Getting started

Hongbo Zhang

May 29, 2013

Contents

The following post assumes the reader is already familiar with OCaml. If you are not familiar with OCaml, `http://ocaml.org/` is recommended for you to learn.

1 Installation

see Installation

2 What language does Fan speak?

Fan speaks OCaml natively, plus a few addons.

There are some minor differences between Fan's concrete syntax and OCaml though, the major differences is that Fan is more strict than OCaml.

Three particular points:

1. Parens are necessary for tuples

```
(** illegal *)
a,b
let a,b = f in
    body

(** correct syntax *)
(a,b )
let (a,b) = f in
    body
```

2. Parens or "begin" "end" necessary for semis

```
(* illegal *)
print_endline "a"; print_endline "b"

(** correct *)
(print_endline "a"; print_endline "b")
begin print_endline "a"; print_endline "b" end
```

3. First vertical bar is necessary for algebraic data type, pattern match.

```
(** illegal *)
type u = A | B
let f = function
    A -> "a"
  | B -> "b"
let f =
  match c with
    A -> "a"
  | B -> "b"
```

```
(** correct *)
type u =
  | A
  | B

let f = function
  | A -> "a"
  | B -> "b"

let f =
  match c with
  | A -> "a"
  | B -> "b"
```

4. $ is a reserved operator, please don't take it as a function.

3 Compiing with Fan

3.1 Hello world

Create a file hello.ml as follows:

```
print_endline "hell, Fan"
```

The compile is quite simple, make sure `fan.byte` or `fan.native` is in your search path.

```
$ ocamlc -pp 'fan.native' hello.ml -o test
$ ./test
```

As you may notice, adding ~-pp 'fan.native'~ flag is enough to switching to Fan. Using `fan.byte` or `fan.native` is up to you, for the time being, only the performance matters here. So, compiling with the following command line does also work.

```
$ ocamlc -pp 'fan.native' hello.ml -o test
```

3.2 First class lexer

Writing hello world is not very interesting, for the following example, we show you how DDSL fits into Fan. Suppose we want to write a lexical analyzier to filter nested comments in OCaml, the traditional way is to write a complex regex expression, or start a new file to write a lexer. The first way is hackish, inefficient, unmaintainable in the long run while the second way is too heavy weight, since lexer generator is a standard alone external DDSL which introduces another staging phase.

Within Fan, we show how easy it is now:

```
let depth = ref 0

let rec f  = {:lexer|
 | "(*" -> comment lexbuf
 | '"' -> (print_char '"'; string lexbuf)
 | _ as c -> (print_char c; f lexbuf)
 | ! -> exit 0
|}
and comment  = {:lexer|
 | "*)" ->
     if !depth = 0 then f lexbuf
     else begin
       decr depth;
       comment lexbuf
     end
 | "(*" -> incr depth
 | _ -> comment lexbuf
 | ! -> failwith "unterminated comment"
|}
and string = {:lexer|
 | '"' -> (print_char '"'; f lexbuf)
 | _ as c -> (print_char c; string lexbuf)
 | ! -> failwith "unterminated string"
|}

let _ = f (Lexing.from_channel (open_in "comment.ml"));;
```

Compiling is the same as the previous example 3.1

```
ocamlc -annot -pp 'fan.native' comment.ml -o comment
```

Here we see the lexer DDSL is first class construct in Fan, the user don't need to create a new file to isolate their lexer, it's as convenient as regex expression in perl. So it works in toplevel, it works with module system, and objects, that said, the user could make lexer reusable by using objects instead of functions.

Abot the internal of *lexer* DDSL, see DDSL:lexer.

4 Fan for toplevel

4.1 Playing with toplevel

If you have ocamlfind installed, the easiest way to explore fan is starting the toplevel:

```
# #require "fan.top";;
/Users/bobzhang1988/.opam/4.00.1/lib/fan: added to search path
/Users/bobzhang1988/.opam/4.00.1/lib/fan/fanTop.cma: loaded
# let a = {:exp-|3|};;
val a : FAstN.exp = `Int "3"
```

Here "exp-" is a built-in DDSL for quasiquotation, see DDSL:quotation

There are two directives added,

1. normal

   ```
   #normal;;
   ```

   This directive would restore the toplevel to the default behavior, it's useful sometimes, for examlpe, you want to load a normal ocaml file instead of file pre-processed by Fan.

2. fan

   ```
   #fan;;
   ```

   It will turn on the featurs of fan.

4.2 Playing with utop

Utop is a toplevel with nice auto-completion support, it's very helpful for explore some libraries which the user is not familiar with.

There is adapter for fan, namely ftop, available here: `https://github.com/bobzhang/ftop`

5 Writing plugins for Fan

There is a paper, which gives a high-level priciple about how Fan works, avail-

Fan: compile-time metaprogramming for OCaml

Hongbo Zhang
University of Pennsylvania
hongboz@seas.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

This paper presents Fan, a general-purpose syntactic metaprogramming system for OCaml. Fan helps programmers create *delimited, domain-specific languages* (DDSLs) that generalize nested quasiquotation and can be used for a variety of metaprogramming tasks, including: automatically deriving "boilerplate" code, code instrumentation and inspection, and defining domain-specific languages. Fan provides a collection of composable DDSLs that support lexing, parsing, and transformation of abstract syntax.

One key contribution is the design of Fan's abstract syntax representation, which is defined using polymorphic variants. The availability of intersection and union types afforded by structural typing gives a simple, yet precise API for metaprogramming. The syntax representation covers all of OCaml's features, permitting overloaded quasiquotation of all syntactic categories.

The paper explains how Fan's implementation itself uses metaprogramming extensively, yielding a robust and maintainable bootstrapping cycle.

1. Introduction

The Lisp community has recognized the power of metaprogramming for decades [37]. For example, the Common Lisp Object System [19], which supports multiple dispatch and multiple inheritance, was built on top of Common Lisp as a library, and support for aspect-oriented programming [11] was similarly added without any need to patch the compiler.

Though the syntactic abstraction provided by Lisp-like languages is powerful and flexible, much of its simplicity derives from the uniformity of s-expression syntax and the lack of static types. Introducing metaprogramming facilities into languages with rich syntax is non-trivial, and bringing them to statically-typed languages such as OCaml and Haskell is even more challenging.

The OCaml community has embraced syntactic abstraction since 1998 [12], when Camlp4 was introduced as a syntactic preprocessor and pretty printer. The GHC community introduced Template Haskell in 2002 [34], and added generic quasiquotation support later [22]. Both have achieved great success, and the statistics from hackage [35] and opam [1] show that both Template Haskell and Camlp4 are widely used in their communities.

Common applications of metaprogramming include: automatically deriving instances of "boilerplate" code (maps, folds, pretty-printers, etc.) for different datatypes, code inspection and instrumentation, and compile-time specialization. More generally, as we will see below, metaprogramming can also provide good integration with domain-specific languages, which can have their own syntax and semantics independent of the host language.

In Haskell, some of these "scrap-your-boilerplate" applications [20, 21] can be achieved through the use of datatype-generic programming [32], relying on compiler support for reifying type information. Other approaches, such as Weirich's RepLib [44], hide the use of Template Haskell, using metaprogramming only internally, while "template-your-boilerplate" builds a high level generic programming interface on top of Template Haskell for efficient code generation [6].

OCaml, in contrast, lacks native support for datatype-generic programming, which not only makes metaprogramming as in Camlp4 more necessary [24–26], but it also makes building a metaprogramming system particularly hard.

Despite their success, both Template Haskell and Camlp4 are considered to be too complex for a variety of reasons [5].[1] For example, Template Haskell's limited quasiquotation support has led to an alternative representation of abstract syntax [23], which is then converted to Template Haskell's abstract syntax, while GHC itself keeps two separate abstract syntax representations: one for the use in its front end, and one for Template Haskell. Converting among several non-trivial abstract syntax representations is tedious and error prone, and therefore does not work very well in practice. Indeed, because not all of the syntax representations cover the complete language, these existing systems are limited. Using Camlp4 incurs significant compilation-time overheads, and it too relies on a complex abstract syntax representation, both of which make maintenance, particularly bootstrapping, a nightmare.

In this paper we address this problem by describing the design and implementation of Fan[2], a library that aims to provide a *practical, tractable, yet powerful metaprogramming system* for OCaml. In particular, Fan supports:

- A uniform mechanism for extending OCaml with *delimited, domain-specific languages* (DDSLs) (described below), and a suite of DDSLs tailored to metaprogramming, of which quasiquotation of OCaml source is just one instance.
- A unified abstract syntax representation implemented via OCaml's polymorphic variants that serves as a "common currency" for various metaprogramming purposes, and one reflective parser for both the OCaml front-end and metaprogramming.
- Nested quasiquotation and antiquotation for the full OCaml language syntax. Quoted text may appear at any point in the grammar and antiquotation is allowed almost everywhere except keyword positions. Quasiquotation can be overloaded, and the meta-explosion function [42] is exported as an object [29] that can be customized by the programmer.
- Exact source locations available for both quoted and antiquoted syntax that are used by the type checker for generating precise error messages to help with debugging.
- Performance that is generally an order of magnitude faster than that of Camlp4.
- A code base that is much smaller and a bootstrapping process that is easier to maintain compared to that of Camlp4.

[1] Don Stewart has called Template Haskell "Ugly (but necessary)" [3].
[2] Fan is available from: https://github.com/bobzhang/Fan/tree/icfp

able here

5.1 Fan's quotation system

Fan's metaprogramming has a quotation system similar to Camlp4.

The differences lie in serveral following aspects

5.1.1 Concrete Syntax

- For quotation, Fan uses `{:quot@loc| |}`, while Camlp4 uses `<:quot@< >>`
- For antiquotation, Fan uses a single `$` or `$(...)`, while Camlp4 uses `$$`
- Fan supports nested quasiquotation, while Camlp4 does not. The following quotation is legal in Fan.

  ```
  {:exp|{:exp| $($x) |}|}
  ```

  Which is simliar to Common Lisp style macros.

  ```
  ``(,,x)
  ```

5.1.2 Abstract Syntax

The definition of Fan's abstract syntax is available here `fansrc/fAst.mli.html` Fan adopts polymorphic variants for encoding abstract syntax, there are several major benefits

- subtyping For user who wants to reuse part of Fan, the subtyping provides much more refined types
- constructor overloading There's no need to add a prefix for each type
- polymorphisim For example, in Fan, to get a location of an ast node, the user only need to write `loc_of`, without writing `loc_of_exp`, `loc_of_pat`, *etc.* Other ast processing libraries are also more generic compared with Camlp4.
- unqualified, easier to use.

The disadvantage: Error message is sometimes a problem, however, in Fan, the default quasiquotation will add type annotation automatically for the user, we have an optional quasiquotation without type annotations for advanced user.

For example:

```
(** with annotations *)
{:exp|3|}
(** after expansion *)
```

```
(‘Int (_loc, "3") : FAst.exp )
(** without annotations *)
{:exp’|3|}
(** after expansion *)
‘Int (_loc, "3")
```

1. location handling From the user's point of view, Fan has two abstract syntax representations, with or without locations. They have exactly the same semantics except handling locations. The abstract syntax without location is derived from abstract syntax with location.

   For example, the definition of `literal` (see fAst) are as follows:

   ```
   (** literal with locations [FAst]*)
   type literal =
     [ ‘Chr of (loc * string)
     | ‘Int of (loc * string)
     | ‘Int32 of (loc * string)
     | ‘Int64 of (loc * string)
     | ‘Flo of (loc * string)
     | ‘Nativeint of (loc * string)
     | ‘Str of (loc * string)]


   (** literal without locations [FAstN]*)
   type literal =
     [ ‘Chr of string
     | ‘Int of  string
     | ‘Int32 of  string
     | ‘Int64 of string
     | ‘Flo of string
     | ‘Nativeint of string
     | ‘Str of string]
   ```

   For ast without locations, the naming convention is a `N` postfix. Programming abstract syntax without caring about locations is way more easier. Location is important for debugging and meaningful error message, however, some scenarios, for example, code generation, don't need precise location.

   There are serveral helpful functions for location handling

(a) `loc_of` it would fetch location from any type in module FAst

(b) Objs.strip_[type]
For example, Objs.strip_ctyp would strip the location for ctyp, their type signature are as follows: (ftop would help!)

```
utop $ Objs.strip_case;;
- : FAst.case -> FAstN.case = <fun>
utop $ Objs.strip_exp;;
- : FAst.exp -> FAstN.exp = <fun>
```

(c) FanAstN.fill_[type] It's opposite to strip

```
utop $ FanAstN.fill_exp;;
- : FLoc.t -> FAstN.exp -> FAst.exp = <fun>
utop $ FanAstN.fill_case;;
- : FLoc.t -> FAstN.case -> FAst.case = <fun>
```

There is also a suite of quasiquotation for ast without locations.

```
(** with annotations, [-] means minus locations *)
{:exp-|3|}
(** after expansion *)
(‘Int  "3" : FAstN.exp )

(** without annotations *)
{:exp-’|3|}
(** after expansion *)
‘Int  "3"
```

5.2 Quotation DDSL

5.3 Parser DDSL