S2.1: Modular Programming in OCaml - Polymorphism Addendum

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

Polymorphism and modules

It is worth pausing our discussion on modules to see how functors provide a form of "ad-hoc" polymorphism.

Parametric polymorphism

- We've seen this many times.
  - `List.length : 'a list -> int`
  - `List.map : ('a -> 'b) -> 'a list -> 'b list`
  - `ListStack.push : 'a -> 'a stack -> 'a stack`
- We can put anything in a list or stack. `int`, `string`, or functions values, etc.
- Key point: we have the same behavior at all type instantiations.

  For `ListStack.push`:
  - `int stack -> int -> int stack`
  - `string stack -> string -> string stack`
  - `('a -> 'b) stack -> ('a -> 'b) -> ('a -> 'b) stack`
- The same thing happens with `push` for every type.
- Thus, there is a single generic implementation that works for all type instantiations.

Ad-hoc polymorphism

- In ad-hoc polymorphism there are some restrictions on the type arguments.
- There are also different behaviors at different instantiations.

  Consider operator overloading and `(<)` with type `'a -> 'a -> bool`
- Is the behavior the same for `int` values and `string` values?

  More specifically, is the implementation the same? No.
- Thus, sometimes is there no valid behavior for certain type instantiations.

  For example, `(<)` doesn't work for functions.

  We can see all of this in `utop`.
- Thus there are different implementation at different types.

Other instances of ad-hoc polymorphism

- In object-oriented programming, sub-classes provide their own implementation of methods declared in a super-class.
- In OCaml functors, each parameter module implements the types and values (functions) in the signature.
- For example, the `ExprF` functor is parameterized by a stack implementing signature `Stack`.

  This is ad-hoc polymorphism in that it restricts what type of modules can be passed in.
- `S.push` as seen in `ExprF ( ListStack.ListStackM )` is a different function than `S.push` as seen in `ExprF ( CustomStack.CustomStackM )`
- We see this in Intervals as well.

Exprssions/Stacks and Intervals/Comparables

- The `ExprF` functor uses `S : Stack.StackS`.
  The type `expr` and `instr` type in `ArithmeticS` do not mention the stack.
  The signature `ArithmeticS` does not expose the stack.
- The `Make_interval` functor uses `Endpoint : Comparable`.
  But the type `t` in `Interval` uses `Endpoint.t`
  The signature `Interval_intf` does expose the endpoint.
- Let's consider some interval examples in `Intervals`.

In different languages

In OCaml

- parametric polymorhism at the "in the small level"
- ad-hoc at the "in the large"

In Haskell

- both types done "in the small"
- ad-hoc polymorphism uses type classes
- very limited module system

In Java

- both via classes with the addition of generic methods

In C++

- Templates - the world's most broken form of parametric polymorphism.