Metaprogramming in OCaml

ygrek

OCaml@Singapore

Jul 30, 2014

Overview

- Introduction
- cppo
- camlp4
- ppx
- metaocaml

Metaprogramming

Literally, *programming a program* — writing a program to manipulate another program (or itself).

- Reflexivity and *eval*
- Meta and object language, homoiconicity
- Two axes: spatial (macros) and temporal (stages)

Approaches

Naive classification by the object of manipulation

- *lexical* — lexemes of the language: `cppo`
- *syntactic* — Abstract Syntax Tree (AST): `camlp4`, `ppx`
- *runtime* — compiled code: `metaocaml`

cppo: The C preprocessor for OCaml

- Simple text-based preprocessor in the spirit of cpp
- Understands OCaml lexical rules
- Usual set of directives: `#define`, `#if`, `#include` and others
- Install: `opam install cppo`
- http://mjambon.com/cppo.html

cppo: Example

Source:

```
#define test(x) if x = 2 then

#if defined OCAML3
let (@@) f x = f x
#elif defined OCAML400
external (@@) : (’a -> ’b) -> ’a -> ’b = "%apply"
#endif

let () = print_endline @@ test(3) "wut?" else "ok!"
```

Compile:

```
ocamlc -pp "cppo -D OCAML400" example_cppo.ml
```

Result of preprocessing:

```

external (@@) : (’a -> ’b) -> ’a -> ’b = "%apply"

let () = print_endline @@ if 3 = 2 then "wut?" else "ok!"
```

camlp4: OCaml PreProcessor and Pretty-Printer

- extensible parser and pretty-printer OCaml library
- included in distribution (< 4.02)
- complex and very powerful, arbitrary syntax modifications
- little documentation
- revised syntax, pa_openin, deriving, bitstring, pa_lwt, etc

camlp4: Example (bitstring syntax)

```
let fail fmt = Printf.ksprintf failwith fmt

(** announce response *)
let announce_response s exp_txn =
  let rec clients rest l =
    bitmatch rest with
    | { ip : 32 ; port : 16 ; rest : -1 : bitstring } -> clients rest ((ip,port)::l)
    | { } -> l
  in
  bitmatch Bitstring.bitstring_of_string s with
  | { 1l : 32 ; txn : 32 ; interval : 32 ; leechers : 32 ; seeders : 32 ;
     rest : -1 : bitstring } ->
       if txn = exp_txn then
         (interval,clients rest [])
       else
         fail "error announce_response txn %ld expected %ld" txn exp_txn
  | { 3l : 32 ; txn : 32 ; msg : -1 : string } -> fail "error announce_response txn %ld : %s"
        txn msg
  | { } -> fail "error announce_response (expected txn %ld) : %S" exp_txn s
```

camlp4: Example (deriving generator)

```
module Deriving (Syntax : Camlp4.Sig.Camlp4Syntax) =
struct
  open Camlp4.PreCast
  include Syntax

  DELETE_RULE Gram str_item: "type"; type_declaration END
  DELETE_RULE Gram sig_item: "type"; type_declaration END

  open Ast

  EXTEND Gram
  str_item:
  [[ "type"; types = type_declaration -> <:str_item< type $types$ >>
    | "type"; types = type_declaration; "deriving"; "("; cl = LIST0 [x = UIDENT -> x] SEP ",";
          ")" ->
       let decls = display_errors loc Type.Translate.decls types in
       let module U = Type.Untranslate(struct let loc = loc end) in
       let tdecls = List.map U.decl decls in
         <:str_item< type $list:tdecls$; $list:List.map (derive_str loc decls) cl$ >>
   ]]
  ;

end

module M = Camlp4.Register.OCamlSyntaxExtension(Id)(Deriving)
```

camlp4: Example (lwt syntax)

```
let get_date () =
  let proc = Lwt_process.open_process ("",[|"date"|]) in
  lwt date = try_lwt Lwt_io.read_line proc#stdout with _ -> Lwt.return "unknown" in
  lwt _ = proc#close in
  Lwt.return date

let () =
  print_endline @@ Lwt_main.run @@ get_date ()
```

```
let get_date () =
  let proc = Lwt_process.open_process ("", [| "date" |]) in
  let __pa_lwt_0 =
    Lwt.catch (fun () -> Lwt_io.read_line proc#stdout)
      (fun _ -> Lwt.return "unknown")
  in
    Lwt.bind __pa_lwt_0
      (fun date ->
         let __pa_lwt_0 = proc#close
         in Lwt.bind __pa_lwt_0 (fun _ -> Lwt.return date))

let () = print_endline @@ (Lwt_main.run @@ (get_date ()))
```

camlp4: Example (cont.)

Compile:

```
ocamlfind ocamlc -verbose -c -syntax camlp4o -package lwt.syntax example_lwt.ml
```

See result of preprocessing:

```
camlp4 -I +camlp4 -I lwt -parser o -parser op -printer o lwt-syntax-options.cma lwt-syntax.cma

example_lwt.ml
```

META:

```
package "syntax" (
 version = "2.4.5"
 description = "Syntactic sugars for Lwt"
 requires = "camlp4 lwt.syntax.options"
 archive(syntax, preprocessor) = "lwt-syntax.cma"
 archive(syntax, toploop) = "lwt-syntax.cma"
 archive(syntax, preprocessor, native) = "lwt-syntax.cmxa"
 archive(syntax, preprocessor, native, plugin) = "lwt-syntax.cmxs"
)
```

ppx: Extension points

- generic attributes attached to AST nodes
- available since OCaml 4.02
- `compiler-libs`
- `ppx_tools`, `ppx_metaquot`
- `-dsource`, `-dparsetree`, `-dtypedtree`

ppx: Example (attributes)

```
type bkpt = {
  number : int;
  typ [@name "type"] : string;
  disp : string;
  enabled : string;
  addr : int;
  func : string;
  file : string option;
  fullname : string option;
  line : int option;
  cond : int option;
  thread_groups [@name "thread-groups"] : string values;
  times : int;
  ignore : int option;
} [@@inject]
```

ppx: Example (generator)

```
let gen_builder tdecl =
  let make_func body =
    let body =
      match find_attr_expr "inject" tdecl.ptype_attributes with
      | Some x when get_lid x = Some "unnamed" -> body
      | _ -> (* this type is represented as name-value pair *)
        let name = make_name tdecl.ptype_name.txt tdecl.ptype_attributes in
        let unwrapped_x = app (evar "named") [name; evar "x"] in
        let_in [Vb.mk (pvar "x") unwrapped_x] body
    in
    Str.value Nonrecursive [Vb.mk (pvar tdecl.ptype_name.txt) (lam (pvar "x") body)]
  in
  match has_attr "inject" tdecl.ptype_attributes, tdecl.ptype_kind, tdecl.ptype_manifest with
  | true, Ptype_record fields, _ ->
      let fields = List.map make_field fields in
      [make_func @@ let_in [Vb.mk (pvar "x") (app (evar "tuple") [evar "x"])] (record fields)]
  | true, Ptype_abstract, Some ty ->
      [make_func @@ app (extract ty tdecl.ptype_loc) [evar "x"]]
  | true, _, _ -> fatal tdecl.ptype_loc "Unsupported usage"
  | false, _, _ -> []
```

MetaOCaml: staged OCaml metaprogramming

- modification to OCaml compiler
- BER = Bracket, Escape, Run
- multiple stages
- Install: `opam switch 4.01.0+BER`
- http://okmij.org/ftp/ML/MetaOCaml.html

MetaOCaml: Example

```
let square x = x * x

let rec power n x =
  if n = 0 then 1
  else if n mod 2 = 0 then square (power (n/2) x)
  else x * (power (n-1) x)
(* val power : int -> int -> int = <fun> *)

let rec spower n x =
  if n = 0 then .<1>.
  else if n mod 2 = 0 then .<square .~(spower (n/2) x)>.
  else .<.~x * .~(spower (n-1) x)>.
(* val spower : int -> int code -> int code = <fun> *)

let spower7_code = .<fun x -> .~(spower 7 .<x>.)>.
(* val spower7_code : (int -> int) code = .<
  fun x_1 ->
     x_1 *
           (((* cross-stage persistent value (id: square) *))
                   (x_1 * (((* cross-stage persistent value (id: square) *)) (x_1 * 1))))>. *)

let spower7 = Runcode.run spower7_code
```

The End