Make CamlP4 A better macro system

Hongbo Zhang[1]

University Of Pennsylvania

September 13, 2012

[1] hongboz@seas.upenn.edu

Outline

Outline

The power of macros (I)

- Syntax Extension

  Elf_parsers in compcert vs ELF_parsers generated code
  Loc: 171 vs 1971

The power of macros (I)

- Syntax Extension

  Elf_parsers in compcert vs ELF_parsers generated code
  Loc: 171 vs 1971
- Code generator: Type-directed code generation for compiler libraries

  Generated Code for typedtree.cmi
  Loc: Tens Of Thousands

The power of macros (I)

- Syntax Extension

  Elf_parsers in compcert vs ELF_parsers generated code
  Loc: 171 vs 1971
- Code generator: Type-directed code generation for compiler libraries

  Generated Code for typedtree.cmi
  Loc: Tens Of Thousands
- My branch of camlp4

  `Bootstrapped branch in original syntax` vs `syntax extension branch` Loc 20610 vs 85614

The power of macros (II)

- A story about CLOS(The Common Lisp Object System) 1986

The power of macros (II)

- A story about CLOS(The Common Lisp Object System) 1986
  1. Lisp was created long before Object-Orieted Paradigm became popular

The power of macros (II)

- A story about CLOS(The Common Lisp Object System) 1986
  1. Lisp was created long before Object-Orieted Paradigm became popular
  2. An object oriented layers which supports **multimethods** (multiple dispatch) and multiple inheritances

The power of macros (II)

- A story about CLOS(The Common Lisp Object System) 1986
  1. Lisp was created long before Object-Orieted Paradigm became popular
  2. An object oriented layers which supports **multimethods** (multiple dispatch) and multiple inheritances
  3. Extensible through standard Lisp macros and reader macros: The Art of MetaObject Protocol (`MOP`)

The power of macros (II)

- A story about CLOS(The Common Lisp Object System) 1986
  1. Lisp was created long before Object-Orieted Paradigm became popular
  2. An object oriented layers which supports **multimethods** (multiple dispatch) and multiple inheritances
  3. Extensible through standard Lisp macros and reader macros: The Art of MetaObject Protocol (`MOP`)
- Macros: The ultimate tool for abstraction

Outline

What's CamlP4

A library that helps to manipulate programs in an easy way.

- CamlP4: A general Pre-Processor-Pretty-Printer in OCaml 1997

What's CamlP4

A library that helps to manipulate programs in an easy way.

- CamlP4: A general Pre-Processor-Pretty-Printer in OCaml 1997
  - Pre-Processor
    The concrete syntax in the front end can be customized

What's CamlP4

A library that helps to manipulate programs in an easy way.

- CamlP4: A general Pre-Processor-Pretty-Printer in OCaml 1997
  - Pre-Processor
    The concrete syntax in the front end can be customized
  - Pretty-Printer
    The backend of the compiler can be changed

What's CamlP4

A library that helps to manipulate programs in an easy way.

- CamlP4: A general Pre-Processor-Pretty-Printer in OCaml 1997
  - Pre-Processor
    The concrete syntax in the front end can be customized
  - Pretty-Printer
    The backend of the compiler can be changed
  - Composable
    You can write in `Prolog` syntax, and have a `C` backend, via `Camlp4`

Minimal Example

```
$camlp4 -parser o -printer r -str 'let a = 3'
value a = 3;

$camlp4 -parser r   -printer o -str 'value a = 3;'
let a = 3
```

Outline

Dynamically Extensible Parser

Easy to reuse an existing parser without copy-paste, mutate the host language on the fly

```
let _ = begin
  EXTEND Gram GLOBAL: arith ;
  arith:
    ["top"
       [ `INT (i,_) -> i | x=SELF ; "+"; y=SELF -> x+y]
    |"simple" [ "("; x = SELF; ")" -> x ]];
  END;
end
```

```
let _ = begin
  EXTEND Gram arith: AFTER "top"
    [[ x=SELF; "*" ;y =SELF -> x + y]] ; END;
end
```

Dynamically Extensible Parser: Example 1

```
let bits = Bitstring.bitstring_of_file "/bin/ls" in
bitmatch bits with
| { 0x7f : 8; "ELF" : 24 : string; (* ELF magic number *)
    e_ident : 12*8 : bitstring;    (* ELF identifier *)
    e_type : 16 : littleendian;    (* object file type *)
    e_machine : 16 : littleendian  (* architecture *)
  } ->
printf "This is an ELF binary, type %d, arch %d\n"
e_type e_machine;
```

Dynamically Extensible Parser: Example 2

```
type t = {u:int ; v:float} with sexp
```

We will discuss a **more modular** type-derivation hook later

Generic Delimited Syntax Extension mechanism

CamlP4 provides a quotation hook to comprehend a piece of string in **any way** you want

```
<:quot_name< >>
```

For the code inside the quote, user can customize their lexer, parser

Generic Delimited Syntax Extension mechanism: Example

- A Quasi-Quotation for OCaml Ast Out Of The Box
  Reflective Parser but with a lift operation. (An Ast which econdes the Ast)

Then What's Quotation?

Denote `Abstract Syntax` using `Concrete Syntax`.
Consider how to construct an Ast node for the program below:

```
let a = List.iter (fun x -> x + 1)
```

Quotation (II)

```
Ast.StVal (loc, Ast.ReNil,
  (Ast.BiEq (loc, (Ast.PaId (loc, (Ast.IdLid (loc, "a"))))),
     (Ast.ExApp (loc,
        (Ast.ExId (loc,
           (Ast.IdAcc (loc, (Ast.IdUid (loc, "List")),
              (Ast.IdLid (loc, "iter"))))))),
        (Ast.ExFun (loc,
           (Ast.McArr (loc, (Ast.PaId (loc, (Ast.IdLid (loc, "x
              (Ast.ExNil loc),
              (Ast.ExApp (loc,
                 (Ast.ExApp (loc, (Ast.ExId (loc, (Ast.IdLid (l
                    (Ast.ExId (loc, (Ast.IdLid (loc, "x"))))))),
                 (Ast.ExInt (loc, "1"))))))))))))))
```

Quotation (III)

With Quotation mechanism, now you can just write

```
<:str_item< let a = List.iter (fun x -> x + 1)>>
```

`str_item` means `structure_item`

So Quasi-Quotation?

A way to abstract over code. For example, when you write

```
let big_ast = <:expr<
List.map (fun y -> (y+1)) [1;2;3]
>>
```

You can decompose it into two smaller Asts instead

```
let ast1 = <:expr< (fun y -> y+1) >>
let mk_big ast1 = <:expr< List.map $exp:ast1$ [1;2;3]>>
let ast2 = mk_big ast1
```

Quasi-Quotation (II)

Fundamental to do Meta-Programming

- It's an easy way to construct pieces of code
- It's the **only** way to do **meta-meta** programming: Complexity grows dramatically when writing macros which generate macros without the help of quasi-quotation.

  To construct a piece of code

  ```
  <:expr< <:expr<  3 >> >>
  ```

  You have to write

  ```
  Ast.ExApp (_loc,
    (Ast.ExApp (_loc,
       (Ast.ExId (_loc,
          (Ast.IdAcc (_loc, (Ast.IdUid (_loc, "Ast")),
             (Ast.IdUid (_loc, "ExInt")))))),
       (Ast.ExId (_loc, (Ast.IdLid (_loc, "_loc")))))),
    (Ast.ExStr (_loc, "3")))
  ```

- Shines when combining with pattern matching

Generic Delimited Syntax Extension mechanism: Example 2: EDSL

```
<:regexp<hello+>>
```

Other non-trivial features

- Bootstrappable: Key feature

  Growing your language with a macro system: More and more expressive!

Other non-trivial features

- Bootstrappable: Key feature

  Growing your language with a macro system: More and more expressive!
- Minimal requirement to the compiler

Outline

Improvement in three aspects

- Design issues

Improvement in three aspects

- Design issues
- Link the compiler (for 4.00)

Improvement in three aspects

- Design issues
- Link the compiler (for 4.00)
- Implementation issues

Design issues

- An Extensible and Programmable Lexer Support orakuda directly

```
let r = $/hello/ -> "world"
| $/bye/ -> "universe" | _ -> "default"
```

  orakuda is a very interesting project by camlspotter to write scripts using Perl's syntax
- An Extensible and Programmable and Functional Parser For expression leveled Scoped syntax extension. Two ways to avoid syntax collision:
  1. Encouraging delimited syntax extension
  2. The smaller of the scope for the Syntax extension, the better!
- Combining with ocaml's compiler!

Programmable Lexer, Parser?

- The description of grammar should be first-class and programmable – Algebraic data types with nice syntax extension support.

```
<:grammar<  SELF + SELF | Let IPATT = EXPR IN EXPR >>
```

- Grammar is first class
- Automaton generation delayed until the run-time
- Grammar separate from Action
- Functional to guarantee scoped level

MetaDSL

- Define An Algebraic Data Type

MetaDSL

- Define An Algebraic Data Type
- Make writing parsers much more concise

MetaDSL

- Define An Algebraic Data Type
- Make writing parsers much more concise
- Derive pretty printer

MetaDSL

- Define An Algebraic Data Type
- Make writing parsers much more concise
- Derive pretty printer
- Derive vistor-pattern and other type hooks for free

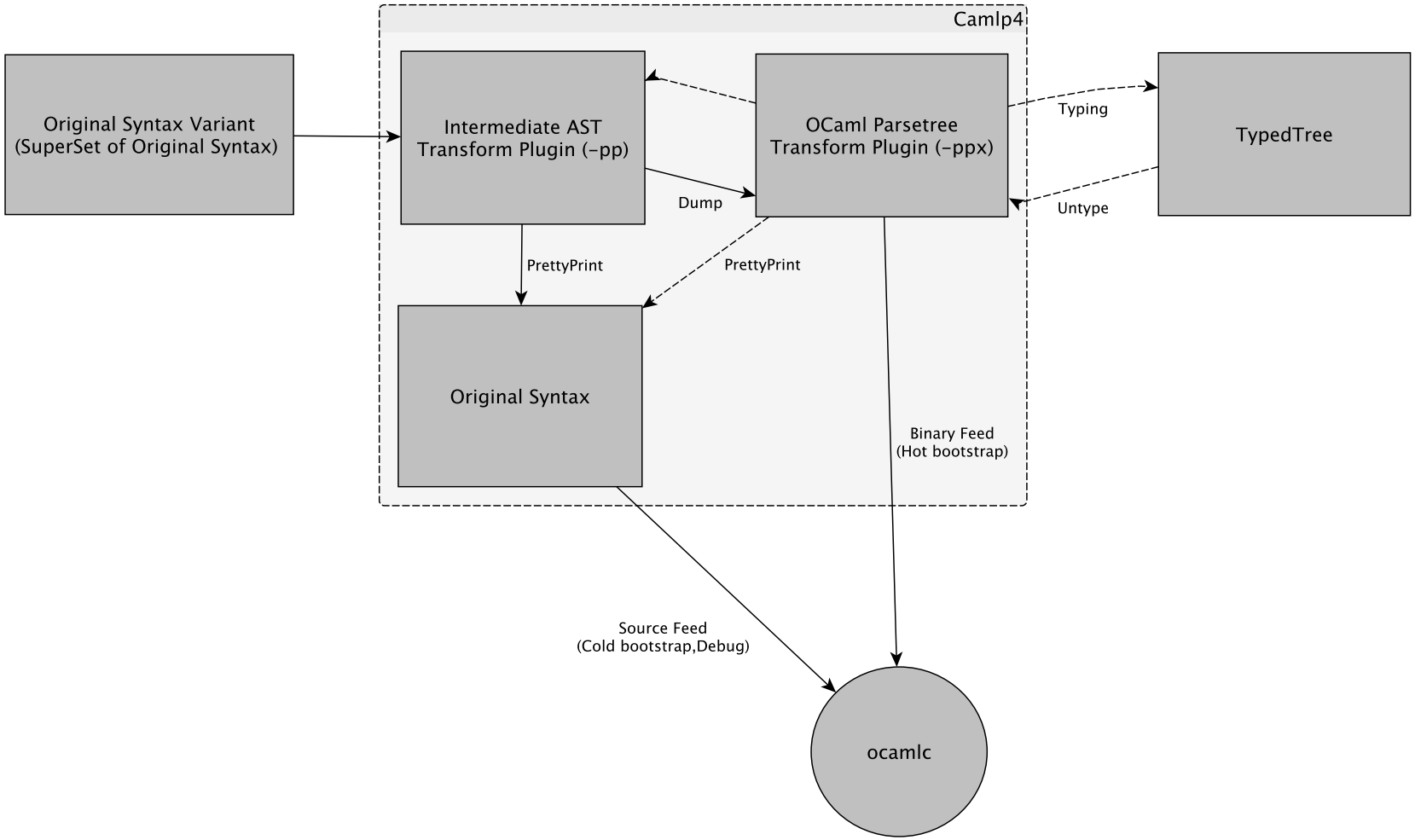
MetaDSL

- Define An Algebraic Data Type
- Make writing parsers much more concise
- Derive pretty printer
- Derive vistor-pattern and other type hooks for free
- Algebraic Data Type operations (open Algebraic Data Type)

Link to ocaml's compiler

```
<:eval#<
  <:fan_config<
 "name":"Eq";
 "extract": (fun {ty_expr;_} -> ty_expr);
 "compose": (fun x y -> <:expr< $x && $y >> );
 "arity": 2;
 "default": <:expr< false >>;
 "prefix":"eq_";
>> ;; >>;;
<:directive#<
  lang "ocaml"; keep on; show_code on; plugin_add "Eq";
 >> ;; (* evaluated at compile time *)
<<
type t = A of int * float
>> ; (* Inject Print Code  at compile time *)
```

Link to ocaml's compiler



Link to ocaml's compiler

- typed tree traversal

Link to ocaml's compiler

- typed tree traversal
- compile time evaluation

Link to ocaml's compiler

- typed tree traversal
- compile time evaluation
- non-intrusive code generation

Wishes!

There are lots of things not mentioned here, but we **can** keep bootstrapping CamlP4 with more and more mini DSLs to grow a more and more expressive system!