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

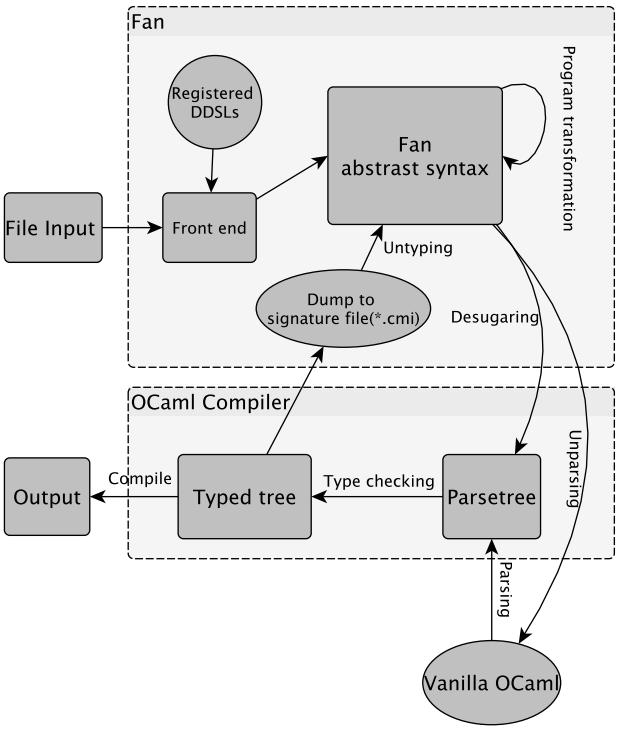

Figure 1. Fan workflow: registered DDSLs parse concrete text into Fan's `Ast`$[nt]$, which is fed to OCaml's standard compilation path.

2. The Fan Approach

2.1 Delimited domain-specific languages

The main syntactic abstraction mechanism provided by Fan is the notion of a *delimited, domain-specific language* (DDSL). For example, Fan's built-in *exp* DDSL provides quasiquotation support for OCaml expressions, so the concrete syntax `{:exp|3+4|}` denotes a "quoted" piece of abstract syntax that is represented by the Fan value (omitting location information):

```
`App (`App (`Id (`Lid "+"), `Int "3"), `Int "4")
```

More generally, the concrete syntax `{:lang|...text...|}` tells Fan to parse the string "`...text...`", interpreting it into abstract syntax via a function registered with the DDSL *lang*. We can therefore think of a DDSL as simply a function:

$$\texttt{parse}_{\texttt{lang}} : \texttt{string} \rightarrow \texttt{Ast}[nt]$$

Here, `Ast`$[nt]$ is the Fan type of abstract syntax for the OCaml grammar nonterminal *nt*. The function $\texttt{parse}_{\texttt{lang}}$ is invoked by the Fan front end whenever it encounters the DDSL syntax, and the resulting `Ast`$[nt]$ is spliced into the parse results, as shown in Figure 1. This process is called *compile-time expansion*.

Fan provides abstract syntax representations for all of OCaml's nonterminals, but the most commonly used include *exp* (expressions), *pat* (patterns), *stru* (toplevel structures), and *ep* (expressions that are patterns). The design of the abstract syntax `Ast`$[nt]$ plays a crucial role in Fan, and we will explain it in detail in Section 4.

One common use case for Fan DDSLs is to embed an arbitrary object language into OCaml. In this case, the parse function could be factored into two steps:

$$\texttt{parse}_{\texttt{object}} : \texttt{string} \rightarrow \texttt{OAst}$$

$$\texttt{compile} : \texttt{OAst} \rightarrow \texttt{Ast}[nt]$$

Here `OAst` is some OCaml datatype (typically an algebraic datatype) that represents the parse trees of the object language. The `compile` function translates object-language constructs into OCaml. The parser and compiler could be written using any OCaml code, but, as we shall see, the Fan namespaces *Fan.Lang* and *Fan.Lang.Lex* provide *lex* and *parser* DDSLs that aid in parsing.

Fan also provides quotation DDSLs to aid in compilation and translation tasks. For example, when implementing `compile`, it is useful to be able to pattern match against the object language's abstract syntax using notation for its concrete syntax [22]. This is provided by a quotation DDSL implemented using the function:

$$\texttt{lift} : \texttt{OAst} \rightarrow \texttt{Ast}[ep]$$

Although `lift`'s type looks similar to `compile`'s, the abstract syntax it returns is itself a representation of an `OAst` parse tree as a value of type `Ast`$[ep]$, *i.e.* it is a *quotation*, not an *interpretation*.[3]

Things become even more interesting when the object language and the meta language are the *same*. That is, when `OAst` *is* `Ast`$[nt]$. In this case, the `lift` function provides quotation of OCaml code, and the result allows for manipulation of quoted OCaml syntax directly, which enables type- and syntax-directed generation of OCaml programs. Indeed, much of Fan's power derives from a suite of DDSLs (in the *Fan.Lang.Meta* and *Fan.Lang.Meta.N* namespaces) that provide just such quasiquotation support; *exp* is one such DDSL. Moreover, the DDSL *fans* implements type-directed derivation of various utility functions such as visitor-pattern traversals and pretty-printers—it can be used to derive `lift` functions automatically.

Fan itself is implemented using its own DDSLs—this nontrivial bootstrapping support, though initially difficult to implement, leads to a robust maintenance strategy that we discussion in Section 6.

2.2 Examples

In this section, we show several examples of DDSLs built on top of Fan, and then turn to the DDSLs for quasiquotation support that are provided in Fan by default.

Embedding a subset of Prolog Our first example is a deep embedding of a foreign language—Prolog—into OCaml in such a way that OCaml code can invoke Prolog programs. Listing 1, which solves the N-queens problem, shows what the resulting DDSL looks like to the programmer.

Listing 1. Deep embedding of Prolog code.

```
1  {:prolog|
2  %:nqueens(+N,?Board)
3  nqueens(N,Board) :-
4    range(1,N,L),  permutation(L,Board),
5    safe(Board).
6
7  %:range(+Start,+Stop,?Result)
8  range(M,N,[M|L]) :-
9      M < N, M1 is M+1, range(M1,N,L).
10 range(N,N,[N]).
11
12 %:permutation(+List,?Result)
13 permutation([],[]).
14 permutation([A|M],N) :-
15   permutation(M,N1), insert(A,N1,N).
16
17 %:insert(+Element,+List,?Result)
18 %:insert(?Element,+List,+Result)
19 %:insert(+Element,?List,+Result)
20 insert(A,L,[A|L]).
21 insert(A,[B|L],[B|L1]) :- insert(A,L,L1).
```

[3] We explain why only *ep* is needed in `lift` instead of both *exp* and *pat* in section 5.

```
22
23 % :safe(+Board)
24 safe([_]).
25 safe([Q|Qs]) :- nodiag(Q,Qs,1), safe(Qs).
26
27 %:nodiag(+Queen,+Board,+Dist)
28 nodiag(_,[],_).
29 nodiag(Q1,[Q2|Qs],D) :-
30   noattack(Q1,Q2,D),  D1 is D+1,
31   nodiag(Q1,Qs,D1).
32
33 %:noattack(+Queen1,+Queen2,+Dist)
34 noattack(Q1,Q2,D) :-
35    Q2-Q1 =\= D,  Q1-Q2 =\= D.
36 |} ;;
37 let _ = nqueens
38   (fun b -> print_endline (string_of_plval b))
39   (Int 10);;
```

This example shows how the DDSL mixes OCaml syntax with Prolog syntax delimited by `{:prolog|` and `|}` brackets. The DDSL exposes Prolog queries as OCaml functions. In this case, when viewed as an OCaml function, `nqueens` takes an output continuation, corresponding to the `?Board` parameter, and provides the input `+N` as a wrapped OCaml value.

This example shows that there is an implicit contract between a DDSL and its host context—in this case, the interpretation of Prolog predicates as OCaml functions (and the order and types of the function arguments) is a part of the DDSL interface, and Fan does not itself attempt to verify that the interface is used correctly. In practice though, IDE support, which allows the programmer to locally expand (and collapse) the DDSL code statically during development, in conjunction with OCaml's type checker, which can be used to inspect the types of identifiers such as `nqueens`, allows the programmer to understand the contract between the DDSL and the host code.

The implementation strategy for the Prolog DDSL, which supports backtracking and unification, follows the first approach described above: a standard OCaml datatype is defined and then Fan's *parser* DDSL is used to write the Prolog parser (though any implementation could be used). The compilation strategy uses metaprogramming to translate (a representation of) each predicate to (a representation of) an OCaml function, following Robinson's techniques [31].

Unlike a Prolog interpreter, this *prolog* DDSL compiles the Prolog source via translation to OCaml and the OCaml backend. The resulting implementation yields performance that is significantly better than an interpreter, and that compares favorably with the SWI-Prolog compiler, a mainstream Prolog implementation ([46] (at least for this subset of the language).

Embedding a parser generator Our second example shows how Fan's *parser* DDSL is used to implement the front-end of the subset of Prolog mentioned above, as shown in Listing 2.

Listing 2. Parsing a subset of Prolog using the *parser* DDSL.

```
1  #import Fan.Lang;;
2  let g =
3    Gram.create_lexer
4      ~annot:"prolog"
5      ~keywords:["\\="; "is"; "=:=";
6                 "=\\="; "<"; ...]
7      ();;
8  {:create|(g:Gram.t) prog rule_or_mask rule
9      body args term mask var arg_mask|};;
10 {:extend|
11    body:
12      [ ":-"; L1 term SEP ","{r} -> r ]
13    (*.... other non-terminals *)
14    term:
15     {[ S{x}; "=";   S{y} -> ... (* action *)
16      | S{x}; "\\="; S{y} -> ...
17      | ... ]
18      (* ... other levels *)
19      "simple" NA   (* non-associative *)
20      [ `Uid x -> Var (x,_loc)
21      | ... ] } |};;
```

Line 1 imports the namespace of *Fan.Lang*, which provides two DDSLs for working with parsers, *create* and *extend*. Lines 2 through 7 create a grammar `g` and specify the keywords that are used to customize the behavior of the underlying lexer. In line 8, we register a list of nonterminals using the *create* DDSL, and between line 10 and line 20 we extend the grammar by groups of productions for each nonterminal.

For example, defining the non-terminal *body* is straightforward: it's composed of a list (`L1` stands for a list contains at least one element) of *term*s separated by commas following "`:-`". The `{r}` pattern matches against the result, binding the outcome to `r`.

Lines 14 through 20 demonstrate a stratified parser for Prolog *term* expressions. Each group of productions, delimited by "`[`" and "`]`", shares the same priority and associativity; priority increases from the top to bottom. The symbol `S` stands for the nonterminal itself, so `S{x}` binds a *term* to the variable `x` for use in the parser action. The `_loc` expression in line 20 stands for the parser location data, which is computed for the left-hand-side of the production and implicitly available to the action on the right hand.

Compared with stand-alone parser generators like `ocamlyacc`, Fan's *parser* DDSL has some advantages. One benefit is that only one tool (Fan) is needed, and staging is handled uniformly, so the build process is streamlined. It is also easy to parameterize the generated parser, since the *parser* DDSL generates normal OCaml toplevel phrases after compile-time expansion. This means that it works seamlessly with OCaml's modules and functors. Also, since type information is available after expansion, IDE support for inspecting OCaml code's types also work—for example, the OCaml type produced by the *term* is available to the programmer while interacting with the code.

Embedding a lexer generator Our third example demonstrates Fan's *lex* DDSL, which is useful for custom lexer generation. Listing 3 shows how this DDSL can be use to implement a program that checks well-nesting of OCaml-style comments.

Listing 3. First Class Lexer: Check nested comments

```
1  #import Fan.Lang.Lex;;
2  let rec token =  {:lex|
3    | eof  ->  exit 0
4    | "\"" ->  string lexbuf
5    | "(*" ->  comment 0 lexbuf
6    | _    ->  token lexbuf |}
7  and string = {:lex|
8    | "\"" ->  ()
9    | eof  ->  failwith "string not finished"
10   | _    ->  string lexbuf |}
11 and comment n = {:lex|
12   | "*)" ->
13      if n <> 0 then comment (n-1) lexbuf
14   | eof  -> failwith "comment not finished"
15   | _    -> comment n lexbuf |};;
```

The syntax for the *lex* DDSL is simple: a *lex* expression consists of a sequence of rules, where the left-hand side is a regular expression (that follows the same conventions as `ocamllex`), and the right-hand side is a normal OCaml expression. The `lexbuf` variable mentioned in line 4 is an implicit parameter that stands for the current lexing buffer—it also follows the same conventions as

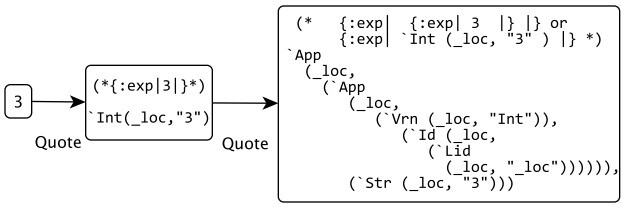

Figure 2. Nested quotation.

`ocamllex`. The *lex* DDSL shares the same advantages of host–DDSL integration as the *parser* above, but unlike *parser*, which generates top-level OCaml phrases, the *lex* DDSL generates OCaml expressions. In some cases, annotating the name of each DDSL is still too heavyweight in practice, *with_stru* DDSL is introduced to set the default name of DDSL for a toplevel phrase(*stru*), so the example above could be even simplified as below:

Listing 4. Simplified Lexer

```
#import Fan.Lang.Lex;;
{:with_stru|lex:
let rec token =  {| ... |}
and string    =  {| ... |}
and comment n =  {| ... |} |};;
```

Quasiquotation As a final example, we return to the *exp* quasiquotation DDSL. As we mentioned earlier, Fan provides support for nested quasiquotation of all of OCaml's syntactic categories, which allows programmers to conveniently manipulate OCaml programs by writing quoted concrete syntax.

Without support for nested quasiquotation, writing metaprograms can be extremely cumbersome. For example, Figure 2 shows the results of two-levels of quotation for the simple expression `3`. One level of quotation yields `` `Int(_loc,"3") ``, which, since Fan is a homoiconic metaprogramming system, is itself a legal expression. That small snippet of abstract syntax has a rather larger quotation—it gets "exploded" as the figure shows. It is much simpler to write `{:exp| {:exp|3|} |}`.

Quotation alone is not very interesting. *Antiquotation* allows the user to quote a piece of program text, but selectively escape elements out of the quotation. Quotation combined with antiquotation, *i.e. quasiquotation*, provides a very intuitive way to abstract over code.

Listing 5. Antiquotation

```
let f  = function
  | {:exp| $x +  $y |} ->
    {:exp| $x +. $y |}
  | x -> x


let f =  function
  | `App (_loc,
      `App (_,
        `Id (_,
          `Lid (_,"+")), x),y)  ->
    `App (_loc,
      `App (_loc,
        `Id (_loc,
          `Lid (_loc, "+.")), x), y)
  | x -> x
```

Listing 5 shows how antiquotation is used in (a slightly artificial, but small) program transformation. This function transforms a piece of abstract syntax by rewriting a top-level occurrence of integer $+$ to floating-point $+.$. Fan's antiquotation allows the pattern variables from the OCaml phrase to be spliced into the quoted syntax. The first version of `f` clearly demonstrates the benefits of working with quasiquotation rather than explicit abstract syntax.

These examples give a taste of the kinds of metaprogramming that are possible with Fan. Other, more sophisticated uses, like the ability to generically derive boilerplate code are described in Section 6.2—they are not only useful for bootstrapping Fan itself, but are also available for general-purpose programming.

3. Design Guidelines

Designing and implementing a compile-time metaprogramming system that is capable of expressing the kinds of examples shown above is a nontrivial task, and there are several desirable traits that are sometimes in tension with one another. Here we identify Fan's primary design objectives and discuss some of the tradeoffs involved.

Transparency, and generality for the user Our first goal is to make metaprogramming convenient and tractable for both the users of DDSLs and their developers. This means that quasiquotation support should be provided for all constructs of the language, and, moreover, the abstract syntax representation should be as close to the concrete syntax as possible. For transparency, the user's tools (like the IDE) should be able to locally expand a DDSL's action on quoted text to obtain compilable, vanilla OCaml without really compiling the program—this is crucial for debugging metaprograms.

Robustness and composability Metaprogramming is a *language-level* capability, not a library- or module-level capability. The means of constructing and manipulating abstract syntax representations should therefore be globally available and stable in the sense that local definitions should not perturb the meaning of already-constructed AST values. Similarly, DDSLs should be isolated from one another, and composing multiple DDSLs should be possible and yield predictable behavior.

Maximum availability and minimal dependency The metaprogramming system should be independent of the compiler and it should be possible to separate the "compile-time dependencies," *i.e.* those libraries needed by the implementation of a DDSL, from the "link-time dependencies," *i.e.* those libraries needed to run the OCaml program generated by using the metaprogramming facilities. Correspondingly, Fan should be distributable as a library (not as a compiler patch) and it should be possible to remove even the compile-time dependency on Fan by exporting vanilla OCaml from Fan source.

Simplicity for the maintainer Implementing a metaprogramming system involves writing a lot of boilerplate code for extremely large datatypes that represent abstract syntax, often with multiple variants (for example with and without source file location information). Maintaining such a large program is both tedious and error prone. The metaprogramming system implementation itself should therefore be automated as much as possible. In a similar vein, only a single representation of abstract syntax should be exposed to the user for both the surface syntax and metaprogramming, and only a single parser should be used for both the front end and quasiquotation.

Performance Using metaprogramming should not incur significant cost in terms of compile-time performance. In particular, the performance hit should be "pay as you go" in the sense that only those DDSLs used a program should affect its compilation time.

Some of these goals are synergistic. For example, minimizing compile-time dependencies and reducing the need for dynamic

loading improve compilation times significantly. Likewise, simplifying the representation of the abstract syntax streamlines things for both DDSL implementors and the Fan maintainers. Simplifying the abstract syntax also enables the automation of boilerplate code generation, which is used in Fan to derive tens of thousands lines of code for various purposes, including overloaded quasiquotation, visitor-patterns, generic maps, fold,s zips, pretty-printing, systematic antiquotation support, *etc*.

On the other hand, making the abstract syntax too simple is at odds with the desire for transparency, since doing so might obscure the connection between the concrete syntax and its abstract representation. The most delicate component of any metaprogramming system is therefore the choice of how to represent syntax, since that design must balance these tradeoffs while achieving all of the goals outlined above. We turn to this issue next.

4. Design of abstract syntax

Metaprogramming is mainly about writing or manipulating other programs as data. Though programs can be represented as plain strings, and indeed there are several widely used textual or token-based preprocessors, *e.g.* the C preprocessor [36], it is common practice to use a hierarchical data structure to encode the abstract syntax. One of the key differences between different metaprogramming systems is what kind of structured data is adopted for the underlying representation. There are many representation options, including: higher-order abstract synax [38], nominally typed [13, 34], structurally typed or just untyped data structures [8].

4.1 Problems with previously used representations

Common Lisp-style s-expressions have several characteristics that make them suitable for metaprogramming tasks. Their structure, consisting only of atoms and anonymous trees of atoms, along with the lack of semantically meaningful tags makes this representation very uniform. As a consequence, because most Lisp dialects are dynamically typed, all the program transformations on the abstract syntax share the same signature: take in an (untyped) s-expression to produce another (untyped) s-expression. However, this uniformity is also a drawback: to represent the rich syntax of OCaml using only s-expressions would require a cumbersome encoding, which is at odds with the goal of transparency. Also, the lack of type information makes debugging s-expression-based metaprograms much more difficult, since the compiler can't aid the programmer in catching missing cases or mis-associated nesting.

Both Template Haskell and Camlp4 adopt algebraic data types as the basis of their abstract syntax. Compared with s-expressions, algebraic data types are safer and more precise for symbolic manipulation. They support type checking, deep pattern matching, and exhaustiveness checks. The data constructors of an algebraic data type give each node a distinct meaning, which is helpful for generating better error messages. Splitting the whole abstract syntax into different sub-syntactic categories helps to make the metaprograms more precise: the type signature can tell the meta-programmer whether a piece of abstract syntax is a pattern, an expression or type declaration at compile time. However, algebraic data types, as found in Haskell and OCaml, are defined *nominally*. Nominal type systems require explicitly named type declarations, and type equivalence is determined by the name, not by the structure of the type. Though nominal typing has many legitimate uses—it prevents "accidental" type equivalences that might be induced by coincidental sharing of structure—for metaprogrammming, nominal typing presents several problems.

One difficulty is is that standard implementations of Damas-Hindley-Milner-style type-inference (as currently used in Haskell and OCaml) disallow sharing of constructor names among distinct algebraic data types. This introduces a practical problem when dividing a large abstract syntax into different subcategories. For example, as Listing 6 shows, although OCaml expressions and patterns share much of their concrete syntax, the corresponding AST nodes (used in Camlp4) are forced to have distinct constructor names.

Listing 6. Non sharable abstract syntax

```
type pat =   (* Pa stands for pattern*)
  | PaNil        of (loc)
  | PaId         of (loc * ident)
  | PaChr        of (loc * string)
  | PaInt        of (loc * string)
  | ... (* more constructors *)

type exp =   (* Ex stands for expression*)
  | ExNil        of (loc)
  | ExId         of (loc * ident)
  | ExChr        of (loc * string)
  | ExInt        of (loc * string)
  | ... (* more constructors *)
```

This naming overhead is leaked to clients of the abstract syntax as well: all of the functions that work with the abstract syntax have to be duplicated too. For example, Camlp4 provides functions to get the location data for each AST node:

Listing 7. Duplicated location functions in Camlp4

```
val loc_of_exp  : exp  -> loc
val loc_of_pat  : pat  -> loc
val loc_of_ctyp : ctyp -> loc
(* more syntactic categories ... *)
```

For every AST operation, a collection of nearly-identical functions, one for each syntactic category, must be implemented. Although we could conceivably automate the generation of such functions[4], that still isn't sufficient to hide the duplication from clients. Listing 8 gives an example of why this verbosity is necessary—the code shows typical "smart" constructors that build larger pieces of syntax out of smaller ones, suitably combining the location data:

Listing 8. Duplicated clients due duplicated location functions

```
let com_exp a b =
  let loc = Loc.merge
     (loc_of_exp a) (loc_of_exp b) in
  Ast.ExCom(loc,a,b)
let com_pat a b =
  let loc = Loc.merge
   (loc_of_pat a) (loc_of_exp b) in
  Ast.PaCom(loc,a,b)
(* more syntactic categorie ... *)
```

The issue is that such functions must fix the type of their inputs *a priori*. OCaml's lack of ad-hoc overloading for algebraic datatypes prevents any chance of code reuse and therefore requires the API to be polluted with nearly identical functions. Even with ad-hoc polymorphism support *à la* Haskell's type classes [18], the library maintainer still has to define one typeclass and write each instance per syntactic category, even though they all behave analogously.

This problem is exacerbated further when we add support for antiquotation to the abstract syntax. Doing so (see the discussion in Section 5) requires adding an extra constructor per syntactic category, the names of which also cannot be shared. Since it is useful to have versions of the types both with and without antiquotation (and, similarly, both with and without source location information), the proliferation of constructor names is multiplicative.

The redundancy described above caused by nominal typing *discourages* the designer from refining a coarse-grained syntactic

[4] Or break the abstraction boundary using Obj.magic (like Camlp4)

category into more precise syntactic categories, yet such refinement is particularly helpful for metaprogramming. For example, in Camlp4, all of the type-related syntactic categories are lumped together into one category, *i.e. ctyp*, which makes type-directed code generation particularly hard to reason about.

Another problem with using a nominal type to represent the syntax is that the name and constructors of the type should be defined or imported prior to their use. Listing 9 shows how quotation expansion might introduce a dependency on a (previously unmentioned) module `Ast` that defines the abstract syntax type.

Listing 9. Quotation Expander

```
let a = {:exp| 3 + 4 |}
(* After meta explosion *)
let a =
  Ast.ExApp
    (_loc,
      (Ast.ExApp
         (_loc,
           (Ast.ExId
             (_loc,
              (Ast.IdLid (_loc, "+")))),
           (Ast.ExInt (_loc, "3")))),
      (Ast.ExInt (_loc, "4")))
```

This need to define the name of the type is inconsistent with the goal of allowing metaprogramming to be visible at the language level without introducing any new dependencies. A related problem is that packaging the abstract syntax type by name creates the possibility of name capture during quotation expansion. Listing 10 shows that shadowing the `Ast` module interferes with the semantics of quasiquotation, since the new module declaration overrides the previous definition.

Listing 10. Module shadowing changes quotation semantics

```
module Ast = struct
  (* Rebind the module Ast to  shadow the
     original Ast module *)
end
(* Ast refers to the new Ast module now *)
let a = {:exp|  3 +  4 |}
```

Whether such dynamic scoping (or anaphoric macro) is a feature or bug in metaprogramming is debatable (we discuss issues of hygiene in section 8), the semantics of quasiquotation for the language itself should be consistent—we argue that no such rebinding should be allowed.

4.2 Structural typing, subtyping and polymorphism

Based on the observations above, Fan adopts *polymorphic variants* [17, 28] for representing its abstract syntax.

Polymorphic variants are a good fit for several reasons. Polymorphic variants permit deeply nested pattern matching, which is one of the main benefits of algebraic datatypes. Unlike algebraic datatypes, however, they also admit structural subtyping, which allows us to refine the syntactic categories of the grammar, yet still conveniently work with coarser-grained categories when needed. Importantly, polymorphic variant data constructors can be shared across type definitions and even manipulated algebraically to form explicit union types. Moreover, since the data constructors are global there is no risk that they will be shadowed or otherwise redefined, and using them incurs no additional linking dependencies. Finally, polymorphic functions naturally generalize to all subtypes of their expected inputs, which allows code to be reused consistently.

Listing 11 shows how sharable constructors (in this case for literals) and union types makes Fan's representation much cleaner than that shown in Listing 6:

Listing 11. Shared type constructors

```
type literal =
  [ `Chr       of (loc * string)
  | `Int       of (loc * string)
  | ...]
type exp =
  [ literal
  | ... (* more data constructors ... *) ]
type pat =
  [ literal
  | ... (* more data constructors ... *)]
```

Some of Fan's syntactic categories are large union types, which enables sharing of large swaths of boilerplate code. For example, Listing 12 shows how extracting location data is implemented in Fan. (In practice `loc_of` is automatically generated using Fan's *deriving* DDSL; see section 6.2.)

Listing 12. Union types

```
type syntax =
  [ exp | pat ...]
let loc_of (x:syntax) =
  match x with
  | `Chr(loc,_)
  | `Int(loc,_)
  | ... -> loc
```

Here, the union type *syntax* is the supertype containing all of Fan's syntactic categories[5].

Structural typing also provides a uniform API for metaprogramming that prevents the proliferation of utility functions across syntact categories. For example, contrast the code in Listing 8 with the first few lines below:

Listing 13. Polymorphic API

```
let (<+>) a b =
  Loc.merge (loc_of a) (loc_of b);;
let com a b =
  let _loc = a <+> b in
    `Com(_loc,a,b);;
let rec list_of_com x acc =
   match x with
   |`Com(_,x,y) ->
      list_of_com x (list_of_com y acc)
   | _ -> x::acc;;
let rec com_of_list = function
  | [] -> failwithf "com_of_list empty"
  | [t] -> t
  | t::ts -> com t (com_of_list ts);;
```

Structural subtyping also opens up the new possibility of selectively reusing part of the host language in a DDSL without losing type safety. As the *lex* DDSL (Section 2.2) shows, one common idiom for DDSLs is combining part of the host language with some new features. For a nominal algebraic datatypes, it's impossible to use only some of the constructors from the original type definition without defining a data type for the new syntax and writing embedding/projection functions for transforming the representations back and forth. In Fan, when introducing a new DDSL, the author can pick a subset of a specific syntactic category instead. As we will show later (Section 5), for the quasiquotation DDSL, a smaller new type *ep*, which is a subtype of both expressions and patterns, is chosen for the metaprogramming DDSL.

[5] In practice, OCaml's type checker could infer the type of *loc_of* without generating the syntax type

4.3 Other considerations

Having chosen to use polymorphic variants as the basis for representing abstract syntax, there are still several design decisions to be made.

Desugaring considered harmful for syntactic metaprogramming The first choice is to what what degree the "abstract" syntax reflects semantic content versus syntactic content. For compiler writers, eliminating redundancy present in the syntax both makes the abstract syntax type more compact and makes subsequent stages (such as typechecking) easier. It is common practice to de-sugar different concrete syntax into the same abstract syntax, for example, the original OCaml front end translates both `let open Module in exp` and `Module.(exp)` into the same representation as shown in Listing 14:

Listing 14. Local module desugaring

```
| LET OPEN mod_longident IN seq_exp
  { mkexp(Pexp_open(mkrhs $3 3, $5)) }
| mod_longident DOT LPAREN seq_exp RPAREN
  { mkexp(Pexp_open(mkrhs $1 1, $4)) }
```

However, meta-programmers should not be expected to know or be allowed to rely on this implementation detail. Fan therefore adopts the philosophy that the abstract syntax should not introduce a new abstraction layer over the concrete syntax—the abstract syntax should be isomorphic to the concrete syntax, which accords with the goal of transparency for the user.

Shared constructors with common conventions Fan also pursues aggressive sharing of constructors across semantically unrelated, but syntactically similar categories. For example, the following connectives appear in many different contexts of the OCaml grammar:

Listing 15. Frequently used connectives

```
'Com  (* a b -> a  ,  b  *)
'Sem  (* a b -> a  ;  b  *)
'Par  (* a   -> (  a  )  *)
'Sta  (* a b -> a  *  b  *)
'Bar  (* a b -> a  |  b  *)
'And  (* a b -> a and b  *)
'Dot  (* a b -> a  .  b  *)
'App  (* a b -> a     b  *)
```

To enable sharing of code, for example as in the `list_of_com` and `com_of_list` functions shown in Listing 13, such constructors are expected to conform to some (unenforced) conventions about their use. For example, every occurrence of `'Com of (loc * ty1 * ty2)` is expected to follow the convention that `ty1 = ty2`, which lets it interact smoothly with the provided generic code for conversion to and from lists.

Planning for metaprogramming In light of the considerations above, the Fan abstract syntax representation introduces a total of (roughly) 170 distinct constructors collected into 53 syntactic categories, which permits extremely precise typing constraints. For example, Fan divides the class of OCaml types *ctyp* (which is just one category in Camlp4) into 10 different subcategories, so that metaprograms that want to process OCaml's types structures (*e.g.* for type-directed programming) need not necessarily consider *all* of OCaml's types. The shear number of constructors is manageable because working with them directly is rare—most of the time you manipulate the abstract syntax via quotation using familiar OCaml notation.

There are two further ways in which Fan's abstract syntax design supports metaprogramming, both having to do with nested quasiquotation. First, for antiquotation support, in which it is convenient to be able to splice together abstract syntax trees, it is much cleaner to work with symmetric, tree-structured data rather than asymmetric, linear data types like OCaml's built-in lists. That is, append is simpler to work with "append" than "cons," largely due to typing constraints. Consequently, all linear data structures in Fan are represented in a symmetric way as illustrated in Listing 16, such design also follows the common conventions we mentioned above:

Listing 16. Symmetric data structure

```
type binding =
  ['And   of (loc * binding * binding)
  |'Bind  of (loc * pat * exp)
  | ant  ]
```

Second, Fan's abstract syntax tries to minimize the set of language constructs used: it is not parametrized by any type variables, and it doesn't doesn't use any parameterized types, it doesn't use OCaml's records, lists, or even option types. It uses recursively-defined, uniformly structured variant types, and that's it—such structural regularity is essential for tractably implementing Fan's meta-explosion operations and bootstrapping, which are described in the next Section.

5. Overloaded quasiquotation

Meta-explosion in Fan To implement quasiquotation, Fan uses *meta-explosion*, which lifts a DDSL's abstract syntax to the meta level at compile-time, as illustrated in Figure 2. Meta-explosion is well studied in the untyped setting [42], and the same basic principle underlies other compile-time metaprogramming systems [12, 22, 34, 41]. In Fan, the meta explosion operations are encapsulated in objects [29], which means that type- and data-constructor-specific behavior can even be overridden by the user.

Listing 17 shows the (hard-coded) meta-explosion object for the primitive types—it provides one method per type:

Listing 17. Meta-exploding primitive types.

```
class primitive = object
  method int _loc i : ep  =
    'Int(_loc,string_of_int i)
  method int32 _loc i : ep =
    'Int32(_loc,Int32.to_string i)
  (* ... other primitive types *)
end
```

The result of a method like `primitive#int` could be given a coarse-grained type like *syntax* (since the result is indeed valid OCaml syntax). However, due to Fan's use of structural typing, it is possible to introduce an intersection type, *ep*, which precisely describes the co-domain of the meta-explosion operations—this precision makes writing meta-meta-programs (*i.e.* DDSLs that manipulate meta-programs) much simpler—a fact that is exploited heavily in Fan's bootstrapping.

For Fan, the co-domain of meta-explosion is a strict subtype of the intersection of expressions (*exp*) and patterns (*pat*), as depicted in Figure 4. Listing 18 shows the type itself, which has only 18 distinct constructors (including 7 for `literal` values). This is a 10x reduction in the number of constructors when compared to the full *syntax*.

Listing 18. Precise co-domain of meta explosion

```
type ep =
  [ 'Id      of (loc * ident)
  | 'App     of (loc * ep * ep)
  | 'Vrn     of (loc * string)
  | 'Com     of (loc * ep * ep)
  | 'Par     of (loc * ep)
```

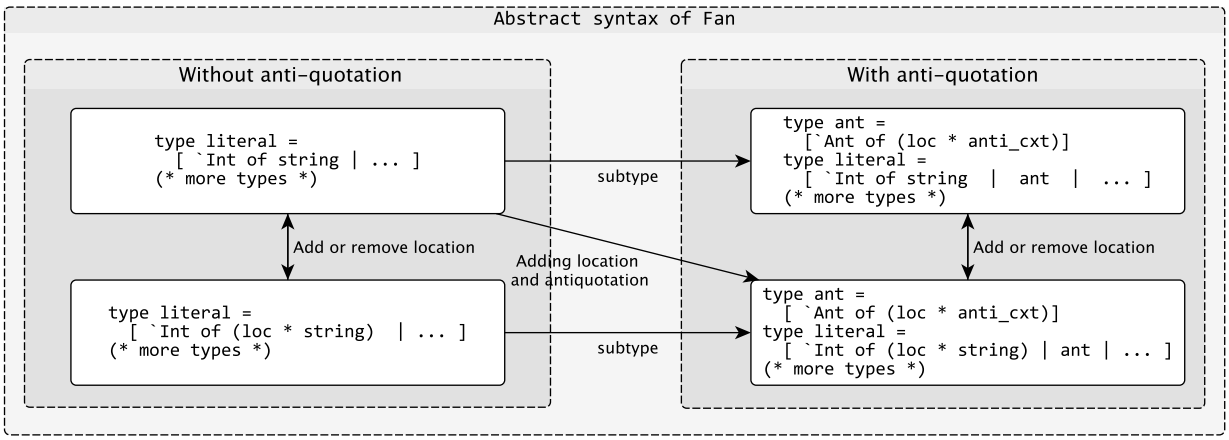

Figure 3. type algebra of abstract syntax

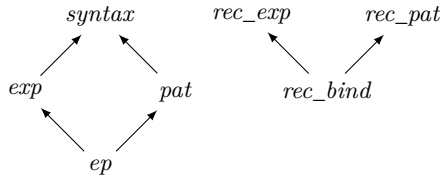

Figure 4. Subtype relation of explosion syntax.

```
    | `Sem     of (loc * ep * ep)
    | `Array   of (loc * ep)
    | `Record  of (loc * rec_bind)
    | `Any     of loc
    | literal
    | ant ]
and rec_bind =
    [ `RecBind of (loc * ident * ep)
    | `Sem     of (loc * rec_bind * rec_bind)
    | `Any     of loc
    | ant ]
```

Having precisely identified *ep* as the co-domain of meta-explosion, we can use the corresponding quotation DDSL to implement meta-explosion for the full *syntax*, as demonstrated in listing 19.

Listing 19. Meta explosion for the whole syntax

```
class meta = object(self)
  inherit primitive
  method exp _loc x  =
   match x with
    |#literal y ->
      self#literal _loc y
    |`Id(loc,s) ->
      {:ep|`Id
             ($(self#loc    _loc loc),
              $(self#ident  _loc s))|}
    |`Vrn (loc,s) ->
      {:ep|`Vrn
             ($(self#loc    _loc loc),
              $(self#string _loc s))|}
    | ... (* other cases *)
  (* ... other methods *)
end
```

There are a few additional observations to make. First, the *ep* quotation DDSL is itself defined in terms of meta, so in the initial version of Fan we must implement meta-explosion for the constructs in Listing 18 by hand—having to do so for only 18 constructors is a big win.

Second, the implementation of the `meta` object shown above is so mechanical that it is derived automatically by a DDSL (see Section 6.2), but that requires support for nested meta-explosion. However (if we ignore locations for the moment), meta-explosion has type $syntax \rightarrow ep$. But, since *ep* is a subtype of *syntax*, meta-explosion is chainable, which means explosion composition *i.e.* $explosion \cdot explosion$ or $explosion^n$ is freely available for nested quotation.

Finally, the precise co-domain not only makes program transformation on the meta-level abstract syntax easier to handle, but it also provides a stable API that will not break, even if the underlying language (*i.e.* OCaml) changes. This is because even if new features are added to *syntax*, the co-domain for $explosion1$ remains the same, so long as we follow Fans conventions about using polymorphic variants to represent the new constructs.

Systematic antiquotation Support One new wrinkle in the types described above is the inclusion of the `ant` type, which contains a single variant `` `Ant of (loc * anti_cxt) `` that supports antiquotation in a systematic way. Because the effect of antiquotation is to "escape" from quotation back to the outer language, its effect with respect to meta-explosion is (by default) simply to act as the identity function, as shown in Listing 20.

Listing 20. Antiquotation support

```
class primitive' = object
  inherit primitive
  method ant x = x
end
```

In practice, it is useful to allow custom processing of antiquoted values, for instance to inject some bit of syntax (like parentheses or back-ticks) into the abstract syntax. Fan supports this by filtering away the antiquotation after meta-explosion, but providing hooks for users to define their own processing. The filter, shown in Listing 21, is implemented as an object that inherits a generic visitor of the syntax and dispatches via the names of the antiquotation syntax to invoke custom rewriters.

Listing 21. Named Antiquotation support

```
class filter = object
  inherit FanAst.map
  method! pat x =
   match x with
    |`Ant(_loc, cxt) -> begin match ...
      (* project the named antiquotation from
          the context *) with
      |("par",_,e)  ->
        {:pat| `Par ($(mloc _loc), $e)|}
      |("lid",_,e) -> ...
    end
  (* override other methods*)
end
```

In this case, the filtering `{:exp|$par:a|}` would inject parentheses when expanding the quotation, yielding: `` `Par (_loc, a) ``

A family of overloaded quasiquotation syntax There are actually four related versions of Fan's abstract syntax, each of which is useful for different purposes, depending on whether antiquotation support or precise location information is needed. Figure 3 shows the relationships among the four versions. The richest includes both precise location information and antiquotation support—it is the standard representation generated by the parser. Filtering, as described above strips out the antiquotation nodes; the version with locations (in the lower left) is what is sent to OCaml's backend. The variants without location information are convenient when programmatically generating abstract syntax, in which case there is no corresponding source file to draw location data from.

Only the type declarations shown in the bottom right of the figure are implemented manually; all of the other representations are derived automatically.

Listing 22. Overloaded quasiquotation

```
#import Fan.Lang.Meta;;
{:exp| $a + $b |}
(* Expanded code *)
`App (_loc,
  (`App (_loc, (`Id (_loc, (`Lid (_loc, "+")))
      ), a)), b)
#import Fan.Lang.Meta.N;;
{:exp|$a+$b|}
(* Expanded code *)
`App(`App(`Id(`Lid "+"),a),b)
```

Essentialy, the quasiquotation DDSL for each syntactic category α in Fan is composed of three compoents as we described above: $filter\#\alpha \cdot meta\#\alpha \cdot parse_\alpha$. The combinations of automatic derviation for the meta explosion *i.e. MetaObj* and transformation between different syntaxes *e.g. RemoveType* make overloading of quasiquotation DDSLs(see Section 6.2) automatically derived. The quasiquotation DDSLs for the different versions of the syntax reside in different namespaces, as shown in Listing 22.

6. Implementation

6.1 Architecture

As shown in Figure 1, Fan includes a parser for OCaml[6], a desugaring function that desugars Fan's abstract syntax to OCaml's internal representation, an unparsing engine that pretty prints Fan's abstract syntax into compilable OCaml code, an untyping engine that converts binary signatures into Fan's abstract syntax and a collection of DDSLs.

The workflow is simple: Fan parses the file input, performs the compile-time expansion, and then desugars Fan's abstract syntax into the OCaml compiler's internal representation. Alternatively,

[6] There are some minor differences between the concrete syntax of Fan and OCaml

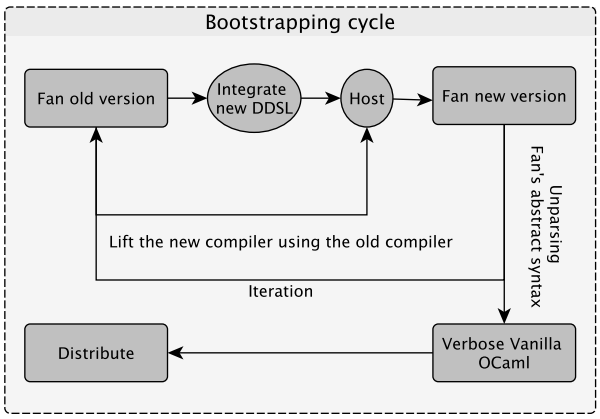

Figure 5. Bootstrapping

Fan can unparse and pretty print its abstract syntax to a compilable source program, removing the dependency on Fan. Fan itself is distributed this way. Such a simple workflow also results in an explicit compile-time runtime: *explicitly* registered DDSLs.

A user's DDSL can be either statically linked or dynamically loaded, but DDSL registration does not have any side effects if the user doesn't actually use it, nor is there implicit dynamic loading. Fan does not have cross-stage persistence, but the benefit is that it does not impose that the code generator for a DDSL depend on the DDSL's runtime, which might trigger a dependency chain at compile time. As a result of Fan's static characteristics, it works well with IDE *e.g.* Emacs(see Section 7).

As Figure 1 shows, without touching the source tree, Fan can still extract the abstract syntax from the binary interface (*i.e.* `cmi`) file for each compilation unit. Such non-invasive code generation is quite similar to Template Haskell's reification. Fan's reification is conservative and will not break abstraction boundaries, though.

6.2 Bootstrapping and evolution

Part of the power of Fan lies in its support for self-extensibility via bootstrapping. Bootstrapping, however, is not just an intellectual game—it is necessary to keep the code base manageable while developing more features. Fan's development history demonstrates the value of bootstrapping: it grew from a front end with only an abstract syntax and a parser to a powerful metaprogramming system that has tens of DDSLs to provide a whole technology stack for embedding new DDSLs. Fan treats OCaml as a cross platform *assembly language*. As Figure 5 shows, after each bootstrapping cycle, all the features available in the current version go into the next.

Listing 23. Automation meta explosion

```
{:fans|
  derive(Map2 Fold2 OIter
     Map Fold  OPrint OEq  Print MapWrapper
     MetaObj RemoveLoc (* more .... *) );|};;
{:ocaml| {:include|"src/Ast.mli"|}; |};;
```

Listing 23 shows how different DDSLs can be composed together to remove the boilerplate and derive utilities for various purposes. The *fans* DDSL accepts the type declarations captured by the *ocaml* DDSL and applies different code generators to the captured type declarations. The *include* DDSL is simply a C-like include macro, except that it also checks whether the syntax is correct. Fan has a deriving framework which makes it very easy to write a customized type-directed code generators, which we explain next.

Eq generator The Eq generator derives equality functions for any datatype expressible in OCaml. Listing 24 shows (a fragment of) how Fan's high-level interface is used to write a customized code generator. The heavy-lifting of name generation and type-structure analysis is hidden in the function `gen_stru`, which is parameterized by several call-backs that say how to process various datatypes. The `id` argument (line 9) indicates that all generated functions will be prefixed with `eq_`, and the `arity` argument says that equality takes two inputs. Lines 1 through 7 show how the case of a variant type is handled, namely by generating code that compares each field pointwise: the recursive calls to appropriate `eq_X` function are combined using `&&`. Snippets of code representing the results of those recursive call are given to `mk_variant` by `gen_stru` in the form of a record containing, among other things, the `info_exp` field.

Listing 24. Eq generator

```
1  let mk_variant _cons fields =
2    match fields with
3    | [] -> {:exp|true|}
4    | ls ->
5      List.reduce_left_with
6      ~compose:(fun x y -> {:exp| $x && $y|})
7      ~project:(fun {info_exp=e;_} -> e) ls ;;
8  let gen_eq =
9    gen_stru ~id:(`Pre "eq_") ~arity:2
10   ~mk_tuple ~mk_record
11   ~mk_variant ~default:{:exp|false|} ();;
12 Typehook.register ("Eq", gen_eq);;
```

Most customized type-derived code generators *e.g.* Map2, Fold2 follow this style, and it is available for both Fan and Fan's users.

RemoveLoc generator RemoveLoc is an ad-hoc code generator specialized for Fan's abstract syntax that removes location data from a datatype. It assumes that the location data has type `loc` and that it appears as the first component of the type. This generator creates the types in the upper half of Figure 3.

Listing 25. Row field transformation

```
1 {:in_exp| row_field: map_row_field (function
2   | {| $par:x of loc |} -> {| $par:x |}
3   | {| $par:x of (loc * $y ) |}
4       ->
5        begin match y with
6        |{:ctyp| $_ * $_ |} ->
7         {| $vrn:x of $par:y |}
8        | _ -> {| $vrn:x of $y |}
9        end
10  | x -> x)|}
```

The basic idea is that RemoveLoc traverses the type, using `map_row_field`, which is a wrapper for a visitor pattern object (derived by `MapWrapper` see listing 23). For each row field, it pattern matches to see whether there is only a loc field (line 2) or whether the is loc followed by more types (line 3). In the latter case, further pattern matching either

The beauty is that `row_field` is a very small fine-grained type that only allows `row_field` appears instead of the whole type universe. `$vrn` and `$par` are named antiquotations which mean polymorphic variant, parenthesis respectively(section 5). When the `row_field` only has one field named `loc`, we simply discard it as shown in line 2. Lines 6 through 7 shows when the `row_field` has more than two arguments, we parenthesize the rest of fields, and line 8 is the case that `row_field` happens to have two fields. One benefit for the extremely simplified representation of Fan's abstract syntax is that during development, it's particularly easy to get a code generator working for Fan's abstract syntax first, and then generalized to arbitrary datatype in OCaml.

MetaObj generator *MetaObj* code generator is a bit different from other type-directed code generator, as shown in listing 19 that the class that needs to be generated already uses quasiquotation, to derive the meta explosion, nested quotation and antiquotation is needed. It's fairly complicated to implement the *MetaObj* code generator since we need to parameterize the meta program, namely macros which write macros, *e.g.* the classic *once-only* macro in Common Lisp[27]. Listing 26 shows how nested antiquotation is used in Fan to parameterize the meta program.

Listing 26. Nested antiquotation

```
1  let mcom x y =
2    {:exp|{:exp| $($x), $($y) |}|};;
3  let mapp x y =
4    {:exp|{:exp| $($x)  $($y) |}|};;
```

To generate the program `{:exp| x y|}` as an expression and parameterize it by *x* and *y* at the same time, we need both nested quotation and antiquotation *i.e.* `{:exp|{:exp|$($x)$($y)|}|}` which shares a similar notation as Lisp: ``` ``(,,x ,,y) ```. The same applies to the definition of *mcom*. Both *mapp* and *mcom* are used in the implementation of *MetaObj* code generator. The native support of nested quotation and antiquotation makes Fan's macro system almost as expressive as Lisp's quasiquotation. Based on such utility functions, *MetaObj* code generator piggybacks on the same function *gen_stru* to generate the class *meta* for each datatype.

7. Discussion

IDE support and transparency Unlike most metaprogramming systems, compile-time runtime in Fan is totally *static* and *explicitly* defined when the user uses Fan, combining with the fact that DDSLs in Fan is always locally delimited, Fan already knows how to expand the quoted DDSL into Fan's abstract syntax without compiling the program. Besides, Fan has an unparsing engine which could pretty print Fan's abstract syntax into compilable program. This means Fan has a nice API which be made use of by IDE to expand or collapse the program. This is unlike slime [2], where the local expansion requires user to load the macros dynamically. One motivation to make all syntax extensions as DDSLs is to have a potential support for the IDE. The static property of Fan also helps get rid of compilation dependency on Fan, with minimum scripting work on build system, the dependency can be removed. Another benefit immediately inherited is that this makes debugging generated program with low level tools, *e.g.* GDB more smoothly.

Error messages with polymorphic variant When we adopt the polymorphic variant, the major worries are about the potential horrible error message emitted by the type checker. Luckily, this does not turn into a big problem in Fan, since most programs around the abstract syntax are written using the quasiquotation DDSL instead of written by hand, besides, for each quasiquotation DDSL, for example, *exp* DDSL, we provide an accompanying DDSL *exp'* which adds the type annotation automatically. An interesting discovery is that we find such type annotation have an obvious impact on the compilation performance, for example when compiling a module named `AstTypGen`, switching from α quasiquotation DDSL into α' changes the compilation time from 3s to 1s.

Performance In Fan, a lot of efforts are put into optimize its performance. One major direction is based on the result discovered above, we generate full type annotations or partial type annotations when information is not enough. Such type annotation is still optional, though. Fan does not rely on dynamic loading, static linking all DDSLs into native program is available in most platforms, which could provide better performance. Besides, most of the program transformations is locally delimited by DDSL, so if the user

does not use the DDSL, the user does not need to pay for registering different DDSLs. On a machine with *2.7 GHz Intel Core i5,, Memory 4 GB 1333 MHz DDR3, uni-processor*, the time to build a native version of the vanilla Fan(Fan's source tree after compile time expansion) takes 28s, while the time to bootstrap Fan using the vanilla Fan is 28s as well, which means preprocessing using Fan or not does not incur perceptible compilation performance downgrading.

Hygienic issues Hygienic is a lovely feature for macros [15], it could help avoid unintentionally name capture. However, this presents two challenges we are unclear how to solve so far: first, introducing hygienic macros would impose the dependency on both the compiler and the runtime where the dependency on runtime is not actually used but would further cause a dependency chain on other libraries, if such dependency is figured out by the compiler itself, such smartness would break the transparency to the user, the actual compile-time runtime is inferred instead of not explicitly specified which may not be a theoretical problem but exists in practice; second, it presents an engineering challenge to dump a *human editable* and compilable program to get rid of compilation dependency on Fan. For example, the program dumped by *-ddump-splices* flag in GHC is not guaranteed to compile, it is barely readable, not alone human editable. From a practical point of view, the fundamental DDSLs, *i.e.* quasiquotation DDSL, get rid of name capture problem by introducing polymorphic variants. To avoid variables polluting the outside environment, OCaml has a rich module system to qualify the generated code and in practice, OCaml's static type system could detect most unintentionally shadowing problems. Capturing the outside environment is mostly recognized as a feature instead of a bug, it provides a chance for type-specific optimizations of type-directed code generation, for example, the Opa compiler [4] intentionally shadows the module `Stream` to make use of Camlp4's stream parser's syntax extension, and our *lex* and *parser* DDSL implicitly brings variable *lexbuf* and *_loc* into scope. With the nice IDE support mentioned above, the name capture problem is largely mitigated in Fan. However, for a runtime metaprogramming system, there is only a run-time runtime and the code can not be inspected actually, hygienic macros is necessary in such case.

Runtime reflection Fan could already silently inject a toplevel phrase which records the abstract syntax of the compilation unit. It's still a type-safe way since the injected phrase is passed to type checker. The advantage compared with Camlp4 or Template Haskell is such injection does not impose any linking dependency on Fan due to structural typing— even the type annotations for Fan's abstract syntax will not impose any linking dependency since `Ast.cmi` does not contain any code. However, there is one challenge to be solved in the future, since OCaml's compilation unit is composed of two components: the implementation and signature, without reading the signature file the runtime reflection mechanism would break the boundary of abstraction, while reading the signature file per compilation may downgrade the compile performance as well.

8. Related Work

Metaprogramming has a long history since Lisp was invented [37], there are a number of meta programming systems in dynamic programming languages such as Stratego [41], Metaborg [7], OMeta [43] and Mython [30]. Most of those practical metaprogramming systems work on interpreted programming languages, this is no surprise since the interpreter usually exposes the whole compiler tool chain in the REPL. In Racket [16], language extensions are provided as libraries which is the same as Fan [39], though Racket is designed with metaprogramming in mind while Fan is built atop OCaml.

Run-time metaprogramming Most runtime metaprogramming languages are multiple staged, such as MetaML [38], MetaOCaml [9], Mint [45] and LMS in Scala [33]. Runtime metaprogramming has a more discipled and typeful representation for abstract syntax [10]. These multi-staged languages exhibit even stronger properties of type safety: type checked code generator can not generate ill typed code. However, some use cases can not be covered by such multi-staged languages, for example, in MetaOCaml, only expressions can be generated which excludes some interesting use cases in Fan. Another difference between run-time metaprogramming and compile-time metaprogramming is that for run-time metaprogramming, the staged code cannot be inspected, which means the domain specific optimizations must be performed before code generation.

Compile-time metaprogramming C++'s template meta programming is one of the most widely used compile time metaprogramming systems in industry. The compile-time metaprogramming happens in the type level of C++, which presents a huge concept gap between the daily programming and template metaprogramming, though C++'s type system is turing complete [40]. Some use cases of metaprogramming, *e.g.* compile time specialization, are less awkward when some new features are added *i.e.* generalized constant expression [14]. Fan is directly inspired from Camlp4 [13], Template Haskell [34].

Template Haskell is designed with a more ambitious goal: a hygienic compile time metaprogramming system with the ability to type check the quoted code. However, Template Haskell relies on implicitly dynamically loading at the compile-time, which results in a significantly slow compilation performance and the dependency on GHCi. More importantly, the implicit dynamically loading makes it hard to work well with IDE. It's not too hard to trigger loading tons of packages at the compile-time even for superficial usage of Template Haskell which may causes some security issues. We don't see an easy way to get rid of the compilation dependency on Template Haskell and the abstract syntax of Template Haskell does not keep the precise location either. The type checking for the quoted expression is helpful to catch bugs but it is not too hard to break it when the user construct the abstract syntax explicitly. The idea of evolving the metaprogramming system by bootstrapping itself is inherited from Camlp4, however, Camlp4 does not make use of this feature actively due to its fragile bootstrapping model and particular slow compilation. Fan takes this step aggressively to evolve itself. Another difference between Camlp4 and Fan is that Camlp4's macro system relies on an imperative style syntax extension heavily, which is not composable. Loading one syntax extension in Camlp4 have a side effect and the order matters, which means it's dangerous to statically linking all the syntax extensions, therefore Camlp4 still partly relies on dynamic loading. To our best knowledge, Fan is the first compile-time metaprogramming system hosted in a strongly typed programming language which has adopted structural typing and native support for nested quotations and antiquotations.

References

[1] Opam: OCaml package manager. `http://opam.ocamlpro.com/`, 2013. [Online; accessed 19-Mar-2013].

[2] Slime: Emacs mode for Common Lisp development. `http://common-lisp.net/project/slime/`, 2013. [Online; accessed 19-Mar-2013].

[3] Stackoverflow: Which GHC extensions should users use/avoid. `http://stackoverflow.com/questions/10845179/`, 2013. [Online; accessed 19-Mar-2013].

[4] The Opa Framework for Javascript. http://opalang.org/, 2013. [Online; accessed 19-Mar-2013].

[5] Working Group: the future of syntax extensions in OCaml. http://lists.ocaml.org/listinfo/wg-camlp4, 2013. [Online; accessed 19-Mar-2013].

[6] Michael D Adams and Thomas M DuBuisson. Template your boilerplate: using template haskell for efficient generic programming. In *Proceedings of the 2012 symposium on Haskell symposium*, pages 13–24. ACM, 2012.

[7] Aivar Annamaa. MetaBorg: Domain-specific Language Embedding and Assimilation. 2009.

[8] A. Bawden et al. Quasiquotation in LISP. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Technical report BRICS-NS-99-1, University of Aarhus*, pages 4–12. Citeseer, 1999.

[9] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[10] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. *ACM SIGPLAN Notices*, 38(9):275–286, 2003.

[11] Pascal Costanza. A short overview of AspectL. In *European Interactive Workshop on Aspects in Software (EIWAS04), Berlin, Germany*, page 8. Citeseer, 2004.

[12] Daniel de Rauglaudre. Camlp4 version 1.07. 2. *Camlp4 distribution*, 1998.

[13] Daniel de Rauglaudre and N Pouillard. Camlp4. *URL: http://caml. inria. fr/camlp4*, 2002.

[14] Gabriel Dos Reis and Bjarne Stroustrup. General constant expressions for system programming languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2131–2136. ACM, 2010.

[15] R Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and symbolic computation*, 5(4):295–326, 1993.

[16] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.

[17] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, volume 230. Baltimore, 1998.

[18] Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[19] Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[20] R. Lämmel and S.P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *ACM SIGPLAN Notices*, volume 38, pages 26–37. ACM, 2003.

[21] R. Lämmel and S.P. Jones. Scrap your boilerplate with class: extensible generic functions. In *ACM SIGPLAN Notices*, volume 40, pages 204–215. ACM, 2005.

[22] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.

[23] Matt Morrow. Translation form haskell-src-exts abstract syntax to Template Haskell. http://hackage.haskell.org/package/haskell-src-meta, 2013. [Online; accessed 19-Mar-2013].

[24] Markus Mottl. Binary protocol code generator. http://forge.ocamlcore.org/projects/bin-prot, 2013. [Online; accessed 19-Mar-2013].

[25] Markus Mottl. S-expression code generator for OCaml. http://forge.ocamlcore.org/projects/sexplib/, 2013. [Online; accessed 19-Mar-2013].

[26] Markus Mottl. Type-conv library for OCaml. http://forge.ocamlcore.org/projects/type-conv/, 2013. [Online; accessed 19-Mar-2013].

[27] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann, 1992.

[28] Didier Rémy. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88. ACM, 1989.

[29] Didier Rémy and Jérôme Vouillon. Objective ML: An Effective Object-Oriented Extension to ML. *TAPOS*, 4(1):27–50, 1998.

[30] Jonathan Riehl. Language embedding and optimization in mython. *SIGPLAN Not.*, 44(12):39–48, October 2009.

[31] JA Robinson. Beyond LogLisp: combining functional and relational programming in a reduction setting. In *Machine intelligence 11*, pages 57–68. Oxford University Press, Inc., 1988.

[32] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C d S Oliveira. Comparing libraries for generic programming in Haskell. *ACM Sigplan Notices*, 44(2):111–122, 2009.

[33] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *ACM SIGPLAN Notices*, volume 46, pages 127–136. ACM, 2010.

[34] T. Sheard and S.P. Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[35] Michael Snoyman. Hackage dependency monitor. http://packdeps.haskellers.com/, 2013. [Online; accessed 19-Mar-2013].

[36] Richard M Stallman and Zachary Weinberg. The C preprocessor. *Free Software Foundation*, 1987.

[37] Guy L Steele Jr and Richard P Gabriel. The evolution of Lisp. In *acm sigplan Notices*, volume 28, pages 231–270. ACM, 1993.

[38] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1):211–242, 2000.

[39] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[40] Todd L Veldhuizen. C++ templates are turing complete. *Available at citeseer. ist. psu. edu/581150. html*, 2003.

[41] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *Rewriting techniques and applications*, pages 357–361. Springer, 2001.

[42] Eelco Visser. Meta-programming with concrete object syntax. In *Generative programming and component engineering*, pages 299–315. Springer, 2002.

[43] Alessandro Warth and Ian Piumarta. OMeta: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19. ACM, 2007.

[44] Stephanie Weirich. RepLib: a library for derivable type classes. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 1–12, New York, NY, USA, 2006. ACM.

[45] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *ACM Sigplan Notices*, volume 45, pages 400–411. ACM, 2010.

[46] Jan Wielemaker. SWI-Prolog 5.3 reference manual. 2004.