Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 6 - Sequence 4: Modules as compilation units

Compiling an OCaml programs

- The file extension for *OCaml* source code is `.ml`.
- *OCaml* enjoys **separate compilation**.
- To produce an executable program:
  1. **Compile** each file separately following dependencies.
  2. **Link** the resulting compilation units altogether.
- `ocamlc` is the compiler to the *OCaml* virtual machine.
- `ocamlopt` is the compiler to native code.
- (In the sequel, we will use `ocamlc`
  but the same commands work with `ocamlopt`.)

Compiling an OCaml program

- Imagine that your project contains `a.ml` and `b.ml` and that `b.ml` uses `a.ml`.
- First, compile `a.ml`:

  ```
  ocamlc -c a.ml
  ```

- This command produces **2** files:
  - `a.cmo`: the bytecode (would be `a.cmx` if native code)
  - `a.cmi`: a compiled interface (see next slide)
- Now, compile `b.ml`:

  ```
  ocamlc -c b.ml
  ```

- And finally, link `a.cmo` and `b.cmo` into an executable `prog`:

  ```
  ocamlc -o prog a.cmo b.cmo
  ```

- The order of the cmo files must follow the dependencies.

Compilation units are modules

- A file named `a.ml` appears as a module `A` in the program.
- Hence, to refer in `b.ml` to a value `x` defined in `a.ml`, just write `A.x`.
- The interface of the module `a.ml` can be written in file named `a.mli`.
- For instance, if `A` exports a type `t` and a value `x` of this type, `a.mli` is:

```
type t
val x : t
```

- When `a.ml` is compiled, the compiler looks for `a.mli` to compile the interface. If it does not exist, it uses the inferred module interface.
- Interfaces can also be compiled independently:

```
ocamlc -c a.mli
```

... produces the file `a.cmi`.

Where is the main function?

- There is no main function in an *OCaml* program.
- The evaluation of a program is the evaluation of its modules.
- The modules are evaluated in the order given in the linking command.

Libraries

- Several modules can be aggregated as a library into one `.cma` file:
  ```
  ocamlc -a a.cmo b.cmo -o lib.cma
  ```

- This library can be used by another program as if it were a compilation unit.
- To install a library in the system, copy the compiled files (`.cmi`, `.cmo` and `.cma`) into an arbitrary directory `some_dir`.
- To use a library to compile another file:
  ```
  ocamlc -I some_dir -c c.ml
  ```

- To use a library during linking:
  ```
  ocamlc -I some_dir -o prog lib.cma c.cmo
  ```

- The `findlib` tool automates
  the library configuration and installation process.

Build system

- *OCaml* comes with a **build system** tool named `ocamlbuild`.
- It automatically **builds** compiled files, libraries and executable programs.
- It automatically **computes needed dependencies**.
- It is configurable through a `_tag` file.
- It interacts with `findlib`.
- It is customizable using plugins.
- To build a program `a.byte` out of `a.ml` and its dependencies, typing:

```
ocamlbuild a.byte
```

. . . usually works.

Package manager

- *OCaml* has a package manager named `opam`.
- Find it at `http://opam.ocamlpro.com/`
- A package may contain libraries and programs useful to other developments.
- This is a simple way to get *OCaml* **packages developed by our community**!
- We look forward to see there **your own package**!