```scala
object ListPQueue {
  type T = List[Int]
  val empty: T = Nil
  def insert(n: Int, pq: T): T = pq match {
    case Nil => List(n)
    case x::xs =>
      if (n < x) {n::pq} else {x::insert(n,xs)}
  }
  def remove(pq:T) = pq match {
    case x::xs => (x,xs) // otherwise error
    // otherwise error
  }
}
```

### Importing

Like packages, objects provide a form of namespace

```scala
object ListPQueue {
  ...
}
val pq = ListPQueue.insert(1,ListPQueue.empty)
import ListPQueue._
val pq2 = remove(pq)
```

Importing can be done inside other scopes (unlike Java)

```scala
def singleton(x: Int) {
  import ListPQueue._
  insert(x,empty)
}
```
ListPQueue isn't abstract

- If we only use the ListPQueue operations, the specification is satisfied.
- However, the ListPQueue.T type allows non-sorted lists.
- So we can violate the specification by passing remove a non-sorted list!

```scala
remove(List(2,1))  // returns 2, should return 1
```

- This violates the (implicit) invariant that ListPQueue.T is a sorted list.
- So, users of this module need to be more careful to use it correctly.

One solution (?)

- As in Java, we can make some components private.

```scala
object ListPQueue {
    private type T = List[Int]
    private val foo: T = List(1)
}
```

- This stops us from accessing foo.

```scala
scala> ListPQueue.foo
<console>:20: error: (foo cannot be accessed)
```

- However, T is still visible as List[Int]!

```scala
scala> ListPQueue.remove(List(2,1))
res10: (Int, List[Int]) = (2,List(1))
```

Interfaces

- Another way to hide information about the implementation of a module is to specify an interface.

```scala
trait PQueue {
    type T = List[Int]
    def empty: T
    def insert(n: Int, pq: T): T
    def remove(pq: T): (Int,T)
}
```

- Scala doesn't exactly have Java-like interfaces, but its traits can play a similar role.

```scala
trait PQueue {
    type T = List[Int]
    val empty: T
    def insert(n: Int, pq: T): T
    def remove(pq: T): (Int,T)
}
```

- (We'll say more about why Scala uses the terms object and trait instead of module and interface later...)
Implementing an interface

- Already, the trait interface hides information about the implementations of the operations. But, now we can go further and hide the definition of T!

```scala
trait PQueue {
  type T // abstract!
}
```

- Now we can specify that ListPQueue implements PQueue using the extends keyword:

```scala
object ListPQueue extends PQueue {...}
```

- This assertion needs be checked to ensure that all of the components of PQueue are present and have the right types!

Checking a module against an interface

```scala
trait PQueue {
  type T
  val empty: T
  def insert(n: Int, pq: T): T
  def remove(pq: T): (Int, T)
}
```

- An implementation needs to define T to be some type τ
- It needs to provide a value empty: τ
- It needs to provide functions insert and remove with the corresponding types (replacing T with τ)
- If any are missing or types don’t match, error.
- (Note: this is related to type inference, and there can be similar complications!)

Interfaces allow multiple implementations

- We can now provide other implementations of PQueue

```scala
object ListPQueue extends PQueue {...}
object SetPQueue extends PQueue {...}
```

- Also, in Scala, objects can be passed as values, and extends implies a subtyping relationship
- So, we can write a function that uses any implementation of PQueue, and run it with different implementations:

```scala
def make(m: PQueue) =
  m.insert(42, m.insert(17, m.empty))
scala> make(ListPQueue)
```

Data abstraction

- Even though ListPQueue satisfies the PQueue interface, its definition of T = List[Int] is still visible
- However, T is abstract to clients that use the PQueue interface
- So, we can’t do this:

```scala
scala> def bad(m: PQueue) = m.remove(List(2,1))
<console>:18: error: type mismatch;
  found   : List[Int]
  required: m.T
```

```scala
def bad(m: PQueue) = m.remove(List(2,1))
```
Implementing multiple interfaces

- An interface gives a “view” of a module (possibly hiding some details).
- Modules can also satisfy more than one interface.

```scala
trait HasSize {
  type T
  def size(x: T): Int
}
object ListPQueue extends PQueue with HasSize {
  ...  
  def size(pq: T) = pq.length
}
```

(This is slightly hacky, since it relies on using the same type name T as PQueue uses. We’ll revisit this later.)

Representation independence

- If we have two implementations of the same interface, how do we know they are providing “equivalent” behavior?
- **Representation independence** means that the clients of the interface can’t distinguish the two implementations using the operations of the interface
  - (even if their actual run time behavior is very different)
- This is much easier in a strongly typed language because the abstraction barrier is enforced by type system
- In other languages, client code needs to be more careful to avoid depending on (or violating) intended abstraction barriers

Modules and interfaces, in general

**Decl ⊨ d ::=**

- `let x = e;`
- `let fun f(x : τ) = e;`
- `let rec f(x : τ) : τ' = e;`
- `type T = τ;`
- `deftype T = τ;`
- `module M {d₁ · · · dₙ} | import q`
- `interface S {s₁ · · · sₙ}`

**Spec ⊨ s ::=**

- `val x : τ;`  
- `type T;`  
- `type T = τ;`

**QName ⊨ q ::=**

- `x | M.q | S.q`  

This a simplified form of the (influential) Standard ML module language. (We aren’t going to formalize the details.)

Note: Allows arbitrary nesting of modules, interfaces

Not shown: need to allow qualified names in code also

Summary

- As programs grow in size, we want to:
  - split programs into components (packages or modules)
  - use package or module scope and structured names to refer to components
  - use interfaces to hide implementation details from other parts of the program
- We’ve given a high-level idea of how these components fit together, illustrated using Scala
- Next time:
  - Object-oriented constructs (objects, classes)