CSci 2041: Advanced Programming Principles Techniques for Modular Code Development

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

Modularity: What and Why

Modularity concerns conceiving of and building large programs through small, independent but interacting components

Some reasons why this is an important notion in programming

- It breaks up a complex task into smaller, coherent parts, thereby making it more manageable
- It provides a structure for a team to work on a project
- It allows us to identify code with more limited functionality, thereby facilitating reuse
- It provides a structure for easily upgrading code

Our objective: To understand something about such structure by seeing how it is specifically supported in OCaml

The Issues Concerning Modularity

At a linguistic level, the issues concern how the ability to construct programs modularly is supported, i.e.

- What facilities are available for constructing smaller, insulated parts?
- What mechanisms are provided for eventually authenticating the functionality realized by these parts?
- What mechanisms exist for abstracting away (or hiding) details of how the functionality is realized?
- What mechanisms are available for realizing interactions between these parts?

We will understand these notions by looking at how they are supported in OCaml

How exactly to exploit these capabilities in a large programming task may be undertaken in CSci 3081

Support for Modularity in OCaml

A birds eye-view of how modularity is realized in OCaml

- Facilities for constructing smaller, insulated parts?

  encapsulating code in *modules* or *structures*

- Mechanisms for authenticating the functionality of parts?

  identifying interfaces in the form of *signatures*

- Mechanisms for abstracting away (or hiding) details?

  using signature qualifications to control visibility

- Mechanisms for realizing interactions between parts?

  Allowing access to visible components using “long names”

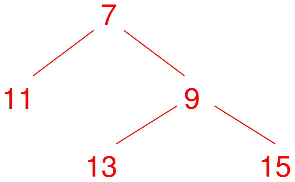
  Providing *functors* for building *parameterized modules*, i.e. for *composing* modules

An Example to Illustrate the Modularity Mechanisms

Consider the implementation of a *heap*

> *A binary tree structure where the root item is smaller than the items in the left and right subtrees*

For example, an integer heap:



To realize such a heap notion we need

- a type declaration for trees
- operations such as `top`, `maxHeap` and `isHeap`

Moreover, we would like an encapsulation of these definitions

Modules or Structures in OCaml

An *objectification* and *naming* of a collection of declarations

The simplest form for such a declaration

```
module <StrName> = <StrObj>
```

where `<StrObj>` has the form

```
struct
      {type, value, function
       exception, etc declarations}
end
```

In other words, surrounding code with `struct` and `end` objectifies it and the module declaration gives it a name

A (Skeleton) Heap Structure in OCaml

```
module IntHeap =
struct
  type item = int

  type tree = L of item | N of item * tree * tree

  let top = function
    | L i -> i
    | N (i,_,_) -> i

  let leq((p:item),(q:item)):bool = p <= q

  let rec isHeap = function
    | L _ -> true
    | N (i,l,r) -> leq (i,top l) && leq(i,top r)
                        && isHeap l && isHeap r

  let max(p,q) = if leq (p,q) then q else p

  let rec maxHeap = function
    | L i -> i
    | N (_,l,r) -> max(maxHeap l, maxHeap r)
end
```

Using the Components of a Structure

Once a structure is present in the environment, objects defined within it can be referred to

The syntax for accessing such components:

```
<struct-name> . <obj-name>
```

For example, relative to `IntHeap` we can write

```
IntHeap.leq(5,7)
```

As an aside, this is what we have been doing with library modules, e.g. when we use `List.map`

Signatures in OCaml

These are the "types" associated with structures

In particular, a signature identifies the structures, types, value and function objects defined within a structure

For example, OCaml infers the following signature for `IntHeap`:

```
module IntHeap :
  sig
    type item = int
    type tree = L of item
              | N of item * tree * tree
    val top : tree -> item
    val leq : item * item -> bool
    val isHeap : tree -> bool
    val max : item * item -> item
    val maxHeap : tree -> item
  end
```

Defining Signatures

Signatures can also be objectified and named in OCaml

The syntax for realizing such objectification and naming

```
module type <Sig-Name> =
         sig <component-decls> end
```

An example of such a declaration

```
module type INTHEAP =
sig
  type tree
  val isHeap : tree -> bool
  val maxHeap : tree -> int
  val top : tree -> int
end
```

This signature is “coarser” than the one inferred for `IntHeap`

Typing Structures with Signatures

Structures can be identified with particular signatures

For example, `IntHeap` could have been defined as

```
module IntHeap : INTHEAP =
struct
      ...
end
```

Notice that a signature can be less specific than what is contained in the structure

In this case, the signature plays two roles:

- it determines what the structure *must* define
  (Enforced via “type checking”)
- it restricts what is accessible from outside
  (Encapsulation, information hiding)

Signature Matching and Opacity

Suppose we exposed the *tree* type in the heap signature but kept the *item* type abstract:

```
module type HEAP =
sig type item
    type tree = L of item | N of item * tree * tree
      ...
end
```

Then consider

```
  module IntHeap : HEAP =  struct ... end
```

The upshot would be that we can use constructors such as `IntHeap.L` but we cannot construct any trees/heaps!

```
# let x = IntHeap.L 3;;
                    ^
Error: This expression has type int but an expression
          was expected of type IntHeap.item
#
```

Signature Matching with Constraints

We could solve the problem by dropping the signature qualification

However, this is not ideal: we do want the checking provided by signature matching

OCaml provides a better solution by allowing signature matching to be qualified by constraints

An example of the use of such a constraint

```
module IntHeap : (HEAP with type item = int) =
  struct ... end
```

The constraint ensures that the `item` type is the same as `int` and it also exposes this outside the module

```
# let x = IntHeap.L 3;;
val x : IntHeap.tree = IntHeap.L 3
#
```

Parameterizing Modules

What we might actually want is a “heap structure generator”

Such a generator would be parameterized by an “item” structure with signature

```
signature ITEM =
  sig
    type item
    val leq : item * item -> bool
  end
```

Heap structures can then be identified by a function that

- takes structures satisfying `ITEM` signature
- produces structures satisfying `HEAP` signature

OCaml supports structure parameterization through *functors*

Functors in OCaml

For example, for heaps we could define

```
module Heap (Item : ITEM) :
        (HEAP with type item = Item.item) =
  struct
     type item = Item.item
     fun leq(p:item,q:item) = Item.leq(p,q)
        ...
  end
```

Particular heap structures are generated by *functor application*:

```
module IntHeap = Heap(IntItem)
module StringHeap = Heap(StringItem)
  ...
```

Notice that functor application is carried out by

- checking "interface constraint" satisfaction
- performing necessary linking

Modules, Files and Separate Compilation

When we build a large system, we often want to be able to compile different conceptual pieces of it separately

To allow for this, OCaml provides a default reading of the code in files as modules

- Code created in two files with names like `A.mli` and `A.ml`
- Compiling them essentially amounts to compiling

```
module A : (sig (* contents of A.mli *) end) =
  struct
       (* contents of A.ml *)
  end
```

- When you use the code in such files in other places, you can refer to the components as `A.<comp-name>`

For more details, look up the module system page in the OCaml manual

Also check out `ocamlbuild` for organizing large systems

Modularity More Generally

- Whether or not the language gives you explicit capabilities, try to break big programs up into modular pieces

- Clearly identify an interface for each piece and try to make it explicit in the program

  For example, for C programs, use the `.h` file for this purpose

- Think of ways in which you can check satisfaction of functionalities before putting the whole together