Modules, Signatures and Functors ★★, ★★★

Topics

- Modules, Signatures, Functors
- First class modules, Dynlink
- Symbolic Document Processing, Abstract Syntax Trees

Introduction

The goal of this series of exercises is to design the architecture of a command line tool for converting documents between rich text formats, and apply processing passes on them.

We want to end up with command line invocations such as:

```
rtftool -from latex -to text:out.txt -pass flatten-maths
rtftool -from latex:f.tex -to html:f.html -pass retrieve-urls
```

General design

In the end, the tool will be made of the following components:

- The `Rtf` intermediate language module.
- A series of input rich text format modules with their own internal representations, either ad hoc or decorator modules around existing libraries.
- Their associated converters to `Rtf` that may fail if the input document is too complex for the common language.
- The same for output formats.
- A functor `Converter.Make` that takes an input and an output format and instanciates a dedicated conversion function.
- A registry associating names to first class format and conversion modules and to processing passes.
- A main program analyzing the command line to load the appropriate modules from the registry or by dynlinking, and launching the conversion and processing.

Exercise 1 – Warm-up

Question 1.1 – Write a module `Rtf`. This module will serve as the core interchange rich text format. It defines a type `t` representing the abstract syntax tree of documents.

The required features are:

- Document structure (paragraphs, sections, chapters, etc.)
- Lists, tables.
- In-line formatting (blod, italic, etc.)
- Internal and external references.
- Equations and pictures.

We may not implement every one of them in every handled format, but we want them to be available in the intermediate representation.

Question 1.2 – Write a proper interface, with `ocamldoc` formatted comments.

Question 1.3 – Write two processing passes: in-line formatting stripping and equation flattening, that wil allow them to be exported to formats that do not handle them natively.

Question 1.4 – Add a processing pass registration mechanism in a `Registry` module: a registration function taking a name and a pass and storing it in an internal global table, and a lookup function that may raise `Not_found`. Pre-register the two previous passes.

Exercise 2 – Formats

The signature for rich text file formats is:

```
module type FORMAT = sig
  type t
  type input_params
  type output_params
  val read : input_params -> t
  val write : output_params -> t -> unit
  val name : string
  val parse_input_params : string -> input_params
  val parse_output_params : string -> output_params
end
```

Defined in a file `converter.ml`

The type `t` is the internal representation specific to this format. We will try and use existing libraries, this is likely to be the types they define.

The types `*_params` are the parameters required for reading and writing files in this format. It will probably be a file name, but may include things such as encoding, author, format version, etc.

Other fields are for registering the format in the system, when we introduce the command line / plugin system.

Question 2.1 – We may not want to provide both input and output for all formats. Rewrite `FORMAT` as a composition of two new interfaces `INPUT_FORMAT` and `OUTPUT_FORMAT`.

Question 2.2 – Implement an `INPUT_FORMAT` `Text` whose only smart behaviour will be to split the text in paragpraphs at double line breaks.

Question 2.3 – Implement an `OUTPUT_FORMAT` `Latex` or `Html`, whichever you feel most comfortable with.

Question 2.4 – The previous will be enough for testing, but feel free to implement any other format. You can try using `omd` for Markdown, `xmlm` for XHTML.

Exercise 3 – Rtf projections

Question 3.1 – Write two interfaces `TO_RTF` and `OF_RTF` that, for a specific type `t` provide the `to_rtf` and `of_rtf` functions, converting to and from the format specific type to the intermediate `Rtf.t` format.

Question 3.2 – Write `RTF_CONVERTERS` that combine both `TO_RTF` and `OF_RTF`, in a way similar to `FORMAT`.

Question 3.3 – Write instances `Text_to_rtf`, `Latex_of_rtf` or `Html_of_rtf` and for any other format you defined in the previous exercise. Your converters may `failwith "unsupported feature"`.

Exercise 4 – Converter program

Question 4.1 – Write a functor `Converter.Make` taking the input and output formats and their Rtf projections and returning a module of signature:

```
module type S = sig
  type input_params
  type output_params
  val convert : input_params -> output_params -> (Rtf.t -> Rtf.t) list -> unit
end
```

The convert function calls the input reader with its parameters, the input to `Rtf.t` converter, applies a series of passes, calls the `Rtf.t` to output converter, and finally the output writer with its parameters.

Question 4.2 – Statically apply the functor and provide a first basic line interface to the instanciated conversion function.

Question 4.3 – Use the `Arg` module to provide a `-pass` option that will apply passes from their registered names on the intermediate document. It will also take `input` and `output` arguments that will be passed to the parameter parsers of the input and output formats.

Exercise 5 – Run-time selection of formats

Question 5.1 – Add to the `Registry` functions to register by name input and output formats as first class modules (with their associated converters). Pre-register the formats and converters you already wrote.

Question 5.2 – Update the main so that it supports the `-input` and `-output` arguments as demonstrated in the header: a format name and its arguments separated by a colon.

The main will obtain the format and converter modules by name from the `Registry` and locally apply the `Converter.Make` functor and call the instanciated `convert` function.

Question 5.3 – If the format specified on the command line is `format.cmxs`, dynlink the module and then try another search for `format` in the registry. The idea is that the format dynlinkable module will register a format at toplevel under its own name.

Exercise 6 – To go further

Question 6.1 – Find a better way to handle features unsupported by a processing pass or converter.

Question 6.2 – Design a stream version of the tool that does not maintain the document in memory.

Question 6.3 – Implement document format autodetection.

Question 6.4 – Using objects, implement customizable map and fold over the type `Rtf.t`.

Modules, Signatures and Functors

SOLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTION

Solution to exercise 1

Makefile:

```
rtftool: rtf.cmx registry.cmx
	ocamlopt $^ -o $@
%.cmx: %.ml
	ocamlopt -c $<
%.cmi: %.mli
	ocamlopt -c $<
rtf.cmx: rtf.cmi
registry.cmx: rtf.cmi
```

File `rtf.ml`:

```
type structure =
  (anchor * structure_item) list
and structure_item =
  | Section of inline option * structure
  | Picture of inline option * url
  | Equation of math
  | Paragraph of inline
and math =
  | Variable of string
  | Op of string * math list
  | Seq of math list
  | Number of float
  | Inline of inline
and inline =
  inline_item list
and inline_item =
  | Text of string
  | External of url
  | Internal of anchor
  | Style of attribute list * inline
and attribute =
  | Bold | Italic | Monospace
  | Color of string
and url = string
and anchor = string

type t = structure

let strip_formatting l =
  let rec structure l =
    List.map (fun (a, item) -> (a, structure_item item)) l
  and structure_item = function
```

```
    | Section (None, s) -> Section (None, structure s)
    | Section (Some i, s) -> Section (Some (inline i), structure s)
    | Picture (None, u) -> Picture (None, u)
    | Picture (Some i, u) -> Picture (Some (inline i), u)
    | Equation _ as eq -> eq
    | Paragraph i -> Paragraph (inline i)
  and inline l =
    List.flatten (List.map inline_item l)
  and inline_item = function
    | Style (_, i) -> inline i
    | Text _ | External _ | Internal _ as untouched -> [ untouched ]
  in structure l

let flatten_maths l =
  let rec structure l =
    List.map (fun (a, item) -> (a, structure_item item)) l
  and structure_item = function
    | Section (t, s) -> Section (t, structure s)
    | Equation m -> Paragraph (math m)
    | Picture _ | Paragraph _ as untouched -> untouched
  and math = function
    | Variable n -> [ Text n ]
    | Op (n, []) -> [ Text n ; Text "()"]
    | Op (n, m :: ms) ->
      [ Text n ; Text "(" ]
      @ math m
      @  List.fold_right
        (fun m acc -> Text "," :: math m @ acc)
        ms [ Text ")" ]
    | Seq ms -> List.flatten (List.map math ms)
    | Number n -> [ Text (string_of_float n) ]
    | Inline i -> i
  in structure l
```

File `rtf.mli`:

```
type structure =
  (anchor * structure_item) list
and structure_item =
  | Section of inline option * structure
  | Picture of inline option * url
  | Equation of math
  | Paragraph of inline
and math =
  | Variable of string
  | Op of string * math list
  | Seq of math list
  | Number of float
  | Inline of inline
and inline =
  inline_item list
and inline_item =
  | Text of string
  | External of url
  | Internal of anchor
```

```
  | Style of attribute list * inline
and attribute =
  | Bold | Italic | Monospace
  | Color of string
and url = string
and anchor = string

type t = structure

val strip_formatting : t -> t
val flatten_maths : t -> t
```

File registry.ml:

```
let all_passes = ref []
let register_pass name pass =
  all_passes := (name, pass) :: !all_passes
let lookup_pass name =
  List.assoc name !all_passes
let () =
  register_pass "strip-formatting" Rtf.strip_formatting ;
  register_pass "flatten-maths" Rtf.flatten_maths
```

File registry.mli:

```
val register_pass : string -> (Rtf.t -> Rtf.t) -> unit
val lookup_pass : string -> (Rtf.t -> Rtf.t)
```

Solution to question 2.1

```
module type INPUT_FORMAT = sig
  type t
  type input_params
  val read : input_params -> t
  val name : string
  val parse_input_params : string -> input_params
end

module type OUTPUT_FORMAT = sig
  type t
  type output_params
  val write : output_params -> t -> unit
  val name : string
  val parse_output_params : string -> output_params
end

module type FORMAT = sig
  type t
  include INPUT_FORMAT with type t := t
  include OUTPUT_FORMAT with type t := t
end
```

Solution to question 2.2

```
type t = string list
type input_params = in_channel
let name = "text"
```

```
let parse_input_params = function
  | "" | "stdin" | "-" -> stdin
  | fn -> open_in fn
let read fp =
  let rec read lacc pacc =
    match input_line fp with
    | "" -> read [] (pack lacc pacc)
    | line -> read (line :: lacc) pacc
    | exception End_of_file -> pack lacc pacc
  and pack lacc pacc =
    if lacc = [] then pacc else String.concat "␣" (List.rev lacc) :: pacc in
  let pars = read [] [] in
  close_in fp ; List.rev pars
```

Solution to question 3.1

```
module type TO_RTF = sig
  type t
  val to_rtf : t -> Rtf.t
end

module type OF_RTF = sig
  type t
  val of_rtf : Rtf.t -> t
end
```

Solution to question 3.2

```
module type RTF_CONVERTERS = sig
  type t
  include TO_RTF with type t := t
  include OF_RTF with type t := t
end
```

Solution to question 3.3

```
type t = string list
let to_rtf l = List.map (fun t -> Rtf.(None, Paragraph [ Text t ])) l
```

Solution to question 4.1

```
module Make
    (IF : INPUT_FORMAT)
    (IC : TO_RTF with type t = IF.t)
    (OF : OUTPUT_FORMAT)
    (OC : OF_RTF with type t = OF.t)
  : S with type input_params = IF.input_params
       and type output_params = OF.output_params = struct
  type input_params = IF.input_params
  type output_params = OF.output_params
  let convert input_params output_params passes =
    let input = IF.read input_params in
    let rtf = IC.to_rtf input in
    let rtf = List.fold_left (fun rtf pass -> pass rtf) rtf passes in
    let output = OC.of_rtf rtf in
    OF.write output_params output
```

```
end
```

Solution to question 4.2

```
module TextToHtml = Converter.Make (Text) (Text) (Html) (Html)
let () =
  let input_params = Text.parse_input_params Sys.argv.(1) in
  let output_params = Html.parse_output_params Sys.argv.(2) in
  TextToHtml.convert input_params output_params []
```

Solution to question 5.1

```
module type INPUT = sig
  type t
  include Converter.INPUT_FORMAT with type t := t
  include Converter.TO_RTF with type t := t
end

module type OUTPUT = sig
  type t
  include Converter.OUTPUT_FORMAT with type t := t
  include Converter.OF_RTF with type t := t
end

let all_input_formats : (string * (module INPUT)) list ref = ref []
let register_input_format name m =
  all_input_formats := (name, m) :: !all_input_formats
let lookup_input_format name =
  List.assoc name !all_input_formats

let all_output_formats : (string * (module OUTPUT)) list ref = ref []
let register_output_format name m =
  all_output_formats := (name, m) :: !all_output_formats
let lookup_output_format name =
  List.assoc name !all_output_formats
let () =
  register_input_format "text" (module Text : INPUT) ;
  register_output_format "html" (module Html : OUTPUT)
```

Solution to question 5.2

```
let () =
  let input = ref ((module Text : Registry.INPUT), "") in
  let output = ref ((module Html : Registry.OUTPUT), "") in
  let passes = ref [] in
  Arg.parse
    [ "-input", Arg.String (fun s -> input := parse_input_arg s),
      "set␣the␣input␣format␣and␣params" ;
      "-output", Arg.String (fun s -> output := parse_output_arg s),
      "set␣the␣output␣format␣and␣params" ;
      "-pass", Arg.String (fun pass -> passes := pass :: !passes),
      "add␣a␣processing␣pass"]
    (fun _ -> raise (Arg.Bad "anonymous␣arguments␣unsupported"))
    "Usage:␣rtftool␣-input␣format:␣params␣-output␣format:params␣pass1␣pass2␣..." ;
  let (module Input), input_params = !input in
```

```
  let (module Output), output_params = !output in
  let input_params = Input.parse_input_params input_params  in
  let output_params = Output.parse_output_params output_params in
  let module Converter =
    Converter.Make (Input) (Input) (Output) (Output) in
  Converter.convert
    input_params output_params
    (List.map Registry.lookup_pass (List.rev !passes))
```