CSci 2041
Advanced Programming Principles
L19: Modularity, Part 2

Eric Van Wyk

Fall 2014

- Functors - 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"
- See Chapter 9 of Real World OCaml for more information.

Intervals

- Return to our interval examples.
  (These examples stay much closer to the examples in Chapter 9 of Real World OCaml.)

- 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 `code-examples/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 created using `Core.Std.String`
  The `Code.Std.String` module matches the `Comparable` signature becuase it has at least the named elements with the same types as in `Comparable`.
- 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.