Syntactic Extension via Pre-Processors

CS 4215: Programming Language Implementation

Chin Wei Ngan

April 7, 2015

Generated on Monday 6th April, 2015, 17:15

Outline

Syntactic Extension

- A good way to extend language to build DSL.
- Macros are simple syntactic extension
- camlp4 also a powerful language extension for OCaml
- Lisp has a simpler yet powerful meta-programming extension system.
- PPX - a new simpler syntactic extension for OCaml

PPX Syntax Extension

- A new API for syntactic extensions for OCaml.
- A simpler replacement for camlp4.
- Developed by Alain Frisch.
- First released in OCaml 4.02.
- References:

```
http://whitequark.org/blog/2014/04/16
 /a-guide-to-extension-points-in-ocaml/
https://blogs.janestreet.com/extension-points-or
 -how-ocaml-is-becoming-more-like-lisp/
http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk
 /experimental/frisch/extension_points.txt?view=log
https://github.com/alainfrisch/ppx_tools
```

Why replace camlp4?

- Allows arbitrary syntactic change. Too flexible.
- Two OCaml syntaxes : revised and original.
- Two OCaml parsers (ocamlc and camlp4)
- Independent set of bugs and error messages.
- Inadequate documentation.
- Confusing to tools (due to arbitrary syntax extension).
- Hard to integrate different camlp4 extensions.

Key Concepts of PPX

- A *syntax extension* is a function that maps OCaml AST to another OCaml AST.
- An *attribute* is an annotation that is used to support *type-driven* code generation.
  - It is ignored by the OCaml compiler by default.
  - It can be attached to any interesting syntactic construct: expressions, types, variant constructors, fields, modules, etc.
- An *extension node* replaces a valid component in syntax tree. Type-checker fails when it encounters an extension node.
- A *quoted string* gives another syntax for writing string literals. It is typically for embedding foreign syntax fragments.

Attributes

- An attribute is written as `[@id ...]` (on expressions, constructors etc), `[@@id ...]` (on items e.g signature) or `[@@@id ...]` (floating).
- Each `...` denotes a payload that can either be a structure (e.g. `[@id x * 3]`), type expression (e.g. `[@id : TYP]`) or pattern (e.g. `[@id ? PAT]`).

Example

An example with different kinds of attributes.

```
module type S = sig
  [@@@id1]
  type t =
    | A [@id2]
    | B [@id3] of int [@id4]   [@@id5]
  [@@@id6] [@@@id7]
  [@@@id8]
  type s
  [@@id9]
end
```

Floating Attributes

`[@@@id1]`, `[@@@id6]`, `[@@@id7]` and `[@@@id8]` are standalone attributes.

```
signature_item (ex4_module.ml[2,20+2]..[2,20+10])
  Psig_attribute "id1"
  []
signature_item (ex4_module.ml[6,96+2]..[6,96+10])
  Psig_attribute "id6"
  []
signature_item (ex4_module.ml[6,96+11]..[6,96+19])
  Psig_attribute "id7"
  []
signature_item (ex4_module.ml[7,116+2]..[7,116+10])
  Psig_attribute "id8"
  []
```

Attributes on Items

`[@@id5]` and `[@@id9]` are attributes on structure items (e.g. types).

```
type_declaration "t" (ex4_module.ml[3,31+7]..[3,31+8])
     (ex4_module.ml[3,31+7]..[5,57+38])
   attribute "id5"
   []
type_declaration "s" (ex4_module.ml[8,127+7]..[8,127+8])
     (ex4_module.ml[8,127+7]..[9,136+9])
   attribute "id9"
   []
```

Attributes on Expressions etc

`[@id2]`, `[@id3]`, and `[@id4]` are attributes on expressions, construtors, type expressions etc.

```
(ex4_module.ml[4,42+6]..[4,42+14])
  "A" (ex4_module.ml[4,42+6]..[4,42+7])
  attribute "id2"
    []
  []
  None
(ex4_module.ml[5,57+6]..[5,57+28])
  "B" (ex4_module.ml[5,57+6]..[5,57+7])
  attribute "id3"
    []
    [
     core_type (ex4_module.ml[5,57+18]..[5,57+21])
      attribute "id4"
      []
      Ptyp_constr "int" (ex4_module.ml[5,57+18]..[5,57+21])
      []
    ]
```

A Simple Example - Convert to Int

- Assume we desire a syntax extension that could automatically synthesize a function to convert from each specified type into its integer value.
- We can supply attribute annotation, as follows:

```
type t = A [@id 1]
       | B [@id 4] of int [@@id_of]
```

- Generated Converter:

```
id_of_t a = match a with
            | A -> 1
            | B _ -> 4
```

Extension Nodes

- Unless transformed, the presence of extension node causes type-checking to fail.
- Two syntaxes exist for extension node.
  1. `[%id s]` is an extension node for expression, type expression, module expression, module type expression, pattern, class expression or class type expression.
  2. `[%%id s]` is an extension node for structure item, signature item, class field or class type fields

An Example

A simple example `foo.ml`

```
let _ = [%getenv "USER"]
```

Parsing/printing works:

```
> ocamlc -dsource foo.ml
let _ = [%getenv "USER"]
File "foo.ml", line 1, characters 10-16:
Uninterpreted extension 'getenv'.
```

Compilation fails at type-checking:

```
> ocamlc -c foo.ml
File "foo.ml", line 1, characters 10-16:
Uninterpreted extension 'getenv'.
```

Example : getenv

Let us build a syntactic extension that would replace `[%getenv "<var>"]` by a string of the environment variable `<var>`.

Thus, given:

```
let _ = [%getenv "USER"]
```

We expect (assuming current user-id is pls2nus) to derive:

```
let _ = "pls2nus"
```

Getting Syntax Tree

Given `foo.ml`:

```
let _ = [%getenv "USER"]
```

We can use the following command to examine its syntax tree:

`ocamlc -dparsetree foo.ml`

```
[
  structure_item (foo.ml[1,0+0]..[1,0+24])
    Pstr_eval
    expression (foo.ml[1,0+8]..[1,0+24])
      Pexp_extension "getenv"
      [
        structure_item (foo.ml[1,0+17]..[1,0+23])
          Pstr_eval
          expression (foo.ml[1,0+17]..[1,0+23])
            Pexp_constant Const_string("USER",None)
      ]
]
```

AST Mapper

Type:

```
type mapper = {
  (* ... *)
  expr: mapper -> expression -> expression;
  (* ... *)
  structure: mapper -> structure -> structure;
  structure_item: mapper -> structure_item -> structure_item;
  typ: mapper -> core_type -> core_type;
  type_declaration: mapper -> type_declaration -> type_declaration;
  type_kind: mapper -> type_kind -> type_kind;
  value_binding: mapper -> value_binding -> value_binding;
  (* ... *)
}
```

Default mapper which changes nothing:

```
val default_mapper : mapper
```

AST Mapper for getenv

```
let getenv s = try Sys.getenv s with Not_found -> ""
let getenv_mapper argv =
 { default_mapper with (* overrides expression. *)
   expr = fun mapper expr ->
    match expr with (* extension node? *)
    | { pexp_desc =
        (* Should have name "getenv". *)
        Pexp_extension ({ txt = "getenv"; loc }, pstr)} ->
       begin match pstr with
       | PStr [{ pstr_desc =
         Pstr_eval ({ pexp_loc  = loc;
             pexp_desc = Pexp_constant (Const_string
         (sym, None))}, _)}] -> (* replace constant string *)
         Exp.constant ~loc (Const_string (getenv sym, None))
       | _ ->
         raise (Location.Error (
          Location.error ~loc "[%getenv] accepts a string,
                 e.g. [%getenv \"USER\"]"))
        end
           (* use default mapper. *)
    | x -> default_mapper.expr mapper x;
  }
let () = register "getenv" getenv_mapper
```

Compiling getenv extension

Using ocamlbuild

```
ocamlbuild -package compiler-libs.common ppx_getenv.native
```

Using ocamlc

```
ocamlc -I +compiler-libs -o ppx_getenv.native \
 ocamlcommon.cma ppx_getenv.ml
```

Using ocamlfind and ocamlc

```
ocamlfind ocamlc -package compiler-libs.common \
 -o ppx_getenv.native ocamlcommon.cma ppx_getenv.ml
```

Executing getenv extension

Using ocamlc:

```
ocamlc -dsource -ppx ./ppx_getenv.native foo.ml
```

Using rewritter:

```
ocamlfind ppx_tools/rewriter ./ppx_getenv.native foo.ml
```

Both execution results in:

```
let _ = "pls2nus"
```

Alternative Syntax for Keywords

All expressions starting with a keyword, e.g. `let`, `fun` and `if`, supports an alternative syntax. Given `<expr> = KW <rest>`:

```
KW[@id s1]..[@id sn] <rest>
  --> <expr> [@id s1]..[@id sn]

KW%id <rest>
  --> [%id <expr>]

KW%id[@id s1]..[@id sn] <rest>
  --> [%id <expr> [@id s1]..[@id sn]]
```

Alternative Syntax for Keywords

Examples:

```
let[@foo] x=2 in x+1
 --> ((let x = 2 in x + 1)[@foo])

begin[@foo][@foo2] x := !x+1; x := !x+1; end
 --> ((x := ((!x) + 1); x := ((!x) + 1))[@foo][@foo2])

match%foo e with | None > 0 | Some n -> n
 --> [%foo match e with | None  -> 0 | Some n -> n]

let%foo x=2 in x+1
 --> [%foo let x = 2 in x + 1]
```

An Example

In `pa_lwt` packgae to support light-weight threads, they allow let-bindings of the form:

```
let%lwt (x,y) = f in x+y
```

which gets translated to monad-like counterpart of the form:

```
Lwt.bind f (fun (x,y) -> x+y)
```

Quoted Strings

- This is meant to insert syntax that are unrelated to OCaml.
- It is captured by {<delim>|...|<delim>}
- Note that <delim> is a sequence of lowercase letters.
- Also, ... denotes its uninterpreted raw string representation.
- Delimiter information is kept inside parse tree. The Astypes.Constr_string scenario is defined as:

```
| Const_string of string * string option
```

Example

- `String.length {|\"|}` returns 2.
- `String.length {foo|\"|foo}` returns 2.
- `String.length {foo|\"|foo}` is parsed as:

```
expression (ex3_quoted.ml[1,0+8]..[1,0+20])
 Pexp_constant Const_string ("\\\"",Some "foo")
```

- Question : What is the parsed tree for `String.length {|\"|}`?

Quasi-Quotation

- `ppx_tools` library supports AST quasi-quotation through extension points.
- Types of quasi-quotation supported:
  - AST expression: e.g.
    `[%expr 2+2]`
  - Inject AST sub-expression: e.g.
    `[%expr 2+[%e number]]`
  - Pattern matching on AST, e.g.

```
match expr with
[%expr [%e? lhs]+[%e? rhs]] -> lhs,rhs
```

Getting Syntax Tree

Given `ex2_quasi.ml`:

```
let x = [%expr 222+222]
```

Two possible compilation commands. The first one prints the internal AST representation, while the second command, using ocamlbuild, generates just the code the AST construction.

```
ocamlfind ocamlc -c -package ppx_tools.metaquot -dsource ex2_quasi.ml
```

```
ocamlbuild -use-ocamlfind -package ppx_tools.metaquot ex2_quasi.byte
```

Large Abstract Syntax Tree

Equivalent AST form:

```
let x =
 { Parsetree.pexp_desc =
   (Parsetree.Pexp_apply
      ({
         Parsetree.pexp_desc =
           (Parsetree.Pexp_ident
              {
                Asttypes.txt = (Longident.Lident "+");
                Asttypes.loc = (Pervasives.(!) Ast_helper.default_loc)
              });
         Parsetree.pexp_loc = (Pervasives.(!) Ast_helper.default_loc);
         Parsetree.pexp_attributes = []
       },
      [("",
        {
         Parsetree.pexp_desc =
           (Parsetree.Pexp_constant (Asttypes.Const_int 222));
            Parsetree.pexp_loc = (Pervasives.(!) Ast_helper.default_lo
            Parsetree.pexp_attributes = []
          });
      ("",
         {
```

PPX deriving

- `ppx_deriving` simplifies type-driven code generation.
- It helps avoid writing tedious boilerplate codes that could be automatically synthesized based on types.
- a set of useful plugins `show`, `ord`, `enum`, `iter`, `map`, `fold`, `create`, `yojson` and , `protobuf`.
- Installation instruction:

```
opam install ppx_deriving
```

- Reference:

```
https://github.com/whitequark/ppx_deriving/blob \
  /master/README.md
```

@@deriving show

- `show` method converts to string counterpart
- Given:

```
type point2d = float * float
[@@deriving show]
```

- Using:

```
ocamlfind ocamlc -package ppx_deriving.std -dsource /
 -c ex6_show.ml
```

We can derive:

```
let rec pp_point2d fmt (a0,a1) =
  Format.fprintf fmt "(@[<hov>";
  ((Format.fprintf fmt "%F") a0;
   Format.fprintf fmt ",@ ";
   (Format.fprintf fmt "%F") a1);
  Format.fprintf fmt "@])"
and show_point2d x = Format.asprintf "%a" pp_point2d x
```

@@deriving eq

- `eq` method test for equality.
- Given:

```
type point2d = float * float
[@@deriving show, eq]
```

Generates an extra method:

```
let rec equal_point2d (lhs0,lhs1) (rhs0,rhs1) =
  ((fun (a : float)  -> fun b  -> a = b) lhs0 rhs0) &&
    ((fun (a : float)  -> fun b  -> a = b) lhs1 rhs1)
```

Deriving for Polymorphic Types

- Deriving works with polymorphic data types. Given

```
type ('a,'b) pair = 'a * 'b
[@@deriving show]

type point2d = (float,float) pair
[@@deriving show]
```

It supports polymorphism through higher-order formatters.

```
let rec pp_pair poly_a poly_b fmt (a0,a1) =
  Format.fprintf fmt "(@[<hov>";
  ((poly_a fmt) a0; Format.fprintf fmt ",@ "; (poly_b fmt) a1);
  Format.fprintf fmt "@])"
and show_pair poly_a poly_b x =
  Format.asprintf "%a" ((pp_pair poly_a) poly_b) x

let rec pp_point2d fmt =
  pp_pair (fun fmt  -> Format.fprintf fmt "%F")
    (fun fmt  -> Format.fprintf fmt "%F") fmt
and show_point2d x = Format.asprintf "%a" pp_point2d x
```

Adding a Custom Formatter

- Allows custom formatters to override default.
- Each formatter of type `t` has a type `Format.formatter -> t -> unit`.
- Example `ex8_show_file.ml` where `perm` field has a customised formatter:

```
type file = {
  name : string;
  perm : int     [@printer fun fmt -> fprintf fmt "0o%03o"];
} [@@deriving show];;
print_endline (show_file {name="hello"; perm=15});;
```

- Execution outputs:

```
{ Ex8_show_file.name = "hello"; perm = 0o017 }
```

Other Customized Formatters

- Possible to use `[@polyprinter]`.
- Given type `int list`.
- `[@printer]` should have type:

```
formatter -> int list -> unit
```

- `[@polyprinter]` should have type:

```
('a -> formatter -> unit) -> formatter -> 'a list -> unit
```

- `[@opaque]` is shorthand for:

```
[@printer fun fmt _ -> Format.pp_print_string fmt "<opaque>"].
```

Deriving Show for Tree

- Given tree type:

```
type 'a tree = Empty | Node of 'a * ('a tree) * ('a tree)
[@@deriving show]

type tree_f = point2d tree
[@@deriving show]

let t2 = let t1=Node(x,Empty,Empty) in  Node(x,t1,t1);;
print_endline (show_tree_f t2);;
```

- Execution yields indented tree:

```
Ex10_tree.Node ((2., 3.),
  Ex10_tree.Node ((2., 3.), Ex10_tree.Empty, Ex10_tree.Empty),
  Ex10_tree.Node ((2., 3.), Ex10_tree.Empty, Ex10_tree.Empty))
```

OCamlDoc

To support documentation generation:

```
val stats : ('a, 'b) t -> statistics
[@@doc
 "[Hashtbl.stats tbl] returns statistics about the table [tbl]:
  number of buckets, size of the biggest bucket, distribution of
  buckets by size."
]
[@@since "4.00.0"]

;;[@@doc section 6 "Functorial interface"]

module type HashedType =
  sig
    type t
      [@@doc "The type of the hashtable keys."]
    val equal : t -> t -> bool
      [@@doc "The equality predicate used to compare keys."]
  end
```

Logging

To support Bolt logging:

```
let funct n =
  [%log "funct(%d)" n LEVEL DEBUG];
  for i = 1 to n do
    print_endline "..."
  done
```

Monad

To support monadic code:

```
begin%monad
  a <-- [1; 2; 3];
  b <-- [3; 4; 5];
  return (a + b)
end
```

cppo

To support macro processing:

```
[%%ifdef DEBUG]
[%%define debug(s) = Printf.eprintf "[%S %i] %s\n%!" \
       __FILE__ __LINE__ s]
[%%else]
[%%define debug(s) = ()]
[%%endif]
debug("test")
```

sedlex

To support lexer based on Unicode:

```
let rec token buf =
  let%regexp ('a'..'z'|'A'..'Z') = letter in
  match%sedlex buf with
  | number -> Printf.printf "Number %s\n"
      (Sedlexing.Latin1.lexeme buf); token buf
  | letter, Star ('A'..'Z' | 'a'..'z' | digit) -> Printf.printf
      "Ident %s\n" (Sedlexing.Latin1.lexeme buf); token buf
  | Plus xml_blank -> token buf
  | Plus (Chars "+*-/") -> Printf.printf "Op %s\n"
      (Sedlexing.Latin1.lexeme buf); token buf
  | Range(128,255) -> print_endline "Non ASCII"
  | eof -> print_endline "EOF"
  | _ -> failwith "Unexpected character"
```