S6: Programming in the Large
6.1: Modules
CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

"Programming in the large"

- This 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

Methodologies

- how best to decompose a problem into components or modules.
- often with the above goals in mind

Mechanisms

Programming language features that support or enable breaking applications into modules.

These 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 by other components of the program.

The kinds of types (disjoint unions, 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

- See Chapters 12 and 13 of the Hickey text.
- OCaml and Standard ML have a very sophisticated module system.
- It is the "gold-standard" for module systems.

Files as simple modules

- A single `.ml` file containing types and value definitions can be viewed as a module.
- Consider putting many of our list processing functions, such as `map` and `foldl`, in a file name `ourList.ml`.
- Another file can refer to these as, for example `OurList.map`.

Module namespaces

- The module defines a namespace in which its types and values can be referenced.
  - The file name is capitalized to form the module name.
  - The dot (`.`) opens the modules name space and interprets the following name based on the components in that module.
- Thus `OurList.map` is the external name of the `map` function in that module. The internal name is just `map`.
- Let's look at `ourList.ml` and `usingLists.ml` in the `SamplePrograms` directory of the public repository.

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.
  - *e.g.*`#mod_use "ourList.ml"`
  - Next, `#use` a file that refers to these module elements using their external names.
  - For example `#use "usingLists.ml"`

Using modules and the OCaml compilers

- So far, we've only run OCaml programs from inside the utop interpreter.
- But OCaml has compilers that will generate byte code (a situation similar to Java and the JVM).

```
% ocamlbuild usingLists.byte
% ./usingLists.byte
```

- This sees the reference to `OurList.sum` and then compiles that module as well.

Or compile to native code

- One can also compile down to native code.

```
% ocamlbuild usingLists.native
% ./usingLists.native
```

- One can start fresh with `% ocamlbuild -clean` Do this to remove saved temporary files. If you get errors about not finding modules that you think are there, run this and try again.

Intervals

Another example of using file as modules can be seen in

- `intInterval.ml` and
- `useIntInterval.ml` in
- `SamplePrograms/Intervals/v1/`.

We will consider several versions in the appropriately named directories.

Interface files

- Interface files, with a `.mli` extension, indicate the types and types of values in a module.
  That is, the interface.
- The `.ml` file contains the implementation of those values.
- Code using a module need only see the interface file.

Separate compilation

Separate compilation: when files change in an application, only recompile those that change.

- In our example, `useIntInterval` uses the module `IntInterval`.
- If the implementation, but not the interface, to `IntInterval` changes, then we would like to re-compile it, but not
  `useIntInterval`.
- We can see this, and other scenarios, using `ocamlbuild`.
- `ocamlbuild` is a wrapper for `ocamlfind` which determines dependencies between modules and recompiles only those files that must be.
- These files are in `code-examples/Intervals/v2/`

Abstract data types

- We may like to hide the implementation of the integer interval so that it is kept abstract.
- That is, the functions in the module provide the only means for inspecting or modifying the value. Its implementation cannot be seen.
- For example, we should not pattern match using constructors `Interval` or `Empty` since they are part of the type and are not visible to the user of the module.
- A abstract data type is one whose implementation is not visible (it is abstract) and the only way to use the data type is through functions that help to isolate the implementation from its use.

Abstract data types

- The way that we hide this data type in OCaml, is to not include it in the interface file.
- It is therefore not visible to other modules using this one.
- We thus define a type `t` which is exposed.
- But its value (which is the type `intInterval`) is not in the interface file and thus not visible.

Inferring interfaces

- Note that OCaml can "infer" an interface file if one doesn't exist.
- But it is better to write them explicitly.
- Also note, in utop, `#mod_use "intInterval.ml"` bypasses our `.mli` file and we can see the abstract type.

  Separate compilation and abstract data types are important in large applications in which the utop interpreter is not the primary way means executing code.

Nested Modules

- We will often want more flexibility than the one-module-per-file approach we saw above.
- OCaml allows multiple modules to be defined in a file, with various ways to combine and access them.
- We've already used these a bit in the `String`, `Int`, and `List` modules.

Opening a Module

- The `List` module is written inside a file as an element of another module.
- This 'open' makes visible all the names declared at the top level.
- We can `open` the `List` module so all elements can be used directly.

Modules and module signatures

- A module is a collection of named types and values.
- The type of a module is called its `signature`. This signature defines the interface that is implemented by a module.
- The syntax for this is

  `module` $\langle name \rangle$ `:` $\langle signature \rangle$ `=` $\langle implementation \rangle$

“in the large and small”

“in the small”

- expressions have types
- expressions denote a value

“in the large”

- modules have signatures
- modules have an implementation

Functors, coming soon, will expand upon this point.

Modules and module signatures

module ⟨*name*⟩ : ⟨*signature*⟩ = ⟨*implementation*⟩

For example:

```
module Username : sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end = struct
  type t = string
  let of_string x = x
  let to_string x = x
end
```


Modules and module signatures

- The `sig` keyword begins signatures
- The `struct` keyword begins module “values”.
- We can also define these separately and then refer to named signatures and structs.
- Consider how signatures and structs can be defined separately in `session_info.ml` in the code examples.

- We can now put nest our `IntInterval` module in another file.
- We then will write an explicit signature for it.
- These files are in `SamplePrograms/Intervals/v3/`

Our "big ideas" so far

- Separating interface from implementation
- Useful for separate compilation
- Required for creating abstract data types.

Next, parameterization of modules using **functors**.

Functors

- Functors are functions over modules
- Sometime called "parameterized modules"
- Allow us to instantiate modules with some component module.
- Parametric polymorphism happens "at the type level"

  Functors happen "at the module level"
- Think of them as functions for "programming in the large"

“in the large and small” - revisited

“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

Return to our interval examples.

- Recall, that we use modules to group types and supporting operations (functions) together as a named component.
- So we create a `Comparable` signature to specify what is required for the end points in an interval.
- The `Make_interval` functor creates an interval module based on a module that implements the `Comparable` signature.
- Examples are in `SamplePrograms/Intervals/v4`.

Functor application

- `module` *functor-name* `(` *input-module* `:` *sig-of-input* `) =` `struct` ...
- See the examples in `code-examples/Intervals/v4`.
- We can also create string intervals.
- Matching here is a bit like sub-typing. Having more elements is OK and the ones with the same names have to have the same types.

Transparent versus Opaque types

- So far, these are all concrete or transparent modules.

  The type of the implementation is exposed.
- A transparent module exposes all of its types.
- A translucent module exposes some of its types.
- A opaque module exposes none of its types.
- To support "representational independence" the type of the ADT must be hidden, that is, abstract.

Opaque intervals

- Let's hide the implementation of intervals.
- See the non-working examples in `code-examples/Intervals/v5`.
- You can see this in utop using
  - `#mod_use "intervals.ml" ;;`
  - `#use "intIntervals.ml" ;;`

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 `code-examples/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 `code-examples/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.