Fan: Meta-Domain-Specific Languages for OCaml

Abstract

A domain-specific language (DSL) is a mechanism that allows problem solutions to be expressed idiomatically. However, the cost of designing, implementing and maintaining DSLs is often high, and there are integration issues when composing multiple DSLs.

To address these challenges, this paper presents Fan, a suite of DSLs tailored to the domain of building domain-specific languages. Fan's DSLs are *delimited*, which permits integration and nesting. An extension of OCaml, Fan and its DSLs benefit from OCaml's type system and toolchain.

Fan comprises over 100 (mostly tiny) delimited DSLs for lexing, dynamic parsing, quasiquotation, meta-level type-directed programming. Fan's quasiquotation can be overloaded via structural subtyping, and its parser generator allows grammar analysis, reflection and abstraction over nonterminals and terminals.

Fan is bootstrapped, and we evaluate its design based on the bootstrapping experience and the implementation of several other example DSLs.

1. Introduction

Domain-specific languages (DSLs) provide a mechanism that allows problem solutions to be expressed idiomatically. There are a wide variety of DSLs, for example, HTML for the web, `regex` expressions for string processing, or `lex` and `yacc` for parsing. Even in specialized domains, for example compiler backends, DSLs have found use. The TableGen description language has been developed for the LLVM [1] toolchain, and according to their website [2]:

> The most important is that it makes it easier to port LLVM because it reduces the amount of C++ code that has to be written, and the surface area of the code generator that needs to be understood before someone can get something working. Second, it makes it easier to change things. In particular, if tables and other things are all emitted by tblgen, we only need a change in one place (tblgen) to update all of the targets to a new interface.

It is clear that DSLs can be a big productivity win. However, the cost of designing, implementing and maintaining them is often high, and there are issues when composing multiple DSLs.

To address these challenges, this paper presents Fan, a suite of DSLs tailored to the domain-specific problem of building domain-specific languages. Fan's DSLs are *delimited*—their scope is restricted—which permits better tooling, integration, and DSL interaction. Fan is implemented via syntactic metaprogramming techniques on top of OCaml, which means that both Fan and its DSLs benefit from OCaml's rich type system, in contrast with many prior approaches to syntactic metaprogramming.

Embracing the DSL model fully, Fan provides over 100 delimited DSLs (mostly tiny, and most mechanically generated) for implementing first-class lexers and dynamically parameterized parsers, quasiquotation of every syntactic category of OCaml's grammar, and utility-DSLs that support generation of DSLs by (meta-level) type-directed programming.

Fan's quasiquotation DSLs can be overloaded via structural subtyping to increase flexibility. By targeting a particular, general token type, Fan's top-down parser generator distributes its work between compile- and run-time, allowing grammar analysis, high-quality error message, and reflection in OCaml's toplevel. The parser grammars can also be parameterized by nonterminals or terminals.

Fan is bootstrapped—it is implemented using its own DSLs. We evaluate the effectiveness of Fan's design based on this bootstrapping experience and a number of other example DSL implementations. In summary this paper makes the following contributions:

- 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 [38] is exported as an object [26] that can be customized by the programmer.
- A suite of front-end DDSLs, which include first class lexers and a programmable and parametric parser.

2. Overview of Fan

2.1 Delimited domain-specific languages

The main syntactic abstraction mechanism provided by Fan is the notion of a *delimited, domain-specific language* (DDSL), indicated by the concrete syntax `%lang{...text...}`. This notation tells Fan to interpret the string "`...text...`" via a semantic function associated with the `lang` DDSL.

For example, Listing 1 shows the use of a Prolog compiler DDSL called `plc` to solve the n-queens problem. The `plc` 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 the input `+N` as a wrapped OCaml value. Behind the scenes, the `plc` semantic function uses syntactic metaprogramming to translate each Prolog predicate to an OCaml function, following Robinson's techniques (which support backtracking and unification) [28]. This translation takes place *statically*, and the resulting code is passed to the OCaml back end for typechecking and compilation. The resulting implementation yields performance that is significantly better than a Prolog interpreter, and that compares favorably with the SWI-Prolog compiler, a mainstream Prolog implementation [41] (at least for this subset of the language).

As illustrated by the example above, unlike a library or typical "embedded" domain-specific language, Fan's DDSLs are not restricted to the host language's syntactic conventions. Moreover, since the DDSLs' semantics are given via syntactic translation into OCaml, Fan's DDSLs can perform static analysis of the embedded code, or restrict its syntax, thereby enabling optimizations, something that is not easily available for "language as a library" approaches to DSLs.

```
%plc{

%:nqueens(+N,?Board)
nqueens(N,Board) :-
  range(1,N,L),  permutation(L,Board),
  safe(Board).

%:range(+Start,+Stop,?Result)
range(M,N,[M|L]) :-
    M < N, M1 is M+1, range(M1,N,L).
range(N,N,[N]).

%:permutation(+List,?Result)
permutation([],[]).
permutation([A|M],N) :-
  permutation(M,N1), insert(A,N1,N).

%:insert(+Element,+List,?Result)
%:insert(?Element,+List,+Result)
%:insert(+Element,?List,+Result)
insert(A,L,[A|L]).
insert(A,[B|L],[B|L1]) :- insert(A,L,L1).

% :safe(+Board)
safe([_]).
safe([Q|Qs]) :- nodiag(Q,Qs,1), safe(Qs).

%:nodiag(+Queen,+Board,+Dist)
nodiag(_,[],_).
nodiag(Q1,[Q2|Qs],D) :-
  noattack(Q1,Q2,D),  D1 is D+1,
  nodiag(Q1,Qs,D1).

%:noattack(+Queen1,+Queen2,+Dist)
noattack(Q1,Q2,D) :-
   Q2-Q1 =\= D,  Q1-Q2 =\= D.

} ;;
let _ = nqueens
  (fun b -> print_endline (string_of_plval b))
  (Int 10);;
```

Listing 1: Deep embedding of Prolog code into OCaml via Fan.

Importantly, Fan's supports *delimited* language extensions. The scope of a DDSL is restricted to the code between the { and } braces. By further imposing a few mild lexical restrictions on the syntax of DDSLs, Fan ensures that DDSLs are *nestable* and *composable*—multiple DDSLs can be used simultaneously.

Besides wholesale embedding of a foreign language (like in the Prolog example), DDSLs can also be used for very fine-grained syntactic support, much like Lisp macros, or for "medium-scale" languages such as lexer or parser generators that incorporate parts of the host-language grammar (e.g. for production actions) with new syntax for matching.

2.2 Fan's workflow

Figure 1 shows the high-level view of the Fan workflow. The programmer registers DDSLs with the Fan front end, which parses input files and translates them into abstract syntax trees that can be desugared into OCaml's intermediate `Parsetree` representation for processing by OCaml's back end.

The heart of Fan is thus a function that takes a piece of quoted text `%lang{...text...}` along with a tag that indicates what syntactic category (of the host-language) the quote should be expanded to fill, and returns an abstract syntax tree of that type:

```
Ast_quotation.expand : quot -> 'a Dyn_tag.t -> 'a
```

Here the `'a` type parameter is determined by the parser—for example it should be `FAst.exp` if the quote appears in an expression position of the host grammar. Internally, the quotation expander dispatches to the appropriate semantic function of the registered DDSL (or produces an error if there is none). Quotations are expanded recursively from top to bottom for nested DDSLs, and the resulting abstract tree is spliced into the parse results. This process is called *compile-time expansion*.

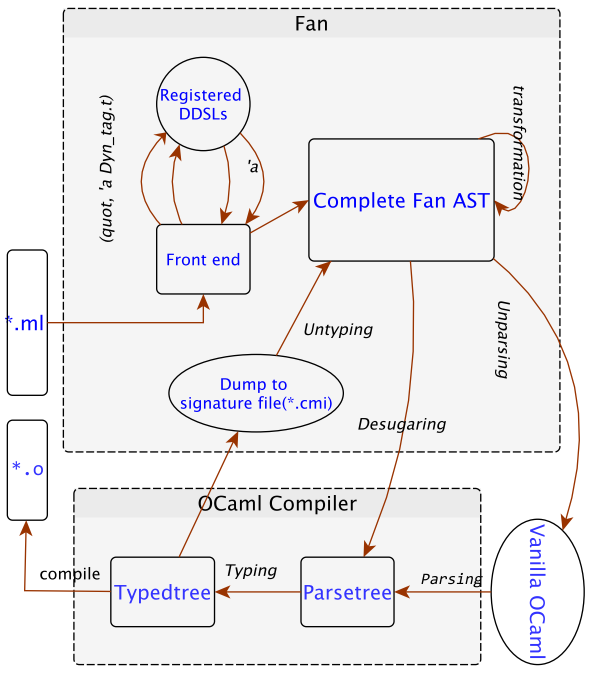


Figure 1. Fan workflow: registered DDSLs expand concrete text into Fan's AST, which is fed to OCaml's standard compilation path.

2.3 Creating DDSLs

Fan fully embraces the DDSL model by providing a suite of domain specific languages tailored to the domain of implementing DDSLs. Just as with any language implementation, the semantic function for a DDSL can usually be factored into lexing, parsing, and compilation steps:

```
val lex : loc -> char Stream.t -> Token.stream
val parse : Token.stream -> OAst
val compile : OAst -> FAst.?
```

Here `OAst` is some datatype that represents the parse trees of the object DDSL. The `compile` function translates object-language constructs into OCaml code represented by Fan's abstract syntax trees, but the precise type depends on the parsing context (and is determined by the `'a Dyn_tag.t` given to the quotation expander).

Fan thus includes DDSLs that help with all of these tasks, including quasiquotation, and various other syntactic metaprogramming constructs. Listing 2 shows several of these DDSLs in action for a simple example meant to illustrate Fan's novel features. The code shown creates a DDSL named `arith`, which supports quasiquotation of arithmetic expressions. For example, the expression `%arith{let x be 3 in 4 * x}` produces the value:

```
`Let ("x", `Int 3, (`Times (`Int 4) (`Var "x"))
```

The first several lines of the listing define the OCaml data type produced by `arith` DDSL expressions. The lexer, defined using the `lex_fan` DDSL in lines 11–18, recognizes several kinds of tokens, treating some strings (like `let` and `**`) as keywords, but otherwise using OCaml defaults for numeric literals and identifiers.

```
    type arith =  (* Abstract syntax for the arith DDSL *)
       [ `Add    of (arith * arith)
       | `Minus  of (arith * arith)
       | `Times  of (arith * arith)
 5     | `Divide of (arith * arith)
       | `Power  of (arith * arith)
       | `Let    of (string * arith * arith)
       | `Int    of int
       | `Var    of string]
10
    let rec token = %lex_fan{
     | @whitespace %{token lexbuf}
     | @ocaml_num_literal
     | @ocaml_lid("let"|"be"|"in")
15   | @ocaml_ant
     | @kwd_symbol("+"|"-"|"*"|"/"|"**"|"("|")")
     | @ocaml_eof
     | @default};;

20  let lexer = Lexing_util.adapt_to_stream token;;

    %create{ arith };;

    let add_binop left nt action prec op = %extend{
25      nt: $prec $bool:left
          [ nt; $key:op; nt   ${action}]}

    let mk_ant nt prec = %extend{
        nt: $prec
30        [ Ant("",x)   %{Tokenf.ant_expand Parsef.ep x}]}

    let mk_action tag e2 _ e1 _loc =
      %ep{($vrn:tag ($e1,$e2) : arith)}

35  let tbl = [("+",  ("Add",    10, true));
               ("-",  ("Minus",  10, true));
               ("*",  ("Times",  20, true));
               ("/",  ("Divide", 20, true));
               ("**", ("Power",  30, false))] ;;
40
    List.iter
     (fun (op,(tag,prec,left))
       -> add_binop left arith (mk_action tag) prec op) tbl ;;

45  %extend{
      arith: 40 true
        [ "("; arith as e; ")" %{e}
        | Lid s                %ep{`Var $str:s}
        | Int i                %ep{`Int $int:i} ]} ;;
50
    mk_ant arith 40 ;;

    %extend{
      arith: 0 false
55      [ "let"; Lid x; "be"; arith as e1; "in"; arith as e2
            %ep{`Let($str:x,$e1,$e2)} ]} ;;

    Ast_quotation.of_ep ~lexer ~name:(Ns.lang,"arith")
      ~entry:arith ()
```

Listing 2: A tiny DDSL for quasiquoting arithmetic expressions.

The parser for `arith` expressions is created on line 22 using Fan's `create` DDSL, and the productions associated with its grammar are incrementally added to that nonterminal by the following commands. As usual, a production is a sequence of symbols—either terminals or nonterminals—separated by `;` and followed by an action, which is either an inlined quoted piece of code or parameterized via antiquotation . (The parser DDSL is explained more thoroughly in Section 5.3.)

The basic binary operators are added to the grammar by iterating through a table that specifies the token, variant name, and associativity, adding each entry via the `add_binop` function. Note that `add_binop` is parameterized by the grammar nonterminal and an action. It uses Fan's `extend` DDSL to add a new production that recognizes an infix binary operator. Antiquotation, as in `$key:op`, escapes out of the `extend` DDSL, splicing in the `op` argument (interpreted as a keyword) from the surrounding context.

Lines 45–50 also use the `extend` DDSL to add productions for identifiers, constants, and parenthesized subexpressions to the grammar. On line 46, the constant 40 is the precedence level and the **`true`** indicates that these should be left associative (these features were parameters of the use of `extend` in `add_binop`).

Line 51 calls a utility function `mk_ant`, which adds antiquotation capabilities to any nonterminal (at a given precedence level). Such a function would be part of Fan's library, since it can apply generically to any grammar, but note how short its definition is—its action is simply to call an antiquotation expander, which we explain in more detail later. Here we use it to add antiquotation support to the `arith` DDSL.

Support for "let" notation is added in lines 53–56, and the last two lines register the `arith` DDSL with Fan, specifying the lexer, its name, and an entry point for its parser.

The "compile" step for this simple example is straightforward, and for simplicity we inline it with the parser actions. Each action produces (abstract syntax for) OCaml code that represents the corresponding value of type `arith`. These actions demonstrate the use of the built-in quasiquotation DDSLs for working with Fan's abstract syntax types. For example, the `ep` DDSL found on lines 48 and 49 (and elsewhere) quotes OCaml code that is valid as both an expression and a pattern. Thus, `%ep{`Var "x"}` generates the OCaml abstract syntax that, when compiled, will produce the OCaml value `` `Var "x"``. This "meta-programming with concrete object syntax" [38] provides a convenient way to work with syntactic objects using familiar notation. Fan provides abstract syntax representations for all of OCaml's syntactic categories, but the most commonly used include `FAst.exp` (expressions), `FAst.pat` (patterns), `FAst.stru` (toplevel structures), and `FAst.ep` (expressions that are patterns). The design of these abstract syntax types plays a crucial role in Fan, and we will explain it in detail next.

3. Design of abstract syntax

Compile time 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 [33], 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 [35], nominally typed [12, 31], structurally typed or just untyped data structures [9].

3.1 Various representations for abstract syntax

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 [31] and Camlp4 [12] 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 metaprogramming, nominal typing presents several problems.

There are three drawbacks for representing abstract syntax using algebraic datatypes:

1. Standard implementations of Damas-Hindley-Milner-style type-inference (as currently used in Haskell and OCaml) discourages sharing of constructor names among distinct algebraic data types. This problem is exacerbated further when we add support for antiquotation to the abstract syntax. Doing so (see the discussion in Section 4.1) 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.
2. The redundancy described above caused by nominal typing *discourages* the designer from refining a coarse-grained syntactic 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.
3. The name and constructors of the type should be defined or imported prior to their use. 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.

3.2 Structural typing, subtyping and polymorphism

Based on the observations above, Fan adopts *polymorphic variants* [14, 25] for representing its abstract syntax.

Polymorphic variants permit deeply nested pattern matching. 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 3 shows how sharable constructors (in this case for literals) and union types enables sharing of large swaths of boilerplate code.

For example, the loc_of function extracts location data uniformly for all syntax, which is the supertype containing all of

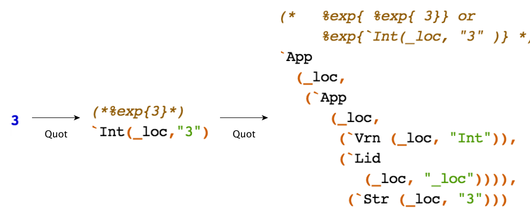
```
type literal =
  [ `Chr of (loc * string)
  | `Int of (loc * string)
  | ...]
type exp =
  [ literal
  | ... (* more *) ]
type pat =
  [ literal
  | ... (* more *)]
...
type syntax =
  [ exp | pat | ... ]
```

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

let (<+>) a b = Loc.merge
    (loc_of a) (loc_of b)

let com a b =
  let _loc = a <+> b in
    `Com(_loc,a,b)
```

Listing 3: Shared type constructors and boilerplate code.



```
3

(*%exp{3}*)
`Int(_loc,"3")

(*   %exp{ %exp{ 3}} or
     %exp{`Int(_loc, "3" )} *)
`App
  (_loc,
   (`App
      (_loc,
       (`Vrn (_loc, "Int")),
       (`Lid
          (_loc, "_loc")))),
     (`Str (_loc, "3")))
```

Figure 2. Nested quotation.

Fan's syntactic categories. (In practice loc_of is automatically generated using Fan's *deriving* DDSL; see section **??**.) Similarly, structural subtyping means that there can be a uniform API for metaprogramming that works generically for all syntactic constructs, as shown by the com operation that adds a comma between two arbitrary pieces of syntax.

Structural subtyping also creates the possibility of selectively reusing parts of the host language in a DDSL without losing type safety. In Fan, when introducing a new DDSL, the author can work with just a subset of a specific syntactic category. For example, in the quasiquotation DDSL (explained below) uses the smaller type ep, which is a subtype of both expressions and patterns.

In light of the considerations above, the Fan abstract syntax representation introduces a total of (roughly) 170 distinct constructors collected into 53 syntactic categories, permitting 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 sheer 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.

4. Backend DDSLs

Quasiquotation 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 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 ele-

$$\begin{tikzcd}
 & syntax & & rec\_exp & & rec\_pat \\
exp \arrow[ur] & & pat \arrow[ul] & & rec\_bind \arrow[ul] \arrow[ur] & \\
 & ep \arrow[ul] \arrow[ur] & & & &
\end{tikzcd}$$

Figure 3: Subtype relation of explosion syntax.

ments out of the quotation. Quotation combined with antiquotation, *i.e. quasiquotation*, provides a very intuitive way to abstract over code. For example, the right side of Listing 4 shows how antiquotation is used in (a slightly artificial, but small) program transformation that swaps the order of arguments in a top-level + abstract syntax node. Fan's antiquotation allows the pattern variables from the OCaml phrase to be spliced into the quoted syntax. The left side shows the equivalent function written without quasiquotation—the difference in readability is clear.

```
let swap =  function           let swap = function
  | `App (_loc, `App (_,          | %exp{$x+$y} ->
     `Lid (_,"+"), x),y)  ->        %exp{$y+$x}
    `App (_loc, `App (_loc,       | x -> x
     `Lid (_loc, "+"), y), x)
  | x -> x
```

Listing 4: Antiquotation

4.1 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 [38], and the same basic principle underlies other compile-time metaprogramming systems [11, 18, 31, 37]. In Fan, the meta-explosion operations are encapsulated in objects [26], which means that type- and data-constructor-specific behavior can be overridden by the user.

Listing 5 shows the (hard-coded) meta-explosion object for the primitive types—it provides one method per type. For example, the call `primitive#int _loc 3` yields the value `` `Int(_loc, "3") ``.

```
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
```

Listing 5: Meta-exploding primitive types.

The result of a method like `primitive#int` could be given a coarse-grained type like `syntax` (since the resulting AST represents 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 3. Listing 6 shows the type itself, which has only 18 distinct constructors (including 7 for `literal` values). This is a ten-fold reduction in the number of constructors as compared to the full `syntax` type.

```
type ep =
  [ vid
  | `App     of (loc * ep * ep)   (* application *)
  | `Vrn     of (loc * string)    (* variant tag *)
  | `Com     of (loc * ep * ep)   (* pair ',' *)
  | `Par     of (loc * ep)        (* parentheses *)
  | `Sem     of (loc * ep * ep)   (* semicolon *)
  | `Array   of (loc * ep)        (* arrays *)
  | `Record  of (loc * rec_bind)  (* records *)
  | `Any     of loc               (* _ *)
  | literal                       (* 7 literals *)
  | ant ]                         (* antiquotation *)
and rec_bind =
  [ `RecBind of (loc * vid * ep)
  | `Sem     of (loc * rec_bind * rec_bind)
  | `Any     of loc
  | ant ]
```

Listing 6: Precise co-domain of meta explosion

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` type, as shown in Listing 7.

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

Listing 7: Meta explosion for the whole syntax

There are a few additional observations to make about this design. Meta-explosion has type `syntax -> ep`. But, since `ep` is a subtype of `syntax`, meta-explosion can be composed with itself, which makes it easy to support nested quotations: `%exp{%exp{%exp{...}}}`.

The precise co-domain not only makes program transformation of 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.

The `ep` quotation DDSL is itself defined in terms of `meta`, so to bootstrap Fan we must first implement meta-explosion for the constructs in Listing 6 by hand—having to do so for only the 18 constructors of `ep` is a big win. Once bootstrapped, the implementation of the `meta` object shown above is so mechanical that, in practice, it is implemented automatically by a DDSL called `derive`, which uses type-directed programming to construct the body of each case of `meta` from the corresponding `syntax` type.

Systematic antiquotation Support One further wrinkle in the types for abstract syntax is the need to include the `ant` type, which contains a single variant `` `Ant of (loc * Token.ant) `` 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 8.

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—the `arith` example from Listing 2 uses such antiquotation filtering to turn a string into a

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

Listing 8: Antiquotation support

variant tag via `$vrn:tag`. Fan supports this by providing hooks for users to define their own filtering during antiquotation. Listing 9 shows the `filter` object, which overrides a generic syntax visitor to dispatch via named antiquotation filter names (*e.g.* `vrn`) to invoke custom rewriters.

```
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
      |("vrn",_,e)  ->
       %pat{`Vrn ($(mloc _loc), $e)}
    |("lid",_,e) -> ...
   end
  (* override other methods*)
end
```

Listing 9: Named Antiquotation support

In this case, the filtering `%exp{$vrn:a}` would inject the `` `Vrn `` constructor when expanding the quotation, yielding: `` `Vrn (_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 4 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. Essentially, the quasiquotation DDSL for each syntactic category `c` in Fan is the composition `(filter #c)o (meta#c)o (parse_c)`. Internally, Fan's implementation uses various DDSLs to automate the process of meta explosion and transformation between different syntaxes.

5. Frontend DDSLs

5.1 Lexing DDSLs

5.2 Lexing DDSLs

Fan provides a general-purpose `lex` DDSL that behaves similarly to the stand-alone tool `ocamllex`. One significant difference is that, being a DDSL, `lex` expressions can be seamlessly integrated with surrounding OCaml code.

However most DDSLs use essentially similar token types with limited customization, so Fan also provides a `lex_fan` DDSL that is specialized to a particular token representation that is shown in Listing 10

This specialization allows the `lex_fan` DDSL to provide named lexical cases for frequently used tokens. For example, the `arith` lexer, defined in lines 11–18 of Listing 2 uses `@whitespace` and `@ocaml_num_literal` cases directly, while line 14 indicates

```
type txt = {                  type t =
   loc : loc ;                  [ `Key  of txt  (* keyword*)
   txt : string}                | `Inf  of op   (* infix op *)
type op = {                     | `Lid  of txt  (* identifier *)
  (* ... fields of txt *)         ....
  level : int ;}                | `Quot of quot (* quotation *)
type ant = {                    | `Ant  of ant  (* antiquot. *)]
  (* ... fields of txt *)
  kind : string;}
type quot = {
  (* ... fields of txt *)
  name : name ;
  (* ...   *)}
```

Listing 10: Definition of Token.t

that all lower-case identifiers follow OCaml conventions (and will be lexed as `` `Lid `` values) except for the given string literals that will be treated as keywords (using `` `Key `` instead). Since `lex_fan` knows the possible string literals at compile time, such promotion uses compile-time pre-hashing which is generally faster than a hand-written lexer.

Lexical convention for Fan's DDSLs As we discussed in Section 2.1, there are a few mild restrictions imposed on Fan's DDSLs to support nesting. Within a DDSL, the `{—}` braces must balance, and the language must respect OCaml's conventions for comments strings, and literal characters. DDSLs that are implemented using `lex_fan` as the tokenizer can `@ocaml_comment`, `@ocaml_string`, `@ocaml_char` to automatically meet these requirements. To support quotation and antiquotation following Fan's conventions the `@ocaml_quotation` and `@ocaml_ant` cases are provided.

5.3 Dynamic Parser DDSL

The main design goal of Fan's parser DDSLs is to make parsing convenient, while sharing grammars and even grammar productions as much as possible. For example, `lex_fan` shares most of its grammar with `lex`, and both of them reuse the grammar for OCaml expressions. Recursively, since it is delimited to expression positions, the grammar of `lex_fan` is reused by the expression grammar as well. Such sharing is protected against conflicts at the lexing level.

Fan uses a novel, dynamic parser engine with incremental semantics; the runtime engine is a variant of Pratt Parsing [24]. Fan's `create` and `extend` DDSLs support the addition (dynamically) of new parser non-terminals and productions associated with them. Just as with `lex_fan`, these DDSLs process tokens of the type `Token.t` defined in Listing 10.

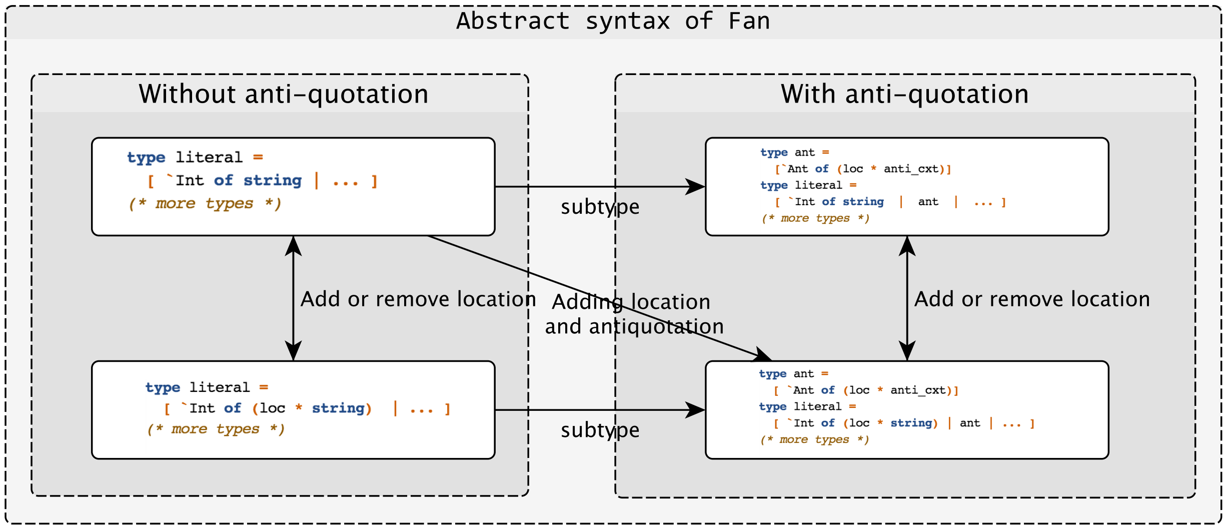
5.3.1 The `extend` DDSL

Listing 11 gives the syntax for the `extend` parser DDSL, which adds productions to nonterminals made by the `create` DDSL.[1] A Fan grammar has a name, optional precedence and associativity annotations (**`true`** for left associative) and a non-empty list of rules. Each rule is given by a (possibly empty) list of symbols and an action specified via quotation or parameterized by antiquotation. There are three kinds of symbols: terminals, which parse tokens, nonterminals, which name another grammar, and meta symbols, which support basic prefix-style regexp expression syntactic sugar (*e.g.* `L1` means "nonempty list").

5.3.2 compile time pre-processing

Effective value of Token Lines 45–59 of Listing 2 show a use of the `extend` DDSL for the `arith` language to add parenthesized

[1] In true metacircular fashion, this grammar is written in syntax that it itself accepts (modulo the actions that we omit).



```
type literal =
  [ `Int of string | ... ]
(* more types *)
```

```
type ant =
  [`Ant of (loc * anti_cxt)]
type literal =
  [ `Int of string  |  ant  |  ... ]
(* more types *)
```

```
type literal =
  [ `Int of (loc * string)  | ... ]
(* more types *)
```

```
type ant =
  [ `Ant of (loc * anti_cxt)]
type literal =
  [ `Int of (loc * string) | ant | ... ]
(* more types *)
```

Figure 4: Type algebra of Fan's abstract syntax.

```
grammar:
  [ Lid ; ":"; ?Int; ?("true"|"false");  rules ]

rules:
  [ "["; L1 rule SEP "|"; "]"]

rule:
  [ L0 symbol SEP ";"; Quot
  | L0 symbol SEP ";"; Ant]

symbol:
  [ terminal_symbol | nt_symbol | meta_symbol]

terminal_symbol:
  [ ... ]

meta_symbol:
  [ ("L1"|"L0"); symbol; "SEP"; symbol
  | ("L1"|"L0"); symbol
  | "?"; symbol]

nt_symbol:
  [ Lid ]
```

Listing 11: (Metacircular) Syntax for the `extend` DDSL.

expressions, literal integers, and lower-case identifiers to the `arith` grammar. Here `"let"` and `Lid s` are terminal symbols of the grammar that would parse the corresponding tokens `` `Key {txt = "let"} `` and `` `Lid _ ``, respectively.

The parser implementation requires two invariants:

1. Terminal symbol constructors must be resolved at compile time. This is crucial for type reconstruction and optimization of the generated OCaml code.
2. All terminal symbols must be comparable at runtime. This is necessary for ranking symbols during tree factorization.

The first invariant is enforced by the grammar for `extend`, which prohibits antiquotation of terminals. For example the following program, in which the constructor of the "terminal" `$x` is not known statically, is not syntactically correct:

```
fun x -> %extend{arith: [ $x %{()} ]}
```

To address the second invariant, we define a datatype that describes the "effective value" semantics of each token (Listing 12). Comparing `terminal_symbols` reduces to testing structural equivalence of their corresponding effective values.

```
type tag =  (* Tag  corresponding to each token *)
  [ `Key  | `Inf | `Lid (* ... *) | `Quot | `Ant ]
type word =
  | A of string
    (* Token matches a single string *)
  | Any (* Token matches any string *)
  | Kind of string (* represents kind in "$kind:x" *)
  | Level of int  (* Specific to Infix *)
type descr =  {
    tag : tag;
    word : word;
}
```

Listing 12: Effective value of token

For most tokens, the effective value is just the constructor tag and the `txt` field. However, there are special rules for quasiquotation tokens. For a `Quot` token, the effective value is only its tag, whereas for a `` `Ant `` token, the effective value captures the escaped code. Listing 13 shows several examples.

```
"a"      ~> { tag = `Key;  word = A "a"}
$key:op  ~> { tag = `Key;  word = A op }
Lid x    ~> { tag = `Lid;  word = Any  }
Lid "x"  ~> { tag = `Lid;  word = A "x"}
Lid $x   ~> { tag = `Lid;  word = A x  }
Quot x   ~> { tag = `Quot; word = Any  }
Quot "x" ~> (* illegal syntax in the extend DDSL *)
```

Listing 13: Examples of effective values for each token

Compile time transformation One challenging problem with Pratt-Parsing in a Hindley-Miler-based type system is that the actions associated with the productions could have arbitrary arities with different types. Since we want to store a list of productions, including the actions, with each nonterminal, the Fan internals use `Obj.magic` to cast actions to the universal `Obj.t` type. To insulate programmers from the use of such potentially unsafe casts, the code generated by Fan for parser actions includes typing constraints synthesized from the symbol list of the production.

Listing 14 shows how the `add_binop` function (from the `arith` example) looks after compile-time expansion. The `action` argument is annotated with a type derived from the symbol list of the production in which it is used. If `action` were used inconsistently in the grammar, the program would fail to typecheck.

```
let add_binop left nt action prec op =
  Gramf.extend_single (nt : 'nt Gramf.t )
    {label = Some prec;
     lassoc = left;
     productions =
     [{symbols =
       [Self;       (* Self refers to this nonterminal *)
        Token { tag = `Key; word = A op };
        Self];
       fn =
       Obj.magic
       (action :    (* synthesized type annotation *)
        'nt -> Token.txt -> 'nt -> loc -> 'nt)}]}
```

Listing 14: Example of compile time expansion for extend DDSL

Each nonterminal is associated with a distinct type variable and each terminal's type is derived syntactically from its tag. To improve efficiency, the tag of the token type is projected away when calculating the type of the associated action. In this example, since `action` second parameter is known to be the `` `Key `` variant of the `Token.t` type, the action can safely expect a `Token.txt` value.

Another important step in expanding grammar productions is normalizing nonterminals by replacing all recursive uses of a nonterminal by a special `Self` symbol. As shown in Listing 14, the two uses of `nt` in the body of the grammar are replaced by `Self`. This transformation is needed for the tree-factorization step of parser generation (described below).

Besides these two transformations, Fan's parser also provides several other handy features that are translated out at this stage, including pattern matching over symbols and symbol cross products that allow for matching alternatives like `"let"|"in"`.

5.3.3 Run time adjusting and interpretation

The `extend` DDSL constructs an efficient parser incrementally for the nonterminals of the grammar. One step in parser generation is to construct a "rectangular" presentation of the grammar, which groups together all of the productions of the same precedence and sorts them. As a debugging and visualization aid, Fan can display such intermediate represenations. Printing the `arith` grammar from Listing 2 in OCaml's toplevel is shown in Listing 15.

```
# Gramf.print Format.std_formatter arith;;
arith:  (* S stands for Self *)
  [  0 R
   ["let"; Lid; "be"; S; "in"; S ]
   | 10
   [ S; "-"; S | S; "+"; S  ]
   | 20
   [ S; "/"; S | S; "*"; S   ]
   | 30 R
   [ S; "**"; S ]
   | 40
   [ Ant | "("; S; ")" | Lid | Int]]
```

Listing 15: The `arith` grammar in rectangular form.

The second stage of parser construction is to generate a tree-shaped representation of the grammar, which has "decision points" as nodes and symbol-dectorated edges. For each precedence level, the productions are split into two *prefix trees*. Each production starting with `Self` is marked as "continuation" rule and its tail is merged into the continuation tree. All other productions are marked as start rules and merged with the start tree. Listing 16 shows the `start` and `cont` trees for the level 10 rules of the `arith` grammar.

Left factorization happens during the merge step, which takes a list of symbols and merges it into the tree. When the head symbol

```
# Gramf.dump Format.std_formatter arith;;
arith:
  [
   | 10
   cont:
     |-"+"---S---.
     `-"-"---S---.
   start: ]
```

Listing 16: Factorized prefix trees (level 10)

of the list matches a symbol on a tree edge, recursively merge the tail with the subtree along that edge. When the head symbol does not match any tree edge, create a new decision point if necessary and then place the head edge following these precedence rules:

1. Nonterminals have higher precedence than terminals.
2. The precedence between nonterminals is decided by insertion order.
3. For nonterminals with the same tag, a concrete symbol has a higher precedence than less specific tokens. For nonterminals with different tags the order does not affect parsing semantics, so a heuristic strategy is applied.

Listing 17 shows the tree obtained by dynamically extending the `arith` grammar with a new production via the the toplevel.

```
# %extend{arith :
            [ arith ; ("+"|"-"); Int; Int %ep{()} ]};;
# Gramf.dump Format.std_formatter arith;;
arith:
  [
   | 10
  cont:
    |-"+"-+-Int---Int---.
    |      `-S---.
    `-"-"-+-Int---Int---.
           `-S---.]
```

Listing 17: Adding rules to the arith grammar (level 10)

After compilation expansion, two rules are added. Both are continuation rules, and since `Int` takes precedence over nonterminal `arith` it will be inserted above the old rules.

Finally, to use a grammar created via the `extend` DDSL, the parsing engine interprets the `start` and `cont` trees. The core algorithm is simple, but relies on .

1. In order of lowest to highest precedence, try walking the `start` trees. Traverse each tree from left to right and top to bottom, matching edges labeled by terminals with tokens from the input stream. For edges labeled by nonterminals there are three cases:
   (a) If the nonterminal is not `Self` or if the nonterminal is `Self` but it is not the last edge of the tree, recursively invoke the parse interpretation on the remaining token stream.
   (b) If it is the `Self` nonterminal, and the grammar is *left* associative parse the token stream starting at the *next* precedence level.
   (c) If it the `Self` nonterminal, and the grammar is *right* associative, parse the token stream starting at the *same* precedence level.
2. If a leaf is reached, run the associated action and then process the `cont` trees for this grammar.
3. If there are no `start` trees raise an error.

Processing the `cont` trees follows nearly the same rules as above. During the tree walk, the parser might need to peek at at most $\max(k, 1)$ tokens for a sequence of k terminal symbols. In this algorithm, the LL(k) backtracking does not require the action to be pure, which means the `extend` DDSL can parse some context sensitive grammars.

6. Related Work

The idea of metacompilers is quite old—META II [30] is the first documented syntax-oriented compiler writing language.

For dynamic languages, the Lisp community has recognized the power of metaprogramming for decades [34]. For example, the Common Lisp Object System [15], which supports multiple dispatch and multiple inheritance, was built on top of Common Lisp as a library, and support for aspect-oriented programming [10] was similarly added without any need to patch the compiler. In Racket [13], language extensions are provided as libraries, just as in Fan [36].

There are stand-alone extensible languages such as Stratego [37], Metaborg [8], and OMeta [39]. Mython [27] is an extensible language on top of Python.

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.

Fan is directly inspired from Camlp4 [12], Template Haskell [31]. The OCaml community has embraced syntactic abstraction since 1998 [11], when Camlp4 was introduced as a syntactic preprocessor and pretty printer. The GHC community introduced Template Haskell in 2002 [31], and added generic quasiquotation support later [18]. Both have achieved great success, and the statistics from hackage [32] and opam [3] 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, 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 [16, 17] can be achieved through the use of datatype-generic programming [29], relying on compiler support for reifying type information. Other approaches, such as Weirich's RepLib [40], 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 [7].

OCaml, in contrast, lacks native support for datatype-generic programming, which not only makes metaprogramming as in Camlp4 more necessary [20–22], 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 [6].[2] For example, Template Haskell's limited quasiquotation support has led to an alternative representation of abstract syntax [19], 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.

dynamic parser in static functional languages The closest work to *extend* DDSL is dypgen [23], a dynamic GLR parser generator, dypgen achieves type safety by aggressively using type variables, our experimentation shows that the time spent in compiling the grammar which describes dypgen itself is even longer than compiling the whole Fan – 56 type parameters introduced for such a medium-sized grammar. Another difference is our extensible parser is built incrementally, which is very fast in the run-time while dypgen requires the engine to re-generate the automata which is cost in practice. Camlp4 also supports dynamic parser, the major difference is that Camlp4 lacks support of the abstraction over grammars and its factorization rule is based on syntax instead of semantics which is fragile, in terms of expressivity, it is essentially a LL(1) grammar.

7. Discussion

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 in a program should affect its compilation time. Fan's DDSLs are always locally expanded, in contrast with with other compiler plugins or AST rewriters, which means if DDSL is not needed, there is no cost.

The DDSLs mechanism provides a way for partial evaluation and opportunities for optimizations as `lex_fan` shows. This strategy is employed in `extend` DDSL as well, since all tags are resolved at compile time, and the representation for a singleton tag is an unboxed integer, which makes incrementally building the start and continuation prefix trees very fast.

There are other interesting discoveries for accelerating compilation time. For example, type inference in OCaml is expensive, but when compiling DDSLs we generate type annotations, which significantly reduces compilation time. On our machine with 2.6 GHz Intel Core i7, Memory 16 GB 1600 MHz DDR3, the time to build a native version of the vanilla Fan (Fan's source tree after compile time expansion) takes about 7s, while the time to bootstrap Fan using vanilla Fan is 7s as well, which means preprocessing using Fan or not does not incur perceptible compilation performance cost. Such a short feed back loop was crucial for the evolution of Fan itself.

Error messages with polymorphic variants When we adopted polymorphic variants, one concern was about the potentially horrible error message emitted by the type checker. Luckily, this does not turn into a big problem, since most programs use quasiquotation DDSLs instead of manipulating syntax trees by hand. Besides, Fan automatically inserts type annotations for such code.

IDE support and transparency The verbosity of meta-exploded code is further mitigated by IDE support. Unlike most metaprogramming systems, Fan's compile-time expansion is totally *static* and *explicitly*. Because of their delimited scope, Fan already knows how to expand the quoted DDSL without compiling the program. Fan's unparsing engine can pretty print Fan's abstract syntax into compilable OCaml source. This yields an API that can be used by IDEs to expand or collapse the program DDSL quotations. This is unlike slime [4], where local expansion requires the user to load macros dynamically. Our prototype integration with emacs has shown this feature to be extremely useful as a debugging aid.

Bootstrapping and evolution The heart 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

[2] Don Stewart has called Template Haskell "Ugly (but necessary)" [5].

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 more than 100 DDSLs that provide a whole technology stack for embedding new DDSLs. The total development time was only 18-months of work. After each bootstrapping cycle, all the features available in the current version go into the next. Most of the mechanical work of building DDSLs, pretty printers, visitor patterns, and meta-explosion for object AST, is derived by other utility DDSLs.

References

[1] LLVM. http://llvm.org, 2013. [Online; accessed 15-Nov-2013].

[2] LLVM Code generator. http://llvm.org/docs/CodeGenerator.html, 2013. [Online; accessed 15-Nov-2013].

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

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

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

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

[7] 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.

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

[9] 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.

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

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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].

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

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

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

[23] Emmanuel Onzon. dypgen User's Manual. http://dypgen.free.fr, 2008. [Online; accessed 15-Nov-2013].

[24] Vaughan R Pratt. Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51. ACM, 1973.

[25] 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.

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

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

[28] 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.

[29] 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.

[30] DV Schorre. Meta ii a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM national conference*, pages 41–301. ACM, 1964.

[31] 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.

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

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

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

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

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