S2.1: Modular Programming in OCaml

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

"Programming in the large"

- The notion of "programming in the large" is related to organizing large applications into separate components, each of manageable size.
- These components are often called **modules**.
- This is to provide boundaries between components
  - to minimize redundancy
  - to maximize opportunities for code re-use
  - so that they can be worked on separately by different developers
  - so that one component can be replaced with another that has the same functionality, but perhaps a faster implementation

Read Chapter 5 in the Cornell text book.

Methodologies

Methodologies ask how best to decompose a problem into components or modules.

Often done with the above "programming in the large" goals in mind.

Mechanisms

What programming language features or mechanisms support or enable breaking applications into modules?

These mechanism often focusing on ways to enforce types of correctness, safety, information hiding, separate compilation, etc.

We will focus on mechanisms.

"Software Engineering" focuses, to some extent, on methodologies.

Modules

In many languages, a **module** is a collection of types and values defined so that they may be used as a component by other components of the program.

The kinds of types (inductive types, classes, etc) and the kinds of values (functions, objects, etc.) differ by language.

But the notion of exporting some collections of named entities (types or values) is consistent.

Internal and External References

How do we refer to a type, function, or value from inside the module in which it is defined?

- Use its "internal reference"

How do we refer to a type, function, or value from outside the module in which it is defined?

- Use its "external reference"

OCaml modules

- Read Chapter 5 in the Cornell text book linked from the `README.md` file in `Course-Resources` directory of the public class repository.
- OCaml and Standard ML have rather sophisticated module systems.
- It is the “gold-standard” for module systems.

“Running” OCaml programs

- One motivation for modules is to be able to write “big” programs.
- These are often programs that interact with the user in some way, perhaps by reading input from a keyboard and printing text to the screen.
- These are programs that we will compile and then run.
- While ‘utop‘ and similar tools are useful for development, we don’t expect actual users to interact with our software using these tools.
- So we start by saying a few things about I/O and compilation.

Compilation

- You’ve compiled OCaml programs before.
- In Hwk 02, you were asked to run these commands

  `ocamlbuild hwk_02.byte` — to compile the code

  and then

  `./hwk_02.byte` — to run it
- But this didn’t “do” anything.
- On the other hand

  ```
  ocamlbuild hwk_02_tests.byte
  ./hwk_02_tests.byte
  ```

  did generate some output.

Side-effects in a functional language

- We've said that "expression evaluation" is how behavior is defined in functional programs.
- But how do we "do something"? Like print a result to a screen or ask the user for input?
- Consider `say_hello.ml`

The `unit` type

- What should the type of `print_endline` be?
- Recall: It is `string -> unit`.
- Recall: The only value of type `unit` is `()`.
- Since `unit` has only one value, it doesn't convey any interesting information.

  At compile time we already know what the value of an expression of type `unit` will be!

- What is the type of `read_line`?

  It should give us something back, but what does it take as input?
- The type of `read_line` is `unit -> string`.

  We give it `()` only to indicate that it is being called.
- We don't need names for values of type `unit`, but can use `_` instead.
- We can use `let`-expressions to control the order in which functions that rely on side-effects are evaluated.
- We see all of this in the samples in `say_hello.ml` and in the testing code from Hwk 02.

Modules

A module in OCaml is a collection of

- value declarations: often functions, sometimes strings, integers, or values for empty data structures
- type declarations: often for the data structure defined in the module.

Inside a module, we can refer to the values and types directly by name.

Outside of a module, we can refer to a type or value by using the module name, a dot, and the internal name.

For example, a module defining a stack data structure might define

- values for pushing and popping a stack
- the value of the empty stack
- the type used to represent stacks

We may write `push` inside the module, but `Stack.push` outside of the module.

Stacks

- We'll consider some some example uses of stacks in a stack-machine for evaluating simple arithmetic expressions.
- These files are in the `StackMachine` directory in the `Sample-Programs` directory of the public class repository.
  - `Monolithic`: all code in one files
  - `Modularity_via_Files`: only uses the file-as-module notion of modules
  - `Modularity_via_Modules`: uses modules and signatures language constructs
  - `Modularity_via_Functors`: uses functors for parameterizing modules

The `Monolithic` example

- Here, all the code is in one file.
- We look at this just to understand the problem.
- Expressions are represented by the `expr` type.
- Stacks are represented as lists.
- Expressions can be evaluated directly or "compiled" into as list of instructions.

  Running these instructions uses the stack to store intermediate values.

The `Modularity_via_Files` example

Recall our earlier definition of module:

> A module in OCaml is a collection of
> - value declarations: often functions, sometimes strings, integers, or values for empty data structures
> - type declarations: often for the data structure defined in the module.

- An OCaml file satisfies this definition.
- Thus, files are modules.
- The name of the module is the capitalized file name.
- We can `open` a module to avoid the sometimes cumbersome use of the longer external names of module elements.

The example in `Modularity_via_Files` demonstrates this.

Signatures to ensure abstraction

- In `Modularity_via_Files`, the file `expr.ml` does not make use of the fact that stacks are implemented as lists.
- In fact, we want to prevent this.
- The stack data structure should be kept abstract.

  The "representation type" of the data type should not be accessible outside of the module.
- Signatures define the interface of a module. They determine what is accessible from outside of the module.
- Users of the module can only use what is given in the signature.

Modules implement signatures

- One way to write signatures for a module is in a file with the same name, but with a `mli` file extension.
- A module implements a signature if
  - if provides definitions for all the declarations in the signature
  - each definition corresponds to the type given in the signature
- See `listStack.mli` for this.
- We will use the terms "signature" and "interface" interchangeably.

A problem with files as modules

- We now consider a different implementation of the stack interface defined in `listStack.mli`.
- Note that the list representation type is not visible in the signature.
- By using file names to bind modules (*e.g.* `listStack.ml`) to the interfaces they must implement (*e.g.* `listStack.mli`)

  we cannot create another implementation (a module) for the same signature.
- We need an explicit way to say "this module implements that signature" and using file names does not allow us to do this.
- We might like to create a second stack implementation module that also implements the same stack signature.

  (In fact, we'll do this in the next example `Modularity_via_Modules`.)

Modules inside files: the `Modularity_via_Modules` example

- We can declare modules and signatures inside files.
- This avoids the problem mentioned above of binding module names and signature names to files. Thus, provides more flexibility.
- The syntax for module definitions is:

  `module` ⟨*name*⟩ `:` ⟨*signature*⟩ `= struct` ⟨*definitions*⟩ `end`

  The `:` ⟨*signature*⟩ part is optional.
- The syntax for signature definitions is:

  `module type` ⟨*name*⟩ `= sig` ⟨*declarations*⟩ `end`
- Modules typically use the name of a signature, but can also write a signature name directly between the `:` and `=`.

Some naming conventions

We adopt some naming conventions to avoid confusion and verbosity.

- A `stack` type implemented in a `Stack` module in a file named `stack.ml` would be referenced as `Stack.Stack.stack`.
- This is not so nice.
- The primary type in a signature is often named `t`, for "type".
- Modules inside a file are given a suffix of `M`.
- Signatures inside a file are given a suffix of `S`.
- By opening file-modules, we get more reasonable names

```
open Stack

let s : StackM.t = ...
```

- Let's see how this plays out in example in `Modularity_via_Modules`.

Using multiple stack implementations

There are two things we'd like next.

1. We'd like to have another implementation of stacks.

   We'll create `customStack.ml` with another module that implements the `StackS` signature.

   We know how to do this.

2. We'd like to be able to easily switch one implementation for another in the `expr` code.

   We can change the definition of `S` at the top of `expr.ml`.

Parameterized modules

- Changing stack implementation by changing the code in `expr.ml` is clumsy.
- But there is something worse that this clumsiness!
- We might like users of our `Expr` module to be able to pick that stack implementation that works best for their application.

  And they should be able to do this without modifying the source code of `expr.ml`.
- For this we need something new.
- We now turn to functors.

  These are essentially parameterized modules.

Functors

- Functors are modules that take modules as arguments.
- The syntax for a functor definition is:

  `module` $\langle functorName \rangle$ `(` $\langle moduleName \rangle$ `:` $\langle signature \rangle$ `) = struct` ... `end`
- Functor application is the creation of a module by applying a module to a functor.

  It is giving a module to a functor, so that is can return a module.
- The syntax for functor application is:

  `module` $\langle name \rangle$ `=` $\langle functorName \rangle$ `(` $\langle modulename \rangle$ `)`

The `Modularity_via_Funtors` example

- Examples of functor definition and application are given in the `Modularity_via_Functors` directory in the `StackMachine` directory.

- In `expr.ml` we see the `ExprF` functor.

- In `go.ml` we apply (or instantiate) the functor with the desired stack implementation.

- We also have a signature for the result of the `ExprF` functor in order to limit what contents can be seen from outside of the functor.
  - We will expose the `expr` type so that expressions can be constructed.
  - We will hide the implementation of the `instr` type so that the instruction values are not directly visible.

    We will only be able to create instructions by compiling an expression.

    We can only use them in the `run` function.

The OCaml compiler

- `ocamlbuild` locates all referenced modules, compiles them, and links the generated code together.

- `ocamlbuild go.byte` does this and generates a byte-code executable that runs on the OCaml virtual machine.

  This is the same idea as used in Java with the Java Virtual Machine.

  This create `go.byte` which can be run.

- `ocamlbuild go.native` will generate machine code that runs directly on the machines processor.

  This create `go.native` which can be run.

  This compilation may take longer but generates quite efficient code.

- `ocamlbuild -clean` removes all generated files in the `_build` directory and any `.byte` or `.native` files.
  These files should not be committed to GitHub.

Tech Tip: a detour

- How can we manage files in `git` that we want `git` to ignore.

  `git` should not tell us that these files haven't been added and it should not let us easily add them.

  We want `git` to just ignore our `_build`, `*.byte`, and `*.native` files.

- To do this, we put these patterns in a file named `.gitignore` at the root of your `repo-user1234` individual repository.

  This file might contain these lines:

  ```
  # OCaml files to ignore
  *.byte
  *.native
  _build
  ```

- Look at `.gitignore` in the `public-class-repo` repository as another example.

"Programming in the large and small"

"in the small"

- expressions have types
- expressions denote a value
- functional value and function application

"in the large"

- modules have signatures
- modules have an implementation
- functors and functor application

Using modules in utop

- We can use modules inside utop and when using the OCaml compilers.
- In utop, we need to first use the `#mod_use` directive to use a file as if it were a module.
  For example:
  - `#mod_use "stack.ml";;`
  - `#mod_use "listStack.ml";;`
  - `#mod_use "customStack.ml";;`
  - `#mod_use "expr.ml";;`
- Next, `#use` a file that refers to these module elements using their external names.

  For example, `#use "go.ml"`

Visibility of type in modules

- A transparent module exposes implementations of all of its types.
  We saw this in `Modularity_via_Files`
- A translucent module exposes implementations of some of its types.
  In `Modularity_via_Functors`, the `ArithmeticS` signature is translucent since some of its type implementations are exposed.
- A opaque module exposes implementations of none of its types.
  In `Modularity_via_Functors`, the `StackS` signature is opaque since none of its type implementations are exposed.
- To support "representational independence" the implementation of the type must be hidden, that is, abstract.

  This is what we saw in the stack type in the `StackMachine` examples so far.

Extending or modifying signatures

- We now consider a different example in which we want to expose a type that a signature has hidden.

  Importantly, we will see that the signature had no choice but to hide this type.

  It will be instantiated in different ways by a functor.
- The example is for an interval data type in `Intervals`.
  - `v1`: modules as files, integer intervals
  - `v2`: modules as files, with signatures
  - `v3`: modules **in** files - we start here
  - `v4`: functors for intervals over different types
  - `v5`: adding signatures, a resulting type error
    Note the `Int_interval` signature in `utop`.
  - `v6`: fixing the type error with "sharing"
    Note the `Int_interval` signature. in `utop`
  - `v7`: fixing the type error with "replacement"
    Note the `Int_interval` signature in `utop`.

Sharing types in signatures

Change the signature of `Make_interval` so that the `endpoint` in the module created by the functor is the same as the type from in the input module to the functor.

In `module M : I with type t1 = t2 ...` the type `t1` and `t2` are the same and `t2` is visible.

See working example in `Intervals/v6`.

Especially pay attention to the signature when `#mod_use "intervals.ml";;` is used.

Replacing types in signatures

Change the signature of `Make_interval` so that the `endpoint` in the module created by the functor is replaced by the type from in the input module to the functor.

But in `module M : I with type t1 := t2` the type `t1` is now removed from the signature of `M`.

Thus using sharing (`=`) instead of destructive substitution (`:=`) is required if the named type `t1` is still to be used.

So `=` (sharing) is useful in places in which `:=` (destructive substitution) does not work.

See working example in `Intervals/v7`.

Especially pay attention to the signature when `#mod_use "intervals.ml";;` is used.

Programming in the large

As we've seen, using the ML-style module system in OCaml does feel like "programming."

We have mechanisms for not just creating signatures but also for manipulating them.

The `with type t1 = t2` clauses provide fine control over module interfaces and modules that is not found in many other languages.

Modules, Signatures and, Functors

How do these compare to other module/class/package systems you know?

For example, `public`, `private` modifiers on class elements in OO languages?

- A signature is code that only shows the interface.

  One need not read the implementation of a class and see these modifiers to know what is available.

- What about `protected` modifier or, in C++, "friends" ?

  Functors make dependencies between modules/classes explicit.

  "Dependency Injection" in Java is an example of sub-optimal techniques that address the lack of a well-designed module system.